Chapter 3

Pipelining

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

<table>
<thead>
<tr>
<th>Task Order</th>
<th>Time</th>
<th>6 PM</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>Midnight</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>B</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>C</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>D</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
</tr>
</tbody>
</table>

- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry

<table>
<thead>
<tr>
<th>Task Order</th>
<th>Time</th>
<th>6 PM</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>Midnight</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup

Pipelining

- Technique for having multiple instructions execute in an overlapped manner
- Assembly Line
- Throughput -
  - Determined by how often an instruction exits the pipeline
- Time to move one step is machine cycle time
  - Determined by the slowest step in the pipeline
  - Often it is clock cycle though clocks may have multiple phases
- Length of Pipe - No of stages
  - Determine latency
  - ...

Pipeline Performance

- Under ideal conditions
  - time per instruction =
    
    \[ \frac{\text{Time per instruction on non-pipelined machine}}{\text{Number of pipe stages}} \]
    
    Speedup = No of stages
    
    BUT
    
    Stages are not balanced
    
    Overhead (10%)
Basic Performance Issues

• Pipeline increases instruction throughput
  -- It does not decrease the time of execution of any single instruction - it may increase it.

• Stage imbalance may yield further inefficiencies

• Overheads due to
  -- register and latches adding to delays and clock skew

Example 1

• Consider 10ns clock, 4 cycle ALU(40%) and Branch(20%) Ops and 5 cycle Mem(40%) operations
• Overhead due to clock skew and setup is 1ns
• What is the speedup due to pipelining?

Avg Instruction Execution Time = 10 [(0.4 + 0.2) 4 + 0.4 x 5] = 44 ns
Average Time per instruction for pipeline machine = 11 ns
Speed up = 44/11 = 4

Example 2

• Time for five stages are 10, 8, 10, 10, 7 ns
• Pipelining adds 1 ns
• What is the speedup?

Average instruction execution time = 10 + 8 + 10 + 10 + 7 = 45 ns
Clock Cycle time for a pipeline stage = 11 ns
Speedup = 45/11 = 4.1

Basic Steps of Execution in DLX

1. Instruction fetch step (IF)
2. Instruction decode/register fetch step (ID)
3. Execution/effective address step (EX)
4. Memory access/branch completion step (MEM)
5. Register write-back step (WB)
**DLX Instruction Format**

### I-type instruction

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>d</td>
<td>s</td>
<td>t</td>
</tr>
</tbody>
</table>

- **Op**: Loads and stores of data, shifts, and rotates
- **d**: Destination register
- **s**: Source register
- **t**: Immediate (8 bits)

### R-type instruction

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>R</td>
<td>S</td>
<td>T</td>
</tr>
</tbody>
</table>

- **Op**: ALU operations (0-7, 11, 12, 13, 14)

### J-type instruction

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
</tr>
</tbody>
</table>

- **Op**: Offset to PC

**Instructions and Functions**

- **Load**: LDR
- **Store**: LDR
- **Shift**: LSL, LSR
- **Rotate**: ROR
- **ALU Operations**: ADD, SUB, ...
### Basic Steps of Execution

#### 4. Memory access/branch completion step (MEM)
- \( PC \rightarrow NPC \)
- \( LMD \leftarrow \text{Mem(\text{ALUoutput}) or Mem(\text{ALUoutput})} \)
- If (cond) \( PC \rightarrow \text{ALUoutput} \)

### Basic Steps of Execution

#### 5. Register write-back step (WB)
- \( \text{Reg}[R_{15}] \rightarrow \text{ALUoutput} \)
- \( \text{Reg}[R_{11..15}] \rightarrow \text{ALUoutput} \)
- \( \text{Reg}[R_{11..15}] \rightarrow \text{LMD} \)

### Basic Pipeline for DLX

<table>
<thead>
<tr>
<th>Instruction number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### DLX Pipeline: datapath
Making Pipeline Work

- Determine what happens at each clock tick
- Assure no resource conflicts
  - Can not use one ALU for address calculation and ALU functions
- All operations in a pipe stage must complete in one clock cycle
  - May have to elongate the clock cycle to accommodate this

Pipeline Structure

- Major functional units are used in different cycles
- Three observations
  - Basic datapath uses separate instruction and data memory
  - Have separate instruction and data caches
  - Memory bandwidth increases in a pipelined system
  - Register file is used in two stages - ID and WB
  - To start a new instruction every clock we must increment and store the PC every clock cycle during IF stage
  - What happens when branches occur?

Pipeline Structure

- Every pipe stage is active in every cycle
  - Values passed from one stage to next must be placed in registers
  - Use pipeline registers or pipeline latches between stages
- Registers used to transfer information from one stage to the next
- Pipeline registers carry both control and data from stage to stage
- Values may have to be copied from one to the next stage if it is needed at a later stage
- An instruction is active in exactly one stage of the pipe at any moment

DLX Pipeline Stages
**DLX Datapath w/o pipelining**

**Events in DLX Pipe**

- **IF Stage**
  
  IF/ID IR = Mem[PC]
  
  IF/ID IR, PC = (if (EX/MEM.opcode == branch) & EX/MEM.cond)
  
  (EX/MEM.ALUOutput) else (PC+4))

- **ID Stage**
  
  ID/EX A = Regs[IF/ID IR, 6..10]; ID/EX B = Regs[IF/ID IR, 11..15]; ID/EX NPC = IF/ID IR;
  
