CS3350B Computer Architecture
MIPS Instruction Representation

Marc Moreno Maza

http://www.csd.uwo.ca/~moreno/cs3350_moreno/index.html
Department of Computer Science
University of Western Ontario, Canada

Tuesday January 30, 2018
Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as **32-bit** instruction words
  - Small number of formats encoding operation code (opcode), register numbers, ...
  - Regularity!
- Register numbers
  - $t0 - t7$ are reg’s 8 - 15
  - $t8 - t9$ are reg’s 24 - 25
  - $s0 - s7$ are reg’s 16 - 23
Overview: MIPS R3000 ISA

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point coprocessor
  - Memory Management
  - Special

- 3 Basic Instruction Formats: all 32 bits wide

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sha</th>
<th>funct</th>
<th>R-format</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td>I-format</td>
</tr>
<tr>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jump target</td>
<td>J-format</td>
</tr>
</tbody>
</table>

Registers

- R0 - R31
- PC
- HI
- LO
## MIPS ISA Selected Instruction Set

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>OP/ funct</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic</strong></td>
<td>add</td>
<td>R</td>
<td>0/32</td>
<td>add $s1, $s2, $s3</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>R</td>
<td>0/34</td>
<td>sub $s1, $s2, $s3</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>I</td>
<td>8</td>
<td>addi $s1, $s2, 6</td>
</tr>
<tr>
<td></td>
<td>or immediate</td>
<td>I</td>
<td>13</td>
<td>ori $s1, $s2, 6</td>
</tr>
<tr>
<td><strong>Data Transfer</strong></td>
<td>load word</td>
<td>I</td>
<td>35</td>
<td>lw $s1, 24($s2)</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>I</td>
<td>43</td>
<td>sw $s1, 24($s2)</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>I</td>
<td>32</td>
<td>lb $s1, 25($s2)</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>I</td>
<td>40</td>
<td>sb $s1, 25($s2)</td>
</tr>
<tr>
<td></td>
<td>load upper imm</td>
<td>I</td>
<td>15</td>
<td>lui $s1, 6</td>
</tr>
<tr>
<td><strong>Cond. Branch</strong></td>
<td>br on equal</td>
<td>I</td>
<td>4</td>
<td>beq $s1, $s2, L</td>
</tr>
<tr>
<td></td>
<td>br on not equal</td>
<td>I</td>
<td>5</td>
<td>bne $s1, $s2, L</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>R</td>
<td>0/42</td>
<td>slt $s1, $s2, $s3</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>I</td>
<td>10</td>
<td>slti $s1, $s2, 6</td>
</tr>
<tr>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Uncond. Jump</strong></td>
<td>jump</td>
<td>J</td>
<td>2</td>
<td>j 250</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>R</td>
<td>0/8</td>
<td>jr $t1</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>J</td>
<td>3</td>
<td>jal 250</td>
</tr>
</tbody>
</table>
MIPS R-format Instructions

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

- Instruction fields
  - **op**: operation code (opcode)
  - **rs**: first source register number
  - **rt**: second source register number
  - **rd**: destination register number
  - **shamt**: shift amount (00000 for now)
  - **funct**: function code (extends opcode)
### R-format Example

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

**add $t0, $s1, $s2**

<table>
<thead>
<tr>
<th>op</th>
<th>$s1</th>
<th>$s2</th>
<th>$t0</th>
<th>shamt</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |

000000100011001001000000000100000002
Logical Operations

- Instructions for **bitwise** manipulation

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td>«</td>
<td>«</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>»</td>
<td>»</td>
<td>srl</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

- Useful for extracting and inserting groups of bits in a word
### Shift Operations

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

- **shamt**: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by \(i\) bits multiplies by \(2^i\)
  - Example: `sll $s0, $t0, 4`
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by \(i\) bits divides by \(2^i\) (unsigned only)
  - Example: `srl $s0, $t0, 4`
Shift Operation Example

