Instructions:

- Language of the Machine
- We’ll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980’s
  - Almost 100 million MIPS processors manufactured in 2002
  - used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, ...

MIPS Technologies

MIPS is the #1 architecture in several high-volume, high-growth market segments

Boxes shipped with MIPS-Based™ Silicon

- Cable STB 75%
- Digital STB 40%
- Satellite STB 33%
- DVD Recorders 63%
- Cable Modems 97%
- Internet Backbone 40%
- DSL 52%
- WLAN 40%
- VoIP 71%

Why MIPS

Assembly Language?

- It’s an offspring of one of the authors (John Hennessy started the MIPS project 20 years ago).
- It’s RISC (Reduced Instruction Set Computer).
- It’s simple!
- It’s in the book.
- It’s still a viable architecture.
MIPS arithmetic

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

Example:

C code: \( a = b + c \)

MIPS 'code': \texttt{add} \( a, b, c \)

"The natural number of operands for an operation like addition is three...requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple"

- Translation from C to MIPS is performed by a \texttt{MIPS arithmetic}

Design Principles of Hardware Technology

- **Design Principle 1:** simplicity favors regularity.
- Of course this complicates some things...

<table>
<thead>
<tr>
<th>C code: ( a = b + c + d; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS code: \texttt{???}</td>
</tr>
</tbody>
</table>

- Operands must be registers, only 32 registers provided
- Each register contains 32 bits

- **Design Principle 2:** smaller is faster.
- Why?

Compiling a C assignment using Registers

\[ f = (g + h) - (i + j) \]

The variables \( f, g, h, i \) and \( j \) are assigned to registers \( $s0, $s1, $s2, $s3, \) and \( $s4 \) respectively.

\( $t0, $t1 \rightarrow \text{temporary registers} \)

What is the compiled MIPS code?

Registers vs. Memory

- Arithmetic instructions operands must be registers, — only 32 registers provided
- Compiler relates variables with registers
- What about programs with lots of variables? Where are the these kept?

![Registers vs. Memory Diagram]
The Datapath

- What type of hardware is the register file composed of?

Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array

Instructions

- Load instruction
- Example: Assume A is an array of 100 words. Variables g and h are associated with reg $s1 and $s2. Starting (base) address of the array is in $s3.

C code: \( g = h + A[8] \)  
MIPS code:  
\( \text{lw} \) $t0, 32($s3)  
# temp. reg $t0 gets A[8]  
add $s1, $s2, $t0  
# put sum in reg corresponding to g.

Hardware-Software Interface

- Bytes are nice, but most data items use larger "words"  
- For MIPS, a word is 32 bits or 4 bytes.  
- In MIPS, words must start at addresses that are multiples of 4.
Instructions

- Compiling using Load and Store

Example:
Assume variable h is associated with register $s2 and the base address of the array is in $s3. What is the MIPS code?

C code:
\[ A[12] = h + A[8]; \]

MIPS code:
\[
\begin{align*}
\text{lw } &\$t0, 32(\$s3) & \text{ # Reg } \$t0 \text{ gets } A[8] \\
\text{add } &\$t0, \$s2, \$t0 & \text{ # Reg } \$t0 \text{ gets } h+A[8] \\
\text{sw } &\$t0, 48(\$s3) & \text{ # stores back into } A[12]
\end{align*}
\]

- Store word has destination last
- Can we write: \text{add 48} ($\$s3$) , \$s2, 32 ($\$s3$)?
- Remember arithmetic operands are registers, not memory!
  Can't write: \text{add 48} ($\$s3$) , \$s2, 32 ($\$s3$)\

General format of a line of any assembly language

\[
[label] \text{ opcode } [arg1 [, arg2 [, arg3]]] \#comment
\]

- \[label\] – optional, target of a jump (go to), usually at the beginning of a block of code
- \text{opcode} – operation to be executed, specific to ISA
- \text{args} – dependent on opcode, usually a fixed number of args for a given opcode
- \#comment – optional, everything from “#” to the end of the line is ignored by the assembler (important to the reader!)

General Format of MIPS instruction word

- All MIPS instructions are 32-bit words
- Instructions are broken into fields
  - op – basic operation, historically, the “opcode”
  - rs: - first register source operand
  - rt: - second register source operand
  - rd: - first register destination operand
  - shamt: - shift amount (zero for non-shift instructions)
  - funct: - variant of the operation, the “function” code

So far we’ve learned:

- MIPS
  - loading/storing words
    - 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 $s1, 48($s2)</td>
<td>$s1 = Memory[$s2 + 12]</td>
</tr>
<tr>
<td>sw $s1, 12($s2)</td>
<td>Memory[$s2 + 3] = $s1</td>
</tr>
</tbody>
</table>

(The base address of the array is in \$s2)
Design Principle 3

- Make the common case fast:

\[
\begin{align*}
\text{l} & \text{w } \$t0, \text{Add}r\text{constant}4(\$s1) & \# & \$t0 = \text{constant } 4 \\
\text{add } & \$s3, \$s3, \$t0 & \# & \$s3 = \$s3 + \$t0 \\
\text{addi } & \$s3, \$s3, 4 & \# & \$s3 = \$s3 + 4 \\
\text{addi} & = \text{add immediate}
\end{align*}
\]

General Format of MIPS instruction word

- All MIPS instructions are 32 bit words
- Instructions are broken into fields
  - op – basic operation, historically, the "opcode"
  - rs: - first register source operand
  - rt: - second register source operand
  - rd: - first register destination operand
  - shamt: - shift amount (zero for non-shift instructions)
  - funct: - variant of the operation, the "function" code

<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>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
</tr>
<tr>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Can this format be used for 'lw' instruction?
Think of the 'shamt' field...

Design Principle 4

- Good design demands good compromises

I-format or I-type (for immediate)

\[
\text{l} \text{w } \$t0, 1200(\$t1) & \# & \$t1 = \text{A}[300]
\]

Next Class...

Representing Instructions and Operations