CS 61C:
Great Ideas in Computer Architecture
Compilers and Floating Point

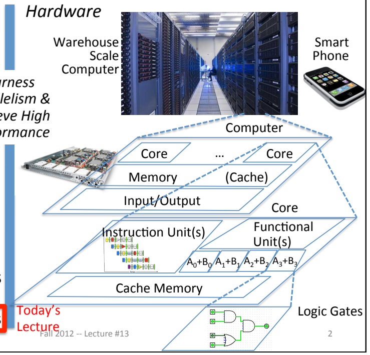
Instructors:
Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12

New-School Machine Structures (It's a bit more complicated!)

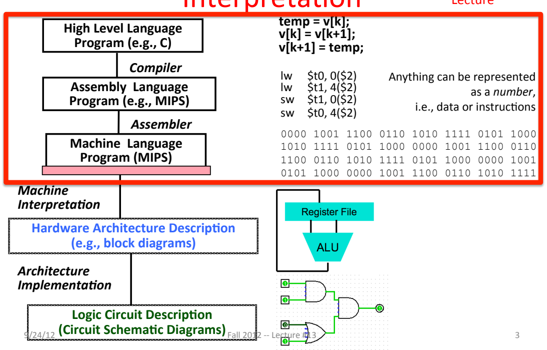
Software | *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance



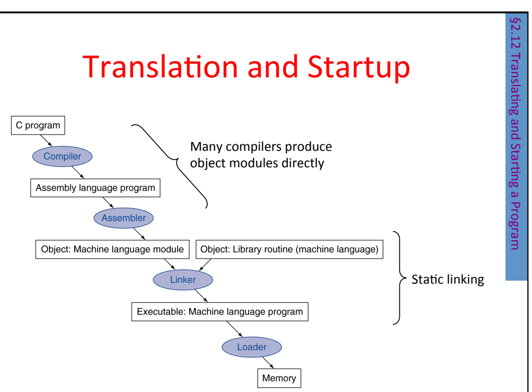
Big Idea #1: Levels of Representation/ Interpretation



Agenda

- Review
- Compilers
- Administrivia
- Floating Point Revisited
- And in Conclusion, …

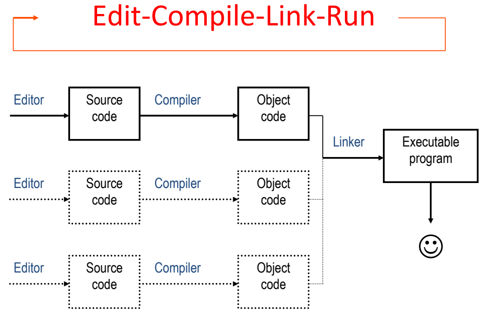
Translation and Startup



What's a Compiler?

- *Compiler*: a program that accepts as input a program text in a certain language and produces as output a program text in another language, *while preserving the meaning of that text*
- Text must comply with the syntax rules of whichever programming language it is written in
- Compiler's complexity depends on the syntax of the language and how much abstraction that programming language provides
  – A C compiler is much simpler than C++ Compiler
- Compiler executes *before* compiled program runs

Compiled Languages: Edit-Compile-Link-Run



Compiler Optimization

- gcc compiler options

-O1: the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time

-O2: Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code

-O3: Optimize yet more. All -O2 optimizations and also turns on the -finline-functions, …

`gcc` Optimization Experiment

	BubbleSort.c	Dhrystone.c
MacAir 3.1 No Opt	3.2 s	7.4 s 3125000 dhrys/s
-O2	1.5 s	2.7 s 8333333 dhrys/s
MacAir 5.1 No Opt	1.9 s (1.7x)	3.0 s (2.4x) 8333333 dhrys/s (2.7x)
-O2	0.8 s (1.9x)	0.7 s (3.8x) 25000000 dhrys/s (3x)

Performance Equation

- Time = $\dfrac{\text{Seconds}}{\text{Program}}$

$$= \boxed{\dfrac{\text{Instructions}}{\text{Program}}} \times \dfrac{\text{Clock cycles}}{\text{Instruction}} \times \dfrac{\text{Seconds}}{\text{Clock Cycle}}$$

Compiler affects this!

