4.5 Pipelining

- Overlapped execution of instructions
- Instruction level parallelism (concurrency)
- Example pipeline: assembly line (“T” Ford)
- Response time for any instruction is the same
- Instruction throughput increases 😊
- Speedup = $k \times$ number of steps (stages)
  - Theory: $k$ is a large constant
  - Reality: Pipelining introduces overhead

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
  - “Storer” 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
- Stall for Dependences

Five-stages of multi-cycle DP

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>IR = Memory[PC]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decoder-fetch</td>
<td></td>
<td>A = Reg[IR(5:2)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B = Reg[IR(0:3)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUOut = PC + (sign-extend [IR(15:0)] × 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address, comp.</td>
<td>ALUOut = A + B</td>
<td>ALUOut = A + sign-extend (RI(15:0))</td>
<td>If (A = 0) then PC = ALUOut</td>
<td></td>
</tr>
<tr>
<td>Execution, branch/jump</td>
<td></td>
<td></td>
<td>PC = PC[31-23] × 2 (IR(25:3) × 2)</td>
<td></td>
</tr>
<tr>
<td>Memory access or R-type</td>
<td>Reg[IR(15:11)] = ALUOut</td>
<td>Load MDR = Memory[ALUOut]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td></td>
<td>or Store MDR = ALUOut = Reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td></td>
<td>Load Reg[Reg(10:8)] = MDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Five Stages of the Load Instruction

- **Ifetch**: 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

Pipelined Execution

- On a processor multiple instructions are in various stages at the same time.
- Assume each instruction takes five cycles
Single Cycle, Multiple Cycle, vs. Pipeline

**Single Cycle Implementation:**
- Cycle 1
  - Load
  - Ifetch
  - Reg
  - Exec
  - Mem
  - Wr
- Cycle 2
  - Store
  - Waste

**Multiple Cycle Implementation:**
- Cycle 1
  - Load
  - Ifetch
  - Reg
  - Exec
  - Mem
  - Wr
- Cycle 2
- Cycle 3
- Cycle 4
- Cycle 5
- Cycle 6
- Cycle 7
- Cycle 8
- Cycle 9
- Cycle 10

**Pipeline Implementation:**
- Load
  - Ifetch
  - Reg
  - Exec
  - Mem
  - Wr
- Store
  - Ifetch
  - Reg
  - Exec
  - Mem
  - Wr
- R-type
  - Ifetch
  - Reg
  - Exec
  - Mem
  - Wr

Why Pipeline?

- Suppose
  - 100 instructions are executed (IC)
  - The single cycle machine has a cycle time of 45 ns (CC)
  - The multicycle and pipeline machines have cycle times of 10 ns
  - The multicycle machine has a CPI of 4.2
- Single Cycle Machine
  - 100 inst x 1 CPI x 45 ns/cycle = 4500 ns
- Multicycle Machine
  - 100 inst x 4.2 CPI x 10 ns/cycle = 4200 ns
- Ideal pipelined machine
  - (100 inst x 1 CPI + 4 cycle overhead) x 10 ns/cycle = 1040 ns
- Ideal pipelined vs. single cycle speedup
  - 4500 ns / 1040 ns = 4.33
- What has not yet been considered?
Graphically Representing Pipelines

- Can help with answering questions like:
  - How many cycles does it take to execute this code?
  - What is the ALU doing during cycle 4?
  - Are two instructions trying to use the same resource at the same time?

Why Pipeline? Because the resources are there!
Can pipelining get us into trouble?

- **Yes: Pipeline Hazards**
  - **structural hazards**: attempt to use the same resource two different ways at the same time
    - Two instructions use the memory at the same time
  - **data hazards**: attempt to use item before it is ready
    - Instruction depends on result of prior instruction still in the pipeline
  - **control hazards**: attempt to make a decision before condition is evaluated (branch instructions)
- Can always resolve hazards by **waiting**
  - Pipeline control must detect the hazard
  - Take action (or delay action) to resolve hazards

---

**Structural Hazard 1: Single Memory**

IM = DM => Read same memory twice in one clock cycle
Structural Hazard 2: Register File

Try read and write to registers simultaneously

Structural Hazards: Solutions

• Structural hazard 1: single memory
  — Two memories? infeasible and inefficient
    => Two Level 1 caches (instruction and data)
• Structural hazard 2: register file
  — Register access takes less than ½ ALU stage time
    => Use the following convention:
      • Always Write during first half of each cycle
      • Always Read during second half of each cycle
  — Both, Read and Write can be performed during the same clock cycle (a small delay between)
Control Hazard: Branch Instr. (1/2)