  ID/EX IR = IF/ID IR; ID/EX IR, 16 = (IF/ID IR, 16) # Regs[IF/ID IR, 16..31]

**Events in DLX Pipe**

**Events in DLX Pipe**

- **MIB**
  
  MIB/MB IR = EZ/MEM.16; MIB/VB.16 = EZ/MEM.16;

  MIB/VB.16 = EZ/MEM.16;

  MIB/VB.ALUOutput f = Regs(EZ/MEM.ALUOutput); or

  Regs(EZ/MEM.ALUOutput) = EZ/MEM.8;
Events in DLX Pipe

- PC must be incremented on each clock
  - Do in IF
  - Separate incrementer as ALU is busy also
- A new instruction must be fetched every cycle
  - Done in IF
- A new data word is needed on every cycle
  - Done in MEM
- ALU output, instruction and next PC have to be saved as needed later in pipeline

Resource Implications for Datapath

Pipeline Hazards: Major Hurdles

- Structural Hazards
  - Resource conflicts
- Data Hazards
  - Data dependencies
- Control Hazards
  - branches and other instructions that change PC

Pipeline Speedup

\[
\text{Pipeline speedup} = \frac{1}{1 + \text{Pipeline Stall cycles per instruction} + \text{Pipeline depth}}
\]
Hazard

- Structural Hazard
- Data Hazard
- Control Hazard

Structural Hazard

- Some combination of instructions result in resource conflicts
  - Some functional units are not fully pipelined
  - Some resource is not duplicated enough
  - Register file write port
  - Single memory pipeline for instruction and data

- Stall pipeline for one cycle
  - Also called a Bubble

Pipeline Stall

<table>
<thead>
<tr>
<th>Inst No</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Single memory pipeline for data and instruction
Structural Hazard and Bubbles

Solutions to Structural Hazard

Resource Duplication example
- Separate I and D caches for memory access conflict
- Time-multiplexed or multiport register file for register file access conflict

Hazards
- Structural Hazard
- Data Hazard
- Control Hazard

Data Hazards
- Order of access to operands is changed by the pipeline vs the normal order

<table>
<thead>
<tr>
<th>Inst No</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Inst</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub Inst</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Types of Data Hazards

- **RAW (read after write)**
  - J tries to read a source before I writes it, so J may get wrong value
- **WAR (write after read)**
  - J tries to write a destination before it is read by I.
  - As reads are early this can not happen in our examples
  - Autoincrement addressing can create it
- **WAW (write after write)**
  - J tries to write an operand before it is written by I.
  - Wrong value may remain in the register.
  - Occurs in pipes which write in more than one stage

Data Hazards

- **WAR**
  - SW 0(R1), R2  
    - IF  
    - ID  
    - EX  
    - MEM1  
    - MEM2  
    - WB  
  - ADD R2, R3, R4  
    - IF  
    - ID  
    - EX  
    - WB
- **WAW**
  - LW R1, 0(R2)  
    - IF  
    - ID  
    - EX  
    - MEM1  
    - MEM2  
    - WB  
  - ADD R1, R2, R3  
    - IF  
    - ID  
    - EX  
    - WB

Solutions to Data Hazard

- **(Internal) Forwarding**
  - Extra hardware
- **Freezing the pipeline**
  - Stalls/bubbles; pipeline interlock
- **Compiler (instruction) scheduling**
  - Delay slot

Data Hazard

- ADD R1, R2, R3
- SUB R4, R5, R1
- AND R6, R1, R7
- OR R8, R1, R9
- XOR R10, R1, R11
Data Hazard

Forwarding Paths

Another Example

Solutions to Data Hazard

• (Internal) Forwarding
  – Extra hardware
• Freezing the pipeline
  – Stalls/bubbles; pipeline interlock
• Compiler (instruction) scheduling
  – Delay slot
Data Hazards Requiring Stalls

• Situations were forwarding is not possible
  – Load interlock

  LW  R1, 0(R2)
  SUB  R4, R1, R5
  AND  R6, R1, R7
  OR  R8, R1, R9

  → pipeline interlock hardware
  – Detects Hazard
  – Stalls the pipe until hazard is cleared

Load cannot bypass results to SUB

Pipeline Interlock Solution

Solutions to Data Hazard

• (Internal) Forwarding
  – Extra hardware

• Freezing the pipeline
  – Stalls/bubbles: pipeline interlock

• Compiler (instruction) scheduling
  – Delay slot
Compiler Scheduling

- Example
  \[ a = b + c; \]
  \[ d = e - f; \]

<table>
<thead>
<tr>
<th>Naive Code</th>
<th>Scheduled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW Ra, b</td>
<td>LW Ra, b</td>
</tr>
<tr>
<td>LW Rc, c</td>
<td>LW Rc, c</td>
</tr>
<tr>
<td>ADD Ra, Rb, Rc</td>
<td>ADD Ra, Rb, Rc</td>
</tr>
<tr>
<td>SW a, Ra</td>
<td>LW Rf, f</td>
</tr>
<tr>
<td>LW Re, e</td>
<td>SW a, Ra</td>
</tr>
<tr>
<td>LW Rf, f</td>
<td>SUB Rd, Re, Rd</td>
</tr>
<tr>
<td>SUB Rb, Re, Rd</td>
<td>SW Rr, Rd</td>
</tr>
<tr>
<td>SW d, Rd</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1, R6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW R2, R7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R3, R1, R2</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW R1, R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>