What is Typical Benefit of Compiler Optimization?

- What is a typical program?
- For now, try a toy program: BubbleSort.c

```
#define ARRAY_SIZE 20000
int main() {
  int iarray[ARRAY_SIZE], x, y, holder;
for(x = 0; x < ARRAY_SIZE; x++)
   for(y = 0; y < ARRAY_SIZE-1; y++)
     if(iarray[y] > iarray[y+1]) {
       holder = iarray[y+1];
       iarray[y+1] = iarray[y];
       iarray[y] = holder;
     }
}
```


Unoptimized MIPS Code

```
$L3:
    lw     $2,80016($sp)
    slt    $3,$2,20000
    bne    $3,$0,$L6
    j      $L4
$L6:
    .set   noreorder
    nop
    .set   reorder
    sw     $0,80020($sp)
$L7:
    lw     $2,80020($sp)
    slt    $3,$2,19999
    bne    $3,$0,$L10
    j      $L5
$L10:
    lw     $2,80020($sp)
    move   $3,$2
    sll    $2,$3,2
    addu   $3,$sp,16
    addu   $2,$3,$2
    lw     $4,80020($sp)
    addu   $3,$4,1
    move   $4,$3
    sll    $3,$4,2
    addu   $4,$sp,16
    addu   $3,$4,$3
    lw     $2,0($2)
    lw     $3,0($3)
    slt    $2,$3,$2
    beq    $2,$0,$L9
    lw     $3,80020($sp)
    addu   $2,$3,1
    move   $3,$2
    sll    $2,$3,2
    addu   $3,$sp,16
    addu   $2,$3,$2
    lw     $3,0($2)
    sw     $3,80024($sp
    lw     $3,80020($sp)
    addu   $2,$3,1
    move   $3,$2
    sll    $2,$3,2
    addu   $3,$sp,16
    addu   $2,$3,$2
    lw     $3,80020($sp)
    move   $4,$3
    sll    $3,$4,2
    addu   $4,$sp,16
    addu   $3,$4,$3
    lw     $4,0($3)
    sw     $4,0($2)
    lw     $2,80020($sp)
    move   $3,$2
    sll    $2,$3,2
    addu   $3,$sp,16
    addu   $2,$3,$2
    lw     $3,80024($sp)
    sw     $3,0($2)
$L11:
$L9:
    lw     $2,80020($sp)
    addu   $3,$2,1
    sw     $3,80020
    ($sp)
    j      $L7
$L8:
$L5:
    lw     $2,80016($sp)
    addu   $3,$2,1
    sw     $3,80016
    ($sp)
    j      $L3
$L4:
$L2:
    li     $12,65536
    ori
    $12,$12,0x38b0
    addu   $13,$12,$sp
    addu   $sp,$sp,$12
    j      $31
```

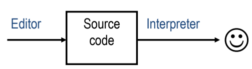
-O2 optimized MIPS Code

```
        li      $13,65536
        ori     $13,$13,0x3890
        addu    $13,$13,$sp
        sw      $28,0($13)
        move    $4,$0
        addu    $8,$sp,16
$L6:
        move    $3,$0
        addu    $9,$4,1
        .p2align 3
$L10:
        sll     $2,$3,2
        addu    $6,$8,$2
        addu    $7,$3,1
        sll     $2,$7,2
        addu    $5,$8,$2
        lw      $3,0($6)
        lw      $4,0($5)
        slt     $2,$4,$3
        beq     $2,$0,$L9
        sw      $3,0($5)
        sw      $4,0($6)
$L9:
        move    $3,$7
        slt     $2,$3,19999
        bne     $2,$0,$L10
        move    $4,$9
        slt     $2,$4,20000
        bne     $2,$0,$L6
        li      $12,65536
        ori     $12,$12,0x38a0
        addu    $13,$12,$sp
        addu    $sp,$sp,$12
        j       $31
        .
```


What's an Interpreter?

