Registers vs. Memory

- Arithmetic instructions operands must be registers, — only 32 registers provided
- Each register: 32 bits = 4 bytes = 1 word
- 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.
Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual **bytes** in memory
  
  - **MIPS**: memory address of a **word** must be multiple of 4 (**alignment restriction**)

- **Big Endian**: leftmost byte is word address
  
  IBM 360/370, Motorola 68k, **MIPS**, Sparc, HP PA

- **Little Endian**: rightmost byte is word address
  
  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)
Instructions: a simple example

• A C statement

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

• \( f, g, h, i, j \) are assigned to \( s_0, s_1, s_2, s_3, s_4 \)

\[
\begin{align*}
\text{add } & t_0, s_1, s_2 \\
\text{add } & t_1, s_3, s_4 \\
\text{sub } & s_0, t_0, t_1
\end{align*}
\]
Load and store instructions

- Load and store instructions

- lw $tn, c_off($S_base)
- sw $tn, c_off($S_base)

- Example:

  C code:  \[ g = h + A[8]; \]
  MIPS code:  \[
  \begin{align*}
    \text{lw } & \text{ } t0, 32(s3) \\
    \text{add } & \text{ } s1, s2, t0
  \end{align*}
  \]

- Spilling registers

  - *doubly* slow
Load and store instructions

- Example:


  MIPS code:  
  \begin{align*} 
  & \text{lw} \ $t0, 32($s3) \\
  & \text{add} \ $t0, $s2, $t0 \\
  & \text{sw} \ $t0, 48($s3) \\
  \end{align*}

- Store word has destination last

- Remember: arithmetic operands are registers, not memory!

  Can’t write: \[ \text{add} \ 48($s3), $s2, 32($s3) \]
So far we’ve learned:

• **MIPS**
  – loading words but addressing bytes
  – arithmetic on registers only

• **Instruction** | **Meaning**

  add $s1, $s2, $s3 | $s1 = $s2 + $s3
  sub $s1, $s2, $s3 | $s1 = $s2 – $s3
  lw $s1, 100($s2)  | $s1 = Memory[$s2+100]
  sw $s1, 100($s2)  | Memory[$s2+100] = $s1
• To use a constant, have to use memory, just like for variables

• Example: add 4 to register $s3

```
lw $t0, AddrConstant4($s1)   # t0 is the constant 4

add $s3, $s3, $t0
```

• Quick add instruction: `addi`

```
addi  $s3, $s3, 4
```

• Design principle: make the common case fast
Representing Instructions in the Computer Machine Language

• Instructions, like registers and words of data, are also 32 bits long
  – Example: $add \; t0, \; s1, \; s2$
  – registers *must* have numbers (why?) $t0=8, \; s1=17, \; s2=18$

• Instruction Format:

  \[
  \begin{array}{cccccc}
  0 & 17 & 18 & 8 & 0 & 32 \\
  000000 & 10001 & 10010 & 01000 & 00000 & 100000 \\
  \end{array}
  \]

  \begin{tabular}{c c c c c c}
  op & rs & rt & rd & shamt & funct \\
  \end{tabular}

  op  6-bits  opcode that specifies the operation
  rs  5-bits  register file address of the first source operand
  rt  5-bits  register file address of the second source operand
  rd  5-bits  register file address of the result’s destination
  shamt  5-bits  shift amount (for shift instructions)
  funct  6-bits  function code augmenting the opcode
## Aside: MIPS Register Convention

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserve on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0 (hardware)</td>
<td>n.a.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>returned values</td>
<td>no</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16-23</td>
<td>saved values</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return addr (hardware)</td>
<td>yes</td>
</tr>
</tbody>
</table>
Machine Language

- What if an instruction needs longer fields
  - e.g.: in `lw`, address of constant may be more than 32 ($2^5$)
  - conflict: keep instruction length same vs. have a single instruction format
  - New principle: *Good design demands a compromise*
  - Here: different formats for different instructions (keep length same)

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

- Example: `lw $t0, 32($s3)`

<table>
<thead>
<tr>
<th>35</th>
<th>19</th>
<th>9</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>16 bit number</td>
</tr>
</tbody>
</table>
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