CS 61C: Great Ideas in Computer Architecture

Running a Program - CALL
(Compiling, Assembling, Linking, and Loading)

Instructors:

Bernard Boser & Randy H. Katz

http://inst.eecs.Berkeley.edu/~cs61c/fa16
Boser and Katz’s Problem with Reef Polling ...
Outline

• Review Instruction Formats
• Multiply and Divide
• Interpretation vs. Translation
• Assembler
• Linker
• Loader
• And in Conclusion ...
Outline

• Review Instruction Formats
  • Multiply and Divide
  • Interpretation vs. Translation
• Assembler
• Linker
• Loader
• And in Conclusion ...
Review

- **I-Format**: instructions with immediates, `lw/sw` (*offset is immediate*), and `beq/bne`
  - But not the shift instructions
  - Branches use PC-relative addressing

```
I: opcode | rs | rt | immediate
```

- **J-Format**: `j` and `jal` (*but not jr*)
  - Jumps use absolute addressing

```
J: opcode | target address
```

- **R-Format**: all other instructions

```
R: opcode | rs | rt | rd | shamt | funct
```
Review: Pseudo Instructions (Green Card)

<table>
<thead>
<tr>
<th>PSEUDOINSTRUCTION SET</th>
<th>NAME</th>
<th>MNEMONIC</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Branch Less Than</td>
<td>blt</td>
<td>if(R[rs]&lt;R[rt]) PC = Label</td>
</tr>
<tr>
<td></td>
<td>Branch Greater Than</td>
<td>bgt</td>
<td>if(R[rs]&gt;R[rt]) PC = Label</td>
</tr>
<tr>
<td></td>
<td>Branch Less Than or Equal</td>
<td>ble</td>
<td>if(R[rs]&lt;=R[rt]) PC = Label</td>
</tr>
<tr>
<td></td>
<td>Branch Greater Than or Equal</td>
<td>bge</td>
<td>if(R[rs]&gt;=R[rt]) PC = Label</td>
</tr>
<tr>
<td></td>
<td>Load Immediate</td>
<td>li</td>
<td>R[rd] = immediate</td>
</tr>
<tr>
<td></td>
<td>Move</td>
<td>move</td>
<td>R[rd] = R[rs]</td>
</tr>
</tbody>
</table>

Valid in assembly language but not in machine language (i.e., maps to multiple machine language instructions)

True Assembly Language (TAL) vs. MIPS Assembly Language (MAL)
Clicker Question

Which of the following place the address of LOOP in $v0?

1) la $t1, LOOP
   lw $v0, 0($t1)

2) jal LOOP
   LOOP: addu $v0, $ra, $zero

3) la $v0, LOOP

1  2  3
A) T, T, T
B) T, T, F
C) F, T, T
D) F, T, F
E) F, F, T
### MIPS Reference Data

#### Core Instruction Set

<table>
<thead>
<tr>
<th>NAME, MNEMONIC</th>
<th>FORMAT</th>
<th>OPERATION (in Verilog)</th>
<th>OPCODE / FUNCT (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump And Link</td>
<td>jal</td>
<td>( R[31] = PC + 8 ); ( PC = \text{JumpAddr} )</td>
<td>( 5 ) ( 3_{\text{hex}} )</td>
</tr>
</tbody>
</table>

#### Basic Instruction Formats

<table>
<thead>
<tr>
<th>R</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21 20 16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J</th>
<th>opcode</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
<td>26 25</td>
</tr>
</tbody>
</table>

\[(5) \text{JumpAddr} = \{ \text{PC} + 4[31:28], \text{address}, 2\text{'b}0 \}\]
Outline

• Review Instruction Formats
• **Multiply and Divide**
• Interpretation vs. Translation
• Assembler
• Linker
• Loader
• And in Conclusion ...
Integer Multiplication (1/3)

• Paper and pencil example (unsigned):

