CSE 141 – Computer Architecture
Fall 2003

Lecture 3

Pramod V. Argade

CSE141: Introduction to Computer Architecture

Web-page: http://www-cse.ucsd.edu/classes/fa03/cse141

Homework: 2.1 through 2.5, 2.15 through 2.17.
3.2, 3.3, 3.6, 3.7, 3.9, 3.10, 3.14, 3.17.
Due Tue. Oct. 7th in class

Quiz: When: Thursday, Oct. 9, 2003, First 10 minutes of the class
Topic: ISA, Chapter 3
Need: Paper, pen, calculator

Next class: Arithmetic for Computers, Chapter 4

CSE 141L: First lecture Friday, October 3, 2003, 8 AM, Center 119

Lab Assignment will be posted on the web page Thursday
Bring a printout to 141L class Friday
## Schedule

<table>
<thead>
<tr>
<th>Lecture #</th>
<th>Date</th>
<th>Day</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sep. 25</td>
<td>Thursday</td>
<td>Introduction, Ch. 1</td>
</tr>
<tr>
<td>2</td>
<td>Sep. 30</td>
<td>Tuesday</td>
<td>Performance, Ch. 2</td>
</tr>
<tr>
<td>3</td>
<td>Oct. 2</td>
<td>Thursday</td>
<td>ISA, Ch. 3</td>
</tr>
<tr>
<td>4</td>
<td>Oct. 7</td>
<td>Tuesday</td>
<td>Arithmec, Ch. 4</td>
</tr>
<tr>
<td>5</td>
<td>Oct. 9</td>
<td>Thursday</td>
<td>Single cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>6</td>
<td>Oct. 14</td>
<td>Tuesday</td>
<td>Single cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>7</td>
<td>Oct. 16</td>
<td>Thursday</td>
<td>Multi-cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>8</td>
<td>Oct. 21</td>
<td>Tuesday</td>
<td>Multi-cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>9</td>
<td>Oct. 23</td>
<td>Thursday</td>
<td>Exceptions and Review for Midterm</td>
</tr>
<tr>
<td>10</td>
<td>Oct. 28</td>
<td>Tuesday</td>
<td>Mid-term Exam</td>
</tr>
<tr>
<td>11</td>
<td>Oct. 30</td>
<td>Thursday</td>
<td>Pipelining, Ch. 6</td>
</tr>
<tr>
<td>12</td>
<td>Nov. 4</td>
<td>Tuesday</td>
<td>Data and control hazards, Ch. 6</td>
</tr>
<tr>
<td>13</td>
<td>Nov. 6</td>
<td>Thursday</td>
<td>Data and control hazards, Ch. 6</td>
</tr>
<tr>
<td>No Class</td>
<td>Nov. 11</td>
<td>Tuesday</td>
<td>Veteran's Day Holiday</td>
</tr>
<tr>
<td>14</td>
<td>Nov. 13</td>
<td>Thursday</td>
<td>Data and control hazards, Ch. 6</td>
</tr>
<tr>
<td>15</td>
<td>Nov. 18</td>
<td>Tuesday</td>
<td>Advanced pipelining issues, Ch. 6</td>
</tr>
<tr>
<td>16</td>
<td>Nov. 20</td>
<td>Thursday</td>
<td>Memory &amp; cache design, Ch. 7</td>
</tr>
<tr>
<td>17</td>
<td>Nov. 25</td>
<td>Tuesday</td>
<td>Memory &amp; cache design, Ch. 7</td>
</tr>
<tr>
<td>No Class</td>
<td>Nov. 27</td>
<td>Thursday</td>
<td>Thanksgiving Holiday</td>
</tr>
<tr>
<td>18</td>
<td>Dec. 2</td>
<td>Tuesday</td>
<td>Memory &amp; cache design, Ch. 7</td>
</tr>
<tr>
<td>19</td>
<td>Dec. 4</td>
<td>Thursday</td>
<td>Review, Instruction ends</td>
</tr>
<tr>
<td></td>
<td>Dec. 11</td>
<td>Thursday</td>
<td>Final Exam</td>
</tr>
</tbody>
</table>

