Chapter 2

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)

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

C code:

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

MIPS code:

`add $t0, $s1, $s2`
`add $s0, $t0, $s3`
`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

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.
Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.
- \(2^{32}\) 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?

Instructions

- Load and store instructions
- Example:


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

- Store word has destination last
- Remember arithmetic operands are registers, not memory!
Logical Operations

• Need instructions for operating on the individual bits of a word
  – For example, since ASCII characters are stored in 8 bits (one byte), we need operations that allow us to examine one byte within a word
• Also, high-level languages such as C and Java support logical operations such as AND, OR, and SHIFT
• MIPS supports bit-by-bit AND, OR, and NOT operations

  AND          and
  OR           or
  NOT          nor

• These instructions are similar to arithmetic instructions in their format and usage
  – For Example,

    and $t1,$t2,$t3 # reg $t1 = reg $t2 & reg $t3
    or $t1,$t2, $t3 # reg $t1 = reg $t2 | reg $t3

Shift instructions

• Shift operations move all the bits in a word to the right or left, filling the emptied bits with 0s
• For example, if register $s0 contained

  0000 0000 0000 0000 0000 0000 0000 1101 (= 9_{10})

  then shifting it left by 4 bits would give us

  0000 0000 0000 0000 0000 1101 0000 (= 144_{10})

• This is accomplished by the instruction sll (shift left logical)

    sll $t1,$s0,4  # reg $t1 = reg $s0 << 4

• Similarly srl (shift right logical) is used to shift a word to the right
• sll and srl can be used to multiply and divide an integer by 2 respectively
So far we’ve learned:

• **MIPS**
  — loading words but addressing bytes
  — arithmetic and logical operations 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>
<tr>
<td>and $t1,$t2,$t3</td>
<td>$t1 = $t2 &amp; $t3</td>
</tr>
<tr>
<td>or $t1,$t2, $t3</td>
<td>$t1 = $t2</td>
</tr>
<tr>
<td>sll $t1,$t2,4</td>
<td>$t1 = $t2 &lt;&lt; 4 bits</td>
</tr>
<tr>
<td>srl $t1,$t2,4</td>
<td>$t1 = $t2 &gt;&gt; 4 bits</td>
</tr>
</tbody>
</table>

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

Stored Program Concept

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

- Processor
- Memory

memory for data, programs, compilers, editors, etc.

- 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
Control

• 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, $s0, $s1
  Label: ....
  ```

• MIPS unconditional branch instructions:
  
  ```
  j label
  ```

• Example:
  
  ```
  if (i!=j)
      h=i+j;
  else
      h=i-j;
  ```

  ```
  beq $s4, $s5, Lab1
  add $s3, $s4, $s5
  j Lab2
  ```

  ```
  Lab1: sub $s3, $s4, $s5
  Lab2: ...
  ```

• Can you build a simple for loop?
Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:
  
  ```
  if $s1 < $s2 then
    $t0 = 1
  slt $t0, $s1, $s2
  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>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$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>
**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  
  andi $29, $29, 6  
  ori $29, $29, 4  
  ```