- Branch decision-making hardware in ALU stage
  - Two more instructions after the branch will *always* be fetched, whether or not the branch is taken
- Desired functionality of a branch
  - if we do not take the branch, don’t waste any time and continue executing normally
  - if we take the branch, don’t execute any instructions after the branch, just go to the desired label

Control Hazard: Branch Instr. (2/2)

- **Initial Solution**: Stall until decision is made
  - Insert “no-op” instructions: those that accomplish nothing, just take time
  - Drawback: branches take 3 clock cycles each (assuming comparator is put in ALU stage)

<table>
<thead>
<tr>
<th>L1: ADD R2, R2, #2</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB R3, R1, #3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>BEQ R5, R4, L1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>ADD R2, R2, #2</td>
<td>S</td>
<td>S</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

\[ S = \text{stall} \]
Branch Prediction

• Longer pipelines can’t readily determine branch outcome early
  — Stall penalty becomes unacceptable
• Predict outcome of branch
  — Only stall if prediction is wrong
• In MIPS pipeline
  — Can predict branches not taken
  — Fetch instruction after branch, with no delay

MIPS with Predict Not Taken
More-Realistic Branch Prediction

• Static branch prediction
  — Based on typical branch behavior
  — Example: loop and if-statement branches
    • Predict backward branches taken
    • Predict forward branches not taken

• Dynamic branch prediction
  — Hardware measures actual branch behavior
    • e.g., record recent history of each branch
  — Assume future behavior will continue the trend
    • When wrong, stall while re-fetching, and update history

Data Hazard

• Instruction depends on result of prior instruction still in the pipeline

  add \texttt{r1},r2,r3
  sub r4, \texttt{r1},r3
  and r6, \texttt{r1},r7
  or r8, \texttt{r1},r9
  xor r10, \texttt{r1},r11

• Problem: \texttt{r1} cannot be read by other instructions before it is written by the add.
Data Hazard on r1:

- Dependencies backwards in time are hazards

<table>
<thead>
<tr>
<th>Time (clock cycles)</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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $r1$,r2,r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $r4$,r1,r3</td>
<td></td>
<td></td>
<td>S</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and $r6$,r1,r7</td>
<td></td>
<td>S</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or  $r8$,r1,r9</td>
<td></td>
<td></td>
<td>ID</td>
<td>EX</td>
<td></td>
<td>MEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor $r10$,r1,r11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data hazard timing

<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>add $r1$,r2,r3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>sub $r4$,r1,r3</td>
<td>IF</td>
<td>S</td>
<td>S</td>
<td>ID</td>
<td></td>
</tr>
<tr>
<td>and $r6$,r1,r7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td>MEM</td>
</tr>
<tr>
<td>or  $r8$,r1,r9</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$S = \text{stall}$

How to delete “Stalls”?
Data Hazard Solution:

• "Forward" result from one stage to another

<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>add r1, r2, r3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>sub r4, r1, r3</td>
<td>IF</td>
<td>S</td>
<td>S</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>and r6, r1, r7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>or r8, r1, r9</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
</tbody>
</table>

Data hazard timing

Without forwarding

With forwarding
Forwarding (or Bypassing): What about Loads

- Dependencies **backwards** in time are hazards

\[
\begin{align*}
&\text{lw } r1, 0(r2) \\
&\text{sub } r4, r1, r3
\end{align*}
\]

- Can’t solve with forwarding:
  - Must delay/stall instruction dependent on loads

**Data hazard requiring stalls**

**Without forwarding**

<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>LD R1, 0(R2)</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB R4, R1, R5</td>
<td>Stall</td>
<td></td>
<td></td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>AND R6, R1, R7</td>
<td>IF</td>
<td></td>
<td></td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>OR R8, R1, R9</td>
<td>IF</td>
<td></td>
<td></td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

**With forwarding**

<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>LD R1, 0(R2)</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB R4, R1, R5</td>
<td>IF</td>
<td></td>
<td></td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>AND R6, R1, R7</td>
<td>Stall</td>
<td>IF</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>OR R8, R1, R9</td>
<td>Stall</td>
<td>IF</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>
Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle
Pipeline Operation

• Cycle-by-cycle flow of instructions through the pipelined datapath
  – “Single-clock-cycle” pipeline diagram
    • Shows pipeline usage in a single cycle
    • Highlight resources used
  – c.f. “multi-clock-cycle” diagram
    • Graph of operation over time
• We’ll look at “single-clock-cycle” diagrams for load & store

IF for Load, Store, …
MEM for Load

WB for Load
WB for Store

Multi-Cycle Pipeline Diagram

- Form showing resource usage
Multi-Cycle Pipeline Diagram

- Traditional form

Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle