Instructions

- Load and store instructions
- Example:
  
  **C code:**
  
  ```
  ```
  
  **MIPS code:**
  
  ```
  lw $t0, 32($s3)
  add $t0, $s2, $t0
  sw $t0, 48($s3)
  ```

- Can refer to registers by name (e.g., \$s2, \$t2) instead of number
- Store word has destination last
- Remember arithmetic operands are registers, not memory!
  
  Can’t write: `add 48($s3), \$s2, 32($s3)`

---

Our First Example

- Can we figure out the code?

  ```
  swap(int v[]), int k):
  |
  |
  |
  ```

  ```
  int temp;
  v[k] = temp;
  ```

  ```
  swap:
  ```

- In C:

  ```
  swap(int v[], int k):
  ```

  ```
  |
  ```

- In MIPS:

  ```
  muli $2, $5, 4
  add $2, $4, $2
  lw $15, 0($2)
  lw $16, 4($2)
  sw $16, 0($2)
  sw $15, 4($2)
  ```

  ```
  jr $31
  ```

---

So far we’ve learned:

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

- Instruction

<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 \$t1, \$s1, \$s2`
  - registers have numbers, \$t1=9, \$s1=17, \$s2=18

- Instruction Format:

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0001</td>
<td>10010</td>
<td>010000</td>
<td>000000</td>
</tr>
</tbody>
</table>

- Can you guess what the field names stand for?
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)

where's the compromise?

Instructions are bits
Programs are stored in memory
- to be read or written just like data

Memory

- 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

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

MIPS conditional branch instructions:
- bne $t0, $t1, Label
- beq $t0, $t1, Label

Example: if (i==j) h = i + j;
- bne $s0, $s1, Label
- add $s3, $s4, $s5

MIPS unconditional branch instructions:
- j label

Example:
- if (i!=j)
- beq $s4, $s5, Lab1
- h=i+j;
- add $s3, $s4, $s5
- else
- j Lab2
- h=i-j;
- Lab1: sub $s3, $s4, $s5
- Lab2: ...

Can you build a simple for loop?
So far:

- 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
  - bne $s4,$s5,L         Next instr. is at label if $s4 ≠ $s5
  - beq $s4,$s5,L         Next instr. is at label if $s4 = $s5

- Formats:
  - op rs rt rd shamt funct
  - op rs rt 16 bit address
  - op 26 bit address

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:
  - if $s1 < $s2 then
    - $t0 = 1
  - else $t0 = 0

- Can use this instruction to build "blt $s1, $s2, Label"
  - can now build general control structures
- Note that the assembler needs a register to do this,
  - there are policy of use conventions for registers

Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:
  - if $s1 < $s2 then
    - $t0 = 1
  - else $t0 = 0

- Can use this instruction to build "blt $s1, $s2, Label"
  - can now build general control structures
- Note that the assembler needs a register to do this,
  - there are policy of use conventions for registers

Policy of Use Conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>No constant value 0</td>
</tr>
<tr>
<td>$at</td>
<td>2-3</td>
<td>Values for results and expression evaluation</td>
</tr>
<tr>
<td>$s1-$s3</td>
<td>4-7</td>
<td>Arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>16-23</td>
<td>Saved</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>

Register 1 ($at) reserved for assembler, 26-27 for operating system

Constants

- Small constants are used quite frequently (50% of operands)
  - e.g., A = A + 5;
  - B = B + 1;
  - C = C - 18;

- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.

- MIPS Instructions:
  - addi $29, $29, 4
  - slti $8, $18, 10
  - ori $29, $29, 6

- Design Principle: Make the common case fast. Which format?
How about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction
  \[
  \text{lui} \; \$t0, \; 1010101010101010 \quad \text{filled with zeros}
  \]
- Then must get the lower order bits right, i.e.,
  \[
  \text{ori} \; \$t0, \; \$t0, \; 1010101010101010
  \]

Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”
- When considering performance you should count real instructions

Other Issues

- Discussed in your assembly language programming lab:
  - support for procedures
  - linkers, loaders, memory layout
  - stacks, frames, recursion
  - manipulating strings and pointers
  - interrupts and exceptions
  - system calls and conventions
- Some of these we’ll talk more about later
- We’ll talk about compiler optimizations when we hit chapter 4.

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

<table>
<thead>
<tr>
<th>R</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>16 bit address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td>2k bit address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- rely on compiler to achieve performance
  - what are the compiler’s goals?
- help compiler where we can
Addresses in Branches and Jumps

- **Instructions:**
  - bne $t4,$t5,Label  Next instruction is at Label if $t4 \neq $t5
  - beq $t4,$t5,Label  Next instruction is at Label if $t4 = $t5
  - j Label       Next instruction is at Label

- **Formats:**
  
  1 | op | rs | st | 16 bit address |
  2 | op |    |    | 16 bit address |

- **Addresses are not 32 bits**
  — How do we handle this with load and store instructions?