CS61C : Machine Structures

Lecture #12 – MIPS Instruction Rep III, Running a Program I
aka Compiling, Assembling, Linking, Loading (CALL)

2007-7-16

Scott Beamer, Instructor

New Direction Service Announced

www.sfgate.com
Review of Floating Point

• Reserve exponents, significands:

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>Denorm</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- fl. pt. #</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- ∞</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

• Integer mult, div uses hi, lo regs
  • mfhi and mflo copies out.

• Four rounding modes (to even default)

• MIPS FL ops complicated, expensive
Clarification Unbiased Rounding

• Round to (nearest) even (default)
  • Normal rounding, almost: 2.5 ⇒ 2, 3.5 ⇒ 4
  • Insures fairness on calculation
  • Half the time we round up, other half down
  • Decimal gives a good initial intuition, but remember computers use binary

• Steps to Use it (in binary)
  • Determine place to be rounded to
  • Figure out the two possible outcomes (its binary so 1 or 0 in last place)
  • If one outcome is closer to current number than other, pick that outcome
  • If both outcomes are equidistant pick the outcome that ends in 0
Decoding Machine Language

• How do we convert 1s and 0s to C code?
  Machine language $\Rightarrow$ C?

• For each 32 bits:
  • Look at opcode: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format.
  • Use instruction type to determine which fields exist.
  • Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number.
  • Logically convert this MIPS code into valid C code. Always possible? Unique?
Decoding Example (1/7)

• Here are six machine language instructions in hexadecimal:

\[
\begin{align*}
00001025_{\text{hex}} \\
0005402A_{\text{hex}} \\
11000003_{\text{hex}} \\
00441020_{\text{hex}} \\
20A5FFFF_{\text{hex}} \\
08100001_{\text{hex}}
\end{align*}
\]

• Let the first instruction be at address 4,194,304_{\text{ten}} (0x00400000_{\text{hex}}).

• Next step: convert hex to binary
Decoding Example (2/7)

• The six machine language instructions in binary:

```
00000000000000000001000000100101
00000000000001010100000000101010
0001000100000000000000000000011
00000000010001000001000000100000
00100000101001011111111111111111
00001000000100000000000000000001
```

• Next step: identify opcode and format

<table>
<thead>
<tr>
<th>R</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0</td>
<td>1, 4-31</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
<tr>
<td>J</td>
<td>2 or 3</td>
<td>target address</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Decoding Example (3/7)

- Select the opcode (first 6 bits) to determine the format:

Format:

| R  | 000000000000000001000000100101 |
| R  | 0000000000000101010000000101010 |
| I  | 0010001000000000000000000000011 |
| R  | 00000000010001000001000000100000 |
| I  | 00100000101001011111111111111111 |
| J  | 00001000000010000000000000000001 |

- Look at opcode:
  0 means R-Format,
  2 or 3 mean J-Format,
  otherwise I-Format.

Next step: separation of fields
Decoding Example (4/7)

- Fields separated based on format(opcode):

<table>
<thead>
<tr>
<th>R</th>
<th>R</th>
<th>I</th>
<th>R</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0</td>
<td>+3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>5</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1,048,577</td>
</tr>
</tbody>
</table>

- Next step: translate (“disassemble”) to MIPS assembly instructions
Decoding Example (5/7)

- **MIPS Assembly (Part 1):**

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>or $2,$0,$0</td>
</tr>
<tr>
<td>0x00400004</td>
<td>slt $8,$0,$5</td>
</tr>
<tr>
<td>0x00400008</td>
<td>beq $8,$0,3</td>
</tr>
<tr>
<td>0x0040000c</td>
<td>add $2,$2,$4</td>
</tr>
<tr>
<td>0x00400010</td>
<td>addi $5,$5,-1</td>
</tr>
<tr>
<td>0x00400014</td>
<td>j 0x100001</td>
</tr>
</tbody>
</table>

- **Better solution:** translate to more meaningful MIPS instructions (fix the branch/jump and add labels, registers)
Decoding Example (6/7)

• MIPS Assembly (Part 2):

```
or    $v0,$0,$0
Loop:  slt     $t0,$0,$a1
        beq     $t0,$0,Exit
        add     $v0,$v0,$a0
        addi    $a1,$a1,-1
        j       Loop
Exit:
```

• Next step: translate to C code (be creative!)
Decoding Example (7/7)

