Pipelining

CIT 595
Spring 2010

Additional material © 2004/2005 Lewis/Martin
Modified by Diana Palsetia

Real-World Pipelines: Car Washes

Sequential

Parallel

Pipelined

Idea
- Divide process into independent stages
- Move objects through stages in sequence
- At any given times, multiple objects being processed

Laundry Example

- Ann, Brian, Cathy and Dave each have one load of clothes to wash, dry and fold
  - Washer takes 30 mins
  - Dryer takes 40 mins
  - "Folder" takes 20 mins

Sequential Laundry

Entire workload takes 6 hours to complete
Pipelined Laundry

- Pipelined Laundry takes only 3.5 hours
- Speedup = 6/3.5 = 1.7
- Pipelining did not reduce completion time for one task but it helps the throughput of the entire workload in turn decreasing the completion time

Instruction Level Pipelining: Big Picture

- Exploit idleness of instruction cycle to allow instructions to be executed in parallel
- Exploit parallelism at instruction level
- For each stage, one phase of instruction is carried out, and the stages are overlapped
- Ideally each stage of the Instruction Processing cycle takes 1 clock cycle

Theoretical Speedup due to Pipelining

- The theoretical speedup offered by a pipeline can be determined as follows:
  - Let $k$ be total number of stages and $t_p$ be the time per stage
  - Let $n$ be the total number of tasks
  - The first task (instruction) requires $k \times t_p$ time to complete in a $k$-stage pipeline.
  - The remaining ($n - 1$) tasks emerge from the pipeline one per cycle
    - Total time to complete the remaining tasks is $(n - 1)t_p$
  - Thus, to complete $n$ tasks using a $k$-stage pipeline requires:
    - $(k \times t_p) + (n - 1)t_p = (k + n - 1)t_p$

  Theoretical Speedup due to Pipelining

- If we take the time required to complete $n$ tasks without a pipeline and divide it by the time it takes to complete $n$ tasks using a pipeline, we find:
  \[ Speedup \, S = \frac{nt_p}{(k + n - 1)t_p} \]

- If we take the limit as $n$ approaches infinity, $(k + n - 1)$ approaches $n$, which results in a theoretical speedup of:
  \[ Speedup \, S = \frac{kt_p}{t_p} = k \]
Implementing a Pipeline

- Since we are overlapping stages
- Control + Data must be remembered per instruction and must be carried through each stage
- This is achieved by placing a n-bit register that can hold the control/data information in between each stage

Impact on Clock cycle time due to Pipelining

- Recall
- \[ \text{CPU Time} = \frac{\text{time program}}{\text{cycle time}} \times \frac{\text{cycles instruction}}{\text{CPI}} \times \frac{\text{instructions program}}{\text{Instruction Count}} \]

- Lower time per cycle → lower CPU time → improves performance
- This implies that we if we shorten the time per pipeline stages, we will lower clock cycle time
- This can be achieved by adding more pipe stages of shorter duration
Impact on Clock cycle time due to Pipelining

- 5 stage
- 10 stage – clock cycle time reduced by one half
- 20 stage – clock cycle time reduced by one fourth

Impact on CPI

- In pipelining, one instruction is in each stage
- Since one instruction will be fetched (or finish) each cycle, the average CPI will equal 1
  - ignoring the very first instruction – cold start
- However, CPI = 1 is barely achieved
  - Assume that the pipeline can be kept filled at all times
    - Structural Hazards
    - Data Hazards
    - Control Hazards

Structural Hazard

- Occurs when hardware cannot support a combination of instructions that we want to execute in parallel
  - In Laundry example: the machine has combined washer or dryer
  - In instruction pipelining: shared memory resources
    - Example: One memory for instruction & data
- Usually overcome by duplicating hardware
  - Memory is separated into instruction and data memory
  - Or memory/register is multi-ported i.e. memory that provides more than one access path to its contents

Data Hazard

- Occurs when an instruction depends on the results of a instruction still in the pipeline
- Example 1:
  - i1: ADD R1, R2, R3
  - i2: AND R5, R1, R4

- Example 2:
  - i1: ADD R1, R2, R3
  - i2: ST R1, A

- Example 3:
  - i1: LD R1, A
  - i2: ADD R2, R1, R2

RAW: Read After Write Dependencies
### Data Hazard due ALU Instruction

**i1:** ADD R1, R2, R3  
**i2:** AND R5, R1, R4

<table>
<thead>
<tr>
<th>Cycle</th>
<th>i1</th>
<th>i2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S1</td>
<td>S1</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S2</td>
</tr>
<tr>
<td>2</td>
<td>S3</td>
<td>S3</td>
</tr>
<tr>
<td>3</td>
<td>S4</td>
<td>S4</td>
</tr>
<tr>
<td>4</td>
<td>S5</td>
<td>S5</td>
</tr>
<tr>
<td>5</td>
<td>S6</td>
<td>S6</td>
</tr>
</tbody>
</table>

- **S1:** Instruction Fetch  
- **S2:** Decode  
- **S3:** Register Fetch  
- **S4:** Execute/EA  
- **S5:** Memory Access (LD/ST)  
- **S6:** Write Back (register write)

### Data Forwarding

- Realize that data value from i1 (to be put in R1) is actually available at end of cycle 4  
- Don’t wait till end of cycle 6 to fetch the register file, instead forward a copy of the data from S4 of i1 to i2’s stage S4

