Announcements

- **Reading Assignment:**
  - Chapter 2: Instructions: Language of the Computer
    - Sections 2.1 - 2.6, 2.9 - 2.13

- **Homework: Due Mon. 10/3/05**
  1. Write a C or Java program to determine endianness of a computer and test it on a Windows PC and a SPARC Station (Sun Solaris Machine). What is the endianness of the two computers?
  2. A 32 bit value 0xcafef00d was written to address 0x13fde9c on a processor with little-endian byte ordering. What byte value will be read at address 0x13fde9e?
  3. Problems from the book: 
     - 2.2, 2.4, 2.5, 2.6, 2.30, 2.32, 2.47

- **Quiz: Mon. 10/3/05**
  - Topic: Instructions: Language of the Computer (Chapter 2)
**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

**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.

<table>
<thead>
<tr>
<th>0</th>
<th>32 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>8</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>12</td>
<td>32 bits of data</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

- Words are aligned on 4-byte boundary for MIPS architecture
  i.e., what are the least 2 significant bits of a word address?

- 32 bits address
  - $2^{32}$ bytes with byte addresses from 0 to $2^{32}-1$
  - $2^{30}$ words with byte addresses 0, 4, 8, ... $2^{32}-4$

Endianness: How to address bytes within words?

- **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)

```
0x00112233
0x44556677
0x8899aabb
0xccddeeff
```

```text
0 1 2 3 4 5 6
0 1 2 3
<table>
<thead>
<tr>
<th>msb</th>
<th>lsb</th>
<th>Word Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
```
Additional Homework Problem

1. Write a C or Java program to determine endianness of a computer and test it on a Windows PC and a SPARC Station (Sun Solaris Machine). What is the endianness of the two computers?

   • **Hint:**

2. A 32 bit value 0xc0cafe00d was written to address 0x12fde9c on a processor with little-endian byte ordering. What value will be read at byte address 0x13fde9e?

---

**Addressing: Alignment**

- **Alignment:** require that objects fall on address that is multiple of their size.

- **MIPS requires address alignment**
  - Word addresses must be multiple of 4
  - Half word addresses must be multiple of 2
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

The Instruction Set Architecture (ISA)

- ISA is the interface between all the software that runs on the machine and the hardware that executes it.

  - Application
  - Operating System
  - Compiler
  - Micro-code
  - I/O system
  - Digital Design
  - Circuit Design

- Provides a "level of abstraction" in both directions
- Modern instruction set architectures:
  - 80x86/Pentium, MIPS, SPARC, PowerPC, ARM, Tensilica, ...
Instruction Set Architecture (ISA)

- Instructions: Words of a machine’s language
- Instruction Set: Machine’s vocabulary
- ISA: A very important abstraction
  - Interface between hardware and low-level software
  - Standardizes instructions, machine language bit patterns, etc.
  - Advantage: different implementations of the same architecture
  - Disadvantage: sometimes prevents using new innovations

Part of the architecture that is visible to the programmer
- opcodes (available instructions)
- number and types of registers
- instruction formats
- storage access, addressing modes
- exceptional conditions

Key ISA Decisions

- Instruction length
  - Fixed length
  - Variable length
- Registers
  - How many?
- Operand access
  - Register
  - Memory
- Instruction format
  - Meaning of group of bits within machine instruction
- Operands
  - How many per instruction, size (byte, word,..)
- Operations
  - ADD, SUB, MUL, ...
Accessing the Operands

- Operands are generally in one of two places:
  - Registers: fast on-chip storage (how many, how wide?)
  - Memory (how many locations?)

- Registers:
  - are easy to specify
  - are 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

Basic ISA Classes

Comparing the Number of Instructions

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

<table>
<thead>
<tr>
<th></th>
<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>
<td>Store C, R3</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1,B</td>
<td>Load R2,B</td>
<td>Add R3, R1, R2</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store C, R1</td>
<td>Store C, R3</td>
<td></td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MIPS Instruction Set Architecture

- Typical “RISC” Instruction Set designed in 1980’s
- MIPS is found in products from:
  - Silicon Graphics
  - NEC
  - Cisco
  - Broadcom
  - Nintendo
  - Sony
  - Ti
  - Toshiba
- We will study various implementations of MIPS instruction set
  - Later part of the course!
MIPS ISA: 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 ISA
  - Simplicity favors regularity:
  - Smaller is faster:
  - Good design demands good compromises:
  - Make the common case fast:

Overview of MIPS ISA

- Fixed 32-bit instructions
- 3-operand, load-store architecture
- 32 general-purpose registers
  - Registers are 32-bits wide (word)
  - R0 always equals zero.
- Addressing modes
  - Register, immediate, base+displacement, PC-relative and pseudo-direct addressing modes
- 3 instruction formats will be covered in the class
Notes on MIPS Assembly

- Comments start with “#”
- Destination first (except for “store” instructions)
  - ADD $t0, $s1, $s2       # $t0 = $s1 + $s2
- Register access $n: register n (e.g. $1 is register 1)
- Register naming convention

<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>

MIPS Instructions (Covered in this course)

- Arithmetic
  - add, sub, addi
- Logical
  - sll, srl, and, andi, or, ori, nor
- Data transfer
  - lw, sw, lb, sb, lui
- Conditional branch
  - beq, bne, slt, slti
- Unconditional jump
  - j, jr
- Jump and link
  - jal
MIPS Arithmetic Instructions

- 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)

- Operands must be registers, only 32 registers provided
- Instruction format:

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>

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

Solutions
- Put 'typical constants' in memory and load them.
- Create hard-wired registers (like $zero) for constants like “1”.
- Specify constant in the instruction: this commonly used

