How do we improve on the performance of the multi-cycle implementation?

Key observation -
- we can be doing multiple things at once

Pipelining -
- implementation technique to execute multiple instructions simultaneously
Pipelining is Natural!

Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- "Folder" takes 30 minutes
- "Stasher" takes 30 minutes to put clothes into drawers

Sequential Laundry
- We have four loads of laundry to do (A,B,C,D)
First, we wash.

Then we dry.
Sequential Laundry

° Now we fold....

° Finally we put the clothes away....
° It took us two hours to do one laundry...yikes!
° We have three loads remaining!
Whew, it’s 10 pm already and two loads to go.

We finish at 2 AM (half asleep)
Sequential laundry takes 8 hours for 4 loads
If they pipelined it, how long would laundry take?
Pipelined Laundry: Start work ASAP

° Let’s start to wash....

° Begin first load with washer
Pipelined Laundry: Start work ASAP

° Move first load to dryer
° Washer is empty, so we can start second load

° Fold first load, dry second load, start third load
Pipelined Laundry: Start work ASAP

- Stash first load, fold second, dry third, wash fourth

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.
- Multiple tasks operating simultaneously using different resources.
- Potential speedup = Number pipe stages.
- Pipeline rate limited by slowest pipeline stage.
- Unbalanced lengths of pipe stages reduces speedup.
- Time to “fill” pipeline and time to “drain” it reduces speedup.
- Stall for Dependences.

Pipelining for Instruction Execution

- Same concept applies for instructions!
- We can pipeline instruction execution.
- For MIPS, there are five classic steps:
  - **FETCH**: Fetch instruction from memory.
  - **DECODE**: Read registers while decoding instruction.
  - **EXECUTE**: Execute operation / calculate an address.
  - **MEMORY**: Access an operand in memory (L/S).
  - **WRITE BACK**: Write result into the register file.
Example - The Five Steps for a Load

- **Fetch**: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**: Calculate the memory address
- **Mem**: Read the data from the Data Memory
- **Wr**: Write the data back to the register file

Pipelining for Instruction Execution - Example

- Let’s consider a single-cycle vs. pipelined implementation of simple MIPS

<table>
<thead>
<tr>
<th>Class</th>
<th>Inst.</th>
<th>Reg</th>
<th>ALU</th>
<th>Mem.</th>
<th>Reg</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td>2 ns</td>
<td>1 ns</td>
<td>8 ns</td>
</tr>
<tr>
<td>Store</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td>2 ns</td>
<td></td>
<td>7 ns</td>
</tr>
<tr>
<td>R-type</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td></td>
<td>1 ns</td>
<td>6 ns</td>
</tr>
<tr>
<td>Branch</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td></td>
<td></td>
<td>5 ns</td>
</tr>
</tbody>
</table>

- For single cycle implementation, the cycle time is stretched to accommodate the slowest instruction
- Cycle time: 8 ns for single cycle implementation
**Single Cycle Implementation**

<table>
<thead>
<tr>
<th>Num.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>lw $1,100($0)</td>
</tr>
<tr>
<td>I2</td>
<td>lw $2,200($0)</td>
</tr>
<tr>
<td>I3</td>
<td>lw $3,300($0)</td>
</tr>
</tbody>
</table>

Time for each instruction is 8 ns - slowest time (for load)
Time between 1st and 4th instruction is 3 * 8 ns = 24 ns
Total time = 24 ns

**Pipelined Implementation**

<table>
<thead>
<tr>
<th>Num.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>lw $1,100($0)</td>
</tr>
<tr>
<td>I2</td>
<td>lw $2,200($0)</td>
</tr>
<tr>
<td>I3</td>
<td>lw $3,300($0)</td>
</tr>
</tbody>
</table>

Each step takes 2 ns (even reg file access) - slowest step is 2 ns
Time between 1st and 4th instruction: 3 * 2 ns = 6 ns
Total time for the three instructions = 14 ns
**Why Pipeline? Because the resources are there!**

**Time (clock cycles)**

---

**Instruction Order**

<table>
<thead>
<tr>
<th>Inst 0</th>
<th>Inst 1</th>
<th>Inst 2</th>
<th>Inst 3</th>
<th>Inst 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Im</td>
<td>Reg</td>
<td>Dm</td>
<td>Reg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Im</td>
<td>Dm</td>
<td>Reg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Im</td>
<td>Reg</td>
<td>Dm</td>
<td>Reg</td>
</tr>
<tr>
<td></td>
<td>Im</td>
<td>Reg</td>
<td>Dm</td>
<td>Reg</td>
</tr>
<tr>
<td></td>
<td>Im</td>
<td>Reg</td>
<td>Dm</td>
<td>Reg</td>
</tr>
</tbody>
</table>

**How does Pipelining Help?**

- Improves *instruction throughput*
- Assuming perfectly balanced stages (all stages take same amount of time):

  \[
  \text{Time between instructions pipeline} = \frac{\text{Time between instructions nonpipelined}}{\text{Number of pipeline stages}}
  \]

*Example: 8 ns for nonpipelined machine*

*What’s the time for five stage pipelined machine?*

\[
8 \text{ ns} / 5 = 1.6 \text{ ns}
\]
Wait Just One Minute!!!

