Lecture 10: Disassembly and Pseudo-instructions
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:

  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
00000000000001010100000000101010
00010001000000000000000000000001
00000000010001000001000000100000
00000000010001000001000000100000
00100000101001011111111111111111
00001000000100000000000000000001

• Next step: identify opcode and format

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>1, 4-31</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>J</td>
<td>2 or 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>target address</td>
</tr>
</tbody>
</table>
Decoding Example (3/7)

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

Format:

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
R & 000000000000000001000000100101 & \\
R & 0000000000000101010000000101010 & \\
I & 00010001000000000000000000000011 & \\
I & 00010000100010000000010000000100000 & \\
J & 00100000101001011111111111111111 & \\
J & 0000100000010001000000000000000001 & \\
\hline
\end{array}
\]

• 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>
</tbody>
</table>
| 0       | 0 0 0 2 0 37  
| R       | 0 0 5 8 0 42  
| I       | 4 8 0 +3  
| R       | 0 2 4 2 0 32  
| I       | 8 5 5 -1  
| J       | 2 1,048,577  

- **Next step: translate (“disassemble”) to MIPS assembly instructions**
### 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)
• 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:
00001025\text{hex}
0005402A\text{hex}
11000003\text{hex}
00441020\text{hex}
20A5FFFE\text{hex}
08100001\text{hex}

After - C code (Mapping below)

$v0$: product
$a0$: multiplicand
$a1$: multiplier

$\text{product} = 0;$
while (multiplier $>$ 0) {
  $\text{product} += \text{multiplicand};$
  multiplier $-\text{=} 1;$
}

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

Demonstrated Idea:
Instructions are just numbers, code is treated like data
Review from before: lui

- So how does lui help us?
  - Example:
    
    \[
    \text{addi} \quad \$t0, \$t0, \text{0xABABCDDC}
    \]

    becomes:
    
    \[
    \text{lui} \quad \$at, \text{0xABAB} \\
    \text{ori} \quad \$at, \$at, \text{0xCDCD} \\
    \text{add} \quad \$t0, \$t0, \$at
    \]
  
  - 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
• **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.
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`
• **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 code 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 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:
```

• Let's convert to MAL
Rewrite TAL as MAL (Answer)

- **TAL:**
  - ori $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: bgt $zero, $a1, Exit
  - add $v0, $v0, $a0
  - sub $a1, $a1, 1
  - j Loop

- **Exit:**
Quickie Quiz

Which of the instructions below are MAL?

i. addi $t0, $t1, 40000  
40,000 > +32,767 => lui, ori

ii. beq $s0, 10, Exit  
Beq: both must be registers
Exit: could be > 2^{15}

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