Computer Architecture: Instruction Set Principles

Berk Sunar        Thomas Eisenbarth

ECE 505
Outline

• Measuring Performance

• Review of Instruction Set Architecture
  1. Classification of ISA
  2. How to analyze Instruction Set architecture
  3. Example: MIPS ISA
Classification of ISAs

... based on operand specification:

- **Stack** architecture: operators on top of stack
- **Accumulator** architecture: one operand is accu
- **General Purpose Register (GPR)** Architecture
  - Most popular: registers are fast and efficient for compiler

Two types of GPR Architecture:

- **Register-memory architecture**: subset of operands directly from memory
- **Load-store architecture**: operates on registers only; operands must be moved to registers first
  - Almost all newer designs are load-store
Examples of ISA classes

Source: H+P book
Code Example: $C = A + B$

**Stack architecture:**
- Push A
- Push B
- Add
- Pop C

**Accumulator architecture:**
- Load A
- Add B
- Pop C

**Register-memory:**
- Load R1, A
- Add R3, R1, B
- Stor R3, C

**Register (load-store):**
- Load R1, A
- Load R2, B
- Add R3, R1, R2
- Stor R3, C
GPR Computers Comparison

**Register-Register** (e.g. ARM, PowerPC, SPARC)
- Simple fixed length instruction encoding
- Instructions take similar clock cycles to execute
- Simple code generation
- Higher instruction count than below architectures
- More instructions + lower density → larger programs

**Register-Memory** (e.g. Intel 80x86, Motorola 68000)
- Data accessible without extra load instruction
- Instruction format easy to encode + good density
- CPI may vary by operand location
- Source operand may be overwritten in binary operations

**Memory-Memory** (VAX)
- Most Compact
- Large variation in instruction size (especially if 3-operand instructions possible)
- Large variation in work per instruction
- Memory accesses create bottleneck → no longer used

ECE 505 Computer Architecture
Memory Addressing

Data available in chunks of
- Bytes (8 bit);
- Half words (16 bit)
- Words (32 bit);
- Double Words (64 bits)

“Alignment restriction” ➞ faster data transfer

Endianness:
how is \(0\text{x}abcdef12\) represented in memory?

Big-endian (e.g. IPv6)

<table>
<thead>
<tr>
<th>Word address</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ab</td>
<td>cd</td>
<td>ef</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>8</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

Little-endian (e.g. x86)

<table>
<thead>
<tr>
<th>Word address</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>ef</td>
<td>cd</td>
<td>ab</td>
</tr>
<tr>
<td>4</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>8</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
Addressing Modes (Selection)

- **Register addressing:** Add R4, R3
  - operands are in registers

- **Immediate addressing:** Add R4, #3
  - operand is a constant within the instruction

- **Base or displacement addressing:** Add R4, 100(R1)
  - operand is at the memory location whose address is the sum of a register(R1) and a constant (100) in the instruction

- **Register indirect addressing:** Add R4, (R1)
  - operand is at the memory location whose address is stored in register (R1)

- **Direct Addressing:** Add R4, (1001)
  - Operand is at the memory location whose address is directly given in the instruction
Operations Classes

- **Arithmetic and logical**
  - Integer arithmetic and logical: add, subtract, and, or, multiply
- **Data transfer**
  - Load and store commands
- **Control**
  - Branch, jump, call, return

Optional:
- **System**: OS call, memory management instructions, VMM support etc.
- **Floating Point**: Floating point arithmetic: add, multiply ...
- **String**: string move, compare, search
- **Graphics**: Pixel and vertex operations
Example: Top 10 80x86 Instructions

<table>
<thead>
<tr>
<th>Rank</th>
<th>80x86 Instruction</th>
<th>% of total executed (SPECint92)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load</td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>Conditional branch</td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>Compare</td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>Store</td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>Add</td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>And</td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>Sub</td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>Move reg-reg</td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>Call/return</td>
<td>1%/1%</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>96%</td>
</tr>
</tbody>
</table>

- Small number of instructions makes majority of executed instructions make common case fast
- Most are data transfer and control instructions
Instructions for Control Flow

Types of Instructions:
- Conditional Branches
- Jumps
- Procedure Calls
- Procedure Returns

Addressing Modes:
Destination address is specified as:
- PC-relative: displacement added to program counter
  - Position independent: helpful when code is dynamically linked
  - Q: How many bits for displacement value?
- Register Indirect: go to address provided in register
Encoding an Instruction Set

How are instructions stored in memory (programs)?
   – Operation specified as opcode
   – How to represent addressing modes?
   – Variable or fixed size instructions?

