CMSC 611: Advanced Computer Architecture

Pipelining

Some material adapted from Mohamed Younis, UMBC CMSC 611 Spr 2003 course slides
Some material adapted from Hennessy & Patterson / © 2003 Elsevier Science
- Washer takes 30 min, Dryer takes 40 min, folding takes 20 min
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?
Pipelining means start work as soon as possible.

Pipelined laundry takes 3.5 hours for 4 loads.
Pipelining Lessons

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduce speedup
- Stall for Dependencies

Slide: Dave Patterson
MIPS Instruction Set

- RISC characterized by the following features that simplify implementation:
  - All ALU operations apply only on registers
  - Memory is affected only by load and store
  - Instructions follow very few formats and typically are of the same size

```
<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>
| 31 | 26 | 21  | 16  | 11    | 6     | 0

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>target address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>26</td>
</tr>
</tbody>
</table>
| 31  | 26             | 0
```
MIPS Instruction Formats

- **R-type (register)**
  - Most operations
    - add $t1, $s3, $s4  \# $t1 = $s3 + $s4
  - rd, rs, rt all registers
  - op always 0, funct gives actual function
MIPS Instruction Formats

• I-type (immediate)
  – ALU with one immediate operand
    • addi $t1, $s2, 32  # $t1 = $s2 + 32
  – Load, store within $\pm 2^{15}$ of register
    • lw $t0, 32($s2)  # $s1 = $s2[32] or *(32+s2)
  – Load immediate values
    • lui $t0, 255  # $t0 = (255<<16)
    • li $t0, 255

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
</tbody>
</table>
MIPS Instruction Formats

- I-type (immediate)
  - PC-relative conditional branch
  - $\pm 2^{15}$ from PC after instruction

  - `beq $s1, $s2, L1` # goto L1 if ($s1 = $s2)
  - `bne $s1, $s2, L1` # goto L1 if ($s1 \neq $s2)

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
</tbody>
</table>
MIPS Instruction Formats

- J-type (jump)
  - unconditional jump
    - \texttt{j L1} \# goto L1
  - Address is concatenated to top bits of PC
    - Fixed addressing within $2^{26}$
Single-cycle Execution

![Diagram of single-cycle execution process]

Figure: Dave Patterson
Multi-Cycle Implementation of MIPS

1. **Instruction fetch cycle (IF)**
   
   IR ← Mem[PC]; NPC ← PC + 4

2. **Instruction decode/register fetch cycle (ID)**
   
   A ← Regs[IR_{6..10}]; B ← Regs[IR_{11..15}]; Imm ← ((IR_{16})^{16} #\#IR_{16..31})

3. **Execution/effective address cycle (EX)**
   
   - Memory ref: ALUOutput ← A + Imm;
   - Reg-Reg ALU: ALUOutput ← A \textit{func} B;
   - Reg-Imm ALU: ALUOutput ← A \textit{op} Imm;
   - Branch: ALUOutput ← NPC + Imm; Cond ← (A \textit{op} 0)

4. **Memory access/branch completion cycle (MEM)**
   
   - Memory ref: LMD ← Mem[ALUOutput] or Mem(ALUOutput) ← B;
   - Branch: if (cond) PC ← ALUOutput;

5. **Write-back cycle (WB)**
   
   - Reg-Reg ALU: Regs[IR_{16..20}] ← ALUOutput;
   - Reg-Imm ALU: Regs[IR_{11..15}] ← ALUOutput;
   - Load: Regs[IR_{11..15}] ← LMD;
Multi-cycle Execution

Figure: Dave Patterson
The load instruction is the longest
All instructions follow at most the following five steps:
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory and update PC
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- WB: Write the data back to the register file
**Instruction Pipelining**

- Start handling next instruction while the current instruction is in progress
- Feasible when different devices at different stages

Time between instructions\(_{\text{pipelined}}\) = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}

Pipelining improves performance by increasing instruction throughput
Example of Instruction Pipelining

Ideal and upper bound for speedup is number of stages in the pipeline
Single Cycle

- Cycle time long enough for longest instruction
- Shorter instructions waste time
- No overlap
• Cycle time long enough for longest stage
• Shorter stages waste time
• Shorter instructions can take fewer cycles
• No overlap
- Cycle time long enough for longest stage
- Shorter stages waste time
- No additional benefit from shorter instructions
- Overlap instruction execution
Pipeline Performance

