Lecture 09: Program Translation
Corrections to Green Sheet

• “Core Instruction Set”)

  1) Opcode wrong for Load Word. It should say 23hex, not 0 / 23hex.

  2) sll and srl should shift values in R[rt], not R[rs]
     i.e. sll/srl: R[rd] = R[rt] <<< shamt
Overview

• Interpretation vs Translation
• Translating C Programs
  • Compiler
  • Assembler
  • Linker
  • Loader
### Language Continuum

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Java bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>Assembly</td>
</tr>
<tr>
<td>C</td>
<td>machine language</td>
</tr>
</tbody>
</table>

- Easy to program
- Inefficient to interpret
- Efficient
- Difficult to program

• In general, we interpret a high level language if efficiency is not critical or translated to a lower level language to improve performance
Interpretation vs Translation

• How do we run a program written in a source language?

• Interpreter: Directly executes a program in the source language

• Translator: Converts a program from the source language to an equivalent program in another language

• For example, consider a Scheme program foo.scm
Interpretation

Scheme program: foo.scm

Scheme Interpreter
Translation

Scheme program: foo.scm

Scheme Compiler

Executable (mach lang pgm): a.out

Hardware

° Scheme Compiler is a translator from Scheme to machine language.
Interpretation

• Any good reason to interpret machine language in software?

• SPIM – useful for learning / debugging

• Apple Macintosh conversion
  • Switched from Motorola 680x0 instruction architecture to PowerPC.
  • Could require all programs to be re-translated from high level language
  • Instead, let executables contain old and/or new machine code, interpret old code in software if necessary
Interpretation vs. Translation?

• Easier to write interpreter

• Interpreter closer to high-level, so gives better error messages (e.g., SPIM)
  • Translator reaction: add extra information to help debugging (line numbers, names)

• Interpreter slower (10x?) but code is smaller (1.5X to 2X?)

• Interpreter provides instruction set independence: run on any machine
  • Apple switched to PowerPC. Instead of retranslating all SW, let executables contain old and/or new machine code, interpret old code in software if necessary
Steps to Starting a Program

C program: foo.c

Assembly program: foo.s

Object (mach lang module): foo.o

Executable (mach lang pgm): a.out

Compiler

Assembler

Linker

Loader

Memory
Compiler

- **Input**: High-Level Language Code (e.g., C, Java such as `foo.c`)
- **Output**: Assembly Language Code (e.g., `foo.s` for MIPS)
- **Note**: Output *may* contain pseudoinstructions

- **Pseudoinstructions**: instructions that assembler understands but not in machine (last lecture) For example:
  - `mov $s1,$s2` ⇒ or `$s1,$s2,$zero`
Where Are We Now?

C program: foo.c
Compiler
Assembly program: foo.s
Assembler
Object (mach lang module): foo.o
Linker
Executable (mach lang pgm): a.out
Loader
Memory
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 Pseudoinstructions

• Produce Machine Language

• Creates Object File
Assembler Directives (p. A-51 to A-53)

• 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
Pseudoinstruction Replacement

- Asm. treats convenient variations of machine language instructions as if real instructions

Pseudo: \[
\begin{align*}
\text{subu } & \$sp, \$sp, 32 \\
\text{sd } & \$a0, 32(\$sp) \\
\text{mul } & \$t7, \$t6, \$t5 \\
\text{addu } & \$t0, \$t6, 1 \\
\text{ble } & \$t0, 100, \text{loop} \\
\text{la } & \$a0, \text{str}
\end{align*}
\]
Real: \[
\begin{align*}
\text{addiu } & \$sp, \$sp, -32 \\
\text{sw } & \$a0, 32(\$sp) \\
\text{sw } & \$a1, 36(\$sp) \\
\text{mul } & \$t6, \$t5 \\
\text{addiu } & \$t0, \$t6, 1 \\
\text{slti } & \$at, \$t0, 101 \\
\text{bne } & \$at, \$0, \text{loop} \\
\text{lui } & \$at, \text{left(str)} \\
\text{ori } & \$a0, \$at, \text{right(str)}
\end{align*}
\]
Producing Machine Language (1/2)

• Simple Case
  • Arithmetic, Logical, Shifts, and so on.
  • All necessary info is within the instruction already.

• What about Branches?
  • PC-Relative
  • So once pseudoinstructions are replaced by real ones, we know by how many instructions to branch.

• So these can be handled easily.
What about jumps (j and jal)?
- Jumps require **absolute address**.

