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

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 (l) 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

Agenda

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

Translation and Startup

- Many compilers produce object modules directly
- Static linking

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
  - bne $at, $zero, L
  - $at (register 1): assembler temporary
Example C program

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i, sum = 0;
    for (i = 0; i < 100; i++)
        sum = sum + i;
    printf("The sum from 0..99 is %d", sum);
    return 0;
}
```

Assembly Language Output from Compiler (using labels)

```
.globl main
main:
    li a0,0
    li a1,0
    jal sum
    li a0,99
    li a1,0
    jal add
```

Assembly Language Output from Compiler (after labels replaced)

```
add:
    li a0,0
    li a1,0
    jal sum
    li a0,99
    li a1,0
    jal add
```

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?

```
  Source File
  |     |
  |     |
  Assembler
  |     |
  |     |
  Object File
  |     |
  |     |
  Linker
  |     |
  |     |
  Executable File
```

Linker Stitches Files Together

```
Object File
  |     |
  |     |
  Relocation record
  |     |
  |     |
  C Library
  |     |
  |     |
  Linker
  |     |
  |     |
  Executable File
```

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

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 items on the succeeding lines should be aligned on a 2n-byte boundary. Hence, `.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.

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.

FIGURE B.1.2 The process that produces an executable file. An assembler translates the 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.

FIGURE B.1.1 The process that produces an executable file. An assembler translates the 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.
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

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. 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

Dynamic MIPS Machine 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.

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

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

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

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

- Java program
  - Class files (Java bytecodes)
  - Java library routines (machine language)

- Simple portable instruction set for the JVM
- Just In Time (JIT) compiler
  - Translates bytecode into machine language just before execution
- Interprets bytecodes
- Compiled Java methods (machine language)
- Interprets bytecodes

Summary

- Translate from text that is 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