Before Hex: • After C code (Mapping below)

00001025_{hex} • $v0: product
0005402A_{hex} • $a0: multiplicand
11000003_{hex} • $a1: multiplier
00441020_{hex} • product = 0;
20A5FFFF_{hex} • while (multiplier > 0) {
08100001_{hex} • product += multiplicand;

Loop:
or $v0,$0,$0
slt $t0,$0,$a1
beq $t0,$0,Exit
add $v0,$v0,$a0
addi $a1,$a1,-1
j Loop

Exit:

Demonstrated Big 61C Idea: Instructions are just numbers, code is treated like data
Administrivia...Midterm in 7 days!

• Project 2 due Friday @ 11:59pm
• Midterm 7/23 @ 7-10pm 60 Evans
• Bring…
  • NO backpacks, cells, calculators, pagers, PDAs
  • 2 writing implements (we’ll provide write-in exam booklets) – pencils ok!
  • One handwritten (both sides) 8.5”x11” paper
  • One green sheet (or copy of it)
• Review Session Friday @ …
Review from before: \texttt{lui}

• So how does \texttt{lui} help us?

  • Example:

    \begin{align*}
    \text{addi} & \quad \$t0, \$t0, 0xABABCDCD \\
    \text{becomes:} & \\
    \text{lui} & \quad \$at, 0xABAB \\
    \text{ori} & \quad \$at, \$at, 0xCDCD \\
    \text{add} & \quad \$t0, \$t0, \$at
    \end{align*}

• Now each I-format instruction has only a 16-bit immediate.

• \textbf{Wouldn’t it be nice if the assembler would this for us automatically?}

  - If number too big, then just automatically replace \texttt{addi} with \texttt{lui}, \texttt{ori}, \texttt{add}
True Assembly Language (1/3)

- **Pseudoinstruction**: A MIPS instruction that doesn’t turn directly into a machine language instruction, but into other MIPS instructions.

- What happens with pseudoinstructions?
  - They’re broken up by the assembler into several “real” MIPS instructions.
  - But what is a “real” MIPS instruction? Answer in a few slides.

- First some examples
Example Pseudoinstructions

• Register Move

\[
\text{move} \quad \text{reg2,reg1}
\]

Expands to:
\[
\text{add} \quad \text{reg2,}$\text{zero,reg1}$
\]

• Load Immediate

\[
\text{li} \quad \text{reg,value}
\]

If value fits in 16 bits:
\[
\text{addi} \quad \text{reg,}$\text{zero,}$\text{value}
\]

else:
\[
\text{lui} \quad \text{reg,upper 16 bits of value}
\]
\[
\text{ori} \quad \text{reg,}$\text{zero,lower 16 bits}$
\]
True Assembly Language (2/3)

• Problem:
  • When breaking up a pseudoinstruction, the assembler may need to use an extra reg.
  • If it uses any regular register, it’ll overwrite whatever the program has put into it.

• Solution:
  • Reserve a register ($\texttt{1}$, called $\texttt{at}$ for “assembler temporary”) that assembler will use to break up pseudo-instructions.
  • Since the assembler may use this at any time, it’s not safe to code with it.
Example Pseudoinstructions

- Rotate Right Instruction

  \[ \text{ror} \quad \text{reg}, \text{value} \]

  Expands to:

  \[ \text{srl} \quad \$at, \text{reg}, \text{value} \]
  \[ \text{sll} \quad \text{reg}, \text{reg}, 32\text{-value} \]
  \[ \text{or} \quad \text{reg}, \text{reg}, \$at \]

- “No OPeration” instruction

  \[ \text{nop} \]

  Expands to instruction = \( 0_{\text{ten}} \)

  \[ \text{sll} \quad \$0, \$0, 0 \]
Example Pseudoinstructions

- Wrong operation for operand

\texttt{addu \ reg,reg,value} \ # \ should \ be \ \texttt{addiu}

If value fits in 16 bits, \texttt{addu} is changed to:
\texttt{addiu \ reg,reg,value}

else:
\texttt{lui \ $at,upper \ 16 \ bits \ of \ value}
\texttt{ori \ $at,$at,lower \ 16 \ bits}
\texttt{addu \ reg,reg,$at}

- How do we avoid confusion about whether we are talking about MIPS assembler with or without pseudoinstructions?
**True Assembly Language (3/3)**

- **MAL** (MIPS Assembly Language): the set of instructions that a programmer may use to code in MIPS; this includes pseudoinstructions

- **TAL** (True Assembly Language): set of instructions that can actually get translated into a single machine language instruction (32-bit binary string)

- A program must be converted from MAL into TAL before translation into 1s & 0s.
Questions on Pseudoinstructions

• Question:
  • How does MIPS recognize pseudoinstructions?