What about references to data?
- `l`a gets broken up into `l`ui and `ori`
- These will require the full 32-bit address of the data.

These can’t be determined yet, so we create two tables...
Symbol Table

• List of “items” in this file that may be used by other files.

• What are they?
  • Labels: function calling
  • Data: anything in the .data section; variables which may be accessed across files

• First Pass: record label-address pairs

• Second Pass: produce machine code
  • Result: can jump to a later label without first declaring it
Relocation Table

• List of “items” for which this file needs the address.

• What are they?
  • Any label jumped to: j or jal
    - internal
    - external (including lib files)
  • Any piece of data
    - such as the la instruction
Object File Format

- **object file header**: size and position of the other pieces of the object file
- **text segment**: the machine code
- **data segment**: binary representation of the data in the source file
- **relocation information**: identifies lines of code that need to be “handled”
- **symbol table**: list of this file’s labels and data that can be referenced
- **debugging information**
Quickie Quiz

1. Assembler *knows where* a module’s data & instructions are in relation to other modules.

2. Assembler will *ignore the instruction* `Loop: nop` because it does nothing.

3. Java designers used an interpreter (rather than a translater) *mainly* because of (at least one of): ease of writing, better error msgs, smaller object code.

```plaintext
1: FFF
2: FFT
3: FTF
4: FTT
5: TFF
6: TFT
7: TTF
8: TTT
```
Quickie Quiz Answer

1. Assembler only sees one compiled program at a time, that’s why it has to make a symbol & relocation table. It’s the job of the linker to link them all together…F!

2. Assembler keeps track of all labels in symbol table…F!

3. Java designers used an interpreter mainly because of code portability…F!

1. Assembler knows where a module’s data & instructions are in relation to other modules.

2. Assembler will ignore the instruction Loop : nop because it does nothing.

3. Java designers used an interpreter (rather than a translator) mainly because of (at least one of): ease of writing, better error msgs, smaller object code.
Where Are We Now?

C program: foo.c

Assembly program: foo.s

Object (mach lang module): foo.o

Linker

Executable (mach lang pgm): a.out

Loader

Memory
Link Editor/Linker (1/3)

• Input: Object Code, information tables (e.g., foo.o for MIPS)

• Output: Executable Code (e.g., a.out for MIPS)

• Combines several object (.o) files into a single executable (“linking”)

• Enable Separate Compilation of files
  • Changes to one file do not require recompilation of whole program
    - Windows NT source is >40 M lines of code!
  • Link Editor name from editing the “links” in jump and link instructions
Link Editor/Linker (2/3)

- `.o` file 1
  - text 1
  - data 1
  - info 1

- `.o` file 2
  - text 2
  - data 2
  - info 2

Linked by the Linker to:
- `a.out`
  - Relocated text 1
  - Relocated text 2
  - Relocated data 1
  - Relocated data 2
Link Editor/Linker (3/3)

• Step 1: Take text segment from each .o file and put them together.

• Step 2: Take data segment from each .o file, put them together, and concatenate this onto end of text segments.

• Step 3: Resolve References
  • Go through Relocation Table and handle each entry
  • That is, fill in all absolute addresses
Four Types of Addresses

• PC-Relative Addressing (beq, bne): never relocate

• Absolute Address (j, jal): always relocate

• External Reference (usually jal): always relocate

• Data Reference (often lui and ori): always relocate
Absolute Addresses in MIPS

• Which instructions need relocation editing?

• J-format: jump, jump and link

  \[ j/jal \quad xxxxx \]

• Loads and stores to variables in static area, relative to global pointer

  \[ lw/sw \quad $gp \quad $x \quad address \]

• What about conditional branches?

  \[ beq/bne \quad $rs \quad $rt \quad address \]

• PC-relative addressing preserved even if code moves
Resolving References (1/2)

• Linker \emph{assumes} first word of first text segment is at address 0x00000000.

• Linker knows:
  • length of each text and data segment
  • ordering of text and data segments

• Linker calculates:
  • absolute address of each label to be jumped to (internal or external) and each piece of data being referenced
Resolving References (2/2)

• To resolve references:
  • search for reference (data or label) in all symbol tables
  • if not found, search library files (for example, for `printf`)
  • once absolute address is determined, fill in the machine code appropriately

• Output of linker: executable file containing text and data (plus header)
Static vs Dynamically linked libraries

• What we’ve described is the traditional way to create a static-linked approach
  • The library is now part of the executable, so if the library updates we don’t get the fix (have to recompile if we have source)
  • In includes the entire library even if not all of it will be used.
• An alternative is dynamically linked libraries (DLL), common on Windows & UNIX platforms
  • 1st run overhead for dynamic linker-loader
  • Having executable isn’t enough anymore!
Where Are We Now?

