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

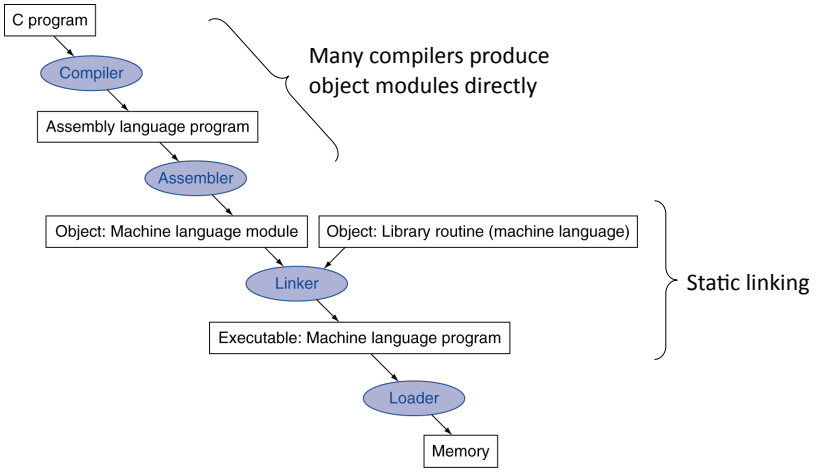
- Review
- Compilers Assemblers and Linkers
- Administrivia
- Technology Break
- Compilers vs. Interpreters

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

§2.12 Translating and Starting a Program



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

```
#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.

Assembly Language Output from Compiler (using labels)

```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        addu    $t9, $t8, $t7
        sw      $t9, 24($sp)
        addu    $t0, $t6, 1
        sw      $t0, 28($sp)
        ble     $t0, 100, loop
        la      $a0, str
        lw      $a1, 24($sp)
        jal     printf
        move    $v0, $0
        lw      $ra, 20($sp)
        addu    $sp, $sp, 32
        jr      $ra


        .data
        .align  0
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

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. 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 fi les. Finally, .asciiz stores a null-terminated string in memory.