• Answer:
  • It looks for officially defined pseudoinstructions, such as `ror` and `move`
  • It looks for special cases where the operand is incorrect for the operation and tries to handle it gracefully
Rewrite TAL as MAL

• TAL:

                   or       $v0,$0,$0
Loop:     slt       $t0,$0,$a1
            beq       $t0,$0,Exit
            add       $v0,$v0,$a0
            addi      $a1,$a1,-1
            j         Loop

Exit:

• This time convert to MAL

• It’s OK for this exercise to make up MAL instructions
Rewrite TAL as MAL (Answer)

**TAL:**

```
or    $v0,$0,$0
Loop:
    slt  $t0,$0,$a1
    beq  $t0,$0,Exit
    add  $v0,$v0,$a0
    addi $a1,$a1,-1
    j    Loop
Exit:
```

**MAL:**

```
li    $v0,0
Loop:
    bge  $zero,$a1,Exit
    add  $v0,$v0,$a0
    decre $a1, 1
    j    Loop
Exit:
```
Which of the instructions below are MAL and which are TAL?

A. `addi $t0, $t1, 40000`
B. `beq $s0, 10, Exit`
C. `sub $t0, $t1, 1`
Peer Instruction Answer

• Which of the instructions below are MAL and which are TAL?
  i. `addi $t0, $t1, 40000`
  ii. `beq $s0, 10, Exit`
  iii. `sub $t0, $t1, 1`

40,000 > +32,767 => `lui`, `ori`

sub:
- both must be registers;
- even if it were `subi`, there is no `subi` in TAL;
- generates `addi $t0, $t1, -1`

Beq:
- both must be registers
- Exit: if > 2
  15

ABC

1: MMM
2: MM
3: M
4: MT
5: MM
6: MT
7: MT
8: TTT
In semi-conclusion…

• Disassembly is simple and starts by decoding \textit{opcode} field.
  • Be creative, efficient when authoring C

• Assembler expands real instruction set (TAL) with pseudoinstructions (MAL)
  • Only TAL can be converted to raw binary
  • Assembler’s job to do conversion
  • Assembler uses reserved register $\at$
  • MAL makes it \textit{much} easier to write MIPS
Overview

• Interpretation vs Translation
• Translating C Programs
  • Compiler
  • Assembler (next time)
  • Linker (next time)
  • Loader (next time)
• An Example (next time)
### Language Continuum

<table>
<thead>
<tr>
<th>Low Efficiency</th>
<th>High Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inefficient to interpret</td>
<td>Efficient</td>
</tr>
<tr>
<td>Difficult to program</td>
<td>Easy to program</td>
</tr>
</tbody>
</table>

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

- 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
(+ assembler & linker)

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
• What if you want to run compiled programs (executables) from another ISA?

• Examples
  • VirtualPC let Windows (compiled to x86) run on old Macs (680x0 or PowerPC)
  • Run old video games on newer consoles
Machine Code Interpretation

• Apple’s Two Conversions
  • In the last 2 years, switched to Intel’s x86 from IBM’s PowerPC
  • Could require all programs to be re-translated from high level language
  • Did so with **minimal disruption** to programmer, and especially the user
    - **Rosetta** allows old PowerPC programs to run on the new x86 systems by runtime translation
    - **Universal Binaries** contain the machine code for both platforms, so both systems can run at native speeds
  • Did a similar thing 13 years ago when they switched from Motorola 680x0 instruction architecture to PowerPC
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

Compiler

Assembly program: foo.s

Assembler

Object (mach lang module): foo.o

Librarian

Executable (mach lang pgm): a.out

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. E.g.,

  • `mov $s1,$s2` ⇒ or `$s1,$s2,$zero`
And in conclusion...

- C program: foo.c
- Assembly program: foo.s
- Object (machine language module): foo.o
- Executable (machine language program): a.out

Diagram:

- Compiler
- Assembler
- Linker
- Loader
- Memory