7 September 2004

- Questions?
- Farmer’s Days?
- Programming the Machine

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 1980s

Design goals: minimize 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...

C code: \[ A = B + C + D; \]
\[ E = F - A; \]
MIPS code: \[ \text{add } \$t0, \$s1, \$s2 \]
\[ \text{add } \$t0, \$t0, \$s3 \]
\[ \text{sub } \$s4, \$s5, \$t0 \]

- 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.
- A memory address is an index into the array.
- "Byte addressing" means that the index points to a byte of memory.

char memory[256*1024*1024]:

0 1 2 3 4 5 6 ...
Memory Organization

- Bytes are nice, but most data items use larger “words”
- For MIPS, a word is 32 bits or 4 bytes.
- Registers hold 32 bits of data
- 2^32 bytes with byte addresses from 0 to 2^32 - 1
- 2^30 words with byte addresses 0, 4, 8, ... 2^30 - 4
- Words are aligned
  i.e., what are the least 2 significant bits of a word address?

Endians?

- What order are the bytes inside the word?
- Is byte 0 the high-order bits of word 0?
- Or is it the low order bits?
- “Big Endian” byte 0 is high-order bits [0 1 2 3]  
  — Macintosh, SPARC
- “Little Endian” byte 0 is low-order bits [3 2 1 0]  
  — Intel, DECStation
When would I care?

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!

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 = $s1 + $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=4, $s1=17, $s2=18
- Instruction Format:
  ```
  000000 00001 00100 00000 000000 000000 000000 000000 000000
  op rs rt rd rd rd rd rd rd
  ```
- Where’s the compromise?
## 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

## Execution Example

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>Memory (32 bits)</th>
<th>Memory (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>204</td>
<td></td>
<td></td>
</tr>
<tr>
<td>208</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Execution Example: Fetch(200)

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>204</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Execution Example: Execute(200)

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>204</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Execution Example: Fetch(204)

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>204</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## MIPS

- Loading words but addressing bytes
- Arithmetic on registers only

- Instruction
  - `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

---

So far we’ve learned:

- Instructions
- 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
### Execution Example: Execute(204)

**Program Counter**: 204

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>204</td>
<td>lw $9, 0($8)</td>
<td>112</td>
</tr>
<tr>
<td>205</td>
<td>add $9,$9,$7</td>
<td>116</td>
</tr>
<tr>
<td>205</td>
<td>sw $9,$8($4)</td>
<td>116</td>
</tr>
</tbody>
</table>

**Registers**

- R0: 0
- R1: 7
- R2: 6
- R3: 3
- R4: 120
- R5: 212
- R6: 316

**Instruction Register**

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>R</td>
<td>lw $9, 0($8)</td>
<td>lw $9, 0($8)</td>
</tr>
<tr>
<td>100000</td>
<td>R</td>
<td>add $9,$9,$7</td>
<td>add $9,$9,$7</td>
</tr>
<tr>
<td>100000</td>
<td>R</td>
<td>sw $9,$8($4)</td>
<td>sw $9,$8($4)</td>
</tr>
</tbody>
</table>

### Execution Example: Fetch(208)

**Program Counter**: 208

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>208</td>
<td>lw $9, 0($8)</td>
<td>112</td>
</tr>
<tr>
<td>209</td>
<td>add $9,$9,$7</td>
<td>116</td>
</tr>
<tr>
<td>209</td>
<td>sw $9,$8($4)</td>
<td>116</td>
</tr>
</tbody>
</table>

**Registers**

- R0: 0
- R1: 7
- R2: 6
- R3: 3
- R4: 120
- R5: 212
- R6: 316

**Instruction Register**

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>R</td>
<td>lw $9, 0($8)</td>
<td>lw $9, 0($8)</td>
</tr>
<tr>
<td>100000</td>
<td>R</td>
<td>add $9,$9,$7</td>
<td>add $9,$9,$7</td>
</tr>
<tr>
<td>100000</td>
<td>R</td>
<td>sw $9,$8($4)</td>
<td>sw $9,$8($4)</td>
</tr>
</tbody>
</table>

### Control

- Decision making instructions
  - alter the control flow
  - i.e., change the "next" instruction to be executed
- MIPS conditional branch instructions:
  - beq $t0, $t1, Label
  - bne $t0, $t1, Label

  **Example:**
  ```
  if (i == j) 
  h = i + j;
  bne $s0, $s1, Label
  add $s3, $s0, $s1
  Label: . . . .
  ```

### MIPS unconditional branch instructions:

- Example:
  ```
  if (i==j)
  
  x=x+1;
  
  else
  
  x=x-1;
  
  ```

### So far:

- Instruction: Meaning
  - add $s1,$s2,$s3 $s1 = $s2 + $s3
  - lw $s1,$s2,$s3 $s1 = Memory[$s2+$s3]
  - beq $s4,$s5,L Label $s4 = $s5 $s4 = $s5 $s4 = $s5 $s4 = $s5
  - bne $s4,$s5,L Label $s4 != $s5 $s4 != $s5 $s4 != $s5 $s4 != $s5

- Formats:
  ```
  R op rs rt cd shamt funct
  I op rs rt 16 bit address
  J op 26 bit address
  ```
Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:
  
  ```
  if $s1 < $s2 then
      s1t $t0, $s1, $s2
      $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>the constant value 0</td>
</tr>
<tr>
<td>at</td>
<td>1</td>
<td>reserved for the assembler</td>
</tr>
<tr>
<td>v0 - v1</td>
<td>2 - 3</td>
<td>values for result and expression evaluation</td>
</tr>
<tr>
<td>a0 - a3</td>
<td>4 - 7</td>
<td>arguments</td>
</tr>
<tr>
<td>t0 - t1</td>
<td>8 - 15</td>
<td>temporaries</td>
</tr>
<tr>
<td>s0 - s7</td>
<td>16 - 23</td>
<td>saved</td>
</tr>
<tr>
<td>t2 - t3</td>
<td>24 - 25</td>
<td>frame pointers</td>
</tr>
<tr>
<td>gp</td>
<td>28</td>
<td>stack pointer</td>
</tr>
<tr>
<td>sp</td>
<td>29</td>
<td>frame pointer</td>
</tr>
<tr>
<td>rp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>rip</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>