Chapter 2 — Instructions: Language of the Computer — 1
Instructions: MIPS ISA

Based on Text: Patterson Henessey

Publisher: Morgan Kaufmann

Edited by Y.K. Malaiya for CS470

Acknowledgements to V.D. Agarwal and M.J. Irwin
### Aside: MIPS Register Convention

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserve on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0 (hardware)</td>
<td>n.a.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>returned values</td>
<td>no</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16-23</td>
<td>saved values</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return addr (hardware)</td>
<td>yes</td>
</tr>
</tbody>
</table>
Unsigned Binary Integers

- Given an n-bit number

\[ x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0 \]

- Range: 0 to +2\(^n\) – 1

- Example
  - 0000 0000 0000 0000 0000 0000 0000 1011_2

\[ = 0 + \cdots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \]

\[ = 0 + \cdots + 8 + 0 + 2 + 1 = 11_{10} \]

- Using 32 bits
  - 0 to +4,294,967,295
2s-Complement Signed Integers

- Given an n-bit number

\[ x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0 \]

- Range: \(-2^{n-1}\) to \(+2^{n-1} - 1\)

- Example
  - \(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2\)
  - \(= -1 \times 2^{31} + 1 \times 2^{30} + \cdots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0\)
  - \(= -2,147,483,648 + 2,147,483,644 = -4_{10}\)

- Using 32 bits
  - \(-2,147,483,648\) to \(+2,147,483,647\)
2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- \(-(-2^{n-1})\) can’t be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 ... 0000
  - \(-1\): 1111 1111 ... 1111
  - Most-negative: 1000 0000 ... 0000
  - Most-positive: 0111 1111 ... 1111
Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

\[ x + x = 1111...111 \_2 = -1 \]
\[ x + 1 = -x \]

- Example: negate +2
  - +2 = 0000 0000 ... 0010\_2
  - –2 = 1111 1111 ... 1101\_2 + 1
    \[ = 1111 1111 ... 1110\_2 \]
Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `1b, 1h`: extend loaded byte/halfword
  - `beq, bne`: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 ⇒ 0000 0000 0000 0010
  - –2: 1111 1110 ⇒ 1111 1111 1111 1110
Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

<table>
<thead>
<tr>
<th>0</th>
<th>0000</th>
<th>4</th>
<th>0100</th>
<th>8</th>
<th>1000</th>
<th>c</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
<td>9</td>
<td>1001</td>
<td>d</td>
<td>1101</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
<td>a</td>
<td>1010</td>
<td>e</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
<td>b</td>
<td>1011</td>
<td>f</td>
<td>1111</td>
</tr>
</tbody>
</table>

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000
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
## Aside: MIPS Register Convention

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserve on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0 (hardware)</td>
<td>n.a.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>returned values</td>
<td>no</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16-23</td>
<td>saved values</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return addr (hardware)</td>
<td>yes</td>
</tr>
</tbody>
</table>
MIPS R-format Instructions

Instruction fields
- **op**: operation code (opcode)
- **rs**: first source register number
- **rt**: second source register number
- **rd**: destination register number
- **shamt**: shift amount (000000 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>special</th>
<th>$s1</th>
<th>$s2</th>
<th>$t0</th>
<th>0</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>

| 0000000 | 10001 | 10010 | 01000 | 00000 | 100000 |

00000010001100100100000000100000₂ = 02324020₁₆
MIPS I-format Instructions

- 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

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

The BIG Picture

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs
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>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</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

- **Shamtm**: how many positions to shift
- **Shift left logical**
  - Shift left and fill with 0 bits
  - `sll` by `i` bits multiplies by $2^i$
- **Shift right logical**
  - Shift right and fill with 0 bits
  - `srl` by `i` bits divides by $2^i$ (unsigned only)
AND Operations

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

And $t0$, $t1$, $t2$

| 1000 0000 0000 0000 0000 0000 1101 1100 0000 |
| 1000 0000 0000 0000 0000 0000 0011 1100 0000 0000 |
| 1000 0000 0000 0000 0000 0000 0000 1100 0000 0000 0000 |
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
  - \( a \text{ NOR } b = \text{ NOT } ( a \text{ OR } b ) \)

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

Register 0: always read as zero

\[
\begin{array}{c}
\$t1 \\
0000 0000 0000 0000 0011 1100 0000 0000
\end{array}
\]

\[
\begin{array}{c}
\$t0 \\
1111 1111 1111 1111 1100 0011 1111 1111
\end{array}
\]
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:
  ```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
  Else: sub $s0, $s1, $s2
  Exit: ...
  ```

Assembler calculates addresses
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: ...
  ```
Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)

- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks
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
  ```
Branch Instruction Design

- Why not `blt`, `bge`, etc?
- Hardware for `<`, `≥`, … slower than `=`, `≠`
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise
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 \Rightarrow $t0 = 1$
- `sltu $t0, $s0, $s1  # unsigned`
  - $+4,294,967,295 > +1 \Rightarrow $t0 = 0$

Most unsigned instructions simply do not cause overflow