Lecture 17 – Introduction to MIPS
Instruction Representation III

2004-10-08

Lecturer PSOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia

World Cyber Games
5-day competition held in SF to determine the top video game players in the world. Halo, Warcraft III, FIFA Soccer, Counter-Strike, etc.

www.worldcybergames.com
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 $\Rightarrow$ 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:

- $00001025_{\text{hex}}$
- $0005402A_{\text{hex}}$
- $11000003_{\text{hex}}$
- $00441020_{\text{hex}}$
- $20A5FFFF_{\text{hex}}$
- $08100001_{\text{hex}}$

• 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
000000000000101010100000000101010
00010001000000000000000000000011
00000000010001000001000000100000
00100000101001001011111111111111
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>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:**

<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>00000000000000000001000000100101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000000001010100000000101010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001000100000000000000000000011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000010001000001000000100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100000101001011111111111111111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001000000010000000000000000001</td>
<td></td>
<td></td>
<td></td>
<td></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>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></td>
<td>+3</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></td>
<td>-1</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>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):

```plaintext
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)

00001025_{hex} \quad \$v0: product

0005402A_{hex} \quad \$a0: multiplicand

11000003_{hex} \quad \$a1: multiplier

00441020_{hex} \quad \text{product} = 0;

20A5FFFF_{hex} \quad \text{while} (\text{multiplier} > 0) \{ \\
08100001_{hex} \quad \quad \quad \quad \quad \quad \quad \text{product} += \text{multiplicand}; \\
\}

or \quad \$v0,$0,$0

Loop: slt \quad \$t0,$0,$a1

beq \quad \$t0,$0,Exit

add \quad \$v0,$v0,$a0

addi \quad \$a1,$a1,-1

j \quad \text{Loop}

Exit:

Demonstrated Big 61C

Idea: Instructions are just numbers, code is treated like data
Kilo, Mega, Giga, Tera, Peta, Exa, Zetta, Yotta

physics.nist.gov/cuu/Units/binary.html

• Common use prefixes (all SI, except K [= k in SI])

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbr</th>
<th>Factor</th>
<th>SI size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kilo</td>
<td>K</td>
<td>$2^{10} = 1,024$</td>
<td>$10^3 = 1,000$</td>
</tr>
<tr>
<td>Mega</td>
<td>M</td>
<td>$2^{20} = 1,048,576$</td>
<td>$10^6 = 1,000,000$</td>
</tr>
<tr>
<td>Giga</td>
<td>G</td>
<td>$2^{30} = 1,073,741,824$</td>
<td>$10^9 = 1,000,000,000$</td>
</tr>
<tr>
<td>Tera</td>
<td>T</td>
<td>$2^{40} = 1,099,511,627,776$</td>
<td>$10^{12} = 1,000,000,000,000$</td>
</tr>
<tr>
<td>Peta</td>
<td>P</td>
<td>$2^{50} = 1,125,899,906,842,624$</td>
<td>$10^{15} = 1,000,000,000,000,000$</td>
</tr>
<tr>
<td>Exa</td>
<td>E</td>
<td>$2^{60} = 1,152,921,504,606,846,976$</td>
<td>$10^{18} = 1,000,000,000,000,000,000$</td>
</tr>
<tr>
<td>Zetta</td>
<td>Z</td>
<td>$2^{70} = 1,180,591,620,717,411,303,424$</td>
<td>$10^{21} = 1,000,000,000,000,000,000,000$</td>
</tr>
<tr>
<td>Yotta</td>
<td>Y</td>
<td>$2^{80} = 1,208,925,819,614,629,174,706,176$</td>
<td>$10^{24} = 1,000,000,000,000,000,000,000,000$</td>
</tr>
</tbody>
</table>

• Confusing! Common usage of “kilobyte” means 1024 bytes, but the “correct” SI value is 1000 bytes

• Hard Disk manufacturers & Telecommunications are the only computing groups that use SI factors, so what is advertised as a 30 GB drive will actually only hold about $28 \times 2^{30}$ bytes, and a 1 Mbit/s connection transfers $10^6$ bps.
kibi, mebi, gibi, tebi, pebi, exbi, zebi, yobi

> en.wikipedia.org/wiki/Binary_prefix

- **New IEC Standard Prefixes [only to exbi officially]**

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbr</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>kibi</td>
<td>Ki</td>
<td>$2^{10} = 1,024$</td>
</tr>
<tr>
<td>mebi</td>
<td>Mi</td>
<td>$2^{20} = 1,048,576$</td>
</tr>
<tr>
<td>gibi</td>
<td>Gi</td>
<td>$2^{30} = 1,073,741,824$</td>
</tr>
<tr>
<td>tebi</td>
<td>Ti</td>
<td>$2^{40} = 1,099,511,627,776$</td>
</tr>
<tr>
<td>pebi</td>
<td>Pi</td>
<td>$2^{50} = 1,125,899,906,842,624$</td>
</tr>
<tr>
<td>exbi</td>
<td>Ei</td>
<td>$2^{60} = 1,152,921,504,606,846,976$</td>
</tr>
<tr>
<td>zebi</td>
<td>Zi</td>
<td>$2^{70} = 1,180,591,620,717,411,303,424$</td>
</tr>
<tr>
<td>yobi</td>
<td>Yi</td>
<td>$2^{80} = 1,208,925,819,614,629,174,706,176$</td>
</tr>
</tbody>
</table>

- **International Electrotechnical Commission (IEC) in 1999 introduced these to specify binary quantities.**
  - Names come from shortened versions of the original SI prefixes (same pronunciation) and *bi* is short for “binary”, but pronounced “bee” :-(
  - Now SI prefixes only have their base-10 meaning and never have a base-2 meaning.

As of this writing, this proposal has yet to gain widespread use…
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 do 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
  
  `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 recognize pseudoinstructions?

• Answer:
  • It looks for officially defined pseudoinstructions, such as \texttt{ror} and \texttt{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:

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

  Exit:

• MAL:

  Loop:  
  li $v0,0  
  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`
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