- Reads and executes source statements executed one at a time
  - No linking
  - No machine code generation, so more portable
- Starts executing quicker, but runs much more slowly than compiled code
- Performing the actions straight from the text allows better error checking and reporting to be done
- Interpreter stays around during execution
  - Unlike compiler, some work is done *after* program starts
- Writing an interpreter is much less work than writing a compiler

Interpreted Languages: Edit-Run



Compiler vs. Interpreter Advantages

Compilation:
- Faster Execution
- Single file to execute
- Compiler can do better diagnosis of syntax and semantic errors, since it has more info than an interpreter (Interpreter only sees one line at a time)
- Can find syntax errors *before* run program
- Compiler can optimize code

Interpreter:
- Easier to debug program
- Faster development time

Compiler vs. Interpreter Disadvantages

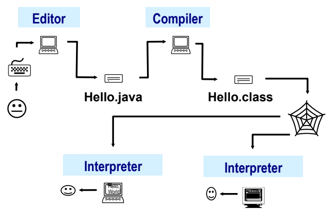
Compilation:
- Harder to debug program
- Takes longer to change source code, recompile, and relink

Interpreter:
- Slower execution times
- No optimization
- Need all of source code available
- Source code larger than executable for large systems
- Interpreter must remain installed while the program is interpreted

Java's Hybrid Approach: Compiler + Interpreter

- A Java compiler converts Java source code into instructions for the *Java Virtual Machine (JVM)*
- These instructions, called *bytecodes*, are same for any computer / OS
- A CPU-specific Java interpreter interprets bytecodes on a particular computer

Java's Compiler + Interpreter



Why Bytecodes?

- Platform-independent
- Load from the Internet faster than source code
- Interpreter is faster and smaller than it would be for Java source
- Source code is not revealed to end users
- Interpreter performs additional security checks, screens out malicious code

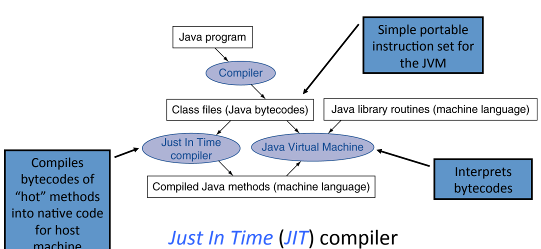
JVM uses Stack vs. Registers

```
a = b + c;
=>
iload b   ;  push b onto Top Of Stack (TOS)
iload c   ;  push c onto Top Of Stack (TOS)
iadd      ;  Next to top Of Stack (NOS) =
          ;  Top Of Stack (TOS) + NOS
istore a  ;  store TOS into a and pop stack
```


Java Bytecodes (Stack) vs. MIPS (Reg.)

Category	Operation	Java bytecode	Size (bits)	MIPS Instr.	Meaning
Arithmetic	add	iadd	8	add	NOS=TOS+NOS; pop
	subtract	isub	8	sub	NOS=TOS–NOS; pop
	increment	iinc I8a I8b	8	addi	Frame[I8a]= Frame[I8a] + I8b
Data transfer	load local integer/address	iload I8/aload I8	16	lw	TOS=Frame[I8]
	load local integer/address	iload_/aload_{0,1,2,3}	8	lw	TOS=Frame[{0,1,2,3}]
	store local integer/address	istore I8/astore I8	16	sw	Frame[I8]=TOS; pop
	load integer/address from array	iaload/aaload	8	lw	NOS=*NOS[TOS]; pop
	store integer/address into array	iastore/aastore	8	sw	*NNOS[NOS]=TOS; pop2
	load half from array	saload	8	lh	NOS=*NOS[TOS]; pop
	store half into array	sastore	8	sh	*NNOS[NOS]=TOS; pop2
	load byte from array	baload	8	lb	NOS=*NOS[TOS]; pop
	store byte into array	bastore	8	sb	*NNOS[NOS]=TOS; pop2
	load immediate	bipush I8, sipush I16	16, 24	addi	push; TOS=I8 or I16
	load immediate	iconst_{–1,0,1,2,3,4,5}	8	addi	push; TOS={–1,0,1,2,3,4,5}
Logical	and	iand	8	and	NOS=TOS&NOS; pop
	or	ior	8	or	NOS=TOS|NOS; pop
	shift left	ishl	8	sll	NOS=NOS << TOS; pop
	shift right	iushr	8	srl	NOS=NOS >> TOS; pop
Conditional branch	branch on equal	if_icompeq I16	24	beq	if TOS == NOS, go to I16; pop2
	branch on not equal	if_icompne I16	24	bne	if TOS != NOS, go to I16; pop2
	compare	if_icomp{lt,le,gt,ge} I16	24	slt	if TOS {<,<=,>,>=} NOS, go to I16; pop2
Unconditional jump	jump	goto I16	24	j	go to I16
	return	ret, ireturn	8	jr	
	jump to subroutine	jsr I16	24	jal	go to I16; push; TOS=PC+3

