CS430 Computer Architecture

Spring 2015
Chapter 15
Reduced Instruction Set Computers

- Reading: pp. 532-538; 545-562
- Major architectural advances since the stored program concept in 1950 include (but are not limited to) the following:
  1. The Family of Computers - companies such as IBM, DEC, Sperry, and so on developed lines of computers around the same architecture with different price/performance ranges.
  2. Microprogrammed Control - provided flexibility and ease in regard to developing the control unit.
  3. Cache Memory - a great performance gain by not having to access main memory for everything.
  4. Pipelining - introduced parallelism into the architecture.
  5. Multi-processing - the use of multiple processors in various computer organizations.
RISC

- We now add RISC
  1. Small/Simple Instructions Set
  2. Large number of GPRs
  3. Emphasis on instruction pipelining
CISC, RISC, Superscalar Characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Complex Instruction Set (CISC) Computer</th>
<th>Reduced Instruction Set (RISC) Computer</th>
<th>Superscalar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IBM 370/168</td>
<td>SPARC</td>
<td>1993</td>
</tr>
<tr>
<td>Year developed</td>
<td>VAX 11/780</td>
<td>MIPS R4000</td>
<td>1996</td>
</tr>
<tr>
<td></td>
<td>Intel 80486</td>
<td>PowerPC</td>
<td>1996</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ultra SPARC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MIPS R10000</td>
<td></td>
</tr>
<tr>
<td>Number of instructions</td>
<td>208</td>
<td>69</td>
<td>225</td>
</tr>
<tr>
<td>Instruction size (bytes)</td>
<td>2–6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Addressing modes</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Number of general-purpose registers</td>
<td>16</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40 - 520</td>
</tr>
<tr>
<td>Control memory size (Kbits)</td>
<td>420</td>
<td>480</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>246</td>
<td>—</td>
</tr>
<tr>
<td>Cache size (KBytes)</td>
<td>64</td>
<td>64</td>
<td>16-32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64</td>
</tr>
</tbody>
</table>
CISC vs RISC

- How did we get to this CISC state anyway?

- As the cost of hardware has dropped, the cost of software has risen.

- The response from researchers has been to develop more powerful and complicated HLLs.

- This resulted in the semantic gap.
Semantic Gap

- How did designers respond to the semantic gap?
  1. Large instruction sets.
  2. Dozens of addressing modes.
  3. Various HLL statements implemented in hardware.

- How do the above help?
  1. Ease the task of the compiler writer.
  2. Improve execution efficiency.
  3. Provide support for even more complicated HLLs.
RISC Advocates

- RISC advocates report the following negative associated with CISC architectures:
  1. It is sometimes hard to find the machine instructions that fit the HLL case because of all of the options. Example, the VAX has 25 different ADD instructions.
  2. Code optimization is very complicated when trying to minimize the code, reduce execution time, and maximize the efficiency of the pipeline.
  3. It is not necessarily the case that a CISC will produce smaller and more efficient code.
RISC Characteristics

1. A fixed instruction length (typically 4 bytes)
2. Few addressing modes (1-5)
3. No addressing that required two memory fetches
4. No arithmetic operations and also load/store
5. At most one MEM op per instruction
6. Minimum of 32 GPRs
Non-pipelined RISC Processor (MIPS)  
5 stages
Instruction Fetch (IF)

- IF (Instruction Fetch)
  IR <- M[PC]
  NPC <- PC + 4
Instruction Decode (ID)

- **ID (Instruction Decode/Register Fetch)**
  
  \[
  \begin{align*}
  A & \leftarrow \text{Regs}[rs] \\
  B & \leftarrow \text{Regs}[rt] \\
  \text{Imm} & \leftarrow \text{sign-extend immediate field of IR}
  \end{align*}
  \]

  **Note1:** Decoding can be done in parallel with reading registers

  **Note2:** In an aggressive implementation the branch can be completed at the end of this stage as we will see later
Execution (EX)

- Memory Reference
  1. ALUOutput <- A + immediate

- Register-Register ALU instruction
  1. ALUOutput <- A function B

- Register-Immediate ALU instruction
  1. ALUOutput <- A opcode Imm

- Branch
  1. ALUOutput <- NPC + (Imm << 2)
  2. Cond <- (A == 0)
Memory Access/Branch Completion (MEM)

- Memory Reference
  1. LMD <- Mem[ALUOutput] ; load from memory
  2. Mem[ALUOutput] <- B ; store to memory

- Branch
  1. if (cond) PC <- ALUOutput
Write Back (WB)

- Register-Register ALU
  1. Regs[rd] <- ALUOutput

- Register-Immediate ALU
  1. Regs[rt] <- ALUOutput

- Load instruction
  1. Regs[rt] <- LMD