IEEE 754 Floating Point Standard (review)

° Biased Notation, where bias is number subtracted to get real number
  - IEEE 754 uses bias of 127 for single precision
  - Subtract 127 from Exponent field to get actual value for exponent
  - 1023 is bias for double precision

° Summary (single precision):

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 23 22</td>
<td>0</td>
</tr>
<tr>
<td>1 bit</td>
<td>8 bits</td>
</tr>
</tbody>
</table>

\((-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - 127)}\)

Double precision exp:11, significand:52 and exponent bias of 1023
IEEE 754 Floating Point Review (2)

° Encodings (Single Precision):

<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>denormal.</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- fl. pt. #</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- infinity</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Denormalized number: no (implied) leading 1, exponent = -126.

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 $4,194,304_{ten}$ ($0x00400000_{hex}$).

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

° The six machine language instructions in binary:

00000000000000000001000000100101
000000000000010101000000010101
0001000100000000000000000000011
000000000100010000010000010000
001000001010010111111111111111
000010000001000000000000000001

° 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:

Format:

| R | 00000000000000000001000000100101 |
| 000000000000010101000000010101 |
| 0001000100000000000000000000011 |
| 000000000100010000010000001000 |
| 001000001010010111111111111111 |
| 000010000001000000000000000001 |

° 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:

Format:

<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></td>
<td>+3</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></td>
<td>-1</td>
</tr>
<tr>
<td>J</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>1,048,577</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 $2,$0,$0
0x00400004  slt $8,$0,$5
0x00400008  beq $8,$0,3
0x0040000c  add $2,$2,$4
0x00400010  addi $5,$5,-1
0x00400014  j 0x100001

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

° MIPS Assembly (Part 2):

```assembly
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 (must be creative!)

Decoding Example (7/7)

Before Hex: ° After C code (Mapping below)

<table>
<thead>
<tr>
<th>Hex</th>
<th>° After C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001025&lt;sub&gt;hex&lt;/sub&gt;</td>
<td>$v0: product</td>
</tr>
<tr>
<td>0005402A&lt;sub&gt;hex&lt;/sub&gt;</td>
<td>$a0: multiplicand</td>
</tr>
<tr>
<td>11000003&lt;sub&gt;hex&lt;/sub&gt;</td>
<td>$a1: multiplier</td>
</tr>
<tr>
<td>00441020&lt;sub&gt;hex&lt;/sub&gt;</td>
<td>product = 0;</td>
</tr>
<tr>
<td>20A5FFFF&lt;sub&gt;hex&lt;/sub&gt;</td>
<td>while (multiplier &gt; 0) {</td>
</tr>
<tr>
<td>08100001&lt;sub&gt;hex&lt;/sub&gt;</td>
<td>product += multiplicand;</td>
</tr>
<tr>
<td></td>
<td>multiplier -= 1;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

```c
or $v0,$0,$0
Loop: 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

- Exam ready to return
- Stats posted on website
- Regrade policy on website:
  - Put it in writing
  - Before one week from today

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 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?

  \texttt{la \ reg,\texttt{label}}

  \texttt{Again if value fits in 16 bits:  
  addi \ reg,\$zero,\texttt{label\_value}}

  \texttt{else:}

  \texttt{lui \ reg,upper 16 bits of value}

  \texttt{ori \ reg,\$zero,lower 16 bits}

True Assembly Language (2/3)

° Problem:

  \begin{itemize}
  \item When breaking up a pseudo-instruction, the assembler may need to use an extra reg.
  \item If it uses any regular register, it’ll overwrite whatever the program has put into it.
  \end{itemize}

° Solution:

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

\begin{align*}
\text{TAL:} & \\
\text{or} & \quad v_0, 0, 0 \\
\text{Loop:} & \\
\text{slt} & \quad t_0, 0, a_1 \\
\text{beq} & \quad t_0, 0, \text{Exit} \\
\text{add} & \quad v_0, v_0, a_0 \\
\text{addi} & \quad a_1, a_1, -1 \\
\text{j} & \quad \text{Loop} \\
\text{Exit:} & \\
\end{align*}

°This time convert to MAL

°It’s OK for this exercise to make up MAL instructions

Rewrite TAL as MAL (Answer)

°TAL:

\begin{align*}
\text{TAL:} & \\
\text{or} & \quad v_0, 0, 0 \\
\text{Loop:} & \\
\text{slt} & \quad t_0, 0, a_1 \\
\text{beq} & \quad t_0, 0, \text{Exit} \\
\text{add} & \quad v_0, v_0, a_0 \\
\text{addi} & \quad a_1, a_1, -1 \\
\text{j} & \quad \text{Loop} \\
\text{Exit:} & \\
\end{align*}

°MAL:

\begin{align*}
\text{MAL:} & \\
\text{li} & \quad v_0, 0 \\
\text{Loop:} & \\
\text{bge} & \quad 0, a_1, \text{Exit} \\
\text{add} & \quad v_0, v_0, a_0 \\
\text{sub} & \quad a_1, a_1, 1 \\
\text{j} & \quad \text{Loop} \\
\text{Exit:} & \\
\end{align*}
Quiz

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`

Quiz Answer

Which of the instructions below are **MAL** and which are **TAL**?

i. `addi $t0, $t1, 40000`  \(40,000 > +32,767\) => `lui, ori`
ii. `beq $s0, 10, Exit`  
Beq: both must be registers
Exit: if > 2\(^{15}\), then MAL
iii. `sub $t0, $t1, 1`  
sub: both must be registers; even if it was `subi`, there is no `subi` in TAL; generates `addi $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