Chapter Goals

- Teach a subset of MIPS assembly language
- Introduce the stored program concept
- Explain how MIPS instructions are represented in machine language
- Illustrate basic instruction set design principles

Instructions:

- Language of the Machine
- More primitive than higher level languages
- Very restrictive
  - e.g., MIPS Arithmetic Instructions

MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed

Example:

\[
\text{C code: } \quad A = B + C
\]
\[
\text{MIPS code: } \quad \text{add }$s0, $s1, $s2
\]

*Design principles: to be found...*  
*Design goals:*
MIPS arithmetic

- Design Principle #1: simplicity favors regularity. Why?
- Of course this complicates some things...

C code: 
```
A = B + C + D;
E = F - A;
```

MIPS code: 
```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

Registers vs. Memory

- Design Principle #2: smaller is faster. Why?
- Therefore, arithmetic instruction operands must be “registers”
  - And only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables?

Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- “Byte addressing” means that the index points to a byte of memory.

Memory Organization

- Bytes are nice, but most data items use larger “words”
- For MIPS, a word is 32 bits or 4 bytes.
- Words are aligned
  i.e., what are the least 2 significant bits of a word address?
Memory Instructions

- Load and store instructions
- Example:

<table>
<thead>
<tr>
<th>C code</th>
<th>MIPS code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[8] = h + A[8];</td>
<td>lw $t0, 32($s3) add $t0, $s2, $t0 sw $t0, 32($s3)</td>
</tr>
</tbody>
</table>

- For lw/sw, address always = register value + offset
- How about this?
  add $t0, 32($s3), $t0

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: add $t0, $s1, $s2
    - registers have numbers, $t0=8, $s1=17, $s2=18
- Instruction Format (r-type):

  \[
  \begin{array}{cccccccc}
  000000 & 10001 & 10010 & 01000 & 00000 & 100000 \\
  \text{op} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{funct} \\
  \end{array}
  \]

So far we’ve learned:

- MIPS
  - loading words but addressing bytes
  - arithmetic on registers only

- Instruction | Meaning
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2+100]</td>
</tr>
<tr>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2+100] = $s1</td>
</tr>
</tbody>
</table>

Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
    - Principle #3: Make the common case fast
    - Principle #4: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - Example: lw $t0, 44($s2)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>18</td>
<td>44</td>
</tr>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>16 bit number</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Where's the compromise?
Example Part 1

- What is the machine code for the following:

\[ A[300] = h + A[300]; \]

Variable \( h \) is assigned register $s2
Array \( A \) base address is assigned register $t1

- Do the assembly code first, then machine language instructions, and then machine code

Example Part 2

- What is the machine code for the following:

\[ A[300] = h + A[300]; \]

Variable \( h \) is assigned register $s2 & Array \( A \) base address is assigned register $t1

- First part of answer:

\[
\begin{align*}
\text{lw} & \ $10, 1200($t1) \ # \text{Temporary reg} \ $10 \text{ gets A[300]} \\
\text{add} & \ $10, s2, s10 \ # \text{Temporary reg} \ $10 \text{ gets } h + A[300] \\
\text{sw} & \ $10, 1200($t1) \ # \text{Stores } h + A[300] \text{ back into A[300]} \\
\end{align*}
\]

- Second part of answer (DECIMAL):

<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>35</td>
<td>9</td>
<td>8</td>
<td>1200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>18</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>43</td>
<td>9</td>
<td>8</td>
<td>1200</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example Part 3

- What is the machine code for the following:

\[ A[300] = h + A[300]; \]

Variable \( h \) is assigned register $s2 & Array \( A \) base address is assigned register $t1

- First part of answer:

\[
\begin{align*}
\text{lw} & \ $10, 1200($t1) \ # \text{Temporary reg} \ $10 \text{ gets A[300]} \\
\text{add} & \ $10, s2, s10 \ # \text{Temporary reg} \ $10 \text{ gets } h + A[300] \\
\text{sw} & \ $10, 1200($t1) \ # \text{Stores } h + A[300] \text{ back into A[300]} \\
\end{align*}
\]

- Second part of answer (BINARY):

<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>100011</td>
<td>01001</td>
<td>01000</td>
<td>0000 0100 1011 0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>10010</td>
<td>01000</td>
<td>01000 0000 010000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101011</td>
<td>01001</td>
<td>01000</td>
<td>0000 0100 1011 0000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stored Program Computers

- 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
- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the "next" instruction and continue