There is one handout today at the front and back of the room!

Lecturer PSOE, new dad Dan Garcia

www.cs.berkeley.edu/~ddgarcia

2005-0xA

SF: Tiger over Daly ⇒

In a treat for the Bay Area, the top golfers descended to Harding Park in SF and saw a treat: the two longest hitters battled in a playoff before Daly “choked”.

sports.espn.go.com/golf/news/story?id=2185968
Review

• 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 - IEEE Four Rounding Modes

• We gave examples in base 10 to show you the 4 modes (only apply to \( .5_{10} \))
• What really happens is…
  1) in **binary**, not decimal!
  2) at the lowest bit of the mantissa with the **guard bit(s)** as our extra bit(s), and you need to decide how these extra bit(s) affect the result if the guard bits are “100…”
  3) If so, you’re half-way between the representable numbers.

E.g., \( 0.1010 \) is 5/8, halfway between our representable 4/8 \([1/2]\) and 6/8 \([3/4]\). Which number do we round to? 4 modes!
Decoding Machine Language

• How do we convert 1s and 0s to C code?
  Machine language ⇒ 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
  00010001000000000000000000000011
  00000000010001000001000000100000
  00100000101001011111111111111111
  00001000000100000000000000000001

- Next step: identify opcode and format

<table>
<thead>
<tr>
<th>R</th>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th>I</th>
<th>1, 4-31</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>2 or 3</td>
<td>target address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 | 0000000000000000000000010000000100101 |
| R | 00000000000001010100000000101010 |
| I | 00010001000000000000000000000011 |
| R | 00000000001000100000010000000001000000 |
| I | 00100000101001011111111111111111 |
| J | 000010000001000000000000000000001 |

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

```mips
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:

00001025_{hex}
0005402A_{hex}
11000003_{hex}
00441020_{hex}
20A5FFFF_{hex}
08100001_{hex}

After C code (Mapping below)

$v0$: product
$sa0$: multiplicand
$sa1$: multiplier

product = 0;
while (multiplier > 0) {
    product += multiplicand;
    multiplier -= 1;
}

Demonstrated Big 61C Idea: Instructions are just numbers, code is treated like data

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

Exit:
Administrivia…Midterm in 7 days!

- Project 2 due Wednesday (ok, Friday)
- Midterm 2005-10-17 @ 5:30-8:30pm Here!
- Covers labs, hw, proj, lec up through 7th wk
- Prev sem midterm + answers on HKN
- 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 (corrections below to bugs from “Core Instruction Set”)

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

2) \text{sll} and \text{srl} should shift values in \text{R[rt]}, not \text{R[rs]}
i.e. \text{sll/srl}: \text{R[rd]} = \text{R[rt]} \ll \text{shamt}
Review from before: \texttt{lui}

• So how does \texttt{lui} help us?

  • Example:

    \begin{verbatim}
    addi $t0,$t0, 0xABABCDCD
    \end{verbatim}

    becomes:

    \begin{verbatim}
    lui $at, 0xABAB
    ori $at, $at, 0xCD
    add $t0,$t0,$at
    \end{verbatim}

• 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 addi with lui, ori, 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
  `move reg2,reg1`
  Expands to:
  `add reg2,$zero,reg1`

• Load Immediate
  `li reg,value`
  If value fits in 16 bits:
  `addi reg,$zero,value`
  else:
  `lui reg,upper 16 bits of value`
  `ori reg,$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 ($1, called $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
  
  \texttt{ror \ reg, value}
  
  Expands to:
  
  \texttt{srl \$at, \ reg, \ value}
  
  \texttt{sll \ reg, \ reg, 32-value}
  
  \texttt{or \ reg, \ reg, \$at}

• “No Operation” instruction
  
  \texttt{nop}
  
  Expands to instruction $= 0_{10}$
  
  \texttt{sll \$0, \$0, 0}
Example Pseudoinstructions

• Wrong operation for operand
  addu reg, reg, value # should be addiu

If value fits in 16 bits, addu is changed to:
  addiu reg, reg, value
else:
  lui $at, upper 16 bits of value
  ori $at, $at, lower 16 bits
  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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>or $v0,$0,$0</td>
<td></td>
</tr>
<tr>
<td><strong>Loop:</strong> slt $t0,$0,$a1</td>
<td></td>
</tr>
<tr>
<td>beq $t0,$0,Exit</td>
<td></td>
</tr>
<tr>
<td>add $v0,$v0,$a0</td>
<td></td>
</tr>
<tr>
<td>addi $a1,$a1,-1</td>
<td></td>
</tr>
<tr>
<td>j Loop</td>
<td></td>
</tr>
</tbody>
</table>

#### 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:
Peer Instruction

1. Converting float -> int -> float produces same float number

2. Converting int -> float -> int produces same int number

3. FP add is associative: 
   \((x+y)+z = x+(y+z)\)
Peer Instruction Answer
Peer Instruction

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

<table>
<thead>
<tr>
<th></th>
<th>ABC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MMM</td>
</tr>
<tr>
<td>2</td>
<td>MMT</td>
</tr>
<tr>
<td>3</td>
<td>MTM</td>
</tr>
<tr>
<td>4</td>
<td>MTT</td>
</tr>
<tr>
<td>5</td>
<td>TMM</td>
</tr>
<tr>
<td>6</td>
<td>TMT</td>
</tr>
<tr>
<td>7</td>
<td>TTM</td>
</tr>
<tr>
<td>8</td>
<td>TTT</td>
</tr>
</tbody>
</table>
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

T

3: M

T

M

4: M

TT

5: T

MM

6: T

M

T

7: TT

8: TTT
<table>
<thead>
<tr>
<th>Week #</th>
<th>Mon</th>
<th>Wed</th>
<th>Thurs Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>#7 This week</td>
<td>MIPS III Running Program I</td>
<td>Running Program II</td>
<td>Running Program</td>
</tr>
<tr>
<td>#8 Midterm week</td>
<td>Midterm @ 5:30-8:30pm Here! (155 Dwin)</td>
<td>Intro to SDS I</td>
<td>SDS</td>
</tr>
</tbody>
</table>

(Review Sun @ 2pm 10 Evans)
In semi-conclusion...

- Disassembly is simple and starts by decoding 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 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>Scheme</th>
<th>Java bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>C</td>
</tr>
<tr>
<td>Assembly</td>
<td>machine language</td>
</tr>
</tbody>
</table>

- Easy to program vs. Efficient
- Inefficient to interpret vs. 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 (+ 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

• 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. E.g.,

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

C program: foo.c

Assembly program: foo.s

C Compiler

Assembler

Object(mach lang module): foo.o

Linker

Executable(mach lang pgm): a.out

Loader

Memory