January 30

- Chun-Fa will demo Spim installation and operation
Instructions:

- Language of the Machine
- More primitive than higher level languages
e.g., no sophisticated control flow
- 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 goals: maximize performance and minimize cost, reduce design time*
MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

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

MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)
MIPS arithmetic

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

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

  MIPS code:
  \[
  \text{add } \$t0, \$s1, \$s2 \\
  \text{add } \$s0, \$t0, \$s3 \\
  \text{sub } \$s4, \$s5, \$s0
  \]

- Operands must be registers, only 32 registers provided
- Design Principle: *smaller is faster*. Why?
Registers vs. Memory

- Arithmetic instructions operands must be registers, — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables?

Diagram:
- Control
- Datapath
- Memory
- Input
- Output
- Processor
- I/O
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.

```
char memory[256*1024*1024];
```
Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>4</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>8</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>12</td>
<td>32 bits of data</td>
</tr>
</tbody>
</table>

Registers hold 32 bits of data

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

• What order are the bytes inside the word?
• Is byte 0 the high-order bits of word 0?
• Or is it the low order bits?

• “Big Endian” byte 0 is high-order bits [0 1 2 3]
  – Macintosh, SPARC
• “Little Endian” byte 0 is low-order bits [3 2 1 0]
  – Intel, DECStation

When would I care?
**Instructions**

- Load and store instructions
- Example:

  **C code:**
  
  \[
  \]

  **MIPS code:**
  
  \[
  lw \ $t0, \ 32($s3)  
  add \ $t0, \ $s2, \ $t0  
  sw \ $t0, \ 32($s3)
  \]

- Store word has destination last
- Remember arithmetic operands are registers, not memory!
So far we’ve learned:

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

- **Instruction**
  - **Meaning**

<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 $s1, 100($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

- Instructions, like registers and words of data, are also 32 bits long
  - Example: \texttt{add $t0, $s1, $s2}
  - registers have numbers, $t0=8$, $s1=17$, $s2=18$

- Instruction Format:

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
 000000 & 10001 & 10010 & 01000 & 00000 & 100000 \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
 op & rs & rt & rd & shamt & funct \\
6 bits & 5 bits & 5 bits & 5 bits & 5 bits & 6 bits \\
\hline
\end{tabular}
\end{center}
Consider the load-word and store-word instructions,
- What would the regularity principle have us do?
- New principle: **Good design demands a compromise**

Introduce a new type of instruction format
- I-type for data transfer instructions
- other format was R-type for register

Example: `lw $t0, 32($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>9</td>
<td>32</td>
</tr>
</tbody>
</table>

Where's the compromise?
Stored Program Concept

- Instructions are bits
- 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

Diagram:
- Processor
- Memory
- Memory for data, programs, compilers, editors, etc.