CSE 141 – Computer Architecture
Fall 2005
Lectures 12

Pipelining

Pramod V. Argade
November 7, 2005
Announcements

● **Reading Assignment**
  – Chapter 5. The Processor: Datapath and Control
    Sections 5.6
  – Chapter 6. Enhancing Performance with Pipelining
    Sections 6.1 - 6.10

● **Homework 5: Due Mon., November 14th in class**
  5.49, 5.50
  6.1, 6.2, 6.3, 6.4, 6.6, 6.14, 6.15, 6.17

● **Quiz**
  **When:** Mon., November 14th
  **Topic:** Pipelining and Hazards
## Course Schedule

<table>
<thead>
<tr>
<th>Lecture #</th>
<th>Date</th>
<th>Day</th>
<th>Lecture Topic</th>
<th>Quiz Topic</th>
<th>Homework Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9/26</td>
<td>Monday</td>
<td>Introduction, Ch. 1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>9/28</td>
<td>Wednesday</td>
<td>ISA, Ch. 2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>10/3</td>
<td>Monday</td>
<td>Arithmetic Part 1, Ch. 4</td>
<td>ISA</td>
<td>#1</td>
</tr>
<tr>
<td>4</td>
<td>10/5</td>
<td>Wednesday</td>
<td>Arithmetic Part 2, Ch. 4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>10/10</td>
<td>Monday</td>
<td>Performance, Ch. 3</td>
<td>Arithmetic</td>
<td>#2</td>
</tr>
<tr>
<td>6</td>
<td>10/12</td>
<td>Wednesday</td>
<td>Single cycle CPU, Ch. 5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>10/17</td>
<td>Monday</td>
<td>Single cycle CPU, Ch. 5</td>
<td>Performance</td>
<td>#3</td>
</tr>
<tr>
<td>8</td>
<td>10/19</td>
<td>Wednesday</td>
<td>Multi-cycle CPU, Ch. 5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>10/24</td>
<td>Monday</td>
<td>Multi-cycle CPU, Ch. 5</td>
<td>Single Cycle CPU</td>
<td>#4</td>
</tr>
<tr>
<td>10</td>
<td>10/26</td>
<td>Wednesday</td>
<td>Review for the Midterm</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10/31</td>
<td>Monday</td>
<td>Mid-term Exam</td>
<td>exceptions, Ch. 5 and Pipelining, Ch. 6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>11/2</td>
<td>Wednesday</td>
<td>Exceptions, Ch. 5 and Pipelining, Ch. 6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>11/7</td>
<td>Monday</td>
<td>Pipelining, Ch. 6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>11/9</td>
<td>Wednesday</td>
<td>Data and control hazards, Ch. 6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>11/14</td>
<td>Monday</td>
<td>Data and control hazards, Ch. 6</td>
<td>Pipeline Hazards</td>
<td>#5</td>
</tr>
<tr>
<td>15</td>
<td>11/16</td>
<td>Wednesday</td>
<td>Memory &amp; cache design, Ch. 7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>11/21</td>
<td>Monday</td>
<td>Memory &amp; cache design, Ch. 7</td>
<td>Cache</td>
<td>#6</td>
</tr>
<tr>
<td>17</td>
<td>11/23</td>
<td>Wednesday</td>
<td>Virtual Memory &amp; cache design, Ch. 7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>11/28</td>
<td>Monday</td>
<td>Course Review</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12/8</td>
<td>Thursday</td>
<td>Final Exam 7 - 10 PM</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
Pipelining: Its Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes
Sequential Laundry takes 6 hours for 4 loads.

If they learned pipelining, how long would laundry take?
Pipelined Laundry: Start work ASAP

- Pipelined laundry takes 3.5 hours for 4 loads
- Sequential laundry takes 6 hours for 4 loads
Pipelining Overview

- What is pipelining?
  - Multiple instructions are overlapped in execution

- Notes:
  - Time for completion of a single instruction is not shorter
  - Multiple tasks operate simultaneously
  - Pipelining does not change latency
  - Pipelining increases the throughput
  - Pipelining rate is limited by the slowest stage
  - Potential speedup = number of pipeline stages
  - Time to “fill” pipeline and time to “drain” it reduces speedup
Pipelining

- Requires separable jobs per stage
- Requires separate resources
- Achieves parallelism with replication
- Pipeline efficiency (keeping the pipeline full) critical to performance
- Time between instructions $T_{\text{pipelined}}$

  \[ T_{\text{pipelined}} = \frac{T_{\text{non-pipelined}}}{\# \text{ Pipe Stages}} \]