<table>
<thead>
<tr>
<th>Cycle</th>
<th>i1</th>
<th>i2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S1</td>
<td>S1</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S2</td>
</tr>
<tr>
<td>2</td>
<td>S3</td>
<td>S3</td>
</tr>
<tr>
<td>3</td>
<td>S4</td>
<td>S4</td>
</tr>
<tr>
<td>4</td>
<td>S5</td>
<td>S5</td>
</tr>
<tr>
<td>5</td>
<td>S6</td>
<td>S6</td>
</tr>
</tbody>
</table>

- **S1:** Instruction Fetch  
- **S2:** Decode  
- **S3:** Register Fetch  
- **S4:** Execute/EA  
- **S5:** Memory Access (LD/ST)  
- **S6:** Write Back (register write)

### Handling Data Forwarding

- Requires additional logic to data path  
- For input \( B \) to ALU, the existing MUX will needed to be expanded  
  - i.e.  \( i1: \) ADD(R1), R2, R3  
  - \( i2: \) ADD(R5, R4, R1)  
- Additional MUX needs to be placed to select between output of the register file and forwarded input for input A to ALU  
- Control will set the MUX control based on the hazard condition  
- Also, forwarded input need to be passed along other stages

### Data Hazard due to a Load Instruction

- Complete Data Forwarding not possible in this case

<table>
<thead>
<tr>
<th>Cycle</th>
<th>i1</th>
<th>i2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S1</td>
<td>S1</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S2</td>
</tr>
<tr>
<td>2</td>
<td>S3</td>
<td>S3</td>
</tr>
<tr>
<td>3</td>
<td>S4</td>
<td>S4</td>
</tr>
<tr>
<td>4</td>
<td>S5</td>
<td>S5</td>
</tr>
<tr>
<td>5</td>
<td>S6</td>
<td>S6</td>
</tr>
</tbody>
</table>

Value from memory (to be put in R1) is received from memory at end of cycle 5 for i1, but i2 needs value of R1 at beginning of cycle 4

- Stall for one cycle and then Forward

<table>
<thead>
<tr>
<th>Cycle</th>
<th>i1</th>
<th>i2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S1</td>
<td>S1</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S2</td>
</tr>
<tr>
<td>2</td>
<td>S3</td>
<td>S3</td>
</tr>
<tr>
<td>3</td>
<td>S4</td>
<td>S4</td>
</tr>
<tr>
<td>4</td>
<td>S5</td>
<td>S5</td>
</tr>
<tr>
<td>5</td>
<td>S6</td>
<td>S6</td>
</tr>
<tr>
<td>6</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Instead of instruction moving forward, a No-operation (NOP) is carried out such that nothing will happen in the pipeline stage

### Data Hazard due to Load Instruction

- Complete Data Forwarding not possible in this case
Control Hazards

- Only known after Decode Stage (S2)
- By then we have already fetched the next sequential instruction
- Branch address is resolved only in the Evaluate Address phase (S4)
- Stall the pipeline till we know which address to fetch from i.e. PC + 1 or Branch Target address
- Instruction before the branch will set Condition Code (NZP)
- One cycle before or the same cycle the branch address is resolved

Control Hazard Solution 1: Reduce the BR Delays

- Compiler resolves branching by rearranging the machine code to fill delay slots after BR instruction
- By inserting useful instruction that can be done without changing the output of the program
  - If no useful instructions then just put NOPs

Control Hazard Solution 2: Prediction

- Many ISAs also often predict the outcome of the branch
  1. In these ISAs address calculation of branch is coupled in the same phase as the branch is discovered
  2. There is prediction unit that records history of branch pattern
  3. Once the branch instruction is discovered, the prediction unit guides processor which instructions to fetch next
  4. The prediction unit is also updated with the actual outcome

Pipeline Hazard with Branch Instruction
Example: Two-bit Branch Predictor

- Keep 2-bit history value for each “recent” BR instruction
- Use 2-bit saturating counter
  - If branch is actually Taken (T), increment the history value
  - If Not Taken (NT), decrement the history value
  - 00 (Strongly NT), 01 (Weakly NT), 10 (Weakly T), 11 (Strongly T)

Typically > 90 % correct predictions

Solution to Multiple Exceptions in a Pipeline

- Maintain exception vector for each instruction
  - Vector is nothing but a n-bit register and each bit position indicating a stage of the pipeline
  - To indicate exception set the appropriate bit position (some hardware logic is needed here)

- When instruction enters the last stage of the pipeline, check the exception vector, and handle the exceptions in instruction order
  - If the bit is set means that the instruction had faulted
  - Abort all instructions following this instruction and restart from the excepting instruction
    - Known as implementing precise exceptions

Pipelining Complications

- Due to overlapping of instruction execution, multiple exceptions can occur in the same clock cycle

<table>
<thead>
<tr>
<th>Stage</th>
<th>Problem Exceptions Occurring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Memory-protection violation, Page fault on instruction fetch, misaligned memory access</td>
</tr>
<tr>
<td>Decode</td>
<td>Undefined Instruction</td>
</tr>
<tr>
<td>Execute</td>
<td>Arithmetic exception</td>
</tr>
<tr>
<td>Memory Access</td>
<td>Memory-protection violation, Page fault on instruction fetch, misaligned memory access</td>
</tr>
</tbody>
</table>

Summary of Pipelining

- Improves performance
  - Improves runtime of program
    - Reducing the clock cycle time
      - Increase Frequency (faster processing)
    - CPI = 1 (ideal)
  - Speedup = #number of pipe stages (ideal)

- However comes at price of greater CPI penalties
  - Data Hazard (Load-use delay)
  - Control Hazard (Branch/Jump delays)
  - Exceptions