Under ideal conditions -

**Speedup from pipelining equals the number of pipeline stages**

\[
speedup = \frac{\text{time nonpipelined}}{\text{time pipelined}} = \frac{8 \text{ ns}}{1.6 \text{ ns}} = 5
\]

But, remember the **maximum stage latency** is 2 ns

Hence, the speedup in this case is really:

\[
speedup = \frac{\text{time nonpipelined}}{\text{time pipelined}} = \frac{8 \text{ ns}}{2 \text{ ns}} = 4
\]

Wait Just One More Minute!!!

° Total time for the three loads was
  - 14 ns on pipelined version
  - 24 ns on nonpipelined version

How can you claim a 4 times speedup?

(Speedup here is 24 ns / 14 ns = 1.7)

Consider 1003 instructions:

Nonpipelined: \(1000 \times 8 \text{ ns} + 24 \text{ ns} = 8024 \text{ ns}\)

Pipelined: \(1000 \times 2 \text{ ns} + 14 \text{ ns} = 2014 \text{ ns}\)

\[8,024 \text{ ns} / 2,014 \text{ ns} = 3.98\]

= approx 8 ns / 2 ns
The Value of Pipelining

Improves performance -

By increasing instruction throughput
As opposed to decreasing execution time!!!

Consider our example for 1003 instructions:

Total program time is: 2,014 ns
But each instruction takes

\[ \text{# pipe stages} \times \text{cycle time} = \]
\[ = 5 \times 2 \text{ ns} \]
\[ = 10 \text{ ns} \]

This is longer than 8 ns for the single cycle version!

Pipelining Complications

° Situations when next instruction can not execute in the next cycle!

° Pipeline hazards - when an instruction is unable to execute (or advance in the pipeline)

° Three types of hazards:
    - Structural hazards
    - Data hazards
    - Control hazards
Structural Hazards

° **Structural hazards**: attempt to use the same resource two different ways at the same time

° Laundry example:
  - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)

° Instruction example:
  - With a single memory
    - Can be fetching an instruction
    - At same time doing a load
  - Only one read: a structural hazard

<table>
<thead>
<tr>
<th>Clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC1</td>
</tr>
<tr>
<td>Load IM Reg</td>
</tr>
<tr>
<td>CC2</td>
</tr>
<tr>
<td>CC3</td>
</tr>
<tr>
<td>CC4</td>
</tr>
<tr>
<td>CC5</td>
</tr>
<tr>
<td>CC6</td>
</tr>
</tbody>
</table>

**Structural Hazards (assuming a single memory)**
Dealing with Structural Hazards

- Arise from *lack of resources*
- We can *eliminate the hazard by adding more resources*!
  - In the previous example, we add a second memory (in effect, we will do this with cache - later in the semester)
  - Fetch and memory data read can happen at the same time
- Another solution:
  - Stall instruction until resource available
Data Hazards

- **Data hazards**: attempt to use item before it is ready
- Laundry example:
  - E.g., one sock of pair in dryer and one in washer; can’t fold until get sock from washer through dryer
- **Instruction execution**:
  - Instruction depends on result of prior instruction still in the pipeline
    
    \[
    \text{add} \quad s0, t0, t1\\
    \text{sub} \quad t2, s0, t3
    \]
    
    $s0$ produced by first add but needed by the second add

Data Hazards

- *Are data hazards common?*
  - You bet!!!
- Programs represent data flow between instructions and that data flow creates these dependences
- Hence, we must do something about data hazards!!
- One solution: Stall until value needed is written back to the register file and we can read it
- Penalty is too high with this solution
### Effect of Stalling on Data Hazard

<table>
<thead>
<tr>
<th>CC0</th>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
</tr>
</tbody>
</table>

- `add $s0,$t0,$t1`
- `sub $t2,$s0,$t3`
### Effect of Stalling on Data Hazard

<table>
<thead>
<tr>
<th></th>
<th>CC0</th>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>F</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>sub</td>
<td>F</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

**Improvement: Register Write in First Half of Cycle, Register Read in Second Half**

<table>
<thead>
<tr>
<th></th>
<th>CC0</th>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>F</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>sub</td>
<td>F</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

