ECE232: Hardware Organization and Design

Part 6: MIPS Instructions II

http://www.ecs.umass.edu/ece/ece232/

Computer Organization - Previously covered

- Byte/Word addressing
- Registers
- Program counter – 32 bits
MIPS Registers

- Fast access to program data
- Register names:
  - $0$–$31$ or R0–R31
- Specialized names based on usage convention
  - Register $R0$/0/$zero$: hardwired to constant zero
  - $t0$–$t7$ ($8$–$15$) - “temporary” registers
  - $s0$–$s7$ ($16$–$23$) - “saved” registers
  - $sp$ - stack pointer
    - Allow relative addressing
  - Other special-purpose registers

MIPS Instruction Types

- Arithmetic & Logical - manipulate data in registers
  - add $s1$, $s2$, $s3$  \( s1 = s2 + s3 \)
  - or $s3$, $s4$, $s5$  \( s3 = s4 \text{ OR } s5 \)

- Instruction usage (assembly)
  - add dest, src1, src2  \( \text{dest} = \text{src1} + \text{src2} \)

- Instruction characteristics
  - Always 3 operands: 2 sources + destination
  - Operand order is fixed
  - Operands are always general-purpose registers
### MIPS Instruction Types - 2

- **Data Transfer** - move register data to/from memory
  - `lw $s1, 100($s2)` → $s1 = Memory[$s2 + 100]
  - `sw $s1, 100($s2)` → Memory[$s2 + 100] = $s1

- **Branch** - alter program flow
  - `beq $s1, $s2, 25` → if ($s1==$s2) skip 25 instructions

### Arithmetic/Logical Instructions: Binary Representation

Each instruction – 32 bits

- **op**: Basic operation of the instruction (opcode)
- **rs**: first register source operand
- **rt**: second register source operand
- **rd**: register destination operand
- **shamt**: shift amount (more about this later)
- **funct**: function - specific type of operation

Also called “R-Format” or “R-Type” Instructions
MIPS Instructions

- All instructions exactly 32 bits wide
- Different formats for different purposes
- Similarities in formats ease implementation

<table>
<thead>
<tr>
<th>R-Format</th>
<th>I-Format (e.g., Load/Store, Branch)</th>
<th>J-Format (e.g., Jump)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td></td>
<td>rt</td>
<td>rd</td>
</tr>
<tr>
<td></td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td></td>
<td>offset</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>26 bits</td>
<td></td>
</tr>
<tr>
<td>op</td>
<td>address</td>
<td></td>
</tr>
</tbody>
</table>

MIPS Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands;</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands;</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant;</td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands;</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands;</td>
</tr>
<tr>
<td>add imm. unsigned</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant;</td>
</tr>
<tr>
<td>multiply</td>
<td>mult $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>multu $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td>divide</td>
<td>div $2,$3</td>
<td>Lo = $2 ÷ $3,</td>
<td>Lo = quotient, Hi = remainder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hi = $2 mod $3</td>
<td></td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3,</td>
<td>Unsigned quotient &amp; remainder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hi = $2 mod $3</td>
<td></td>
</tr>
<tr>
<td>Move from Hi</td>
<td>mfhi $1</td>
<td>$1 = Hi</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mflo $1</td>
<td>$1 = Lo</td>
<td>Used to get copy of Lo</td>
</tr>
</tbody>
</table>
### MIPS Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $1,$2,$3</td>
<td>$1 = $2 &amp; $3$</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>or $1,$2,$3</td>
<td>$1 = $2 \mid $3$</td>
<td>3 reg. operands; Logical OR</td>
</tr>
<tr>
<td>xor</td>
<td>xor $1,$2,$3</td>
<td>$1 = $2 \oplus $3$</td>
<td>3 operands; Logical XOR</td>
</tr>
<tr>
<td>and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = $2 &amp; 10$</td>
<td>Logical AND reg, constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>ori $1,$2,10</td>
<td>$1 = $2 \mid 10$</td>
<td>Logical OR reg, constant</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xori $1$, $2,10</td>
<td>$1 = $2 \oplus 10$</td>
<td>Logical XOR reg, constant</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sll $1,$2,10</td>
<td>$1 = $2 &lt;&lt; 10$</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10$</td>
<td>Shift right by constant</td>
</tr>
</tbody>
</table>

### MIPS Data Transfer Instructions

- Transfer data between registers and memory
- **Instruction format (assembly)**
  
  ```
  lw $dest, offset($addr)  # load word
  sw $src, offset($addr)  # store word
  ```

- **Examples**
  
  ```
  lw $s1, 100($s2)  # $s1 = Memory[$s2 + 100]
  sw $s1, 100($s2)  # Memory[$s2 + 100] = $s1
  ```

- **Uses:**
  - Accessing a variable in main memory
  - Accessing an array element

- **32-bit base address**
- **16-bit offset**
**Example - Loading a Variable**

Example: `lw R5,8(R2)`

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0 (constant)</td>
</tr>
<tr>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>0x10</td>
</tr>
<tr>
<td>R3</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td>692310</td>
</tr>
<tr>
<td>R30</td>
<td></td>
</tr>
<tr>
<td>R31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable X</td>
<td>0x10</td>
</tr>
<tr>
<td>Variable Y</td>
<td>0x14</td>
</tr>
<tr>
<td>Variable Z = 692310</td>
<td>0x18</td>
</tr>
</tbody>
</table>

**Data Transfer Instructions Binary Representation**

- **6 bits** | **5 bits** | **5 bits** | **16 bits** |
  - **op** | **rs** | **rt** | **offset** |

- **op**: Basic operation of the instruction (**opcode**)
- **rs**: first register source operand
- **rt**: second register source operand

- **offset**: 16-bit signed address offset (-32,768 to +32,767)
- Also called “I-Format” or “I-Type” instructions
I-Format vs. R-Format Instructions

- Compare with R-Format

<table>
<thead>
<tr>
<th></th>
<th>I-Format</th>
<th>R-Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td>rs</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td>rt</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td>rd</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td>funct</td>
<td>6 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td>shamt</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
</tbody>
</table>

Note similarity!

