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

- **Programming Languages**
  }
Agenda

- Review: Single + Double Precision FP
- Assemblers
- Administrivia
- Linkers
- Compilers vs. Interpreters
- And in Conclusion, ...
Agenda

- Review: Single + Double Precision FP
- Assemblers
- Administrivia
- Linkers
- Compilers vs. Interpreters
- And in Conclusion, ...

Bias Notation (+127)

<table>
<thead>
<tr>
<th>How it is interpreted</th>
<th>How it is encoded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal Exponent</td>
<td>signed 2's complement</td>
</tr>
<tr>
<td>∞, NaN</td>
<td>01111111</td>
</tr>
<tr>
<td>Getting closer to zero</td>
<td>00000010</td>
</tr>
<tr>
<td>Zero</td>
<td>00000000</td>
</tr>
<tr>
<td>-1</td>
<td>11111111</td>
</tr>
<tr>
<td>-2</td>
<td>11111110</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>-126</td>
<td>10000010</td>
</tr>
<tr>
<td>For Denorms</td>
<td>1000000</td>
</tr>
</tbody>
</table>
Review: Single Precision Floating Point Numbers

• 32-bit word has $2^{32}$ 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)
  \[-1]^s \times (1 + F) \times 2^E\]
• Ranges from $2.0 \times 10^{-38}$ to $2.0 \times 10^{38}$

Review: Double Precision Floating Point Numbers

• What about bigger or smaller numbers?
• IEEE 754 Floating Point Standard:
  \textit{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)
  \[-1]^s \times (1 + F) \times 2^E\]
• Range from $2.0 \times 10^{-308}$ to $2.0 \times 10^{308}$
Floating Point Pitfalls

• \( A = (1000000.0 + 0.000001) - 1000000.0 \)
• \( B = (1000000.0 - 1000000.0) + 0.000001 \)
• 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?

Agenda

• Review: Single + Double Precision FP
• Assemblers
• Administrivia
• Linkers
• Compilers vs. Interpreters
• And in Conclusion, ...
Assembler

- Input: Assembly Language Code (e.g., foo.s for MIPS)
- Output: Object Code, information tables (e.g., foo.o for MIPS)
- Reads and Uses Directives
- Replace Pseudo-instructions
- Produce Machine Language
- Creates Object File

Translation

Many compilers produce object modules directly.
Assembly and Pseudo-instructions

- Turning textual MIPS instructions into machine code called assembly, program called assembler
  - Calculates addresses, maps register names to numbers, produces binary machine language
  - Textual language called assembly language
- Can also accept instructions convenient for programmer but not in hardware
  - Load immediate (li) allows 32-bit constants, assembler turns into lui + ori (if needed)
  - Load double (ld) uses two lwc1 instructions to load a pair of 32-bit floating point registers
  - Called Pseudo-Instructions

Assemble Directives (P&H Appendix A)

- Give directions to assembler, but do not produce machine instructions
  
  .text: Subsequent items put in user text segment
  .data: Subsequent items put in user data segment
  .globl sym: declares sym global and can be referenced from other files
  .asciiz str: Store the string str in memory and null-terminate it
  .word w1…wn: Store the n 32-bit quantities in successive memory words
Assembler Pseudo-instructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudo-instructions: figments of the assembler’s imagination
  
  move $t0, $t1 → add $t0, $zero, $t1
  blt $t0, $t1, L → slt $at, $t0, $t1
  bne $at, $zero, L
  
  -- $at (register 1): assembler temporary

More Pseudo-instructions

- Asm. treats convenient variations of machine language instructions as if real instructions
  
  Pseudo: Real:
  addu $t0,$t6,1 ______________
  subu $sp,$sp,32 ______________
  sd $a0,32($sp) ______________
  la $a0,str ______________
Agenda

• Review: Single + Double Precision FP
• Assemblers
• **Administrivia**
• Linkers
• Compilers vs. Interpreters
• And in Conclusion, ...
Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Separate Compilation and Assembly

- No need to compile all code at once
- How to put pieces together?

FIGURE 8.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file. Copyright © 2009 Elsevier, Inc. All rights reserved.
Many compilers produce object modules directly.

Static linking

Linker Stitches Files Together

FIGURE B.3.1 The linker searches a collection of object files and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files. Copyright © 2009 Elsevier, Inc. All rights reserved.
Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Often slower than compiling
  - All the machine code files must be read into memory and linked together

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space (covered later in semester)
  3. Copy text and initialized data into memory
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
    - Copies arguments to $a0, ... and calls main
    - When main returns, do “exit” systems call
Agenda

- Review: Floating Point Numbers
- Assemblers
- Administrivia
- Linkers
- Compilers vs. Interpreters
- And in Conclusion, ...

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, ...
What is Typical Benefit of Compiler Optimization?