---

### Effect of Stalling on Data Hazard

<table>
<thead>
<tr>
<th></th>
<th>CC0</th>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>F</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>sub</td>
<td>F</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

**Improvement: Register Write in First Half of Cycle, Register Read in Second Half**

<table>
<thead>
<tr>
<th></th>
<th>CC0</th>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>F</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>sub</td>
<td>F</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
Effect of Stalling on Data Hazard

<table>
<thead>
<tr>
<th>CC0</th>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s0,$t0,$t1</td>
<td>F</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>sub $t2,$s0,$t3</td>
<td>F</td>
<td>Stall</td>
<td>Stall</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>ID</td>
</tr>
</tbody>
</table>

**Improvement: Register Write in First Half of Cycle, Register Read in Second Half**

<table>
<thead>
<tr>
<th>CC0</th>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s0,$t0,$t1</td>
<td>F</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>sub $t2,$s0,$t3</td>
<td>F</td>
<td>Stall</td>
<td>Stall</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>ID</td>
</tr>
</tbody>
</table>

In first half of CC4, write to reg file, and in second half of CC4, read from reg file.

A Better Solution: Forwarding

- Write/Read register file in different half of cycle
- **Forwarding** on ALU output
  - Add path from ALU back to one of its inputs!
A Better Solution: Forwarding

- Write/Read register file in different half of cycle
- **Forwarding** on ALU output
  - Add path from ALU back to one of its inputs!

**Def:** *Forwarding* passes result from later stage to an earlier one

---

Forwarding Memory Result

- Just like we forward from ALU
  - The result from a load may be needed by the very next instruction
  - Hence, we need a forwarding path

---

The value needed by the sub isn’t read from the reg file - it comes directly from the result output from doing the add operation.
Forwarding Memory Result

- Just like we forward from ALU
  - The result from a load may be needed by the very next instruction
  - Hence, we need a forwarding path

- lw $s0,20($t1)
- sub $t2,$s0,$t3

Have to stall one cycle - the loaded value isn’t available until DM

Control Hazards

- Control hazards: attempt to make a decision before condition is evaluated
- Laundry example:
  - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
- Instruction execution:
  - Branch instructions
    - beq $1,$2,L0
    - add $4,$5,$6
    - ...
    - L0: sub $7,$8,$9

Which instruction do we fetch next into the pipe following the branch??
Dealing with Control Hazards

° We can stall until branch outcome is known
  • Once branch is known, then fetch
  • But this is wasteful

\[ \text{Time (ns)} \]

\begin{align*}
\text{add} & \ $4,\$5,\$6 \\
\text{beq} & \ $1,\$2,40 \\
\text{lw} & \ $3,300(\$0)
\end{align*}

\[ \text{2ns} \quad \text{4ns} \quad \text{2ns} \]

Dealing with Control Hazards - Predict Branch

° Predict that the branch is not taken
  • Attempt to get next instruction from the fall thru of the branch (i.e., next sequential address)

° We are gambling that the branch isn’t ever going to be taken

° When we’re right - there is no stall
° But what happens when we’re wrong???
Predicting Branch as Not Taken

The three implementations

CPU time = IC × CPI × CC

For same instruction set (IC same):

*Single cycle*: CPI = 1, long CC

*Multi cycle*: CPI > 1, probably 3-4, short CC

*Pipelined*: CPI > 1, probably 1.2-1.4, short CC
Let’s compare

° Suppose 5-step MIPS implementation
  • Single cycle: 10 ns
  • Multi-cycle: 3.9 CPI, 2 ns
  • Pipelined: 1.2 CPI, 2 ns

° What is the speedup of
  • Multi-cycle vs. single cycle
  • Pipelined vs. multi-cycle
  • Pipelined vs. single cycle

Multi-cycle vs single cycle

° CPU time single = IC × 1 × 10ns = IC × 10 ns
° CPU time multi = IC × 3.9 × 2ns = IC × 7.8 ns

° Speedup of multi vs. single cycle
  Speedup = IC × 10 ns / IC × 7.8 ns =
  = 10 ns / 7.8 ns
  = 1.28x
Pipelined vs. multi cycle

° CPU time multi-cycle = IC × 3.9 × 2ns = IC × 7.8ns
° CPU time pipeline = IC × 1.2 × 2ns = IC × 2.4ns

° Speedup of pipelined vs. multi-cycle
  Speedup = IC × 7.8ns / IC × 2.4ns = 3.25x

° Speedup of pipelined vs. single cycle
  Speedup = IC × 10ns / IC × 2.4ns = 4.17x

The End

Thank you!