Starting Java Applications



Just In Time (*JIT*) compiler translates bytecode into machine language just before execution

Agenda

- Review
- Compilers
- Administrivia
- Floating Point Revisited
- And in Conclusion, …

Administrivia

- Lab #5: MIPS Assembly
- HW #4 (of six), due Sunday
- Project 2a: MIPS Emulator, due Sunday
- Midterm, two weeks from Tuesday

THE CLOUD FACTORIES

Power, Pollution and the Internet

Ethan Pines for The New York Times

Data centers are filled with servers, which are like bulked-up desktop computers, minus screens and keyboards, that contain chips to process data.

By JAMES GLANZ
Published: September 22, 2012 | 273 Comments

SANTA CLARA, Calif. — Jeff Rothschild's machines at Facebook had a problem he knew he had to solve immediately. They were about to melt.

FACEBOOK
TWITTER
GOOGLE+

CS61c in the News

"Most data centers, by design, **consume vast amounts of energy in an incongruously wasteful manner**, interviews and documents show. Online companies typically run their facilities at maximum capacity around the clock, whatever the demand. As a result, data centers can waste 90 percent or more of the electricity they pull off the grid, The Times found."

"Worldwide, the digital warehouses use about **30 billion watts of electricity**, roughly equivalent to the output of 30 nuclear power plants, according to estimates industry experts compiled for The Times."

"The consulting firm McKinsey & Company analyzed energy use by data centers and found that, on average, they were **using only 6 percent to 12 percent** of the electricity powering their servers to perform computations. The rest was essentially used to keep servers idling and ready in case of a surge in activity that could slow or crash their operations."

THE CLOUD FACTORIES

Data Barns in a Farm Town, Gobbling Power and Flexing Muscle

Data Barns

Kyle Bair/Bair Aerial

1 2 3 4 5 6 7 8 9 10 11

Relatively low-cost hydroelectric power has lured technology giants to central Washington State.

By JAMES GLANZ
Published: September 23, 2012 | 72 Comments

QUINCY, Wash. — Set in the dry hills and irrigated farmland of Central Washington, Grant County is known for its robust harvest of apples, potatoes, cherries and beans. But for Microsoft, a prime lure was the region's other valuable resource: cheap electrical power.

FACEBOOK
TWITTER
GOOGLE+
E-MAIL

Agenda

- Review
- Compilers
- Administrivia
- **Floating Point Revisited**
- And in Conclusion, …

Goals for Floating Point

- Standard arithmetic for reals for all computers
  - Like two's complement
- Keep as much precision as possible in formats
- Help programmer with errors in real arithmetic
  - +∞, -∞, Not-A-Number (NaN), exponent overflow, exponent underflow
- Keep encoding that is somewhat compatible with two's complement
  - E.g., 0 in Fl. Pt. is 0 in two's complement
  - Make it possible to sort without needing to do floating point comparison

Floating Point: Representing Very Small Numbers

- Zero: Bit pattern of all 0s is encoding for 0.000
  - ⇒ But 0 in exponent should mean most negative exponent (want 0 to be next to smallest real)
  - ⇒ Can't use two's complement (1000 0000_{two})
- *Bias notation*: subtract bias from exponent
  - Single precision uses bias of 127; DP uses 1023
- 0 uses 0000 0000_{two} => 0-127 = -127; ∞, NaN uses 1111 1111_{two} => 255-127 = +128
  - Smallest SP real can represent: 1.00…00 x 2^{-126}
  - Largest SP real can represent: 1.11…11 x 2^{+127}

What If Operation Result Doesn't Fit in 32 Bits?

