Digital film network ⇒
The UK is investing in 150 digital cinemas! Each will get a 100 GiB lossless digital copy of the film and show it on digital 2K (2048x1080) projectors. USA?!

news.bbc.co.uk/1/hi/technology/4297865.stm
Clarification - IEEE Four Rounding Modes

• **This is just an example in base 10 to show you the 4 modes.**

• **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!
Outline

• Disassembly

• Pseudoinstructions and “True” Assembly Language (TAL) v. “MIPS” Assembly Language (MAL)
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:

```
0000000000000000000001000000100101
00000000000001010100000000101010
00010001000000000000000000000011
00000000010001000001000000100000
00100000101001000010000001000000
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>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1, 4-31</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>2 or 3</td>
<td></td>
<td></td>
<td></td>
<td>target address</td>
<td></td>
</tr>
</tbody>
</table>
Decoding Example (3/7)

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

Format:

R

<table>
<thead>
<tr>
<th>000000000000000001000000100101</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000000010101000000001010</td>
</tr>
<tr>
<td>000100010000000000000000000011</td>
</tr>
<tr>
<td>000000000100010000010000001000</td>
</tr>
<tr>
<td>001000001010010111111111111111</td>
</tr>
<tr>
<td>000010000000010000000000000001</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>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>+3</td>
<td></td>
<td></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>
<td></td>
</tr>
<tr>
<td>I</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>2</td>
<td></td>
<td></td>
<td>1,048,577</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):

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x004000000</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
  slt      $t0,$0,$a1
  beq      $t0,$0,Exit
  add      $v0,$v0,$a0
  addi     $a1,$a1,-1
  j        Loop

- 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
- `$a0$: multiplicand
- `$a1$: multiplier

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

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

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

• Thanks to TAs who filled in last week
• SIGCSE 2005 was GREAT
• Your midterm is in 7 days!
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 \texttt{addi} with \texttt{lui}, \texttt{ori}, \texttt{add}
• **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
  
  \texttt{move\ reg2,reg1}

  Expands to:
  
  \texttt{add\ reg2,$zero,reg1}

- Load Immediate
  
  \texttt{li\ reg,value}

  If value fits in 16 bits:
  
  \texttt{addi\ reg,$zero,value}

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

  \texttt{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**

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

Expands to:

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

- **“No Operation”** instruction

\[ \text{nop} \]

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

\[ \text{sll} \; \$0, \$0, 0 \]
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
    sub $a1,$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`

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</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>

ABC
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