C program: foo.c
Compiler
Assembly program: foo.s
Assembler
Object(mach lang module): foo.o
Linker
Executable(mach lang pgm): a.out
Lib.o
Loader
Memory
Loader (1/3)

• Input: Executable Code (e.g., a.out for MIPS)
• Output: (program is run)
• Executable files are stored on disk.
• When one is run, loader’s job is to load it into memory and start it running.
• In reality, loader is the operating system (OS)
  • loading is one of the OS tasks
Loader (2/3)

• So what does a loader do?

• Reads executable file’s header to determine size of text and data segments

• Creates new address space for program large enough to hold text and data segments, along with a stack segment

• Copies instructions and data from executable file into the new address space (this may be anywhere in memory)
Loader (3/3)

• Copies arguments passed to the program onto the stack

• Initializes machine registers
  • Most registers cleared, but stack pointer assigned address of 1st free stack location

• Jumps to start-up routine that copies program’s arguments from stack to registers and sets the PC
  • If main routine returns, start-up routine terminates program with the exit system call
Example: \texttt{C} \Rightarrow \texttt{Asm} \Rightarrow \texttt{Obj} \Rightarrow \texttt{Exe} \Rightarrow \texttt{Run}

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i, sum = 0;
    for (i = 0; i <= 100; i++)
        sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d
", sum);
}
```
Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```
.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)
.addiu $sp,$sp,32
.jr $ra

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

Where are 7 pseudo-instructions?
Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```assembly
.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)
```

```assembly
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)
addiu $sp,$sp,32
jr $ra
```

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

7 pseudo-instructions underlined

Compiling, Assembling, Loading, Linking (CALL) I (38)
Symbol Table Entries

• Symbol Table
  Label  Address
  main:   ?
  loop:   ?
  str:    
  printf:

• Relocation Table
  Address  Instr. Type  Dependency
**Example**: C $\Rightarrow$ Asm $\Rightarrow$ **Obj** $\Rightarrow$ Exe $\Rightarrow$ Run

- Remove pseudoinstructions, assign addresses

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td><code>addiu $29, $29, -32</code></td>
<td>30</td>
<td><code>addiu $8, $14, 1</code></td>
<td></td>
</tr>
<tr>
<td>04</td>
<td><code>sw $31, 20($29)</code></td>
<td>34</td>
<td><code>sw $8, 28($29)</code></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td><code>sw $4, 32($29)</code></td>
<td>38</td>
<td><code>slti $1, $8, 101</code></td>
<td></td>
</tr>
<tr>
<td>0C</td>
<td><code>sw $5, 36($29)</code></td>
<td>3C</td>
<td><code>bne $1, $0, loop</code></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td><code>sw $0, 24($29)</code></td>
<td>40</td>
<td><code>lui $4, l.str</code></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td><code>sw $0, 28($29)</code></td>
<td>44</td>
<td><code>ori $4, $4, r.str</code></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td><code>lw $14, 28($29)</code></td>
<td>48</td>
<td><code>lw $5, 24($29)</code></td>
<td></td>
</tr>
<tr>
<td>1C</td>
<td><code>multu $14, $14</code></td>
<td>4C</td>
<td><code>jal printf</code></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td><code>mflo $15</code></td>
<td>50</td>
<td><code>add $2, $0, $0</code></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td><code>lw $24, 24($29)</code></td>
<td>54</td>
<td><code>lw $31, 20($29)</code></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td><code>addu $25, $24, $15</code></td>
<td>58</td>
<td><code>addiu $29, $29, 32</code></td>
<td></td>
</tr>
<tr>
<td>2C</td>
<td><code>sw $25, 24($29)</code></td>
<td>5C</td>
<td><code>jr $31</code></td>
<td></td>
</tr>
</tbody>
</table>
Symbol Table Entries

• Symbol Table
  • Label Address
    main: 0x00000000
    loop: 0x00000018
    str: 0x10000430
    printf: 0x000003b0

• Relocation Information
  • Address Instr. Type Dependency
    0x00000040 lui l.str
    0x00000044 ori r.str
    0x0000004c jal printf
Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

