Clarification - IEEE Four Rounding Modes

- We gave examples in base 10 to show you the 4 modes (only apply to $0.5_{10}$)
- 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 $\Rightarrow$ 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:

  \[
  \begin{array}{l}
  \text{00001025}_{\text{hex}}\\
  \text{0005402A}_{\text{hex}}\\
  \text{11000003}_{\text{hex}}\\
  \text{00441020}_{\text{hex}}\\
  \text{20A5FFFF}_{\text{hex}}\\
  \text{08100001}_{\text{hex}}
  \end{array}
  \]

- Let the first instruction be at address $4,194,304_{\text{ton}}$ ($0x00400000_{\text{hex}}$).
- Next step: convert hex to binary

Decoding Example (2/7)

- The six machine language instructions in binary:

  \[
  \begin{array}{l}
  \text{0000000000000000000001000000010101}\\
  \text{0000000000000000000001011000010110}\\
  \text{00010010000000000000000000000000}\\
  \text{00000000100101111111111111111111}\\
  \text{00001000000100000000000000000000}\\
  \text{00000000000000000000000000000000}
  \end{array}
  \]

- 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>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1, 4-31</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>2 or 3</td>
<td>target address</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:

<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>000000</td>
<td>00000000000000000000</td>
<td>01000001010101010101</td>
<td>00010001000000000000</td>
<td>00000000000000000001</td>
<td></td>
</tr>
</tbody>
</table>

- 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>1,048,577</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Next step: translate (“disassemble”) to MIPS assembly instructions

Decoding Example (5/7)

- MIPS Assembly (Part 1):

  Address:  Assembly instructions:
  0x00400000  or  $v0,$0,$0
  0x00400004  slt  $8,$0,$5
  0x00400008  beq  $8,$0,$5
  0x00400010  addi  $5,$5,-1
  0x00400014  j  0x100001

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

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

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)

| 00001025hex | $v0: product |
| 0005402Ahex | $a0: multiplicand |
| 11000003hex | $a1: multiplier |
| 00410120hex | product = 0; |
| 20A5FFFFhex | while (multiplier > 0) {
| 08100001hex | product += multiplicand;

| or  | $v0,$0,$0 |
| Loop:  | slt  | $t0,$0,$a1 |
| beq  | $t0,$0,Exit |
| addi  | $v0,$v0,$a0 |
| j  | Loop |

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 23hex, not 0 / 23hex.
  2) sll and srl should shift values in R[rt], not R[rs].
  le.sll/16:R[rd] = R[rt] << shamt
Review from before: lui

• So how does lui help us?
  • Example:
    ```
    addi $t0,$t0, 0xABABCDCD
    becomes:
    lui $at, 0xABAB
    ori $at, $at, 0xCDCD
    add $t0,$t0,$at
    ```
  • Now each I-format instruction has only a 16-bit immediate.
  • 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:
  ```
  add 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
  ```
  ror reg, value
  ```
  Expands to:
  ```
  srl $at, reg, value
  sll reg, reg, 32-value
  or reg, reg, $at
  ```

• “No OPeration” instruction
  ```
  nop
  ```
  Expands to instruction = 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,$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**

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`
    - `decr $a1, 1`
    - `j Loop`
  - `Exit:`

Peer Instruction

1. Converting float \(\rightarrow\) int \(\rightarrow\) float produces same float number
2. Converting int \(\rightarrow\) float \(\rightarrow\) int produces same int number
3. FP add is associative:
   \[(x+y)+z = x+(y+z)\]

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`

ABC

1: FFF
2: FTT
3: FFF
4: FT
5: TFF
6: TTT
7: TFF
8: TTT
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)

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

Compiler

Assembly program: foo.s

Assembler

Object (mach lang module): foo.o

Linker

lib.o

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

 Compiler

Assembly program: foo.s

Assembler

Object (mach lang module): foo.o

Linker

lib.o

Executable (mach lang pgm): a.out

Loader

Memory