- *Overflow*: calculate too big a number to represent within a word
- Unsigned numbers: 1 + 4,294,967,295 (2^{32}-1)
- Signed numbers: 1 + 2,147,483,647 (2^{31}-1)
- How to handle depends on the programming language
  - C signed number arithmetic ignores overflow
  - Other languages want overflow signal on signed numbers (e.g., Fortran)
  - What's a computer architect to do?

MIPS Solution: Offer Both

- Instructions that can trigger overflow:
  - `add`, `sub`, `mult`, `div`, `addi`, `multi`, `divi`
- Instructions that don't overflow are called "unsigned" (really means "no overflow"):
  - `addu`, `subu`, `multu`, `divu`, `addiu`, `multiu`, `diviu`
- Given semantics of C, always use unsigned versions
- Note: `slt` and `slti` do signed comparisons, while `sltu` and `sltiu` do unsigned comparisons
  - Nothing to do with overflow
  - When would get different answer for `slt` vs. `sltu`?

Student Roulette?

What About *Real* Numbers in Base 2?

- r x E^i, *E* where is exponent (2), *i* is a positive or negative integer, *r* is a real number ≥ 1.0, < 2
- Computers version of normalized scientific notation called *Floating Point* notation

Floating Point Numbers

- 32-bit word has 2^{32} patterns, so must be approximation of real numbers ≥ 1.0, < 2
- IEEE 754 Floating Point Standard:
  - 1 bit for *sign (s)* of floating point number
  - 8 bits for *exponent (E)*
  - 23 bits for *fraction (F)* (get 1 extra bit of precision if leading 1 is implicit)

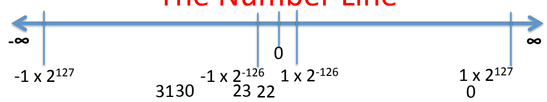
$(-1)^s \times (1 + F) \times 2^E$

- Can represent from 2.0 x 10^{-38} to 2.0 x 10^{38}

Bias Notation (+127)

	How it is interpreted		How it is encoded	
	Decimal Exponent	signed 2's complement	Biased Notation	Decimal Value of Biased Notation
∞, NaN	For infinities		11111111	255
	127	01111111	11111110	254
	...	...	...	...
Getting closer to zero	2	00000010	10000001	129
	1	00000001	10000000	128
	0	00000000	01111111	127
	-1	11111111	01111110	126
	-2	11111110	01111101	125
	...	...	...	...
	-126	10000010	00000001	1
Zero	For Denorms	10000001	00000000	0

The Number Line



	31	30 23	22 0	
$(-1)^S \times (1 + F) \times 2^E$	S	E	F	
∞	0	11111111	00000000000000000000000	
1×2^{127}	0	11111110	00000000000000000000000	127 + 127 = 254
1×2^{-126}	0	00000001	00000000000000000000000	-126 + 127 = 1
0	0	00000000	00000000000000000000000	
-1×2^{-126}	1	00000001	00000000000000000000000	-126 + 127 = 1
-1×2^{127}	1	11111110	00000000000000000000000	127 + 127 = 254
-∞	1	11111111	00000000000000000000000	

Floating Point Numbers

- What about bigger or smaller numbers?
- IEEE 754 Floating Point Standard: *Double Precision* (64 bits)
  - 1 bit for *sign (s)* of floating point number
  - 11 bits for *exponent (E)*
  - 52 bits for *fraction (F)* (get 1 extra bit of precision if leading 1 is implicit)

$(-1)^s \times (1 + F) \times 2^E$

- Can represent from 2.0×10^{-308} to 2.0×10^{308}
- 32 bit format called *Single Precision*

More Floating Point

- What about 0?
  - Bit pattern all 0s means 0, so no implicit leading 1
- What if divide 1 by 0?
  - Can get infinity symbols +∞, -∞
  - Sign bit 0 or 1, largest exponent, 0 in fraction
- What if do something stupid? (∞ – ∞, 0 ÷ 0)
  - Can get special symbols NaN for Not-a-Number
  - Sign bit 0 or 1, largest exponent, not zero in fraction
- What if result is too big? ($2 \times 10^{308} \times 2 \times 10^{2}$)
  - Get *overflow* in exponent, alert programmer!
- What if result is too small? ($2 \times 10^{-308} \div 2 \times 10^{2}$)
  - Get *underflow* in exponent, alert programmer!

