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: \[ \text{add } \$s0, \$s1, \$s2 \]

(associated with variables by compiler)
MIPS arithmetic

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

\[
\begin{align*}
\text{C code:} & \quad A &= B + C + D; \\
& \quad E &= F - A;
\end{align*}
\]

\[
\begin{align*}
\text{MIPS code:} & \quad \text{add } $t0, $s1, $s2 \\
& \quad \text{add } $s0, $t0, $s3 \\
& \quad \text{sub } $s4, $s5, $s0
\end{align*}
\]

- 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 of Processor components: Control, Datapath, Memory, Input, Output]
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 addressing diagram]

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

| 0  | 32 bits of data |
| 4  | 32 bits of data |
| 8  | 32 bits of data |
| 12 | 32 bits of data |

Registers hold 32 bits of data

- $2^{32}$ bytes with byte addresses from 0 to $2^{32}-1$
- $2^{20}$ 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?
**Instructions**

- Load and store instructions
- Example:
  
  
  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!

---

**Policy of Use Conventions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>
Our First Example

- Can we figure out the code?

```c
swap(int v[], int k);
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```mips
sw $15, 4($2)
sw $16, 4($2)
jr $31
```

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 $s1, 100($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: add $t0, $s1, $s2
  - registers have numbers, $t0=9, $s1=17, $s2=18

- Instruction Format:

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

  - op  rs  rt  rd  shamt  funct

- Can you guess what the field names stand for?

Machine Language

- 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
  - l-type for data transfer instructions
  - other format was R-type for register

- Example: lw $t0, 32($s2)

```
35 18 9 32
```

  - op  rs  rt  16 bit number

- Where’s the compromise?
• 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