- Fundamentally invisible to the programmer
Pipelining: Instructions Implemented

- Only following instructions will be implemented
  - Memory: LW, SW
  - Arithmetic: ADD, SUB, AND, OR, SLT
  - Branch: BEQ

<table>
<thead>
<tr>
<th>Instruction Class</th>
<th>Instruction Fetch</th>
<th>Register Read</th>
<th>ALU Operation</th>
<th>Data Access</th>
<th>Register Write</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Word (lw)</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td>100 ps</td>
<td>800 ps</td>
</tr>
<tr>
<td>Store Word (sw)</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td></td>
<td>700 ps</td>
</tr>
<tr>
<td>R-Format</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td></td>
<td>100 ps</td>
<td>600 ps</td>
</tr>
<tr>
<td>Branch (beq)</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td></td>
<td></td>
<td>500 ps</td>
</tr>
</tbody>
</table>
Non-Pipelined vs. Pipelined Execution

Non-Pipelined: Time for 3 instructions $3 \times 800$ ps

Pipelined: Time for 3 instructions $3 \times 200$ ps

Pipelining offers 4x improvement in this example

Register file writes in the first half and reads in the second half of the cycle
MIPS ISA and Pipelining

- All instructions are the same length
  - Easier to fetch from instruction memory
  - Easier to decode in second stage
- Only a few instruction formats
  - Register field location(s) fixed
  - Operand fetch and instruction decode in parallel
- Load/store architecture
  - Memory operands appear only in load and store instructions
  - Execute stage calculates memory address and result for R-type
- Operands must be aligned in memory
  - One memory data transfer requires a single memory access
- Following instructions to be implemented
  - LW, SW, ADD, SUB, AND, OR, SLT, BEQ
Pipelining Challenges

- **Hazards**: Situations where next instruction cannot execute
  - **Structural hazards**:
    - Suppose we had only one memory for instructions and data
  - **Control hazards**:  
    - Need to worry about branch instructions
  - **Data hazards**:  
    - An instruction depends on the result of preceding instruction

- We’ll talk about modern processors and what really makes pipelining hard:
  - Exception handling
  - Trying to improve performance with out-of-order execution, etc.
Review: Single-cycle CPU

The diagram illustrates the flow of instructions in a single-cycle CPU. Here's a breakdown of the components and their interactions:

1. **Instruction Memory**:
   - The PC (Program Counter) is used to access the instruction memory, where the instructions are stored.

2. **Control Unit**:
   - It generates the control signals for various components.

3. **ALU (Arithmetic Logic Unit)**:
   - It performs arithmetic and logical operations.

4. **Registers**:
   - Contains registers for holding data and results.

5. **Adder**:
   - Performs addition operations.

6. **ALU Control**:
   - Determines the operation to be performed by the ALU.

7. **Zero**:
   - Checks if the result of an operation is zero.

8. **Address**:
   - Contains the address for accessing memory.

9. **Data Memory**:
   - Stores data that can be read or written.

10. **Instruction**:
    - The instructions are fetched from memory and executed.

11. **Shift Left 2**:
    - Shifts the data left by two bits.

12. **Jump Address**:
    - Generates the jump address for branching.

13. **Branch**:
    - Determines if a branch instruction should be executed.

14. **Read Address**:
    - Reads the memory address.

15. **Read Data**:
    - Reads data from memory.

16. **Write Data**:
    - Writes data to memory.

17. **Write Register**:
    - Writes data to registers.

18. **MemRead**:
    - Reads from memory.

19. **MemWrite**:
    - Writes to memory.

20. **MemtoReg**:
    - Copies memory data to registers.

21. **ALUOp**:
    - Determines the ALU operation.

22. **RegWrite**:
    - Writes to registers.

23. **RegDst**:
    - Determines the destination register.

24. **ALUSrc**:
    - Determines the source for ALU operations.

25. **MemtoReg**:
    - Copies memory data to registers.

26. **Zero**:
    - Checks if the result is zero.

27. **Sign Extend**:
    - Extends the sign of a number.

28. **Instruction [31–0]**:
    - The full 32-bit instruction is fetched.

29. **Instruction [25–0]**:
    - Generates the jump address.

30. **Instruction [25–20]**:
    - Determines the ALU operation.

31. **Instruction [20–16]**:
    - Determines the memory access mode.

32. **Instruction [15–11]**:
    - Performs shift operations.

33. **Instruction [15–0]**:
    - Performs sign extension.

