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

- 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

Big Idea #1: Levels of Representation/Interpretation

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
What is Typical Benefit of Compiler Optimization?

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

```c
#include 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;
            }
}
```

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 -ffine-functions, ...

Performance Equation

\[
\text{Time} = \frac{\text{Instructions} \times \text{Clock cycles}}{\text{Seconds} \times \text{Clock Cycle}}
\]

Compiler affects this!

gcc Optimization Experiment

<table>
<thead>
<tr>
<th></th>
<th>BubbleSort.c</th>
<th>Dhrystone.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacAir 3.1 No Opt</td>
<td>3.2 s</td>
<td>7.4 s</td>
</tr>
<tr>
<td>MacAir 5.1 No Opt</td>
<td>1.9 s (1.7x)</td>
<td>3.0 s (2.4x)</td>
</tr>
<tr>
<td><code>-O2</code></td>
<td>1.5 s</td>
<td>2.7 s</td>
</tr>
<tr>
<td><code>-O2</code></td>
<td>0.8 s (1.9x)</td>
<td>0.7 s (3.8x)</td>
</tr>
</tbody>
</table>
-O2 optimized MIPS Code

```mips
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

**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

- Editor
- Compiler
- Hello.java
- Hello.class
- Interpreter
- Interpret

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.)

Starting Java Applications

- Simple portable instruction set for the JVM
- Compiles bytecodes of "hot" methods into native code for host machine
- 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

---

**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.”

---

**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
  - NaN, 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

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

- **Overflow**: calculate too big a number to represent within a word
- **Unsigned numbers**: $1 + 4,294,967,295 \times (2^{31} - 1)$
- **Signed numbers**: $1 + 2,147,483,647 \times (2^{31} - 1)$
- 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?

What About Real Numbers in Base 2?

- $r \times 2^i$, $i$ where is exponent (2), $i$ is a positive or negative integer, $r$ is a real number $\geq 1.0, < 2$
- Computers version of normalized scientific notation called Floating Point notation

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,\text{max})
- **Bias notation**: subtract bias from exponent
  - Single precision uses bias of 127; DP uses 1023
- 0 uses $0000\ldots00_2 = 0\cdot 2^{0-127} = -127$;
  - $\infty$, NaN uses $1111\ldots11_2 = 2^{127} + 128$
  - Smallest SP real can represent: $1.00\ldots00 \times 2^{126}$
  - Largest SP real can represent: $1.11\ldots11 \times 2^{127}$

MIPS Solution: Offer Both

- Instructions that can trigger overflow:
  - add, sub, mul, div, addi, multi, diviu
- 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?

Floating Point Numbers

- 32-bit word has $2^{23}$ patterns, so must be approximation of real numbers $\geq 1.0, < 2$
- 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 $2.0 \times 10^{38}$ to $2.0 \times 10^{38}$
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 \(2.0 \times 10^{-308}\) to \(2.0 \times 10^{308}\)
• 32 bit format called Single Precision

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?

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 \(+\infty, -\infty\)
  - Sign bit 0 or 1, largest exponent, 0 in fraction
• What if do something stupid? (\(\infty - \infty, 0 \div 0\))
  - 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? (\(2 \times 10^{308} \times 2 \times 10^{308}\))
  - Get overflow in exponent, alert programmer!
• What if result is too small? (\(2 \times 10^{-308} + 2 \times 10^{-308}\))
  - Get underflow in exponent, alert programmer!

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: $f0, f1, \ldots, f31$
  - Double precision uses adjacent even-odd pairs of registers:
    - $f0$ and $f1$, $f2$ and $f3$, $f4$ and $f5$, $f6$ and $f7$
- Need data transfer instructions for these new registers
  - lwcl (load word), swcl (store word)
  - Double precision uses two lwcl instructions, two swcl instructions

Who is Bigger than Whom?

What's the Representation?

$0000 0000 0000 0000 0000 0000 0000 0000 = 0$
$1111 1111 1111 1111 1111 1111 1111 1111 = -1$
$1111 1111 1111 1111 1111 1111 1111 0100 = \text{Tiny}$
$1111 1111 1111 1111 1111 1111 1111 1000 = \text{Big}$
$1000 0000 0000 0000 0000 0000 0000 0000 = \text{Sign and Magnitude}$
$1000 0000 0000 0000 0000 0000 0000 0000 = \text{IEEE FP Standard}$
$1000 0000 0000 0000 0000 0000 0000 0000 = \text{More than one can apply}$

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 $4.32 \times 10^{17}$), Tiny to a small number (e.g., seconds/femtosecond or $1.0 \times 10^{-15}$), BigNegative = -Big. Here are two conditionals:

1. (Big * Tiny) * BigNegative == (Big * BigNegative) * Tiny
2. (Big + Tiny) + BigNegative == (Big + BigNegative) + Tiny

Which statement about these is correct?

Orange. I. is false and II. is true
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 $4.32 \times 10^{17}$), Tiny to a small number (e.g., seconds/femtosecond or $1.0 \times 10^{-15}$), BigNegative = -Big. Here are two conditionals:

1. (Big * Tiny) * BigNegative == (Big * BigNegative) * Tiny
2. (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!
1. Works ok if no overflow, but because exponents add, if Big * BigNeg overflows, then result is overflow, not -1
2. 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

Who is Bigger than Whom?

What’s the Representation?

$0000 0000 0000 0000 0000 0000 0000 0000 = 0$
$1111 1111 1111 1111 1111 1111 1111 1111 = -1$
$1111 1111 1111 1111 1111 1111 1111 0100 = \text{Tiny}$
$1111 1111 1111 1111 1111 1111 1111 1000 = \text{Big}$
$1000 0000 0000 0000 0000 0000 0000 0000 = \text{Sign and Magnitude}$
$1000 0000 0000 0000 0000 0000 0000 0000 = \text{IEEE FP Standard}$
$1000 0000 0000 0000 0000 0000 0000 0000 = \text{More than one can apply}$
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