Brief Review of the MIPS Instruction Set Architecture

RISC Instruction Set Basics

• All operations on data apply to data in registers and typically change the entire register
• The only operations that affect memory are load and store operations
• The instruction formats are few in number with all instructions typically one size

• Text uses MIPS64
  – Instructions generally have a D at the start or end of the mnemonic, e.g. DADD is 64 bit ADD
MIPS ISA

• 32 registers
  – Register 0 always has the value 0
• Three classes of instructions
  – ALU instructions
    • Register to register or immediate to register
    • Signed or unsigned
    • Floating point or Integer
    • NOT to memory
  – Load/Store instructions
    • Base register added to signed offset to get an effective address
  – Branches and Jumps
    • Branch based on condition bit or comparison between pair of registers

MIPS arithmetic

• Most instructions have 3 operands
• Operand order is fixed (destination first)

Example:

HLL code: \[ A = B + C; \]

MIPS code: \texttt{DADD }$s0,\; s1,\; s2$

($s0, \; s1$ and $s2$ are associated with variables by compiler)
MIPS arithmetic

HLL code: \[ A = B + C + D; \]
\[ E = F - A; \]

MIPS code: \[ DADD \, \$t0, \, \$s1, \, \$s2 \]
\[ DADD \, \$s0, \, \$t0, \, \$s3 \]
\[ DSUB \, \$s4, \, \$s5, \, \$s0 \]

Operands must be registers
- Compiler tries to keep as many variables in registers as possible
- Some variables can not be allocated
  - large arrays
  - aliased variables (variables accessible through pointers)
  - dynamically allocated variables on the heap or stack
- Compiler may run out of registers; this is called **spilling**

Memory layout: Alignment

- Words are aligned (32 bit in this example)
- Big-endian or Little-endian depending on the OS
Instructions: load and store

Example:


MIPS code: \[
\begin{align*}
\text{LW} & \quad t0, 24(s3) \\
\text{DADD} & \quad t0, s2, t0 \\
\text{SW} & \quad t0, 24(s3)
\end{align*}
\]

- 8 bytes per word \(\rightarrow\) offset to 3\(^{rd}\) word \(\rightarrow\) 24 byte displacement
- \(h\) already in register \(s2\)
- Store word operation has no destination (reg) operand

Swap example

C

```c
swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

MIPS32

```mips
swap:
    MULI $2, $5, 4
    ADD $2, $4, $2
    LW $15, 0($2)
    LW $16, 4($2)
    SW $16, 0($2)
    SW $15, 4($2)
    JR $31
```

Explanation:
- index \(k: \$5\)
- base address of \(v: \$4\)
- address of \(v[k]\) is \(\$4 + 4*\$5\)
**MIPS32 Instruction Formats**

### Register-Register

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>6 5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td>SA</td>
<td>Op2</td>
<td>Opx</td>
</tr>
</tbody>
</table>

### Register-Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td>immediate</td>
<td></td>
</tr>
</tbody>
</table>

### Branch

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2/Opx</td>
<td>immediate</td>
<td></td>
</tr>
</tbody>
</table>

### Jump / Call

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Target (offset added to PC)</td>
<td></td>
</tr>
</tbody>
</table>

---

**Control**

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- MIPS unconditional branch
  
  \[
  \text{J Label}
  \]

- MIPS conditional branch instructions:
  
  \[
  \begin{align*}
  & \text{BNE } $t0, $t1, Label \\
  & \text{BEQ } $t0, $t1, Label
  \end{align*}
  \]

- Example: 
  
  \[
  \text{if } (X==Y) \\
  \quad \text{A = B + C;}
  \]

  \[
  \begin{align*}
  & \text{BNE } $s4, $s5, Label \\
  & \text{DADD } $s3, $s0, $s1
  \end{align*}
  \]

  Label: ....
Control Flow

• We have: BEQ, BNE, what about Branch-if-less-than?
• New instruction:

<table>
<thead>
<tr>
<th>meaning:</th>
</tr>
</thead>
<tbody>
<tr>
<td>if $s1 &lt; $s2 then $t0 = 1</td>
</tr>
<tr>
<td>else $t0 = 0</td>
</tr>
</tbody>
</table>

SLT $t0, $s1, $s2

• Can follow this with BNE $t0, $zero, Label
to get branch if less than

MIPS compiler conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$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-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved (by callee)</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$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>
What’s this do? 32 bits

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>LI</td>
<td>$3</td>
<td>4</td>
<td></td>
<td># load immediate</td>
</tr>
<tr>
<td>MULI</td>
<td>$2</td>
<td>$3, 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>$2</td>
<td>$1, $2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>$15, 0</td>
<td>($2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDI</td>
<td>$15, $15</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>$15, 0</td>
<td>($2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDI</td>
<td>$3</td>
<td>$3, -1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>$3</td>
<td>$zero,</td>
<td>Foo</td>
<td></td>
</tr>
</tbody>
</table>

Brief look at the 80x86

- Textbook appendix has more details

- Complexity:
  - Instructions from 1 to 17 bytes long
  - One operand must act as both a source and destination
  - One operand can come from memory
  - Complex addressing modes
    - E.g., “base or scaled index with 8 or 32 bit displacement”

- Saving grace:
  - The most frequently used instructions are not too difficult to build
  - Compilers avoid the portions of the architecture that are slow

- Implementation on later processors translates x86 instructions into RISC-like instructions internally, allowing it to adopt many of the RISC innovations