Chapter 2
Instruction Set Principles and Examples

CSE4210
Fall 2006
Based on Slides by
Prof. L. Bhuyan (UCR)
Prof. M. Shaaban (RIT)
Computer Architecture

- In designing the ISA of a computer, we have to make many design choices about many aspects of the computer.
  - Memory access
  - RISC vs. CISC
  - Addressing modes
  - Operands and operations
  - Applications?
  - Control flow?

- We briefly discuss them

Instruction Set

- Stack
- Accumulator
- Register-Memory
  - 1 memory operand
- Memory-memory
  - 2 or 3 memory operands
- Load-store
# Instruction Set

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Reg-memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>load A</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R3,R1,B</td>
</tr>
<tr>
<td>Add C</td>
<td>Store C</td>
<td>Store R3,C</td>
</tr>
</tbody>
</table>

## Mem-mem

<table>
<thead>
<tr>
<th>Load-store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load R1,A</td>
</tr>
<tr>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add R3,R1,R2</td>
</tr>
<tr>
<td>Store C,R3</td>
</tr>
</tbody>
</table>

## Memory Addressing

- Memory is byte addressable and provides access to bytes (8), half words (16), words (32), and for many computers double words (64).
- **Big Endian**: address of the most significant byte is the word address (Motorola, MIPS, SPARC)
- **Little Endian**: address of the least significant byte is the word address (x86, alpha)
- Only important when exchanging data between computers.
Memory Addressing

- In many computers, access to objects larger than a byte is aligned.
- An access to object of size $s$ at byte address $A$ if $A \text{ mod } s = 0$.
- Since memory is typically aligned on words (or double words), misaligned access may take multiple aligned memory access.
- Even for aligned access, access to bytes and half words requires alignment network.

![Diagram](image)
### Addressing Modes

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4, R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4, #3</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4, 10 (R1)</td>
</tr>
<tr>
<td>Indirect</td>
<td>Add R4, (R1)</td>
</tr>
<tr>
<td>Indexed</td>
<td>Add R3, (R1 + R2)</td>
</tr>
<tr>
<td>Absolute</td>
<td>Add R1, (1001)</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1, @ (R3)</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>Add R1, (R2) +</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>Add R1, - (R2)</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1, 100 (R2) [R3]</td>
</tr>
</tbody>
</table>

### Memory Addressing

#### Frequency of addressing modes

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>GCC</th>
<th>Spice</th>
<th>TEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Displacement</td>
<td>50.00</td>
<td>40.00</td>
<td>30.00</td>
</tr>
<tr>
<td>Immediate</td>
<td>10.00</td>
<td>20.00</td>
<td>30.00</td>
</tr>
<tr>
<td>Reg. Indirect</td>
<td>20.00</td>
<td>30.00</td>
<td>40.00</td>
</tr>
<tr>
<td>Scaled</td>
<td>10.00</td>
<td>20.00</td>
<td>30.00</td>
</tr>
<tr>
<td>Mem. Indirect</td>
<td>0.00</td>
<td>10.00</td>
<td>20.00</td>
</tr>
</tbody>
</table>

Old VAX architecture

why VAX?
Data collected on a computer with a 16-bit displacement (Alpha with full optimization), what can they tell us about displacement > 16?

Number of bits needed for immediate (on ALPHA with a max. of 16 bits)
Addressing Modes for DSP

• Circular addressing mode needed to deal with circular buffers (common in DSP).
• Bit-reversal algorithms are very common in DSP, without that mode, an extra cycle is needed.

Type and size of Operands

• For general-purpose computers, byte, half-word, word, and double word are common (but also depends on the applications, double FP is rare in compilers).
• For DSP applications, we may require different sizes (graphics 24-bit with 8-bit per basic color).
Operations

- Addition, subtractions, shift and logical are standard in any CPU.
- Should we have an integer multiplication? What about FP multiplication? Division?
- Sometimes, even SQRT? String operations?
- For DSP, MAC (multiply and accumulate) is a very important operation
- Paired operations? 2 32-bit operations are done simultaneously on a 64-bit register (must suppress carry). Saturation arithmetic? SIMD?