The diagram illustrates the flow of instructions and data through these components, showing how they interact to execute a single cycle of instruction processing.
Review: Multi-cycle CPU
Instruction Latencies and Throughput

• **Single-Cycle CPU**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
</tr>
</tbody>
</table>

• **Multiple Cycle CPU**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
</tr>
</tbody>
</table>

• **Pipelined CPU**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipeline Performance Considerations

- CPU throughput = Instructions Per Cycle (IPC)
  
  Number of instructions completed per cycle = 1/CPI

- Execution Time = (Instruction Count) * CPI * (Cycle Time)

- Complexity has a cost
  - e.g., Pipeline register overhead
  - Uneven stage latencies

- Pipeline clock cannot run faster than
  - Slowest pipeline stage

- Pipeline overhead (logic needed to control the pipeline)

- Can’t always keep the pipeline full
  - Why not?
Pipeline Stages

IF: Instruction fetch
ID: Instruction decode and register fetch
EX: Execution and effective address calculation
MEM: Memory access
WB: Write back
Pipelining: Basic Idea

What do we need to add to actually split the datapath into stages?
Pipelined Datapath

Instruction Fetch  Instruction Decode/ Register Fetch  Execute/ Address Calculation  Memory Access  Write Back

- Instruction Fetch
  - Instruction memory
  - PC
  - Address

- Instruction Decode/ Register Fetch
  - Instruction
  - Registers
  - Read register 1
  - Read register 2
  - Write register
  - Write data

- Execute/ Address Calculation
  - Add
  - Shift left 2
  - ALU
  - Zero
  - Read data 1
  - Read data 2
  - Write data
  - ALU result
  - Address

- Memory Access
  - Data memory
  - Read data
  - Write data

- Write Back
  - 1 Mux 0
  - 0 Mux 1

Pramod Argade  CSE 141, Fall 2005  12-19
LW Through the Pipeline
LW in EX Pipeline Stage
Problem: Destination Register
Corrected Pipelined Datapath
Observations

• Instructions advance from one stage to another every clock

• Instructions and data move from left to right
  – Exceptions
    - WB stage writes to register file (Potential data hazard)
    - PC = Branch address from Mem stage (Potential control hazard)

• No registers in WB stage
  – Write registers already exist
Graphical Representation of a Pipeline

- Shading indicates that the element is used by the instruction
  - e.g. ADD instruction does not use MEM
- Shading in left half means that the element is written in that stage
- Shading in the right half means that the element is read in that stage
Execution in a Pipelined Datapath

CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC7 | CC8 | CC9
--- | --- | --- | --- | --- | --- | --- | --- | ---
IF  | ID  | EX  | MEM | WB  | IM  | Reg | DM  | Reg
lw  | IM  | Reg | ALU | DM  | Reg | ALU | DM  | Reg
lw  | IM  | Reg | ALU | DM  | Reg | ALU | DM  | Reg
lw  | IM  | Reg | ALU | DM  | Reg | ALU | DM  | Reg
lw  | IM  | Reg | ALU | DM  | Reg | ALU | DM  | Reg
lw  | IM  | Reg | ALU | DM  | Reg | ALU | DM  | Reg

steady state
Mixed Instructions in the Pipeline

Resource conflict!
Pipeline Principles

- All instructions that share a pipeline must have the same stages in the same order
  - Therefore, \textit{add} does nothing during Mem stage
  - \textit{SW} does nothing during WB stage

- All intermediate values must be registered each cycle

- There is no functional block reuse
Pipeline control

- We have 5 stages. What needs to be controlled for each instruction?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back

- Centralized control?
  - Too complicated?

- Better approach
  - Control embedded in the pipeline
  - Compute control information in decode stage
    - Pass it along through pipeline registers
Pipeline Control

- Use combinational Logic!
  - Signals generated once, but follow instruction through the pipeline

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution/Address Calculation stage control lines</th>
<th>Memory access stage control lines</th>
<th>Write-back stage control lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg</td>
<td>Dst</td>
<td>ALU Op1</td>
</tr>
<tr>
<td>R-format</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Pipelined Datapath and Control
Would our pipeline design work in any case?

- What happens when...
  add $3$, $10$, $11$
  lw $8$, 1000($3$)
  sub $11$, $8$, $7$
Announcements

• **Reading Assignment**
  - Chapter 5. The Processor: Datapath and Control
    Sections 5.6
  - Chapter 6. Enhancing Performance with Pipelining
    Sections 6.1 - 6.10

• **Homework 5:** Due Mon., November 14th in class
  5.49, 5.50
  6.1, 6.2, 6.3, 6.4, 6.6, 6.14, 6.15, 6.17

• **Quiz**
  **When:** Mon., November 14th
  **Topic:** Pipelining and Hazards