### Instruction Set Architecture (ISA)

#### General Considerations
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

Stored Program Concept

The Instruction Execution Cycle

- **Instruction Fetch**: Obtain instruction from program storage
- **Instruction Decode**: Determine required actions and instruction size
- **Operand Fetch**: Locate and obtain operand data
- **Execute**: Compute result value or status
- **Result Store**: Deposit results in storage for later use
- **Next Instruction**: Determine successor instruction
Key ISA decisions

- Operations
  - how many?
  - which ones
- Operands
  - how many?
  - location
  - types
  - how to specify?
- Instruction format
  - size
  - how many formats?

\[
y = x + b
\]

Basic ISA Classes
Comparing the Number of Instructions

Code sequence for \( C = A + B \) for four classes of instruction sets:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-memory)</th>
<th>Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1,A</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1,B</td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store C, R1</td>
<td>Add R3, R1, R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td>Store C, R3</td>
</tr>
</tbody>
</table>
Accessing the Operands

- Operands are generally in one of two places:
  - Registers (32 int, 32 fp)
  - Memory (2^{32} locations)
- Registers are
  - Easy to specify
  - Close to the processor
  - Provide fast access
  - Can read two operands and write one result per clock cycle
- The idea that we want to access registers whenever possible led to load-store architectures.
  - Normal arithmetic instructions only access registers
  - Only access memory with explicit loads and stores

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?

Addressing: Endian-ness and alignment

- **Big Endian:** address of most significant byte = word address
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- **Little Endian:** address of least significant byte = word address
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

Alignment: require that objects fall on address that is multiple of their size.
MIPS requires address alignment
MIPS Instruction Set Architecture

Key Points

- MIPS is a general-purpose register, load-store, fixed-instruction-length architecture.
- MIPS is optimized for fast pipelined performance, not for low instruction count
- Four principles of IS architecture
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
Overview of MIPS ISA

- 3-operand, load-store architecture
- 32 general-purpose registers (integer, floating point)
  - R0 always equals 0.
- 2 special-purpose integer registers, HI and LO, because multiply and divide produce more than 32 bits.
- Registers are 32-bits wide (word)
- Register, immediate, base+displacement, PC-relative and pseudo-direct addressing modes
- Fixed 32-bit instructions
- 3 instruction formats

Is this sufficient?

- Measurements on the VAX show that these addressing modes (immediate, direct, register indirect, and base+displacement) represent 88% of all addressing mode usage.

- Similar measurements show that 16 bits is enough for the immediate 75 to 80% of the time

- Measurements also show that 16 bits is enough of a displacement 99% of the time.
MIPS Addressing Modes

1. Immediate addressing
   \[ op\ \text{rs}\ \text{rt}\ \text{Immediate} \]

2. Register addressing
   \[ op\ \text{rs}\ \text{rt}\ \text{funct} \]

3. Base addressing
   \[ op\ \text{rs}\ \text{rt}\ \text{Address} \]

4. PC-relative addressing
   \[ op\ \text{rs}\ \text{rt}\ \text{Address} \]

5. Pseudodirect addressing
   \[ op\ \text{Address}\ \text{PC} \]

Instructions

- Arithmetic
  - add, sub, addi

- Logical
  - (not covered here)

- Data transfer
  - lw, sw, lb, sb, lui

- Conditional branch
  - beq, bne, slt, slti

- Unconditional jump
  - j, jr, jal
MIPS Arithmetic Instructions

- Design Principle: simplicity favors regularity. Why?
- 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)

- Operands must be registers, only 32 registers provided
- Design Principle: smaller is faster. Why?