  Multiplicand  1000   8
  Multiplier  x1001   9

  
  
  
  
  
  0000

  
  
  
  
  
  0000

  
  
  
  
  
  +1000

  
  
  
  
  
  
  01001000

  
  
  
  
  
  \[ \text{m bits x n bits} = m + n \text{ bit product} \]
Integer Multiplication (2/3)

• In MIPS, we multiply registers, so:
  – 32-bit value x 32-bit value = 64-bit value

• Syntax of Multiplication (signed):
  – mult register1, register2
  – Multiplies 32-bit values in those registers & puts 64-bit product in special result registers:
    • Puts product upper half in hi, lower half in lo
  – hi and lo are 2 registers separate from the 32 general purpose registers
  – Use mfhi register & mflo register to move from hi, lo to another register
Integer Multiplication (3/3)

• Example:
  – in C: \( a = b \times c; \)
  – in MIPS:
    • Let \( b \) be \( s2 \); let \( c \) be \( s3 \); and let \( a \) be \( s0 \) and \( s1 \) (since it may be up to 64 bits)

```assembly
mult $s2,$s3  # b*c
mfhi $s0       # upper half of
               # product into $s0
mflo $s1       # lower half of
               # product into $s1
```

• Note: Often, we only care about the lower half of the product
  – Pseudo-inst. \texttt{mul} expands to \texttt{mult/mflo}
Integer Division (1/2)

• Paper and pencil example (unsigned):

\[
\begin{array}{c|c}
\text{Divisor} & 1000 \\
\hline
\text{Dividend} & 1001010 \\
\hline
-1000 & \\
\hline
10 & \\
101 & \\
1010 & \\
-1000 & \\
\hline
10 & \text{Remainder} \\
\end{array}
\]

Quotient

(\text{or Modulo result})

• Dividend = Quotient \times \text{Divisor} + \text{Remainder}
Integer Division (2/2)

• Syntax of Division (signed):
  – div register1, register2
  – Divides 32-bit register 1 by 32-bit register 2:
  – puts remainder of division in hi, quotient in lo

• Implements C division (/) and modulo (%)

• Example in C: 
  \[ a = c / d; \quad b = c \% d; \]

• in MIPS:
  \[ a\leftarrow s0; \quad b\leftarrow s1; \quad c\leftarrow s2; \quad d\leftarrow s3 \]

  \[ \text{div } s2, s3 \quad \# \text{ lo} = c/d, \text{ hi} = c\%d \]

  \[ \text{mflo } s0 \quad \# \text{ get quotient} \]

  \[ \text{mfhi } s1 \quad \# \text{ get remainder} \]
Outline

• Review Instruction Formats
• Multiply and Divide
• Interpretation vs. Translation
• Assembler
• Linker
• Loader
• And in Conclusion ...
Levels of Representation/Interpretation

- High Level Language Program (e.g., C)
  - Compiler
  - Assembly Language Program (e.g., MIPS)
  - Assembler
  - Machine Language Program (MIPS)

+ How to take a program and run it

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

Anything can be represented as a number, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```
Language Execution Continuum

- **Interpreter** is a program that executes other programs

  - Python
  - Java
  - C++
  - C
  - Assembly
  - Machine code

  - Java bytecode

- Language translation gives us another option

- In general, we interpret a high-level language when efficiency is not critical and translate to a lower-level language to increase 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
Interpretation

- For example, consider a Python program `foo.py`

- Python interpreter is just a program that reads a python program and performs the functions of that python program
Interpretation

- WHY interpret machine language in software?
- MARS (Lab #3): MIPS simulator useful for learning / debugging
- E.g., Apple Macintosh conversion
  - Switched from Motorola 680x0 instruction architecture to PowerPC (before x86)
  - 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 (emulation)
Interpretation vs. Translation? (1/2)

