Lecture 10 -

- Today's objectives:
  - No more laundry 🧺
- What does pipelining help with?

Instruction execution review

- Executing a MIPS instruction can take up to five steps.

<table>
<thead>
<tr>
<th>Step</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>IF</td>
<td>Read an instruction from memory.</td>
</tr>
<tr>
<td>Instruction Decode</td>
<td>ID</td>
<td>Read source registers and generate control signals.</td>
</tr>
<tr>
<td>Execute</td>
<td>EX</td>
<td>Compute an R-type result or a branch outcome.</td>
</tr>
<tr>
<td>Memory</td>
<td>MEM</td>
<td>Read or write the data memory.</td>
</tr>
<tr>
<td>Writeback</td>
<td>WB</td>
<td>Store a result in the destination register.</td>
</tr>
</tbody>
</table>

- However, as we saw, not all instructions need all five steps.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Steps required</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>IF ID EX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>IF ID EX WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: Instruction Fetch (IF)

- Let's quickly review how IF is executed in the single-cycle datapath.
- We'll ignore PC incrementing and branching for now.
- In the Instruction Fetch (IF) step, we read the instruction memory.

Instruction Decode (ID)

- The Instruction Decode (ID) step reads the source registers from the register file.

Execute (EX)

- The third step, Execute (EX), computes the effective memory address from the source register and the instruction's constant field.

Memory (MEM)

- The Memory (MEM) step involves reading the data memory, from the address computed by the ALU.
Writeback (WB)

- Finally, in the Writeback (WB) step, the memory value is stored into the destination register.

A bunch of lazy functional units

- Notice that each execution step uses a different functional unit.
- In other words, the main units are idle for most of the 8ns cycle.
  - The instruction RAM is used for just 2ns at the start of the cycle.
  - Registers are read once in ID (1ns), and written once in WB (1ns).
  - The ALU is used for 2ns near the middle of the cycle.
- Reading the data memory only takes 2ns as well.
- That’s a lot of hardware sitting around doing nothing.

Putting those slackers to work

- We shouldn’t have to wait for the entire instruction to complete before we can re-use the functional units.
- For example, the instruction memory is free in the Instruction Decode step as shown below, so...

Decoding and fetching together

- Why don’t we go ahead and fetch the next instruction while we’re decoding the first one?

Executing, decoding and fetching

- Similarly, once the first instruction enters its Execute stage, we can go ahead and decode the second instruction.
- But now the instruction memory is free again, so we can fetch the third instruction.

Making Pipelining Work

- We’ll make our pipeline 5 stages long, to handle load instructions as they were handled before.
  - Stages are: IF, ID, EX, MEM, and WB.
- We want to support executing 5 instructions simultaneously: one in each stage.
  - Hmm, we’ll need something
### Pipelining Performance

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock Cycle</th>
<th>Filling</th>
<th>Execution Time on Ideal Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t0, 405(p)</td>
<td>1 2 3 4 5 6 7 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $t1, 105(p)</td>
<td>1 2 3 4 5 6 7 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $t2, 125(p)</td>
<td>1 2 3 4 5 6 7 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $t3, 205(p)</td>
<td>1 2 3 4 5 6 7 8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution time on ideal pipeline:
- Time to fill the pipeline: one cycle per instruction
- N instructions = N cycles = (N + 3) in for 2ns clock period

Compare with other implementations:
- Single Cycle: N cycles or 3N in for 2ns clock period
- Pipelined: N/2 cycles or N/2 in for 2ns clock period
- How much faster is pipelining for N=1000?

### Pipeline Datapath: Resource Requirements

- We need to perform several operations in the same cycle.
  - Increment the PC and add registers at the same time.
  - Fetch one instruction while another one reads or writes data.
- Thus, like the single-cycle datapath, a pipelined processor duplicates hardware elements that are needed several times in the same clock cycle.
Pipelining other Instruction types

- R-type instructions only require 4 stages: IF, ID, EX, and WB
  - We don’t need the MEM stage
- What happens if we try to pipeline loads with R-type instructions?

\[
\begin{array}{c|c|c|c|c}
\text{add} & 1 & 2 & 3 & 4 \\
\text{sub} & 5 & 6 & 7 & 8 \\
\text{lw} & 9 & 10 & 11 & 12 \\
\text{or} & 13 & 14 & 15 & 16 \\
\text{lw} & 17 & 18 & 19 & 20 \\
\end{array}
\]

A solution: Insert NOP stages

- Enforce uniformity
  - Make all instructions take 4 cycles
  - Make them have the same stages, in the same order
  - Some stages will do nothing for some instructions

\[
\begin{array}{c|c|c|c|c|c}
\text{add} & 1 & 2 & 3 & 4 & 5 \\
\text{sub} & 6 & 7 & 8 & 9 & 10 \\
\text{lw} & 11 & 12 & 13 & 14 & 15 \\
\text{or} & 16 & 17 & 18 & 19 & 20 \\
\text{lw} & 21 & 22 & 23 & 24 & 25 \\
\end{array}
\]

Summary

- Pipelining attempts to maximize instruction throughput by overlapping the execution of multiple instructions.
- Pipelining offers amazing speedup.
  - In the best case, one instruction finishes on every cycle, and the speedup is equal to the pipeline depth.
- The pipeline datapath is much like the single-cycle one, but with added pipeline registers
  - Each stage needs its own functional units
  - Next time we’ll see the datapath and control, and walk through an example execution.