• Edit Addresses: start at 0x0040000

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>addiu $29,$29,-32</td>
<td>30</td>
<td>addiu $8,$14, 1</td>
</tr>
<tr>
<td>04</td>
<td>sw $31,20($29)</td>
<td>04</td>
<td>sw $8,28($29)</td>
</tr>
<tr>
<td>08</td>
<td>sw $4, 32($29)</td>
<td>08</td>
<td>sw $8,28($29)</td>
</tr>
<tr>
<td>0c</td>
<td>sw $5, 36($29)</td>
<td>0c</td>
<td>slti $1,$8, 101</td>
</tr>
<tr>
<td>10</td>
<td>sw $0, 24($29)</td>
<td>10</td>
<td>bne $1,$0, -10</td>
</tr>
<tr>
<td>14</td>
<td>sw $0, 28($29)</td>
<td>14</td>
<td>lui $4, 4096</td>
</tr>
<tr>
<td>18</td>
<td>lw $14, 28($29)</td>
<td>18</td>
<td>ori $4,$4, 1072</td>
</tr>
<tr>
<td>1c</td>
<td>multu $14, $14</td>
<td>1c</td>
<td>jal 812</td>
</tr>
<tr>
<td>20</td>
<td>mflo $15</td>
<td>20</td>
<td>add $2, $0, $0</td>
</tr>
<tr>
<td>24</td>
<td>lw $24, 24($29)</td>
<td>24</td>
<td>lw $5,24($29)</td>
</tr>
<tr>
<td>28</td>
<td>addu $25,$24,$15</td>
<td>28</td>
<td>lw $31,20($29)</td>
</tr>
<tr>
<td>2c</td>
<td>sw $25, 24($29)</td>
<td>2c</td>
<td>addiu $29,$29,32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5c</td>
<td>jr $31</td>
</tr>
</tbody>
</table>
Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```
004000 010011111011111011111111111111
004004 111011111101111111101111111111
004008 01001111110201001000000000010000
00400c 11101111111010010100000000010010
004010 11110000000000000000000000111000
004014 11110000000000000000000000111100
004018 11110000000000000000000000111111
00401c 11110000000000000000000000111111
004020 10010111100101000000000001100100
004024 10010111100100000000000001100100
004028 10010111100100000000000001100100
00402c 10010111100100000000000001100100
004030 10010111110100000000000001100100
004034 10010111110100000000000001100100
004038 10010111110100000000000001100100
00403c 10010111110100000000000001100100
004040 10010111110100000000000001100100
004044 10010111110100000000000001100100
004048 10010111110100000000000001100100
00404c 10010111110100000000000001100100
004050 10010111110100000000000001100100
004054 10010111110100000000000001100100
004058 10010111110100000000000001100100
00405c 10010111110100000000000001100100
```

Compiling, Assembling, Loading, Linking (CALL) I (43) Fall 2005
Quickie Quiz

Which of the following instr. may need to be edited during link phase?

Loop: 
```assembly
lui $at, 0xABCD } # A
ori $a0,$at, 0xFEDC
jal add_link # B
bne $a0,$v0, Loop # C
```

1: FFF
2: FFT
3: FTF
4: FTT
5: TFF
6: TFF
7: TTF
8: TTT
Quickie Quiz Answer

Which of the following instr. may need to be edited during link phase?

Loop: lui $at, 0xABCD         } # A
    ori $a0,$at, 0xFEDC        subroutine; relocate
    jal add_link               # B
    bne $a0,$v0, Loop          PC-relative branch; OK

    data reference; relocate
    subroutine; relocate
Things to Remember (1/3)

C program: foo.c
Compiler

Assembly program: foo.s
Assembler

Object (mach lang module): foo.o
Linker

Executable (mach lang pgm): a.out
Loader

Memory

lib.o
Things to Remember (2/3)

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudoinstructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). This changes each .s file into a .o file.
- Linker combines several .o files and resolves absolute addresses.
- Loader loads executable into memory and begins execution.
Things to Remember 3/3

• Stored Program concept mean instructions just like data, so can take data from storage, and keep transforming it until load registers and jump to routine to begin execution
  • Compiler $\Rightarrow$ Assembler $\Rightarrow$ Linker ($\Rightarrow$ Loader )

• Assembler does 2 passes to resolve addresses, handling internal forward references

• Linker enables separate compilation, libraries that need not be compiled, and resolves remaining addresses