Design Goals:
• Flexibility: Address as many registers and address modes as possible
• Keep code size small: Impact of above on average instruction size, hence program size
• Speedy execution: Instructions should align (no odd sizes) and consumable by pipeline (later).
Variations in Instruction Encoding

**Fixed Format:** e.g. ARM, MIPS, PowerPC

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address field 1</th>
<th>Address field 2</th>
<th>Address field 3</th>
</tr>
</thead>
</table>

**Variable Format:** e.g. Intel 80x86

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address Specifier 1</th>
<th>Address field 1</th>
<th>...</th>
<th>Address Specifier n</th>
<th>Address field n</th>
</tr>
</thead>
</table>

**Mixed Format:** e.g. Thumb, MIPS16
- Always same number of operands, with addressing modes specified as part of opcode
- Fields do not vary in location, but by how content is used/interpreted
Example Instructions for Intel 80386

**Instruction encoding:**
- Variable format
- Length: 1 - 17 bytes
- High flexibility

➔ 80x86 usually has smaller programs than RISC Architectures

---

ECE 505 Computer Architecture
Typical 80386 Instruction Formats

**JUMP**
- Condition: 4 bits
- Displacement: 8 bits

if negative (condition code) go to [EIP + displacement]

**CALL**
- Offset: 32 bits

CALL name
- \( SP = SP - 4 \)
- \( M[SP] = EIP + 5 \)
- \( EIP = Offset \)

**MOV**
- d: 1 bit
- w: 1 bit
- r/m postbyte: 8 bits
- displacement: 8 bits

**MOV** EBX, [EDI+45]

direction
byte or word?
specify target register, which registers are used in address calculation, how?
An Example: the MIPS Architecture

A RISC processor, with design choices:
• General purpose register architecture, Load-store version
• Addressing modes:
  – Displacement (with offset of approx. 16 bits)
  – Immediate (16 bits)
  – Register indirect and Absolute are derived
• Supports:
  – common data sizes: 8, 16, 32, 64 bit integer (+float)
  – Simple instructions: load, store add, sub, move reg-reg, shift
  – Branches: Compare equal, compare not equal, compare less, branch, jump, call, return
• Fixed (performance) or variable (code size) instruction encoding; MIPS is fixed
The MIPS Architecture

- Simple load-store instruction set
- Designed for pipelining efficiency (fixed instruction set encoding)
- Efficiency as compiler target
The MIPS 64

- 32x 64-bit General Purpose Registers (R0 – R31)
- 32x 64-bit (or 32bit) floating point registers (Fi)
- R0 = 0 always
- Byte addressable memory with 64-bit address
- Endianness is choosable, aligned memory access
- 32-bit instructions
  - 6-bit primary opcode
  - Up to 3 registers (5-bit each)
  - Three different types
Three types of MIPS Instructions

**R-type instruction**: Register-register ALU operations and moves

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

**I-type instruction**: Immediate, including load/store, conditional branch

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>Immediate (e.g. address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

**J-type instruction**: Unconditional branches: jump/trap/return

<table>
<thead>
<tr>
<th>op</th>
<th>Offset added to PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>26</td>
</tr>
</tbody>
</table>
R-Type Instruction format

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

- **op**: *opcode* (type of the operation)
- **funct**: selects the specific variant of the operation in the op field. *Function* but sometimes called *function field*.
- **rs**: The 1<sup>st</sup> register source operand
- **rt**: The 2<sup>nd</sup> register source operand
- **rd**: The 3<sup>rd</sup> register destination operand
- **shamt**: shift amount
I-Type Instruction Format

- MIPS designers elect to introduce a new type of instruction format by keeping the length same:
  - I-type for Immediate

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **Example**: `LW R8, 200(R18)`

Notice the 2\textsuperscript{nd} source register becomes the target register
J-Type Instruction Format

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

J-type instructions offer extra long immediate as offset to PC

Example:
**Jump:** \( J \) \( \text{name} \) \( \text{###} PC\downarrow {36..63} \rightarrow \text{name} \)

**Jump+Link:** \( JAL \) \( \text{name} \) \( \text{###} \text{Regs[R31]} \rightarrow PC+8 \)
\( \text{###} PC\downarrow {36..63} \rightarrow \text{name} \)

Branch delay slot
## Example Instructions

<table>
<thead>
<tr>
<th>Example Instruction</th>
<th>Instruction Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DADDU R1, R2, R3</td>
<td>Add unsigned</td>
<td>$\text{Regs}[R1] \leftarrow \text{Regs}[R2] + \text{Regs}[R3]$</td>
</tr>
</tbody>
</table>
# Example Instructions