• Generally easier to write interpreter
• Interpreter closer to high-level, so can give better error messages (e.g., MARS)
  – Translator reaction: add extra information to help debugging (line numbers, names)
• Interpreter slower (10x?), code smaller (2x?)
• Interpreter provides instruction set independence: run on any machine
Interpretation vs. Translation? (2/2)

• Translated/compiled code almost always more efficient and therefore higher performance:
  – Important for many applications, particularly operating systems

• Translation/compilation helps “hide” the program “source” from the users:
  – One model for creating value in the marketplace (e.g., Microsoft keeps all their source code secret)
  – Alternative model, “open source”, creates value by publishing the source code and fostering a community of developers.
Steps in Compiling a C Program

```
 gcc -O2 -S -c foo.c
```
Compiler

• Input: High-Level Language Code (e.g., $foo.c$)
• Output: Assembly Language Code (e.g., $foo.s$ for MIPS)
• Note: Output $may$ contain pseudo-instructions
• Pseudo-instructions: instructions that assembler understands but not in machine

For example:

- $move \; s1,s2 \Rightarrow add \; s1,s2,\$zero$
Outline

• Review Instruction Formats
• Multiply and Divide
• Interpretation vs. Translation

• Assembler
• Linker
• Loader
• And in Conclusion ...
Where Are We Now?

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

CS164
Assembler

• Input: Assembly Language Code (MAL) (e.g., foo.s for MIPS)
• Output: Object Code, information tables (TAL) (e.g., foo.o for MIPS)
• Reads and Uses Directives
• Replace Pseudo-instructions
• Produce Machine Language
• Creates Object File
Assembler Directives
(See Appendix A-1 to A-4)

• Give directions to assembler, but do not produce machine instructions

  .text: Subsequent items put in user text segment (machine code)

  .data: Subsequent items put in user data segment (binary rep of data in source file)

  .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
Pseudo-instruction Replacement

- Assembler treats convenient variations of machine language instructions as if real instructions

**Pseudo:**

- `subu $sp,$sp,32`
- `sd $a0, 32($sp)`
- `mul $t7,$t6,$t5`
- `addu $t0,$t6,1`
- `ble $t0,100,loop`
- `la $a0, str`

**Real:**

- `addiu $sp,$sp,-32`
- `sw $a0, 32($sp)`
- `sw $a1, 36($sp)`
- `mult $t6,$t5`
- `mflo $t7`
- `addiu $t0,$t6,1`
- `slti $at,$t0,101`
- `bne $at,$0,loop`
- `lui $at,left(str)`
- `ori $a0,$at,right(str)`
Clicker/Peer Instruction

Which of the following is a correct TAL instruction sequence for `la $v0, FOO`?*

%hi(label), tells assembler to fill upper 16 bits of label’s addr
%lo(label), tells assembler to fill lower 16 bits of label’s addr

A: ori $v0, %hi(FOO)
   addiu $v0, %lo(FOO)

B: ori $v0, %lo(FOO)
   lui $v0, %hi(FOO)

C: lui $v0, %lo(FOO)
   ori $v0, %hi(FOO)

D: lui $v0, %hi(FOO)
   addiu $v0, %lo(FOO)

E: la $v0, FOO is already a TAL instruction

*Assume the address of FOO is 0xABCD0124
Administrivia

• 2nd C Guerrilla Help Session: Wednesday, September 21, 7:30-9:30 PM, in 293 Cory and 405 Soda
• Project #1 due THURSDAY September 22 @ 23:59:59
• Midterm #1 in 1 week: September 27!
  – IN CLASS! Pauley Ballroom, 3:40-5 PM
  – Covers Number Representations, C (including Project #1), MIPS Assembly and Machine Language, Compiler/Assembly/Linking>Loading (thru next Tuesday’s lecture)
  – Two sided 8.5” x 11” cheat sheet + MIPS Green Card that we give you
  – Review session preceding Sunday (September 25) 1-3 PM Dwinelle 155
  – DSP students: please make sure we know about your special accommodations (contact Derek and Stephan the co-Head TAs)
Producing Machine Language (1/3)

