Lecture 9: Disassembly

CSE 30: Computer Organization and Systems Programming
Winter 2010

Rajesh Gupta / Ryan Kastner
Dept. of Computer Science and Engineering
University of California, San Diego
Instruction Formats

- **I-format**: used for instructions with immediates, `lw` and `sw` (since the offset counts as an immediate), and the branches (`beq` and `bne`),
  - (but not the shift instructions; later)
- **J-format**: used for `j` and `jal`
- **R-format**: used for all other instructions
- It will soon become clear why the instructions have been partitioned in this way
R–Format Instructions

- Define “fields” of the following number of bits each: $6 + 5 + 5 + 5 + 5 + 6 = 32$

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

- For simplicity, each field has a name:

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

- Important: On these slides and in book, each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer.

  - Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63.
**R–Format Instructions**

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

- **What do these field integer values tell us?**
  - **opcode**: partially specifies what instruction it is
    - Note: This number is equal to 0 for all R-Format instructions.
  - **funct**: combined with opcode, this number exactly specifies the instruction
  - **Question**: Why aren’t opcode and funct a single 12-bit field?
    - **Answer**: We’ll answer this later
R–Format Instructions

| opcode | rs | rt | rd | shamt | funct |

- **More fields:**
  - **rs** (Source **Register**): *generally* used to specify register containing first operand
  - **rt** (Target **Register**): *generally* used to specify register containing second operand (note that name is misleading)
  - **rd** (Destination **Register**): *generally* used to specify register which will receive result of computation
**R–Format Instructions**

- **Final field:**
  - **shamt**: This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is redundant, so this field is only 5 bits (so it can represent the numbers 0-31).

- This field is set to 0 in all but the shift instructions.

- For a detailed description of field usage for each instruction, see back inside cover of P&H textbook.
What about instructions with immediates?

- 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
- Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise

Define new instruction format that is partially consistent with R-format:

- First notice that, if instruction has immediate, then it uses at most 2 registers.
I–Format Instructions

- Define “fields” of the following number of bits each: $6 + 5 + 5 + 16 = 32$ bits

| 6 | 5 | 5 | 16 |

- Again, each field has a name:

| opcode | rs | rt | immediate |

- **Key Concept**: Only one field is inconsistent with R-format. Most importantly, opcode is still in same location.
I–Format Instructions

What do these fields mean?

- **opcode**: same as before except that, since there’s no *funct* field, opcode uniquely specifies an instruction in I-format

- This also answers question of why R-format has two 6-bit fields to identify instruction instead of a single 12-bit field: in order to be consistent with other formats
I–Format Instructions

- More fields:
  - $rs$: specifies the *only* register operand (if there is one)
  - $rt$: specifies register which will receive result of computation (this is why it’s called the *target* register “rt”)

I–Format Instructions

- The Immediate Field:
  - addi, slti, sltiu, the immediate is sign-extended to 32 bits. Thus, it’s treated as a signed integer.
  - 16 bits can be used to represent immediate up to $2^{16}$ different values.
  - This is large enough to handle the offset in a typical lw or sw, plus a vast majority of values that will be used in the slti instruction.
J–Format Instructions

- For branches, we assumed that we won’t want to branch too far, so we can specify change in PC.
- For general jumps (j and jal), we may jump to anywhere in memory.
- Ideally, we could specify a 32-bit memory address to jump to.
- Unfortunately, we can’t fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise.
J–Format Instructions

- Define “fields” of the following number of bits each:

| 6 bits | 26 bits |

- As usual, each field has a name:

| opcode | target address |

- Key Concepts
  - Keep opcode field identical to R-format and I-format for consistency.
  - Combine all other fields to make room for large target address.
J-Format Instructions

- For now, we can specify 26 bits of the 32-bit bit address.

- Optimization:
  - Note that, just like with branches, jumps will only jump to word aligned addresses, so last two bits are always 00 (in binary).
  - So let’s just take this for granted and not even specify them.
J–Format Instructions

- Now specify 28 bits of a 32-bit address
- Where do we get the other 4 bits?
  - By definition, take the 4 highest order bits from the PC.
  - Technically, this means that we cannot jump to anywhere in memory, but it’s adequate 99.9999…% of the time, since programs aren’t that long
    - only if straddle a 256 MB boundary
  - If we absolutely need to specify a 32-bit address, we can always put it in a register and use the `jr` instruction.
J–Format Instructions

Summary:
- New PC = PC[31..28] || target address || 00

Note: || means concatenation
4 bits || 26 bits || 2 bits = 32 bit address
- 1010 || 11111111111111111111111111 || 00 = 1010111111111111111111111100
Decoding Machine Language

- How do we convert 1s and 0s to assembly language and to C code?