Assembly Language Output from Compiler (after labels replaced)

```
addiu    $29, $29, -32
sw       $31, 20($29)
sw       $4,  32($29)
sw       $5,  36($29)
sw       $0,  24($29)
sw       $0,  28($29)
lw       $14, 28($29)
lw       $24, 24($29)
multu    $14, $14
addiu    $8,  $14, 1
slti     $1,  $8, 101
sw       $8,  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.

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?

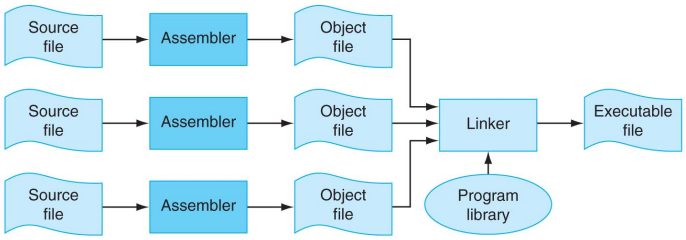


FIGURE B.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file. Copyright © 2009 Elsevier, Inc. All rights reserved.

Linker Stitches Files Together

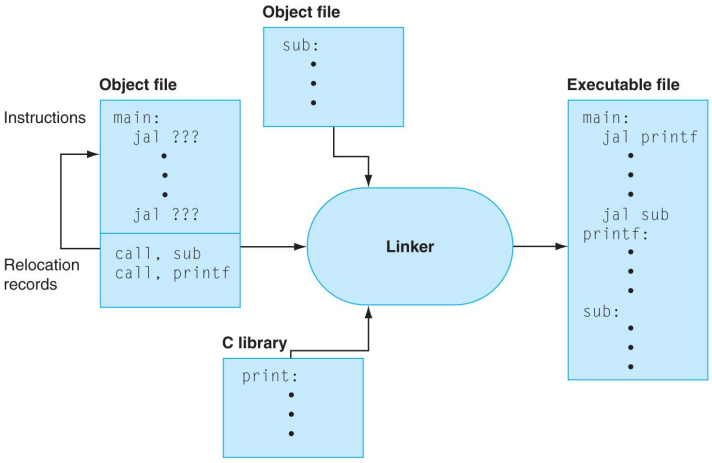


FIGURE B.3.1 The linker searches a collection of object fi les and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files. Copyright © 2009 Elsevier, Inc. All rights reserved.

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

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000011110000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

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.

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

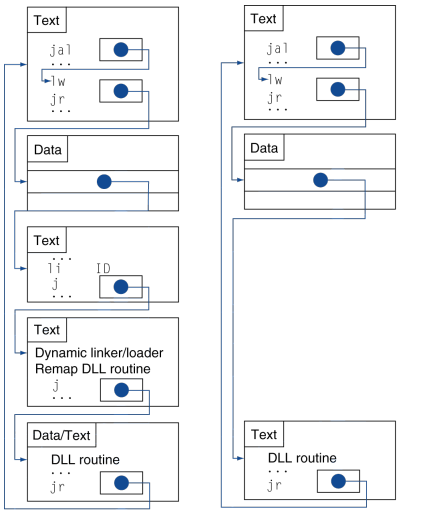
Dynamic Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine

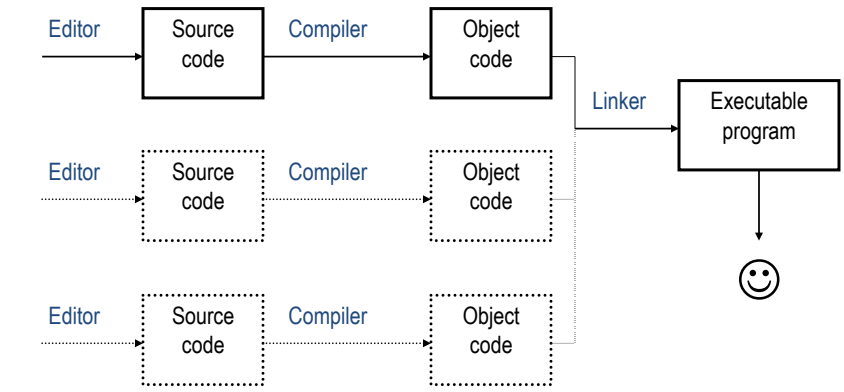
Agenda

- 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.

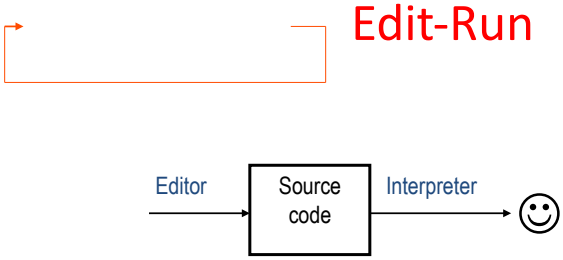
Compiled Languages: Edit-Compile-Link-Run



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



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

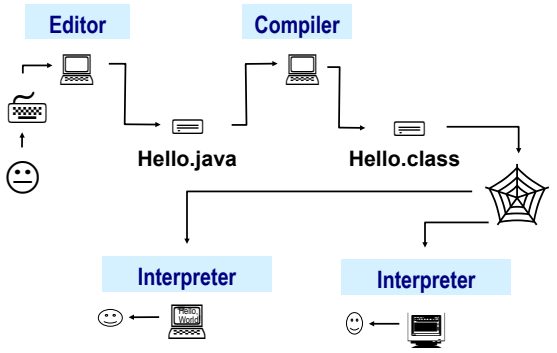
Interpreter disadvantages

- 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



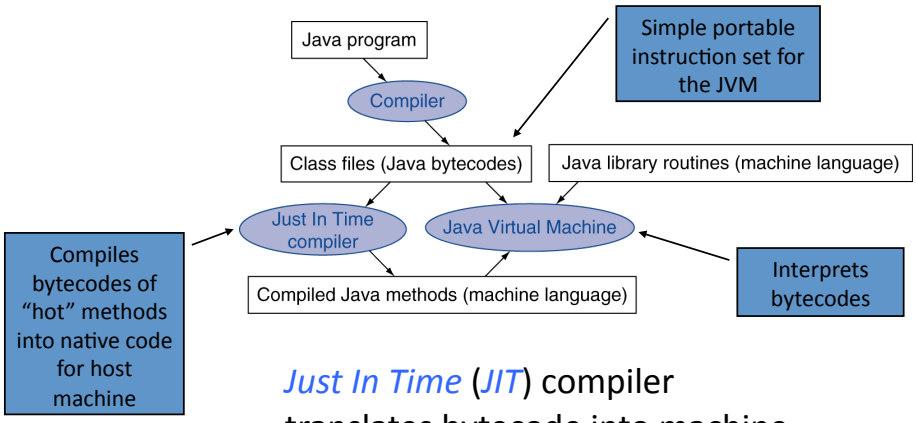
Why Bytecodes?

- 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.)

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

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