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

```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)  addu $2,$3,$2
slt $3,$2,20000  addu $3,$4,$1
bne $3,$0,$14     move $4,$3
lor $3,$5,$3
j $14             sll $2,$3,2

$L6:          .set no reorder
      .set reorder
sw $0,80020($sp)

$L7:          lw $2,80020($sp)  addu $2,$3,$1
      slt $2,$0,$19
      bne $3,$2,19999
      bne $3,$0,$110
      j $15

$L10:         lw $2,80020($sp)  move $3,$2
      sll $2,$3,2
      addu $3,$sp,16
      addu $2,$3,$2
      lw $3,0($2)
      sw $3,80024($sp)

$L11:         lw $3,80020($sp)  addu $2,$3,$1
      addu $3,$2,$3
      sll $2,$3,2
      addu $3,$sp,16
      addu $2,$3,$2
      lw $3,0($2)
      sw $3,80024($sp)

$L12:         lw $3,80020($sp)  addu $2,$3,$1
      addu $3,$2,$3
      sll $2,$3,2
      addu $3,$sp,16
      addu $2,$3,$2
      lw $3,0($2)
      sw $3,80024($sp)
      sw $3,0($2)

$L13:         li $12,65536
      ori $12,$12,0x38b0
      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
gcc BubbleSort.c with and without Optimization

```
4.c  exponential.c
dhcp-47-110:c examples randykatz$ gcc BubbleSort.c
dhcp-47-110:c examples randykatz$ ./a.out
Before = 7049650589779
After = 7053593428157
Diff = 3942838378
Seconds = 1.971
2 seconds of execution
dhcp-47-110:c examples randykatz$ gcc -O2 BubbleSort.c
dhcp-47-110:c examples randykatz$ ./a.out
Before = 7116398255825
After = 7118036173650
Diff = 1637917825
Seconds = 0.819
1 seconds of execution
dhcp-47-110:c examples randykatz$
```
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 the 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

\[
a = b + c;
\]

\[
=>
\]

\[
i_{load} \ b \ ; \ \text{push } b \ \text{onto Top Of Stack (TOS)}
\]

\[
i_{load} \ c \ ; \ \text{push } c \ \text{onto Top Of Stack (TOS)}
\]

\[
i_{add} \ ; \ \text{Next to top Of Stack (NOS) =}
\]

\[
\text{Top Of Stack (TOS) + NOS}
\]

\[
i_{store} \ a \ ; \ \text{store TOS into } a \ \text{and pop stack}
\]
### Starting Java Applications

- **Java program**
- **Compiler**
- **Class files (Java bytecodes)**
- **Java Virtual Machine**
- **Java library routines (machine language)**
- **Compiled Java methods (machine language)**
- **Interprets bytecodes**

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

---

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Operation</th>
<th>Java bytecode</th>
<th>Size (bits)</th>
<th>MIPS Inst.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add</td>
<td>8</td>
<td>add NOS:TOS=NOS; pop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub</td>
<td>8</td>
<td>NOS:TOS=NOS; pop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>increment</td>
<td>inc I8/I16</td>
<td>8</td>
<td>Frame(I8)+ Frame(I16)</td>
<td></td>
</tr>
<tr>
<td>Data transfer</td>
<td>load local integer/address</td>
<td>load IR/aload IR</td>
<td>16</td>
<td>lw TOS=Frame[I8]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>store local integer/address</td>
<td>store IR/astore IR</td>
<td>16</td>
<td>lw Frame[I8]=TOS; pop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>load integer/address from array</td>
<td>load astore/aload astore IR</td>
<td>8</td>
<td>lw NOS*NOS[TOS]; pop2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>store integer/address into array</td>
<td>astore/astore</td>
<td>8</td>
<td>NOS*NOS[TOS]; pop2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>load half from array</td>
<td>lui I8</td>
<td>8</td>
<td>NOS=NOS[TOS]; pop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>store half into array</td>
<td>sth I8</td>
<td>8</td>
<td>NOS*NOS[TOS]; pop2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>load byte from array</td>
<td>lb</td>
<td>8</td>
<td>NOS*NOS[TOS]; pop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>store byte into array</td>
<td>sb</td>
<td>8</td>
<td>NOS*NOS[TOS]; pop2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>load immediate</td>
<td>add I8</td>
<td>16, 24</td>
<td>push; TOS=I8 or I16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>load immediate constant</td>
<td>add I8</td>
<td>16, 24</td>
<td>add push; TOS=I8</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>and</td>
<td>and</td>
<td>8</td>
<td>NOS-TOS=NOS; pop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>or</td>
<td>8</td>
<td>NOS*NOS or TOS[TOS]; pop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>shift left</td>
<td>isl</td>
<td>8</td>
<td>NOS&lt;TOS; NOS &lt;&lt; TOS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>shift right</td>
<td>uisl</td>
<td>8</td>
<td>NOS&gt;NOS &gt;&gt; TOS; pop</td>
<td></td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>beq</td>
<td>24</td>
<td>if TOS == NOS, go to I16; pop2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne</td>
<td>24</td>
<td>if TOS != NOS, go to I16; pop2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>compare</td>
<td>bgt</td>
<td>24</td>
<td>if TOS &gt; NOS, go to I16; pop2</td>
<td></td>
</tr>
<tr>
<td>Unconditional</td>
<td>jump</td>
<td>jal</td>
<td>24</td>
<td>go to I16; push; TOS=PC+3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return</td>
<td>jr</td>
<td>8</td>
<td>go to I16; push; TOS=PC+3</td>
<td></td>
</tr>
</tbody>
</table>

---

9/24/13
And, in Conclusion, ...

- Assemblers can enhance machine instruction set to help assembly-language programmer
- Translate from text that easy for programmers to understand into code that machine executes efficiently: Compilers, Assemblers
- Linkers allow separate translation of modules
- Interpreters for debugging, but slow execution
- Hybrid (Java): Compiler + Interpreter to try to get best of both
- Compiler Optimization to relieve programmer