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

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

• So these can be handled
Producing Machine Language (2/3)

• “Forward Reference” problem
  – Branch instructions can refer to labels that are “forward” in the program:
    
    ```
    or    $v0, $0, $0
    L1:  slt  $t0, $0, $a1
    beq  $t0, $0,   L2
    addi  $a1, $a1, -1
    j     L1
    L2:  add  $t1, $a0, $a1
    
    – Solved by taking two passes over the program
      • First pass remembers position of labels
      • Second pass uses label positions to generate code
    ```
Producing Machine Language (3/3)

• What about jumps (\texttt{j} and \texttt{jal})?
  – Jumps require \texttt{absolute address}
  – So, forward or not, still can’t generate machine instruction without knowing the position of instructions in memory

• What about references to static data?
  – \texttt{la} gets broken up into \texttt{lui} and \texttt{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
Relocation Table

• List of “items” whose address this file needs
What are they?

  – Any label jumped to: j or jal
    • Internal
    • External (including lib files)

  – Any piece of data in static section
    • Such as the 1a 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 static data in the source file
- **relocation information**: identifies lines of code that need to be fixed up later
- **symbol table**: list of this file’s labels and static data that can be referenced
- **debugging information**
- A standard format is ELF (except MS)

http://www.skyfree.org/linux/references/ELF_Format.pdf
Outline

• Review Instruction Formats
• Multiply and Divide
• Interpretation vs. Translation
• Assembler
• Linker
• Loader
• And in Conclusion ...
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

lib.o
Linker (1/3)

• Input: Object code files, information tables (e.g., foo.o, libc.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 the whole program
    • Windows NT source was > 40 M lines of code!
  – Old name "Link Editor" from editing the "links" in jump and link instructions
Linker (2/3)

The Linker combines the .o files from the compilation stage into a single executable file, a.out. The Linker relocates the text and data sections from the .o files into the final executable. The figure shows the process:

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

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

3. a.out
   - Relocated text 1
   - Relocated text 2
   - Relocated data 1
   - Relocated data 2

The Linker processes each .o file, relocating the necessary sections into the final executable, a.out.
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; handle each entry
  – I.e., fill in all absolute addresses
Four Types of Addresses

• PC-Relative Addressing (beq, bne)
  – Never need to relocate
• Absolute Function Address (j, jal)
  – Always relocate
• External Function Reference (usually jal)
  – Always relocate
• Static 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**
    
    - Loads and stores to variables in static area, relative to global pointer
      
      **lw/sw**
      
      - What about conditional branches?
        
        **beq/bne**
        
        - PC-relative addressing *preserved* even if code moves
Resolving References (1/2)

• Linker assumes first word of first text segment is at address \texttt{0x04000000}.
  – (More later when we study “virtual memory”)

• 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 “user” symbol tables
  – If not found, search library files (e.g., for `printf`)
  – Once absolute address is determined, fill in the machine code appropriately

• Output of linker: executable file containing text and data (plus header)
Outline

• Review Instruction Formats
• Multiply and Divide
• Interpretation vs. Translation
• Assembler
• Linker
• Loader
• And in Conclusion ...
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

lib.o
Loader Basics

• 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 ... What Does It 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
- 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 & sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call
Example: $\mathbf{C} \rightarrow \mathbf{Asm} \rightarrow \mathbf{Obj} \rightarrow \mathbf{Exe} \rightarrow \mathbf{Run}$

**C Program Source Code: prog.c**

```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 of sq from 0 .. 100 is %d\n", sum);
}
```

“printf” lives in “libc”
Compilation: MAL

```
.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)
add $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 of sq from 0 .. 100 is %d\n"
```

Where are 7 pseudo-instructions?
.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 of sq from 0 .. 100 is %d\n"
Assembly Step 1:
Remove pseudoinstructions, assign addresses