Floating Point Add Associativity?

- A = (1000000.0 + 0.000001) - 1000000.0
- B = (1000000.0 - 1000000.0) + 0.000001
- In single precision floating point arithmetic, A does not equal B
  - A = 0.000000, B = 0.000001
- Floating Point Addition is not Associative!
  - Integer addition is associative
- When does this matter?

MIPS Floating Point Instructions

- C, Java has single precision (`float`) and double precision (`double`) types
- MIPS instructions: .s for single, .d for double
  - Fl. Pt. Addition single precision: add.s
    Fl. Pt. Addition double precision: add.d
  - Fl. Pt. Subtraction single precision: sub.s
    Fl. Pt. Subtraction double precision: sub.d
  - Fl. Pt. Multiplication single precision: mul.s
    Fl. Pt. Multiplication double precision: mul.d
  - Fl. Pt. Divide single precision: div.s
    Fl. Pt. Divide double precision: div.d

MIPS Floating Point Instructions

- C, Java have single precision (`float`) and double precision (`double`) types
- MIPS instructions: `.s` for single, `.d` for double
  - Fl. Pt. Comparison single precision:
    Fl. Pt. Comparison double precision:
  - Fl. Pt. branch:
- Since rarely mix integers and Floating Point, MIPS has separate registers for floating-point operations: `$f0`, `$f1`, …, `$f31`
  - Double precision uses adjacent even-odd pairs of registers:
  - `$f0` and `$f1`, `$f2` and `$f3`, `$f4` and `$f5`, …, `$f30` and `$f31`
- Need data transfer instructions for these new registers
  - `lwc1` (load word), `swc1` (store word)
  - Double precision uses two `lwc1` instructions, two `swc1` instructions

Who is Bigger than Whom? What's the Representation?

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$
$> 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$
$> 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$
$> 1111\ 1111\ 0111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$

- Ones Complement
- Twos Complement
- Sign and Magnitude
- IEEE FP Standard
- **More than one can apply**

Who is Bigger than Whom? What's the Representation?

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$
$> 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$
$> 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$
$> 1111\ 1111\ 0111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$

- Ones Complement
- Twos Complement
- Sign and Magnitude
- IEEE FP Standard
- **More than one or none can apply**

Peer Instruction Question

Suppose Big, Tiny, and BigNegative are floats in C, with Big initialized to a big number (e.g., age of universe in seconds or 4.32 x 10^{17}), Tiny to a small number (e.g., seconds/femtosecond or 1.0 x 10^{-15}), BigNegative = - Big.
Here are two conditionals:

I. (Big * Tiny) * BigNegative == (Big * BigNegative) * Tiny
II. (Big + Tiny) + BigNegative == (Big + BigNegative) + Tiny

Which statement about these is correct?

Orange. I. is false and II. is false
Green. I. is false and II. is true
Pink. I. is true and II. is false
Yellow. I. is true and II. is true

Peer Instruction Answer

Suppose Big, Tiny, and BigNegative are floats in C, with Big initialized to a big number (e.g., age of universe in seconds or 4.32 x 10^{17}), Tiny to a small number (e.g., seconds/femtosecond or 1.0 x 10^{-15}), BigNegative = - Big.
Here are two conditionals:

I. (Big * Tiny) * BigNegative == (Big * BigNegative) * Tiny
II. (Big + Tiny) + BigNegative == (Big + BigNegative) + Tiny

Which statement about these is correct?

Pink. I. is true and II. is false (if we don't consider overflow)—but there are cases where one side overflows while the other does not!

I. Works ok if no overflow, but because exponents add, if Big * BigNeg overflows, then result is overflow, not -1
II. Left hand side is 0, right hand side is tiny

Pitfalls

- Floating point addition is NOT associative
- Some optimizations can change order of floating point computations, which can change results
- Need to ensure that floating point algorithm is correct even with optimizations

And, in Conclusion, …

- Interpreters for debugging, but slow execution
- Hybrid (Java): Compiler + Interpreter to try to get best of both
- Compiler Optimization to relieve programmer
- Floating point is an approximation of reals