MIPS Instructions:
\[
\text{addi } \$s3, \$t0, 4 \\
\text{slti } \$s0, \$t1, 10
\]

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
  - Put 'typical constants' in memory and load them.
  - Create hard-wired registers (like $zero) for constants like “1”.
  - Specify constant in the instruction: this commonly used

MIPS Instructions:
\[
\text{addi } \$s3, \$t0, 4 \\
\text{slti } \$s0, \$t1, 10
\]

Instruction Format

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>16 bit address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.,
  
  \[
  \text{ori } \$t0, \ \$t0, \ 1010101010101010
  \]

Memory Access Instructions

- Load and store instructions
- Example:
  
  \[
  \begin{align*}
  \text{C code:} & \quad A[8] = h + A[8]; \\
  \text{MIPS code:} & \quad \text{lw } \$t0, \ 32(\$s3) \\
  & \quad \text{add } \$t0, \ \$s2, \ \$t0 \\
  & \quad \text{sw } \$t0, \ 32(\$s3)
  \end{align*}
  \]

- Store word has destination last
- Remember arithmetic operands are registers, not memory!
- Instruction format:
  
  \[
  \begin{array}{c|c|c|c}
  \text{I} & 6 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} \\
  \text{op} & \text{rs} & \text{rt} & \text{16 bit address}
  \end{array}
  \]

Control Transfer Instructions

- 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: ....
- Instruction format:

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
</tr>
</tbody>
</table>

MIPS Conditional Branches

- MIPS branches use PC-relative addressing
- BEQ, BNE
  - BEQ $1, $2, addr => if( r1 == r2 ) goto addr
- MIPS has no Branch-If-Lless-Than
  - It is in timing critical path
- Set-Less-Than, SLT
  - SLT $1, $2, $3 => if( $2 < $3) $1 = 1; else 1 = 0
- BEQ, BNE, SLT combined with $0 can implement all branch conditions:
  - always, never, !=, ==, <=, <, >, >(unsigned), <=(unsigned), ...
MIPS Jump Instructions

- Need to transfer control
  - Jump to an absolute address
  - Jump to an address in a register
  - Jump-And-Link to do procedure call and return

- Jump example:
  - j 0x20000 => PC = 0x20000

- Jump register example:
  - jr $31 => PC = $31 (This is return instruction!)

- Jump and Link example
  - jal 0x40000 => $31 = PC+4, PC = 0x40000

---

Unconditional Jumps

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

- Instruction format:
  
  \[ \text{j} \quad \text{op} \quad \text{26 bit address} \]

- Jump uses pseudo-direct addressing mode

- Program Counter

  \[ \begin{array}{c}
  4 \quad 26 \\
  \end{array} \]

- Instruction

  \[ \begin{array}{c}
  6 \quad 26 \\
  \end{array} \]

- Jump Destination Address

  \[ \begin{array}{c}
  4 \quad 26 \quad 00 \\
  \end{array} \]
MIPS Instruction Format & Machine code

- Instruction format

<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>J</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>16 bit address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The opcode tells the machine which format
- Machine instruction `add r1, r2, r3` has
  - opcode=0, funct=32, rs=2, rt=3, rd=1, sa=0
  - Machine code:
    
    | 000000 | 00010 | 00011 | 00001 | 00000 | 100000 |
    | 0x00430820 |

- Expected to assemble and disassemble machine code

Summary of MIPS Instructions

<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>$s1, s2, s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>$s1, s2, s3</td>
<td>$s1 = $s2 - $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>$s1, s2, 100</td>
<td>$s1 = $s2 + 100</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>$s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Load word from memory to register</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>$s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Store word from register to memory</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>$s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Load byte from memory to register</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>$s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Store byte from register to memory</td>
</tr>
<tr>
<td></td>
<td>load upper immediate</td>
<td>$s1, 100</td>
<td>$s1 = 100 * 2^31</td>
<td>Load immediate in upper 16 bits</td>
</tr>
<tr>
<td></td>
<td>branch on equal</td>
<td>j $s1, 25</td>
<td>if ($s1 == 100) go to PC + 400</td>
<td>Branch on equal; branch on equal</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>j $s1, 25</td>
<td>if ($s1 != 100) go to PC + 400</td>
<td>Branch on not equal; branch on not equal</td>
</tr>
<tr>
<td></td>
<td>not less than immediate</td>
<td>j $s1, 25</td>
<td>if ($s1 &lt; 100) go to PC + 400</td>
<td>Not less than; branch on not less than immediate</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>jump</td>
<td>1, 2500</td>
<td>go to target address</td>
<td>Jump to target address</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>$s1, 2500</td>
<td>$s1 = PC + 400</td>
<td>Jump to target address</td>
</tr>
<tr>
<td></td>
<td>jump absolute</td>
<td>$s1, 2500</td>
<td>$s1 = PC + 4 + 100000</td>
<td>Jump to target address</td>
</tr>
</tbody>
</table>

Pramod Argade
CSE 141, Fall, 2005
1-29
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
  - Used by the processor at run time
- Assembly can provide 'pseudo-instructions'
  - 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
- 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!*
Intel IA-32 Architecture

- 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
- 1999 Pentium II (same architecture)
- 2000 Pentium 4 (144 new multimedia instructions)
- 2001 Itanium (new ISA which executes x86 code)

A dominant architecture: 80x86

- See Section 2-16 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
Summary

- Instruction complexity is only one variable
  - Lower instruction count vs. higher CPI / lower clock rate
- Four principles of ISA
  - Simplicity favors regularity:
  - Smaller is faster:
  - Good design demands good compromises:
  - Make the common case fast:
- Instruction set architecture
  - A very important abstraction indeed!
- In subsequent lectures we will show how to implement a subset of this ISA