Control Flow

- Conditional branches, Jumps, Procedure calls and returns.
- How to specify the target address? Most commonly used is a displacement from the PC \textit{PC-relative}
- Register indirect jumps allows dynamically determine the address (passing function as arguments in C)
Evaluating Branch Condition

- Condition Code CC: Tests special bits set by the ALU operations
- Condition Register: Tests arbitrary register (part of the instruction)
- Compare and branch: Comparison is a part of the branch instruction

Evaluating Branch Condition

- In some DSP applications, the small size of the loop makes the overhead in determining the condition too much.
- In some DSP, a *repeat* instruction is used to repeat a block of instructions a specific number of times.
- Three registers could be used, block start, block end, and a repeat counter
MIPS64 Branch Instructions

BEQZ R4, name  Branch equal zero
   if (Regs[R4] == 0) PC ← name;
   \((PC+4) - 2^{17}\) ≤ name < \((PC+4) + 2^{17}\)

BNEZ R4, Name  Branch not equal zero
   if (Regs[R4] != 0) PC ← name
   \((PC+4) - 2^{17}\) ≤ name < \((PC + 4) + 2^{17}\)

Procedure Call

- Who should save the registers, caller or callee?
- What is a global variable was allocated to a register, the procedure may change that global variable?
### MIPS64 JUPM

**J** name  
Jump  
\[ \text{PC}_{36..63} \leftarrow \text{name} \]

**JAL** name  
Jump and link  
\[ \begin{align*}  
\text{Regs}[31] & \leftarrow \text{PC}+4; \quad \text{PC}_{36..63} \leftarrow \\
\text{name}; ((\text{PC}+4)-2^{27}) & \leq \text{name} < ((\text{PC} + 4) + 2^{27}) 
\end{align*} \]

**JALR** R2  
Jump and link register  
\[ \begin{align*}  
\text{Regs}[\text{R31}] & \leftarrow \text{PC}+4; \quad \text{PC} \leftarrow \\
\text{Regs}[\text{R2}] & \leftrightarrow 
\end{align*} \]

---

### The Role of Compilers

**Dependencies**

- Language dependent  
- Machine dependent

- Somewhat Language dependent largely machine independent

- Small language dependencies  
  machine dependencies slight  
  (e.g. register counts/types)

- Highly machine dependent  
  language independent

**Function:**

- Transform Language to Common intermediate form

- For example procedure inlining and loop transformations

- Include global and local optimizations + register allocation

- Detailed instruction selection  
  and machine-dependent optimizations; may include or be followed by assembler

---
The Role of Compilers

• Usually compilation is done in passes or phases
• There is no going back to undo something in the previous phase (too complex).
• For example, common expression elimination is done before register allocation
• It may be even slower if the common expression is not assigned to a register.

Level0: no optimization  Level1: local optimization, code scheduling and register allocation  Level2: Global optimization and loop transformation  Level4: Procedure integration (Alpha)
How the Architect Can Help

- Provide regularity: orthogonal data type, operations, and addressing. It helps simplify code generation.
- Provide primitives not solutions: solutions that are tied to a HLL usually do not help much.
- Simplify trade-offs among alternatives: to help compiler writer to understand the different cost of alternatives and choose the best one.

RISC Philosophy

- While theoretically we can talk about complicated addressing modes and instructions, the ones we actually use in programs are the simple ones.
MIPS

- Use general purpose registers with a load-store architecture: **YES**
- Provide at least 16 general purpose registers plus separate floating-point registers: **31 GPR & 32 FPR**
- Support basic addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred; **YES: 16 bits for immediate, displacement (disp=0 => register deferred)**
- All addressing modes apply to all data transfer instructions: **YES**
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size: **Fixed**
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: **YES**
- Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return: **YES, 16b**
- Aim for a minimalist instruction set: **YES**

### MIPS Instruction Format

#### I - type instruction

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>rs</td>
<td>rt</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

Encodes: Loads and stores of bytes, words, half words. All immediates (rt ← rs op immediate) Conditional branch instructions (rs1 is register, rd unused) Jump register, jump and link register (rd = 0, rs = destination, immediate = 0)

#### R - type instruction

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>func</td>
</tr>
</tbody>
</table>

Register-register ALU operations: rd ← rs func rt Function encodes the data path operation: Add, Sub, .. Read/write special registers and moves.

#### J - Type instruction

<table>
<thead>
<tr>
<th>6</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Offset added to PC</td>
</tr>
</tbody>
</table>

Jump and jump and link. Trap and return from exception
Decoding

Register (direct)  
\[ \text{op} \rightarrow \text{register} \rightarrow \text{rd} \]

Immediate  
\[ \text{op} \rightarrow \text{immed} \]

Base+index  
\[ \text{op} \rightarrow \text{register} \rightarrow \text{mem} \]

PC-relative  
\[ \text{op} \rightarrow \text{PC} \rightarrow \text{mem} \]