• Pipeline increases the instruction throughput
  – not execution time of an individual instruction

• An individual instruction can be slower:
  – Additional pipeline control
  – Imbalance among pipeline stages

• Suppose we execute 100 instructions:
  – Single Cycle Machine
    • 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
  – Multi-cycle Machine
    • 10 ns/cycle x 4.2 CPI (due to inst mix) x 100 inst = 4200 ns
  – Ideal 5 stages pipelined machine
    • 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns

• Lose performance due to fill and drain
Pipeline Datapath

- Every stage must be completed in one clock cycle to avoid stalls.
- Values must be latched to ensure correct execution of instructions.
- The PC multiplexer has moved to the IF stage to prevent two instructions from updating the PC simultaneously (in case of branch instruction).
**Pipeline Stage Interface**

<table>
<thead>
<tr>
<th>Stage</th>
<th>Any Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IF</strong></td>
<td>IF/ID.IR ← MEM[PC]; IF/ID.NPC, PC ← (if (EX/MEM.opcode == branch) &amp; EX/MEM.cond) {EX/MEM.ALUOutput} else {PC + 4};</td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td>ID/EX.A = Regs[IF/ID.IR 8..10]; ID/EX.B ← Regs[IF/ID.IR 11..15]; ID/EX.NPC ← IF/ID.IR; ID/EX.Imm ← (IF/ID.IR 16) <strong>16</strong> # IF/ID.IR 16..31;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALU</th>
<th>Load or Store</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EX</strong></td>
<td>EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B; \ Or \ EX/MEM.ALUOutput ← ID/EX.A op ID/EX.Imm; EX/MEM.cond ← 0;</td>
<td>EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm;</td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td>MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;</td>
<td>MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; \ Or \ Mem[EX/MEM.ALUOutput] ← EX/MEM.B;</td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td>Regs[MEM/WB.IR 16..20] ← EM/WB.ALUOutput; \ Or \ Regs[MEM/WB.IR 11..15] ← MEM/WB.ALUOutput;</td>
<td>For load only: Regs[MEM/WB.IR 11..15] ← MEM/WB.LMD;</td>
</tr>
</tbody>
</table>
Pipeline Hazards

• Cases that affect instruction execution semantics and thus need to be detected and corrected
• Hazards types
  – Structural hazard: attempt to use a resource two different ways at same time
    • Single memory for instruction and data
  – Data hazard: attempt to use item before it is ready
    • Instruction depends on result of prior instruction still in the pipeline
  – Control hazard: attempt to make a decision before condition is evaluated
    • branch instructions
• Hazards can always be resolved by waiting
Visualizing Pipelining

Time (clock cycles)

Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7

Instr. Order

Slide: David Culler
Example: One Memory
Port/Structural Hazard

Time (clock cycles)

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7

Instr. Order

Instr 1

Instr 2

Instr 3

Instr 4

Load Ifetch Reg ALU DMem Reg

Instr 1 Ifetch Reg ALU DMem Reg

Instr 2 Ifetch Reg ALU DMem Reg

Instr 3 Ifetch Reg ALU DMem Reg

Instr 4 Ifetch Reg ALU DMem Reg

Structural Hazard

Slide: David Culler
Resolving Structural Hazards

1. Wait
   - Must detect the hazard
     • Easier with uniform ISA
   - Must have mechanism to stall
     • Easier with uniform pipeline organization

2. Throw more hardware at the problem
   - Use instruction & data cache rather than direct access to memory
Detecting and Resolving Structural Hazard

Time (clock cycles)

Cycle 1  Cycle 2  Cycle 3  Cycle 4  Cycle 5  Cycle 6  Cycle 7

Instr Order

Load

Instr 1

Instr 2

Stall

Instr 3

Slide: David Culler
Stalls & Pipeline Performance

Pipelining Speedup = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}

\text{Ideal CPI pipelined} = 1

\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall cycles per instruction}
\quad = 1 + \text{Pipeline stall cycles per instruction}

\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}

\text{Assuming all pipeline stages are balanced}

\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}