ADDRESSING MODES AND PIPELINING

Mahdi Nazm Bojnordi
Assistant Professor
School of Computing
University of Utah
Overview

- Announcement
  - Tonight: Homework 1 release (due on Sept. 4th)
    - Verify your uploaded files before deadline

- This lecture
  - RISC vs. CISC
  - Addressing modes
  - Pipelining
ISA Types

- Operand locations

<table>
<thead>
<tr>
<th></th>
<th>Push A</th>
<th>Load A</th>
<th>Load R1,A</th>
<th>Load R2,B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R3,R1,B</td>
<td>Add R3,R1,R2</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store R3,C</td>
<td></td>
<td>Store R3,C</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td></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 x CPI x 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 decreased
## RISC vs. SISC

<table>
<thead>
<tr>
<th>RISC ISA</th>
<th>CISC ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple operations</td>
<td>Complex operations</td>
</tr>
<tr>
<td>- Simple and fast FU</td>
<td>- Costly memory access</td>
</tr>
<tr>
<td>- Fixed length</td>
<td>- Variable length</td>
</tr>
<tr>
<td>- Simple decoder</td>
<td>- Complex decoder</td>
</tr>
<tr>
<td>- Limited inst. formats</td>
<td>- Limited registers</td>
</tr>
<tr>
<td>- Easy code generation</td>
<td>- Hard code generation</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) \( \ldots + \text{Mem}[\text{Reg}[1]+\text{Reg}[2]] \)

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

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

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

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

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

- **Auto-decrement**
  - Add r1, -(r2)  \[\text{Reg}[2]=\text{Reg}[2]-\text{d}\]
  - ...+\text{Mem}[\text{Reg}[2]]

- **Scaled**
  - Add r1, 100(r2)[r3]  \[\text{...+Mem}[100+\text{Reg}[2]+\text{Reg}[3] \times \text{d}]\]
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 \( r2, 200(r1) \)
  - \( r2 = r2 + \text{Mem}[300] \)

- Add \( r2, (r1) \)
  - \( r2 = r2 + \text{Mem}[100] \)

- Add \( r2, @(r1) \)
  - \( r2 = r2 + \text{Mem}[400] \)

<table>
<thead>
<tr>
<th>Registers</th>
<th>( r1 )</th>
<th>( r2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>700</td>
</tr>
<tr>
<td></td>
<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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>RS</th>
<th>RT</th>
<th>Immediate</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>RS</th>
<th>RT</th>
<th>RD</th>
<th>ShAmnt</th>
<th>Funct</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
</tr>
</thead>
</table>
Pipelining
Processing Instructions

- Every RISC instruction may require multiple processing steps.
Every RISC instruction may require multiple processing steps.
Every RISC instruction may require multiple processing steps:
- Instruction Fetch (IF)
- Instruction Decode (ID)
- Register Read (RR)
  - All instructions?
- Execute Instructions (EXE)
- Memory Access (MEM)
  - All instructions?
- Register Write Back (WB)
Single-cycle RISC Architecture

- Example: simple MIPS architecture
  - Critical path includes all of the processing steps
Single-cycle RISC Architecture

Example program

CT=6ns; CPU Time = ?

```
AND R1,R2,R3
XOR R4,R2,R3
SUB R5,R1,R4
ADD R6,R1,R4
MUL R7,R5,R6
```

CPU Time = CI x CPI x CT
Example program

- \( CT = 6\text{ns} \); CPU Time = \( 5 \times 1 \times 6\text{ns} = 30\text{ns} \)

```
AND R1, R2, R3
XOR R4, R2, R3
SUB R5, R1, R4
ADD R6, R1, R4
MUL R7, R5, R6
```

How to improve?

CPU Time = \( CI \times CPI \times CT \)