Search your code!  
Go to http://www.google.com/codesearch, which allows you to specify regular expressions to search public source code.  
See what is there…  
See if you can find code you wrote!

Review

- Reserve exponents, significands:
  - Exponent: Significand: Object
    - 0: 0: 0
    - 0: nonzero: Denorm
    - 1-254: anything: +/- fl. pt. #
    - 255: 0: +/- ∞
    - 255: nonzero: NaN

- 4 rounding modes (default: unbiased)
- MIPS FL ops complicated, expensive

Outline

- Disassembly
- Pseudoinstructions and “True” Assembly Language (TAL) v. “MIPS” Assembly Language (MAL)

Decoding Machine Language

- How do we convert 1s and 0s to assembly language and to C code?
  - Machine language ⇒ assembly ⇒ C?
- For each 32 bits:
  1. Look at opcode to distinguish between R-Format, J-Format, and I-Format.
  2. Use instruction format to determine which fields exist.
  3. Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number.
  4. Logically convert this MIPS code into valid C code.  Always possible? Unique?

Decoding Example (1/7)

- Here are six machine language instructions in hexadecimal:
  - 00001025_{hex}
  - 0005402A_{hex}
  - 11000003_{hex}
  - 00441020_{hex}
  - 20A5FFFF_{hex}
  - 08100001_{hex}
- Let the first instruction be at address 4,194,304 (0x00400000_{hex}).
- Next step: convert hex to binary

Decoding Example (2/7)

- The six machine language instructions in binary:
  - 00000000000000000001000000100101
  - 00000000000001010100000000101010
  - 00010001000000000000000000000011
  - 00000000010010000001000000100000
  - 00100000101001011111111111111111
  - 00001000000100000000000000000001
- Next step: identify opcode and format

<table>
<thead>
<tr>
<th>R</th>
<th>I</th>
<th>L</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
</tr>
<tr>
<td>1, 4-31</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
<tr>
<td>2 or 3</td>
<td>target address</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>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>I</td>
</tr>
<tr>
<td>J</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>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>I</td>
</tr>
<tr>
<td>J</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):

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

- Next step: translate to C code (must be creative!)

Decoding Example (7/7)
- Before Hex:

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly instructions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001025</td>
<td>addi $t0, $t0, 0xABABCDCD</td>
</tr>
<tr>
<td>0x005402A</td>
<td>lui $at, 0xABAB</td>
</tr>
<tr>
<td>11000003</td>
<td>ori $at, $at, 0xCDCD</td>
</tr>
<tr>
<td>00441020</td>
<td>add $v0, $v0, $a0</td>
</tr>
<tr>
<td>20A5FFFF</td>
<td>addi $a1, $a1, -1</td>
</tr>
<tr>
<td>08100001</td>
<td>j Loop</td>
</tr>
</tbody>
</table>

- After C code (Mapping below):

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

Review from before: lui
- So how does lui help us?

- Example:

<table>
<thead>
<tr>
<th>Assembly instructions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>lui $at, 0xABAB</td>
</tr>
<tr>
<td>ori $at, $at, 0xCDCD</td>
</tr>
<tr>
<td>add $t0, $t0, $at</td>
</tr>
</tbody>
</table>

- 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 pseudo-instructions?
  - They’re broken up by the assembler into several “real” MIPS instructions.
  - Some examples follow

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
  ```

Example Pseudoinstructions

- **Load Address**: How do we get the address of an instruction or global variable into a register?

  ```
  la reg, label
  ```

  Again if value fits in 16 bits:
  
  ```
  addi reg, $zero, label_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 pseudo-instruction, 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_{16b}
  
  ```
  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 assembler / SPIM recognize pseudo-instructions?
- **Answer:**
  - It looks for officially defined pseudo-instructions, 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: ble $a1, $zero, Exit
         add $v0, $v0, $a0
         sub $a1, $a1, 1
         j Loop
  Exit:
  ```

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`

In 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