MIPS Conditional Branch Instructions

- Conditional branches allow decision making

  - `beq R1, R2, LABEL` if `R1==R2` goto `LABEL`
  - `bne R3, R4, LABEL` if `R3!=R4` goto `LABEL`
  - `beq $s1, $s2, 25` if `($s1==$s2)` PC = `PC + 4 + 4*25`
    else `PC = PC + 4`

- Example C Code
  
  ```c
  if (i==j) goto L1;
  f = g + h;
  L1: f = f - i;
  ```

- Assembly
  
  ```assembly
  beq $s3, $s4, L1
  add $s0, $s1, $s2
  L1: sub $s0, $s0, $s3
  ```
Example: Compiling C if-then-else

- **Example**
  - **C Code**
    ```c
    if (i==j) f = g + h;
    else f = g - h;
    ```
  - **Assembly**
    ```
    bne $s3, $s4, Else
    add $s0, $s1, $s2
    j Exit;  # new: unconditional jump
    Else:
    sub $s0, $s1, $s2
    Exit:
    ```

- New Instruction: Unconditional jump
  ```
  j LABEL  # goto Label
  ```

Binary Representation - Branch

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
</tr>
</tbody>
</table>

- Branch instructions use I-Format
- **offset** is added to PC when branch is **taken**

  ```
  beq r0, r1, offset
  ```

  has the effect:

  ```
  Conversion to byte offset
  ```

  ```
  if (r0==r1) pc = pc + 4 + (offset << 2)
  else pc = pc + 4;
  ```

- Offset is specified in instruction **words** (why?)
- What is the **range** of the branch target addresses?
Comparisons - What about <, ≤, >, ≥?

- bne, beq provide equality comparison
- slt (set on less than) provides magnitude comparison
  \[ \text{slt } $t0,$s3,$s4 \# if } s3<s4 \text{ } $t0=1; \]
  \[ \# \text{ else } s3>s4 \text{ } $t0=0; \]
- Combine with bne or beq to branch:
  \[ \text{slt } $t0,$s3,$s4 \# if (a<b) \]

condition register

\[ \text{bne } $t0,$zero,Less \quad \# \text{ goto Less;} \]

- Why not include a blt instruction in hardware?
  \[ \begin{itemize}
  \item Supporting in hardware would lower performance
  \item Assembler provides this function if desired
  \end{itemize} \]

(binary Representation - Jump)

- Jump Instruction uses J-Format (op=2)
- What happens during execution?
  \[ \text{PC} = \text{PC}[31:28] : (\text{IR}[25:0] \ll 2) \]
  \[ \text{Concatenate upper 4 bits of PC to form complete 32-bit address} \]
  \[ \text{Conversion to byte offset} \]
Constants / Immediate Instructions

- Small constants are used quite frequently (50% of operands)
  
  `e.g.,   A = A + 5;`
  `B = B + 1;`
  `C = C - 18;`

- MIPS Immediate Instructions (I-Format):
  
  `addi $29, $29, 4`
  `addi $29, $29, -8`
  `slti $8, $18, 10`
  `andi $29, $29, 6`
  `ori $29, $29, 4`

  \[ \text{Arithmetic instructions sign-extend immed.} \]
  \[ \text{Logical instructions don't sign extend immed.} \]

- Allows up to 16-bit constants
- How do you load just a constant into a register?

  `ori $5, $zero, 666`

Larger Constants

- Immediate operations provide only for 16-bit constants
  
  - Because 16 bits fit neatly in a 32-bit instruction
  - Because most constants are small (i.e., < 16 bits)

- What about when we need larger constants?

- Use \textit{load upper immediate - lui} (I-Format)
  `lui $t0, 1010101010101010`

- Then use \textit{ori} to fill in lower 16 bits:
  `ori $t0, $t0, 1110111011101100`

\[ \text{Then, $t0$ looks like:} \]

\[ \begin{array}{c}
\text{original contents} \\
\hline
\text{filled with zeros} \\
\hline
1010101010101010 \\
0000000000000000
\end{array} \]

\[ \begin{array}{c}
\text{ori $t0$, $t0$, 1110111011101100} \\
\hline
1010101010101010 \\
1110111011101100
\end{array} \]
MIPS Registers and Usage

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>$v0–$v1</td>
<td>2–3</td>
<td>values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0–$a3</td>
<td>4–7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0–$t7</td>
<td>8–15</td>
<td>temporary registers</td>
</tr>
<tr>
<td>$s0–$s7</td>
<td>16–23</td>
<td>saved registers</td>
</tr>
<tr>
<td>$t8–$t9</td>
<td>24–25</td>
<td>more temporary registers</td>
</tr>
<tr>
<td>$k0–$k1</td>
<td>26–27</td>
<td>reserved for Operating System kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>