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

sll $s0, $t0, 4

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>$t0</th>
<th>$s0</th>
<th>4</th>
<th>shift left</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

0000000 00000 01000 10000 00100 000000

0000000000001000100000010000000002
AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```plaintext
and $t0, $t1, $t2
```

<table>
<thead>
<tr>
<th></th>
<th>$t2</th>
<th>$t1</th>
<th>$t0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000 0000 0000 0000 0000 1101 1100 0000</td>
<td>0000 0000 0000 0000 0011 1100 0000 0000</td>
<td>0000 0000 0000 0000 0000 1100 0000 0000</td>
</tr>
</tbody>
</table>
```
OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

or $t0$, $t1$, $t2$

<table>
<thead>
<tr>
<th>$t2$</th>
<th>0000 0000 0000 0000 0000 0000 1101 1100 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1$</td>
<td>0000 0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
<tr>
<td>$t0$</td>
<td>0000 0000 0000 0000 0000 0011 1101 1100 0000</td>
</tr>
</tbody>
</table>
NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - \( a \text{ NOR } b = \neg (a \text{ OR } b) \)

\[
\text{nor } \$t0, \$t1, \$zero
\]

# Register 0: always read as zero

<table>
<thead>
<tr>
<th>$t1</th>
<th>0000 0000 0000 0000 0011 1100 0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t0</td>
<td>1111 1111 1111 1111 1100 0011 1111 1111</td>
</tr>
</tbody>
</table>
MIPS I-format Instructions

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **Immediate** arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: $-2^{15}$ to $+2^{15} - 1$
  - Address: offset added to base address in rs

- **Design Principle 4**: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible
Conditional Operations

- Branch to a labeled instruction if a condition is true
  Otherwise, continue sequentially
- beq rs, rt, L1
  - if (rs == rt) branch to instruction labeled L1;
- bne rs, rt, L1
  - if (rs != rt) branch to instruction labeled L1;
- j L1
  - unconditional jump to instruction labeled L1
Compiling If Statements

- C code:
  ```c
  if (i==j) { f = g+h; }
  else { f = g-h; }
  ```
  - `f, g, ...` in `$s0, $s1, ...$

- Compiled MIPS code:
  ```assembly
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
  ```
  Else:
  ```assembly
  sub $s0, $s1, $s2
  # Assembler calculates addresses
  ```
  Exit: ...
Compiling Loop Statements

- **C code:**
  ```c
  while (save[i] == k) { i += 1; }
  ```
  - `i` in `$s3`, `k` in `$s5`, address of `save[]` in `$s6`

- **Compiled MIPS code:**
  ```mips
  Loop: sll $t1, $s3, 2
        add $t1, $t1, $s6
        lw $t0, 0($t1)
        bne $t0, $s5, Exit
        addi $s3, $s3, 1
        j Loop
  Exit: ...
  ```
More Conditional Operations

- Set result to 1 if a condition is true
  Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with `beq, bne`
  - `slt $t0, $s1, $s2` # if ($s1 < $s2)
  - `bne $t0, $zero, L` # branch to L
Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111$
  - $s1 = 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001$
  - `slt $t0, $s0, $s1` # signed
    -1 < +1 ⇒ $t0 = 1$
  - `sltu $t0, $s0, $s1` # unsigned
  - +4,294,967,295 > +1 ⇒ $t0 = 0
Could use bitwise operations

- **MIPS** *byte/halfword load/store*
  - String processing is a common case
  - Sign extend to 32 bits in rt
    \[
    \text{lb } rt, \text{ offset}(rs) \quad \text{lh } rt, \text{ offset}(rs)
    \]
  - Zero extend to 32 bits in rt
    \[
    \text{lbu } rt, \text{ offset}(rs) \quad \text{lhu } rt, \text{ offset}(rs)
    \]
  - Store just rightmost byte/halfword
    \[
    \text{sb } rt, \text{ offset}(rs) \quad \text{sh } rt, \text{ offset}(rs)
    \]
(1) **Register addressing** - **operand** is in a register

```
| op | rs | rt | rd | funct |
```

In register rs

`word operand`

Example: `add $rd, $rs, $rt` # `$rd = $rs + $rt`
(2) **Base (displacement) addressing** - operand is at the memory location whose address is the sum of a register and a 16-bit constant contained within the instruction.

