CPI Calculation

- CPI stands for average number of **Cycles Per Instruction**
- Assume an instruction mix of 24% loads, 12% stores, 44% R-format, 18% branches, and 2% jumps
- \[ \text{CPI} = 0.24 \times 5 + 0.12 \times 4 + 0.44 \times 4 + 0.18 \times 3 + 0.02 \times 3 = 4.04 \]
- Speedup?
- Question: Can we achieve a CPI of 1???
Speeding up through pipelining

- 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

- Sequential laundry takes 8 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!

### 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
### Pipelining Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fetch (10 ns)</th>
<th>Decode (6 ns)</th>
<th>Execute (8 ns)</th>
<th>Memory (10 ns)</th>
<th>Write back (6 ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F D EX M W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F D EX M W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F D EX M W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F D EX M W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F D EX M W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Single Cycle, Multiple Cycle, vs. Pipeline

#### Single Cycle Implementation:
- Load in Cycle 1
- Store in Cycle 3
- Waste in Cycle 2

#### Multiple Cycle Implementation:
- Load in Cycle 1
- Store in Cycle 5
- Fetch in Cycle 2
- Reg in Cycle 3
- Exec in Cycle 4
- Mem in Cycle 6
- Wr in Cycle 8
- R-type in Cycle 7

#### Pipeline Implementation:
- Load: Ifetch, Reg, Exec, Mem, Wr
- Store: Ifetch, Reg, Exec, Mem, Wr
- R-type: Ifetch, Reg, Exec, Mem, Wr
Why Pipeline?

- Suppose we execute 100 instructions
- Single Cycle Machine
  - $45 \text{ ns/cycle} \times 1 \text{ CPI} \times 100 \text{ inst} = 4500 \text{ ns}$
- Multicycle Machine
  - $10 \text{ ns/cycle} \times 4.04 \text{ CPI (for the given inst mix)} \times 100 \text{ inst} = 4040 \text{ ns}$
  - Instruction mix of 24% loads, 12% stores, 44% R-format, 18% branches, and 2% jumps
- Ideal pipelined machine (with 5 stages)
  - $10 \text{ ns/cycle} \times (1 \text{ CPI x 100 inst + 4 cycle drain}) = 1040 \text{ ns}$
- Speedup=4.33 vs. single-cycle
- 3.88 vs. multi-cycle (for the given inst mix)

Why Pipeline? Because the resources are there!
Pipelining Rules

- Forward traveling signals at each stage are latched
- Only perform logic on signals in the same stage
  - signal labeling useful to prevent errors,
  - e.g., IR\textsubscript{R}, IR\textsubscript{A}, IR\textsubscript{M}, IR\textsubscript{W}
- Backward travelling signals at each stage represent hazards

MIPS Pipelined Datapath

- **State registers** between pipeline stages to isolate them
### Pipeline Hazards

- **Data hazards**: an instruction uses the result of a previous instruction (RAW)
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register 1</th>
<th>Register 2</th>
<th>Register 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>R1, R2, R3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>R4, R1, R5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>R1, 4(R2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>R3, 4(R2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Control hazards**: the address of the next instruction to be executed depends on a previous instruction
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register 1</th>
<th>Register 2</th>
<th>Register 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>R1, R2, CONT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>R6, R7, R8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  ... CONT: ADD R3, R4, R5

- **Structural hazards**: two instructions need access to the same resource
  
  - e.g., single memory shared for instruction fetch and load/store

### Structural Hazard

**Time (clock cycles)**

- **Inst 1**: Reading data from memory
- **Inst 2**: Reading instruction from memory

- **Fix with separate instruction and data memories (I$ and D$)**
Data Hazards (RAW)

Time (in cycles)

Instruction

F D EX M W

Write Data to R1 Here

F D EX M W

Get data from R1 Here

ADD R1, R2, R3
SUB R4, R1, R5

One Way to handle a Data Hazard

Instr. Order

add $1, ...

stall

stall

stall

sub $4, $1, $5

By waiting – introducing stalls – but impacts CPI
**Must allow Wr/Rd in REG in same cycle**

Split cycle into two halves

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst 1</td>
</tr>
<tr>
<td><img src="image1.png" alt="Diagram of pipeline stages" /></td>
</tr>
</tbody>
</table>

**Only two stall cycles**

<table>
<thead>
<tr>
<th>Instr. Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $1, ...</td>
</tr>
<tr>
<td>stall</td>
</tr>
<tr>
<td>stall</td>
</tr>
<tr>
<td>sub $4, $1, $5</td>
</tr>
<tr>
<td>and $6, $1, $7</td>
</tr>
<tr>
<td><img src="image2.png" alt="Diagram of pipeline stages" /></td>
</tr>
</tbody>
</table>

Write in 1st half, Read in 2nd half
Another Way to “Fix” a Data Hazard

- **add $1, …**
- **sub $4, $1, $5**
- **and $6, $1, $7**
- **or $8, $1, $9**
- **xor $4, $1, $5**

```
add $1, ...
sub $4, $1, $5
and $6, $1, $7
or $8, $1, $9
xor $4, $1, $5
```

Register File (write and then read)

- **add $1, …**
- **Inst 1**
- **Inst 2**
- **or $8, $1, $9**

Fix register file access hazard by doing reads in the second half of the cycle and writes in the first half.

```
add $1, ...
Inst 1
Inst 2
or $8, $1, $9
```
Internal data forwarding

Fix data hazards by forwarding results as soon as they are available to where they are needed.

ALU-to-ALU forwarding vs. full forwarding

Forwarding with Load-use Data Hazards

- sub needs to stall
- Will still need one stall cycle even with forwarding
3 Types of Data Hazards

- **RAW (read after write)**
  - only hazard for ‘fixed’ pipelines
  - later instruction must *read* after earlier instruction *writes*

- **WAW (write after write)**
  - variable-length pipeline
  - later instruction must *write* after earlier instruction *writes*

- **WAR (write after read)**
  - instruction with late read (e.g., waiting for an execution unit)
  - later instruction must *write* after earlier instruction *reads*
**Control Hazard**

Simple solution: Flush Instruction fetch until branch resolved