CS 61C: Great Ideas in Computer Architecture (Machine Structures)
Compilers, Interpreters, ...

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/fa10

Agenda

• Review
• Compilers Assemblers and Linkers
• Administrivia
• Technology Break
• Compilers vs. Interpreters
Review

• Request-Level Parallelism
  – High request volume, each largely independent of the other
  – Use replication for better request throughput and availability

• Map-Reduce Data Parallelism
  – Divide large data set into pieces for independent parallel processing
  – Combine and process intermediate results to obtain final result

Translation and Startup

Many compilers produce object modules directly

Static linking

C program
  \[\text{Compiler} \rightarrow \text{Assembler} \rightarrow \text{Object: Machine language module}, \text{Object: Library routine (machine language)} \rightarrow \text{Linker} \rightarrow \text{Executable: Machine language program} \rightarrow \text{Loader} \rightarrow \text{Memory} \]
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*

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: 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
Example C program

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    int sum = 0;
    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

FIGURE B.1.5 The routine written in the C programming language. Copyright © 2009 Elsevier, Inc. All rights reserved.

Assembly Language Output from Compiler (using labels)

```
.text
.globl main
main:
    subu $sp, $sp, 32
    sw $t0, 0($sp)
    sw $t3, 2($sp)
    sw $t0, 4($sp)
    sub $t0, $t0, 2
    beq $t0, 100, loop
    jal printf
    jr $ra
    .asciiz "The sum from 0 .. 100 is Zn"
```

FIGURE B.1.4 The same routine written in assembly language with labels, but no comments. The commands that start with periods are assembler directives (see pages B-47–49). .text indicates that succeeding lines contain instructions .data indicates that they contain data .align n indicates that the items on the succeeding lines should be aligned on a 2^n byte boundary. .align 2 means the next item should be on a word boundary. .globl main declares that main is a global symbol that should be visible to code stored in other files. Finally, .asciiz stores a null-terminated string in memory. Copyright © 2009 Elsevier, Inc. All rights reserved.
Assembly Language Output from Compiler (after labels replaced)

```assembly
addiu $29, $29, -32
sw $31, 0($29)
sw $4, 32($29)
sw $5, 36($29)
sw $0, 24($29)
sw $10, 28($29)
lw $14, 28($29)
lw $24, 24($29)
multi $14, $14
addiu $0, $14, 1
slt $1, $0, 101
sw $0, 28($29)
mflo $15
addu $25, $24, $15
bne $1, $0, -9
sw $25, 24($29)
lui $4, 4096
lw $5, 24($29)
jal 1048812
addiu $4, $4, 1072
lw $31, 20($29)
addiu $29, $29, 32
jr $31
move $2, $0
```

FIGURE B.1.3 The same routine written in assembly language. However, the code for the routine does not label registers or memory locations nor include comments. Copyright © 2009 Elsevier, Inc. All rights reserved.

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 put pieces together?

![Diagram](image)

FIGURE B.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.

Linker Stitches Files Together

![Diagram](image)

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 a slower than compiling
  – all the machine code files must be read into memory and linked together

Resulting MIPS Machine Code

```
001001111011101111111111110000
101011110101111111111111110000
10101111010110000000010100
10101111010110000000010100
10101111010000000000011000
10101111010000000000011000
10001111010111000000011000
00000011101100000000001100
00101001101000000000000101
01010000000000010000000110
10101111101000000000011100
00000000000001111111000100
00000110001111111111111111
10101111101001100000000110
00110000000100000000000110
00101001100001000000000110
00101111111111111111111111
00000001110000000000000110
00000001111111111111111111
10101111101000000000011100
00000110001111111111111111
10101111101001100000000110
00110000000100000000000110
00101001100001000000000110
00101111111111111111111111
00000001110000000000000110
00000001111111111111111111
```

FIGURE B.1.2 MIPS machine language code for a routine to compute and print the sum of the squares of integers between 0 and 100. Copyright © 2009 Elsevier, Inc. All rights reserved.
Loading a Program

• Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space (cover 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

Dynamic Linking

• Only link/load library procedure when it is called
  – Requires procedure code to be relocatable
  – Avoids image bloat caused by static linking of all (transitively) referenced libraries
  – Automatically picks up new library versions
Dynamic Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code

Agenda

• Review
• Compilers Assemblers and Linkers
• Administrivia
• Technology Break
• Compilers vs. Interpreters
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.
- The text must comply with the syntax rules of whichever programming language it is written in.
- A 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.

Compiled Languages: Edit-Compile-Link-Run

- Editor → Source code → Compiler → Object code → Linker → Executable program
What’s an Interpreter?

• It reads and executes source statements executed one at a time
  – No linking
  – No machine code generation, so more portable
• Start executing quicker, but run much more slowly than compiled code
• Performing the actions straight from the text allows better error checking and reporting to be done
• The interpreter stays around during execution
• Writing an interpreter is much less work than writing a compiler

Interpreted Languages:

```
Edit-Run
```

```
Editor -> Source code -> Interpreter
```
Compilation Advantages

• 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

Compilation Disadvantages

• Harder to debug program
• Takes longer to change source code, recompile and relink
Interpreter Advantages

- Easier to debug program
- Faster development time

Interpreter disadvantages

- 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
- These instructions, called bytecodes, are the same for any computer / operating system.
- A CPU-specific Java interpreter interprets bytecodes on a particular computer.

