Lecture 10

- Objectives:
  - Intro to Pipelining

- What is the most boring human activity ever?

A relevant question

- Assumptions you've got:
  - One washer (takes 30 minutes)
  - One drier (takes 40 minutes)
  - One “folder” (takes 20 minutes)

- It takes 90 minutes to wash, dry, and fold 1 load of laundry.
  - How long does 4 loads take?

\[ 360 \text{ min} = 6 \text{ hrs} = 1 \text{ 1/2 days} \]

The slow way

- If each load is done sequentially it takes 6 hours

Laundry Pipelining

- Start each load as soon as possible
  - Overlap loads

- Pipelined laundry takes 3.5 hours

Pipelining Lessons

- Pipelining doesn't help latency of single load, it helps throughout of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "draw" it reduces speedup

Pipelining

- Pipelining is a general-purpose efficiency technique
  - It is not specific to processors

- Pipelining is used in:
  - Assembly lines
  - Bucket brigades
  - Fast food restaurants

- Pipelining is used in other CS disciplines:
  - Networking
  - Server software architecture

- Useful to increase throughput in the presence of long latency
  - More on that later...
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>IF</td>
<td>Instruction Fetch</td>
<td>Read an instruction from memory.</td>
</tr>
<tr>
<td>ID</td>
<td>Instruction Decode</td>
<td>Read source registers and generate control signals.</td>
</tr>
<tr>
<td>EX</td>
<td>Execute</td>
<td>Compute an R-type result or a branch outcome.</td>
</tr>
<tr>
<td>WB</td>
<td>Writeback</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>
</tr>
</thead>
<tbody>
<tr>
<td>lw, sw</td>
<td>IF ID EX HMEM</td>
</tr>
<tr>
<td>br, beq, bne</td>
<td>IF ID EX WB</td>
</tr>
</tbody>
</table>

Single-cycle review

- All five execution steps occur in one clock cycle.

- This means the cycle time must be long enough to accommodate all the steps of the most complex instruction—a “lw” in our instruction set.
  - If the register file has a 1ns latency and the memories and ALU have a 2ns latency, “lw” will require 3ns.
  - Thus all instructions will take ≥3ns to execute.

- Each hardware element can only be used once per clock cycle.
  - A “lw” or “beq” must access memory twice (in the IF and HMEM stages), so there are separate instruction and data memories.
  - There are multiple adders, since each instruction increments the PC (IF) and performs another computation (ID). On top of that, branches also need to compute a target address.

Example: Instruction Fetch (IF)

- Let’s quickly review how lw 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.
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 3ns 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 in the multi-cycle implementation.
- Stages are: IF, ID, EX, MEM, and WB.
- We want to support executing 5 instructions simultaneously; one in each stage.

Pipelining Loads

<table>
<thead>
<tr>
<th>Clock cycle</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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t0, 45(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>lw $t1, 57(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>lw $t2, 102(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>lw $t3, 45(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>lw $t4, 20(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
</tbody>
</table>

6 PM

Time:

30 40 40 40 40 20

Pipelining Performance

<table>
<thead>
<tr>
<th>Clock cycle</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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t0, 45(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>lw $t1, 57(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>lw $t2, 102(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>lw $t3, 45(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>lw $t4, 20(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
</tbody>
</table>

| A pipeline diagram shows the execution of a series of instructions.
- The instruction sequence is shown vertically, from top to bottom.
- Clock cycles are shown horizontally, from left to right.
- Each instruction is divided into its component stages. (We show five stages for every instruction, which will make the control unit easier.)
- This clearly indicates the overlapping of instructions. For example, there are three instructions active in the third cycle above.
- The "lw" instruction is in its Execute stage.
- Simultaneously, the "add" is in its Instruction Decode stage.
- Also, the "add" instruction is just being fetched.

Pipeline terminology

<table>
<thead>
<tr>
<th>Clock cycle</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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t0, 45(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>lw $t1, 57(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>lw $t2, 102(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>lw $t3, 45(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>lw $t4, 20(gp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
</tbody>
</table>

- The pipeline depth is the number of stages—in this case, five.
- In the first four cycles, the pipeline is filling, since there are unused functional units.
- In cycle 5, the pipeline is full. Five instructions are being executed simultaneously, so all hardware units are in use.
- In cycles 6-9, the pipeline is emptying.
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
- What happens if we try to pipeline loads with R-type instructions?

A solution: Insert NOP stages

- Enforce uniformity
  - Make all instructions take 5 cycles.
  - Make them have the same stages, in the same order
  - Some stages will do nothing for some instructions

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 new functional units
- Next time we’ll see the datapath and control, and walk through an example execution.