Machine language $\Rightarrow$ assembly $\Rightarrow$ C?

- For each 32 bits:
  - Look at opcode to distinguish between R-Format, J-Format, and I-Format.
  - Use instruction format 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

- 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

- The six machine language instructions in binary:
  00000000000000000001000000100101
  00000000000001010100000000101010
  00010001000000000000000000000011
  00000000100010000000100000100000
  00100000101001011111111111111111
  00001000000100000000000000000001

- Next step: identify opcode and format

<table>
<thead>
<tr>
<th>R</th>
<th>I</th>
<th>J</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1, 4–62</td>
<td>2 or 3</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td>target address</td>
<td></td>
</tr>
</tbody>
</table>
Decoding Example

- 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>R</td>
</tr>
<tr>
<td>I</td>
</tr>
<tr>
<td>J</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R-Format</td>
</tr>
<tr>
<td>2 or 3</td>
<td>J-Format</td>
</tr>
<tr>
<td>Others</td>
<td>I-Format</td>
</tr>
</tbody>
</table>

Next step: separation of fields
Decoding Example

- Fields separated based on format/opcode:

<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>0</td>
<td>+3</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>
</tr>
<tr>
<td>I</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1,048,577</td>
</tr>
</tbody>
</table>

- Next step: translate (“disassemble”) to MIPS assembly instructions
Decoding Example

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 0x1000001</td>
</tr>
</tbody>
</table>

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

- 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
## Decoding Example

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

### After C code (Mapping below)

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

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

### Demonstrated Big Idea:

Instructions are just numbers, code is treated like data.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>$v0,0,0$</td>
</tr>
<tr>
<td>Loop: slt</td>
<td>$t0,0,a1$</td>
</tr>
<tr>
<td>beq</td>
<td>$t0,0,$0,Exit</td>
</tr>
<tr>
<td>add</td>
<td>$v0,v0,a0$</td>
</tr>
<tr>
<td>addi</td>
<td>$a1,a1,-1$</td>
</tr>
<tr>
<td>j</td>
<td>Loop</td>
</tr>
</tbody>
</table>
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**
  - lireg,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
  ```
Problem:
- When breaking up a pseudo-instruction, the assembler may need to use an extra register.
- If it uses any regular register, it’ll overwrite whatever the program has put into it.

Solution:
- Reserve a register ($1$, called $@\text{at}$ 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**

  \[
  \text{ror} \quad \text{reg}, \quad \text{value} \\
  \text{Expands to:} \\
  \text{srl} \quad \text{at}, \quad \text{reg}, \quad \text{value} \\
  \text{sll} \quad \text{reg}, \quad \text{reg}, \quad 32-\text{value} \\
  \text{or} \quad \text{reg}, \quad \text{reg}, \quad \text{at} \\
  \]

- **“No Operation” instruction**

  \[
  \text{nop} \\
  \text{Expands to instruction} = 0_{\text{ten}}, \\
  \text{sll} \quad \text{0, 0, 0} \\
  \]
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**

- **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
 Rewire TAL as MAL

- TAL:
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source 1</th>
<th>Source 2</th>
<th>Source 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>or $v0,$0,$0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop: slt $t0,$0,$a1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq $t0,$0,Exit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $v0,$v0,$a0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $a1,$a1,-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j Loop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exit:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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?

i.  addi $t0, $t1, 40000
ii. beq $s0, 10, Exit
iii. sub $t0, $t1, 1

<table>
<thead>
<tr>
<th></th>
<th>ABC</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>
Peer Instruction 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
Conclusion

- Define instructions to be same size as data word (one word) so that they can use the same memory (compiler can use `lw` and `sw`).
- Computer actually stores programs as a series of these 32-bit numbers.
- **MIPS Machine Language Instruction:** 
  32 bits representing a single instruction

<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- target address
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