<table>
<thead>
<tr>
<th>Example Instruction</th>
<th>Instruction Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DADDU R1, R2, R3</td>
<td>Add unsigned</td>
<td>( \text{Regs}[R1] \leftrightarrow \text{Regs}[R2] + \text{Regs}[R3] )</td>
</tr>
</tbody>
</table>
| MOVZ R1, R2, R3     | Conditional move if zero  | \( \text{If}(\text{Regs}[R3] = 0) \)\[
|                     |                           | \( \text{Regs}[R1] \leftarrow \text{Regs}[R2] \) |
# Example Instructions

<table>
<thead>
<tr>
<th>Example Instruction</th>
<th>Instruction Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DADDU R1, R2, R3</td>
<td>Add unsigned</td>
<td>Regs[R1] ← Regs[R2] + Regs[R3]</td>
</tr>
<tr>
<td>MOVZ R1, R2, R3</td>
<td>Conditional move if zero</td>
<td>If (Reg[R3] = 0) Regs[R1] ← Regs[R2]</td>
</tr>
<tr>
<td>LD R1, 30(R2)</td>
<td>Load Double Word</td>
<td>Regs[R1] ← Mem[30 + Regs[R2]]</td>
</tr>
</tbody>
</table>
## Example Instructions

<table>
<thead>
<tr>
<th>Example Instruction</th>
<th>Instruction Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DADDU R1, R2, R3</td>
<td>Add unsigned</td>
<td>$\text{Regs}[R1] \leftarrow \text{Regs}[R2] + \text{Regs}[R3]$</td>
</tr>
<tr>
<td>MOVZ R1, R2, R3</td>
<td>Conditional move if zero</td>
<td>If($\text{Regs}[R3]=0$) \hspace{1cm} $\text{Regs}[R1] \leftarrow \text{Regs}[R2]$</td>
</tr>
<tr>
<td>LD R1, 30(R2)</td>
<td>Load Double Word</td>
<td>$\text{Regs}[R1] \leftarrow \text{Mem}[30+\text{Regs}[R2]]$</td>
</tr>
<tr>
<td>LD R1, 1000(R0)</td>
<td>Load Double Word</td>
<td>$\text{Regs}[R1] \leftarrow \text{Mem}[1000+0]$</td>
</tr>
</tbody>
</table>
# Example Instructions

<table>
<thead>
<tr>
<th>Example Instruction</th>
<th>Instruction Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DADDU R1, R2, R3</td>
<td>Add unsigned</td>
<td>$\text{Regs}[R1] \leftarrow \text{Regs}[R2] + \text{Regs}[R3]$</td>
</tr>
</tbody>
</table>
| MOVZ R1, R2, R3     | Conditional move if zero  | If($\text{Regs}[R3]=0$)  
                        | $\text{Regs}[R1] \leftarrow \text{Regs}[R2]$                     |
| LD R1, 30(R2)       | Load Double Word          | $\text{Regs}[R1] \leftarrow \text{Mem}[30+\text{Regs}[R2]]$          |
| LD R1, 1000(R0)     | Load Double Word          | $\text{Regs}[R1] \leftarrow \text{Mem}[1000+0]$                      |
| BNE R3, R4, name    | Branch Not Equal          | If($\text{Regs}[R3]!=$\text{Regs}[R4]$) 
                        | PC$\leftarrow$\text{name}  
                        | $((\text{PC}+4)-2^{17})\leq\text{name}\leq((\text{PC}+4)+2^{17})$   |

ECE 505 Computer Architecture
<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>% of total executed (SPECint2000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load</td>
<td>26%</td>
</tr>
<tr>
<td>2</td>
<td>Conditional branch</td>
<td>12% (20% on x86)</td>
</tr>
<tr>
<td>3</td>
<td>Compare</td>
<td>5% (16% on x86)</td>
</tr>
<tr>
<td>4</td>
<td>Store</td>
<td>10%</td>
</tr>
<tr>
<td>5</td>
<td>Add</td>
<td>19% (8% on x86)</td>
</tr>
<tr>
<td>6</td>
<td>And</td>
<td>4%</td>
</tr>
<tr>
<td>7</td>
<td>Sub</td>
<td>3%</td>
</tr>
<tr>
<td>8</td>
<td>OR (includes Move reg-reg)</td>
<td>9% (1% on X86)</td>
</tr>
<tr>
<td>9</td>
<td>Call/return</td>
<td>1%/1%</td>
</tr>
</tbody>
</table>

Total 90%

- *Still* Most are data transfer and control instructions
- Less branching is good for pipeline
Outlook

Next Time:

Pipelining