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

Create relocation table and symbol table

• Symbol Table

<table>
<thead>
<tr>
<th>Label</th>
<th>address (in module)</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:</td>
<td>0x0000000000</td>
<td>global text</td>
</tr>
<tr>
<td>loop:</td>
<td>0x000000018</td>
<td>local text</td>
</tr>
<tr>
<td>str:</td>
<td>0x0000000000</td>
<td>local data</td>
</tr>
</tbody>
</table>

• Relocation Information

<table>
<thead>
<tr>
<th>Address</th>
<th>Instr. type</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000040</td>
<td>lui</td>
<td>l.str</td>
</tr>
<tr>
<td>0x000000044</td>
<td>ori</td>
<td>r.str</td>
</tr>
<tr>
<td>0x00000004c</td>
<td>jal</td>
<td>printf</td>
</tr>
</tbody>
</table>
Assembly Step 3

Resolve local PC-relative labels

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

• Generate object (.o) file:
  – Output binary representation for
    • text segment (instructions)
    • data segment (data)
    • symbol and relocation tables
  – Using dummy “placeholders” for unresolved absolute and external references
Link step 1: combine prog.o, libc.o

- Merge text/data segments
- Create absolute memory addresses
- Modify & merge symbol and relocation tables

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 ...
Link Step 2:

- Edit Addresses in relocation table
  - (shown in TAL for clarity, but done in binary)

```
00 addiu $29,$29,-32
04 sw$31,20($29)
08 sw$4, 32($29)
0c sw$5, 36($29)
10 sw $0, 24($29)
14 sw $0, 28($29)
18 lw $14, 28($29)
1c multu $14, $14
20 mflo $15
24 lw $24, 24($29)
28 addu $25,$24,$15
2c sw $25, 24($29)
30 addiu $8,$14, 1
34 sw$8,28($29)
38 slti $1,$8, 101
3c bne $1,$0, -10
40 lui $4, 4096
44 ori $4,$4, 1072
48 lw$5,24($29)
4c jal 944
50 add $2, $0, $0
54 lw $31,20($29)
58 addiu $29,$29,32
5c jr$31
```
Link Step 3:

• Output executable of merged modules
  – Single text (instruction) segment
  – Single data segment
  – Header detailing size of each segment

• **NOTE:**
  – Preceeding example was a much simplified version of how ELF and other standard formats work, meant only to demonstrate the basic principles
Static vs. Dynamically Linked Libraries

• What we’ve described is the traditional way: **statically-linked** approach
  – 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)
  – Includes the **entire** library even if not all of it will be used
  – Executable is self-contained

• Alternative is **dynamically linked libraries** (DLL), common on Windows & UNIX platforms
Dynamically Linked Libraries

- Space/time issues
  + Storing a program requires less disk space
  + Sending a program requires less time
  + Executing two programs requires less memory (if they share a library)
    - At runtime, there’s time overhead to do link

- Upgrades
  + Replacing one file (`libXYZ.so`) upgrades every program that uses library “XYZ”
    - Having the executable isn’t enough anymore

*Overall, dynamic linking adds quite a bit of complexity to the compiler, linker, and operating system. However, it provides many benefits that often outweigh these.*
Dynamically Linked Libraries

• Prevailing approach to dynamic linking uses machine code as the “lowest common denominator”
  – Linker does not use information about how the program or library was compiled (i.e., what compiler or language)
  – Can be described as “linking at the machine code level”
  – This isn’t the only way to do it ...
Outline

• Review Instruction Formats
• Multiply and Divide
• Interpretation vs. Translation
• Assembler
• Linker
• Loader

• And in Conclusion ...
And In Conclusion...

- Compiler converts a single HLL file into a single assembly language file
- Assembler removes pseudo-instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table): A `.s` file becomes a `.o` file
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several `.o` files and resolves absolute addresses
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution