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
- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo, Silicon Graphics, Sony

Design principles: to be found...

Design goals:
**MIPS arithmetic**

- All instructions have 3 operands
- Operand order is fixed

Example:

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

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

C code: 
\[
A = B + C + D;  \\
E = F - A;
\]
MIPS code: 
\[
\text{add} \ $t0, \ $s1, \ $s2  \\
\text{add} \ $s0, \ $t0, \ $s3  \\
\text{sub} \ $s4, \ $s5, \ $s0
\]

**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.

- \( 2^{32} \) bytes with byte addresses from 0 to \( 2^{32} - 1 \)
- \( 2^{30} \) words with byte addresses 0, 4, 8, ... \( 2^{30} - 4 \)
- Words are aligned
  i.e., what are the least 2 significant bits of a word address?

Memory Instructions

- Load and store instructions
- Example:
  - MIPS code: 
    \[
    \begin{align*}
    \text{lw} \ & \$t0, \ 32(\$a3) \\
    \text{add} \ & \$t0, \ \$s2, \ \$t0 \\
    \text{sw} \ & \$t0, \ 32(\$a3)
    \end{align*}
    \]

- How about this?
  - \( \text{add} \ \$t0, \ 32(\$a3), \ \$t0 \)

So far we’ve learned:

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</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 $s1, 100($a2)</td>
<td>$s1 = Memory[$a2+100]</td>
</tr>
<tr>
<td>sw $s1, 100($a2)</td>
<td>Memory[$a2+100] = $s1</td>
</tr>
</tbody>
</table>

Exercise #1

- What is the MIPS assembly code for the following:
  \( g = g + h \cdot i \);
  Variables \( g, h, \) & \( i \) are assigned registers \$s1, \$s2, and \$s4

Exercise #2

• What is the MIPS assembly code for the following:
  \[ g = h + A[3]; \]
  Variables g, h, \& i are assigned registers $s1$, $s2$, and $s4$
  Array A base address is assigned register $s3$

Exercise #3

• What is the MIPS assembly code for the following:
  \[ g = h + A[i]; \]
  Variables g, h, \& i are assigned registers $s1$, $s2$, and $s4$
  Array A base address is assigned register $s3$

Extra space

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):

```
 000000 10001 10010 01000 00000 100000
```

- `op`
- `rs`
- `rt`
- `rd`
- `shamt`
- `funct`
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:  
  \[
  lw \quad s0, 44(s2)
  \]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>18</td>
<td>8</td>
<td>44</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:
  \[
  lw \quad s0, 1200(s1) \quad # \text{Temporary reg $s0$ gets } A[300]
  \]
  \[
  add \quad s0, s2, s0 \quad # \text{Temporary reg $s0$ gets } h + A[300]
  \]
  \[
  sw \quad s0, 1200(s1) \quad # \text{Stores } h + A[300] \text{ back into } A[300]
  \]
- Second part of answer:

<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>0</td>
<td>18</td>
<td>8</td>
<td>8</td>
<td>32</td>
<td></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 $s1
- First part of answer:
  \[
  lw \quad s0, 1200(s1) \quad # \text{Temporary reg $s0$ gets } A[300]
  \]
  \[
  add \quad s0, s2, s0 \quad # \text{Temporary reg $s0$ gets } h + A[300]
  \]
  \[
  sw \quad s0, 1200(s1) \quad # \text{Stores } h + A[300] \text{ back into } A[300]
  \]
- Second part of answer:

<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>0100</td>
<td>01000</td>
<td>00000</td>
<td>01000</td>
<td>01001</td>
</tr>
</tbody>
</table>
Stored Program Concept

• Instructions are composed of bits / bytes / words
• Programs are stored in memory
  — to be read or written just like data

- 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

memory for data, programs, compilers, editors, etc.