Example: `lw $rt, offset($rs)` # $rt = Memory($rs+offset)
Immediate addressing - operand is a 16-bit constant contained within the instruction

\[
\begin{array}{cccc}
\text{op} & \text{rs} & \text{rt} & \text{operand} \\
\end{array}
\]

Example: addi $rt, $rs, operand # $rt = $rs + operand
(1) **PC-relative addressing** - instruction address is the sum of the PC and a 16-bit constant contained within the instruction.

Example: `beq` and `bne`:  # if rs==rt (or rs!=rt), go to offset (PC=PC+4+4*offset)
(2) **Pseudo-direct addressing** - instruction address is the 26-bit constant contained within the instruction concatenated with the upper 4 bits of the PC.

Example: `j` (jump): PC | xxxx | jump address | 00 |
Addressing Mode Summary

1. Immediate addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Immediate} \]

2. Register addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{rd} \quad \ldots \quad \text{funct} \]
   Registers
   
   \[ \text{Register} \]

3. Base addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Address} \]
   
   \[ \text{Register} \quad + \quad \text{Memory} \]
   
   \[ \text{Byte} \quad \text{Halfword} \quad \text{Word} \]

4. PC-relative addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Address} \]
   
   \[ \text{PC} \quad + \quad \text{Memory} \]
   
   \[ \text{Word} \]

5. Pseudodirect addressing
   \[ \text{op} \quad \text{Address} \]
   
   \[ \text{PC} \quad : \quad \text{Memory} \]
   
   \[ \text{Word} \]
Caution:
Addressing Mode is NOT Instruction Types

- Addressing mode is how an address (memory or register) is determined.
- Instruction type is how the instruction is put together.
- Example: addi, beq, and lw are all I-Format instructions. But,
  - addi uses immediate addressing mode (and register)
  - beq uses pc-relative addressing (and register)
  - lw uses base addressing (and register)
Summary of MIPS Addressing Modes

- **Register**: a source or destination operands specified as content of one of the registers $0$-$31$.
- **Immediate**: a numeric value embedded in the instruction is the actual operand.
- **PC-relative**: a data or instruction memory location is specified as an offset relative to the incremented PC.
- **Base**: a data or instruction memory location is specified as a signed offset from a register.
- **Register-direct**: the value of the effective address is in a register.
- **Pseudo-direct**: the memory address is (mostly) embedded in the instruction.
MIPS Organization So Far
Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

- Layers of software/hardware
  - Compiler, assembler, hardware

- MIPS: typical of RISC ISAs
  - c.f. x86
Aside: Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory: byte-addressable
  - it means that a byte is the smallest unit with its address
  - Naturally aligned data: doublewords that lie on addresses that are multiples of eight, words that lie on addresses that are multiples of four, halfwords that lie on addresses that are multiples of two, and single bytes that lie at any byte address. Such data is located on its natural size boundary, to maximize storage potential and to provide for fast, efficient memory access.

- Little Endian: rightmost byte is word address
  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

- Big Endian: leftmost byte is word address
  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- MIPS memory is byte-addressable; supports 32-bit address (an address is given as a 32-bit unsigned integer)
Aside: Compiler storage of data objects by byte alignment

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, bool</td>
<td>1</td>
<td>Located at any byte address</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>Located at any address that is evenly divisible by 2</td>
</tr>
<tr>
<td>float, int, long, pointer</td>
<td>4</td>
<td>Located at an address that is evenly divisible by 4</td>
</tr>
<tr>
<td>long long, double, long double</td>
<td>8</td>
<td>Located at an address that is evenly divisible by 8</td>
</tr>
</tbody>
</table>