How to specify 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
  ```
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

```
  lui $t0, 1010101010101010
```

```
  fill with zeros
```

```
  1010101010101010 0000000000000000
```

- Then must get the lower order bits right, i.e.,

```
  ori $t0, $t0, 1010101010101010
```

```
  1010101010101010 0000000000000000
```

```
  0000000000000000 1010101010101010
```

```
  ori
```

```
  1010101010101010 1010101010101010
```

Memory Access 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!
Control Transfer Instructions

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

- MIPS conditional branch instructions:
  
  $\text{bne } \$t0, \$t1, \text{Label}$
  $\text{beq } \$t0, \$t1, \text{Label}$

- Example: $\text{if (i==j) h = i + j;}$
  
  $\text{bne } \$s0, \$s1, \text{Label}$
  $\text{add } \$s3, \$s0, \$s1$
  $\text{Label: } \ldots$

Unconditional Jumps

- MIPS unconditional branch instructions:
  
  $\text{j label}$

- Example:
  
  $\text{if (i!=j) }$  $\text{beq } \$s4, \$s5, \text{Lab1}$
  $\text{h=i+j;}$  $\text{add } \$s3, \$s4, \$s5$
  $\text{else}$  $\text{j Lab2}$
  $\text{h=i-j;}$  $\text{Lab1:sub } \$s3, \$s4, \$s5$
  $\text{Lab2: } \ldots$
So far:

- **Instruction**  
  **Meaning**
  
  - `add $s1,$s2,$s3`  
    #$s1 = $s2 + $s3$
  
  - `sub $s1,$s2,$s3`  
    #$s1 = $s2 - $s3$
  
  - `lw $s1,100($s2)`  
    #$s1 = \text{Memory}\[s2+100\]$
  
  - `sw $s1,100($s2)`  
    #Memory[$s2+100] = $s1
  
  - `bne $s4,$s5,LNext`  
    #instr. is at Label if  
    #$s4 \neq $s5$
  
  - `beq $s4,$s5,LNext`  
    #instr. is at Label if  
    #$s4 = $s5$
  
  - `j Label`  
    #instr. is at Label

---

**Policy of Use Conventions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$szero$</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>$s8-s9$</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>
Summary of MIPS Instructions

### MIPS Operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s0-$s7, $t0-$t9, $zero, $at</td>
<td></td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>$fp, $sp, $ra</td>
<td></td>
<td>MIPS register at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>Memory[0], Memory[4]</td>
<td></td>
<td>Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

### MIPS Assembly Language

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>addi $s1, $s2, 100</td>
<td>$s1 = $s2 - 100</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>lui $s1, 100</td>
<td>$s1 = 100 * 2</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>lb $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>sb $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Byte from register to memory</td>
</tr>
<tr>
<td></td>
<td>branch on equal</td>
<td>beq $s1, $s2, 25</td>
<td>if ($s1 == $s2) go to $PC + 4 + 100</td>
<td>Equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1, $s2, 25</td>
<td>if ($s1 != $s2) go to $PC + 4 + 100</td>
<td>Not equal test; PC-relative branch</td>
</tr>
<tr>
<td>Conditional</td>
<td>set less than</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than; for beq, bne</td>
</tr>
<tr>
<td>branch</td>
<td>jump</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td>Unconditional</td>
<td>jump register</td>
<td>jr $ra</td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500 $ra</td>
<td>$ra = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>

### MIPS Machine Code

- The opcode tells the machine which format
- So `add r1, r2, r3` has
  - opcode=0, funct=32, rs=2, rt=3, rd=1, sa=0
  - Machine code:
    000000 00001 00011 00001 00000 100000
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

- Things we are not going to cover
  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 about later
- We've focused on architectural issues
  - basics of MIPS assembly language and machine code
  - we'll build a processor to execute these instructions.
Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI

- Sometimes referred to as “RISC vs. CISC”
  - Virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy
    instructions from 1 to 54 bytes long!

- We’ll look at PowerPC and 80x86

80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions
  (mostly designed for higher performance)
- 1997: MMX is added

“This history illustrates the impact of the “golden handcuffs” of compatibility”
“an architecture that is difficult to explain and impossible to love”
A dominant architecture: 80x86

- See your textbook for a more detailed description
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    - e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

“what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective”
Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction indeed!