INSTRUCTION SET ARCHITECTURE

Mahdi Nazm Bojnordi
Assistant Professor
School of Computing
University of Utah
What is ISA?

- Instruction Set Architecture
  - Well-defined interfacing contract between hardware and software
  - Does define
    - The functional operations of units
    - How to use each functional unit
  - Does not define
    - How functional units are implemented
    - Execution time of operations
    - Energy consumption of operations
Example Problem

- Which one may be guaranteed by a ISA?
  - The number of instructions supported by processor
  - The number of multipliers used by processor
  - The width of operands
  - Sequence of instructions that results in an error
  - Sequence of instructions that results in lower energy consumption
  - The total number of instructions for an application program
  - The total amount of main memory (e.g., DRAM)
Example Problem

Which one may be guaranteed by a ISA?

YES □ The number of instructions supported by processor
NO □ The number of multipliers used by processor
YES □ The width of operands
YES □ Sequence of instructions that results in an error
NO □ Sequence of instructions that results in lower energy consumption
NO □ The total number of instructions for an application program
NO □ The total amount of main memory (e.g., DRAM)
ISA to Programmer Interface

- Internal machine states
  - Architectural registers, control registers, program counter
  - Memory and page table

- Operations
  - Integer and floating-point operations
  - Control flow and interrupts

- Addressing modes
  - Immediate, register-based, and memory-based
ISA Types

- Operand locations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Store to TOS</td>
</tr>
<tr>
<td>Push B</td>
<td>Store to B</td>
</tr>
<tr>
<td>Add</td>
<td>Add B, TOS</td>
</tr>
<tr>
<td>Add R1,A</td>
<td>Add R3,R1,B</td>
</tr>
<tr>
<td>Load R1,A</td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Store C</td>
<td>Store R3,C</td>
</tr>
<tr>
<td>Add R3,R1,R2</td>
<td></td>
</tr>
<tr>
<td>Store R3,C</td>
<td></td>
</tr>
</tbody>
</table>
Which Set of Instructions?

- ISA influences the execution time
  - CPU time = IC x CPI x CT
- Complex Instruction Set Computing (CISC)
- Reduced Instruction Set Computing (RISC)
Which Set of Instructions?

- ISA influences the execution time
  - CPU time = IC \times CPI \times CT

- Complex Instruction Set Computing (CISC)
  - May reduce IC, increase CPI, and increase CT
  - CPU time may be increased

- Reduced Instruction Set Computing (RISC)
  - May increases IC, reduce CPI, and reduce CT
  - CPU time may be improved
## RISC vs. SISC

<table>
<thead>
<tr>
<th>RISC ISA</th>
<th></th>
<th>CISC ISA</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>□ Simple operations</td>
<td>□ Complex operations</td>
<td>□ Costly memory access</td>
<td></td>
</tr>
<tr>
<td>□ Simple and fast FU</td>
<td>□ Variable length</td>
<td>□ Complex decoder</td>
<td></td>
</tr>
<tr>
<td>□ Fixed length</td>
<td>□ Limited registers</td>
<td>□ Hard code generation</td>
<td></td>
</tr>
<tr>
<td>□ Simple decoder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>□ Limited inst. formats</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>□ Easy code generation</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory Addressing

- Register
  - Add r4, r3

- Immediate
  - Add r4, #3

- Displacement
  - Add r4, 100(r1)

- Register indirect
  - Add r4, (r1)
Memory Addressing

- **Register**
  - Add r4, r3 \( \text{Reg}[4] = \text{Reg}[4] + \text{Reg}[3] \)

- **Immediate**
  - Add r4, #3 \( \text{Reg}[4] = \text{Reg}[4] + 3 \)

- **Displacement**
  - Add r4, 100(r1) \( \ldots + \text{Mem}[100+\text{Reg}[1]] \)

- **Register indirect**
  - Add r4, (r1) \( \ldots + \text{Mem}[\text{Reg}[1]] \)
Memory Addressing

- Indexed
  - Add r3, (r1+r2)

- Direct
  - Add r1, (1001)

- Memory indirect
  - Add r1,@(r3)

- Auto-increment
  - Add r1, (r2)
Memory Addressing

- Indexed
  - Add r3, (r1+r2) \( \ldots + \text{Mem[Reg[1]+Reg[2]]} \)

- Direct
  - Add r1, (1001) \( \ldots + \text{Mem[1001]} \)

- Memory indirect
  - Add r1, @(r3) \( \ldots + \text{Mem[Mem[Reg[3]]]} \)

- Auto-increment
  - Add r1, (r2)+ \( \ldots + \text{Mem[Reg[2]]} \)
Memory Addressing

- Auto-decrement
  - Add r1, -(r2)

- Scaled
  - Add r1, 100(r2)[r3]
Memory Addressing

- **Auto-decrement**
  - Add r1, -(r2)  
  - ...
  - ... + Mem[Reg[2]]

- **Scaled**
  - Add r1, 100(r2)[r3]
  - ...
Example Problem

- Find the effective memory address
  - Add r2, 200(r1)
  - Add r2, (r1)
  - Add r2, @(r1)

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>100</td>
</tr>
<tr>
<td>r2</td>
<td>200</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>400</td>
</tr>
<tr>
<td>200</td>
<td>500</td>
</tr>
<tr>
<td>300</td>
<td>600</td>
</tr>
<tr>
<td>400</td>
<td>700</td>
</tr>
<tr>
<td>500</td>
<td>800</td>
</tr>
</tbody>
</table>
Example Problem

- Find the effective memory address
  - Add r2, 200(r1)
    - r2 = r2 + Mem[300]
  - Add r2, (r1)
    - r2 = r2 + Mem[100]
  - Add r2, @(r1)
    - r2 = r2 + Mem[400]

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>...</td>
</tr>
<tr>
<td>r2</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>400</td>
</tr>
<tr>
<td>200</td>
<td>500</td>
</tr>
<tr>
<td>300</td>
<td>600</td>
</tr>
<tr>
<td>400</td>
<td>700</td>
</tr>
<tr>
<td>500</td>
<td>800</td>
</tr>
</tbody>
</table>
Instruction Format

- A guideline for generating/interpreting instructions
- Example: MIPS
  - Fixed size 32-bit instructions
  - Three opcode types
    - I-type: load, store, conditional branch
    - R-type: ALU operations
    - J-type: jump