Java’s Compiler + Interpreter

Diagram showing the process from editing a file to running the program.
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

<table>
<thead>
<tr>
<th>Category</th>
<th>Operation</th>
<th>Java bytecode</th>
<th>Size (bits)</th>
<th>MIPS Instr.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add</td>
<td>8</td>
<td>add</td>
<td>NOS=TOS+NOS; pop</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub</td>
<td>8</td>
<td>sub</td>
<td>NOS=TOS-NOS; pop</td>
</tr>
<tr>
<td></td>
<td>increment</td>
<td>inc</td>
<td>8</td>
<td>add</td>
<td>Frame[0:8]+Frame[9:16] + Ilb</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load local int/address</td>
<td>load/aload 18</td>
<td>56</td>
<td>lw</td>
<td>TOS-Frame[18]</td>
</tr>
<tr>
<td></td>
<td>load local int/address</td>
<td>load/aload [0,1,2,3]</td>
<td>8</td>
<td>lw</td>
<td>TOS-Frame[0,1,2,3]</td>
</tr>
<tr>
<td></td>
<td>store local int/address</td>
<td>store/astore 18</td>
<td>56</td>
<td>sw</td>
<td>Frame[18]=TOS; pop</td>
</tr>
<tr>
<td></td>
<td>load int/address from array</td>
<td>load/aload</td>
<td>8</td>
<td>lw</td>
<td>NOS-NOS(TOS); pop</td>
</tr>
<tr>
<td></td>
<td>store int/address into array</td>
<td>store/astore</td>
<td>8</td>
<td>sw</td>
<td>NOS*NOS(TOS); pop+pop2</td>
</tr>
<tr>
<td></td>
<td>load half from array</td>
<td>saaload</td>
<td>8</td>
<td>lh</td>
<td>NOS=NOS(TOS); pop</td>
</tr>
<tr>
<td></td>
<td>store half into array</td>
<td>saastore</td>
<td>8</td>
<td>sh</td>
<td>*NOS(NOS)=TOS; pop+pop2</td>
</tr>
<tr>
<td></td>
<td>load byte from array</td>
<td>baload</td>
<td>8</td>
<td>lb</td>
<td>NOS*NOS(TOS); pop</td>
</tr>
<tr>
<td></td>
<td>store byte into array</td>
<td>basteore</td>
<td>8</td>
<td>b</td>
<td>*NOS(NOS)=TOS; pop+pop2</td>
</tr>
<tr>
<td></td>
<td>load immediate</td>
<td>bipush 16,ajump [16]</td>
<td>16,24</td>
<td>add</td>
<td>push; TOS=I8 or I16</td>
</tr>
<tr>
<td>Logical</td>
<td>and</td>
<td>and</td>
<td>8</td>
<td>and</td>
<td>NOS-TOS=NOS+pop</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>or</td>
<td>8</td>
<td>or</td>
<td>NOS-TOS=NOS; pop</td>
</tr>
<tr>
<td></td>
<td>shift left</td>
<td>sll</td>
<td>8</td>
<td>sll</td>
<td>NOS=NOS &lt;&lt; TOS; pop</td>
</tr>
<tr>
<td></td>
<td>shift right</td>
<td>sra</td>
<td>8</td>
<td>srl</td>
<td>NOS=NOS &gt;&gt; TOS; pop</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>if_icmpeq 16</td>
<td>24</td>
<td>beq</td>
<td>if TOS == NOS, go to I16; pop+pop2</td>
</tr>
<tr>
<td></td>
<td>branch not equal</td>
<td>if_icmpeq 16</td>
<td>24</td>
<td>bne</td>
<td>if TOS != NOS, go to I16; pop+pop2</td>
</tr>
<tr>
<td></td>
<td>compare</td>
<td>if_icmpeq[lt,gt,eq] 16</td>
<td>24</td>
<td>st</td>
<td>if TOS (&lt;,&gt;,&gt;)=NOS, go to I16; pop+pop2</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>jump</td>
<td>goto 116</td>
<td>24</td>
<td>j</td>
<td>go to I16</td>
</tr>
<tr>
<td></td>
<td>return</td>
<td>ret,retum</td>
<td>8</td>
<td>jr</td>
<td>return</td>
</tr>
<tr>
<td></td>
<td>jump to subroutine</td>
<td>jal 16</td>
<td>24</td>
<td>jal</td>
<td>go to I16; push; TOS=PC+3</td>
</tr>
</tbody>
</table>
Starting Java Applications

- Java program
  - Compiles bytecodes of "hot" methods into native code for host machine
  - Interprets bytecodes

- Simple portable instruction set for the JVM
  - Class files (Java bytecodes)
  - Java library routines (machine language)
  - Just In Time (JIT) compiler translates bytecode into machine language just before execution
  - Java Virtual Machine
  - Compiled Java methods (machine language)

Summary

- 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