Pipelining Analogy

Pipelined laundry: overlapping execution

- Parallelism improves performance

Four loads:
- serial throughput: 0.5 load/hr
- pipelined throughput: 1.14 load/hr
- speedup: $\frac{8}{3.5} \approx 2.3$

Non-stop speedup:
$$\frac{2n}{(0.5n + 1.5)} \approx 4$$
Basic Idea

What if we think of the simple datapath as a linear sequence of stages?

We have 5 stages, which will mean that on any given cycle up to 5 different instructions will be in various points of execution.

Can we operate the stages independently, using an earlier one to begin the next instruction before the previous instruction has completed?
<table>
<thead>
<tr>
<th>Stage</th>
<th>Actions</th>
<th>For</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Instruction fetch from memory</td>
<td>all</td>
</tr>
<tr>
<td>ID</td>
<td>Instruction decode &amp; register read</td>
<td>decode for all; read for all but j</td>
</tr>
<tr>
<td>EX</td>
<td>Execute operation or calculate address</td>
<td>all but j</td>
</tr>
<tr>
<td>MEM</td>
<td>Access memory operand</td>
<td>lw, sw</td>
</tr>
<tr>
<td>WB</td>
<td>Write result back to register</td>
<td>lw, R-type</td>
</tr>
</tbody>
</table>
Timing Assumptions

Assume time for stages is
- 100ps for register read or write
- 200ps for other stages

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction fetch</th>
<th>Register read</th>
<th>ALU operation</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
<tr>
<td>j</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>

QTP: how does j fit in here?
Pipeline Timing Details

Each stage is allotted 200ps, and so that is the cycle time. That leads to "gaps" in stages 2 and 5:

We stipulate that register writes take place in the first half of a cycle and that register reads take place in the second half of a cycle.

QTP: why?
Non-pipelined Performance

Single-cycle ($T_c = 800\text{ps}$)

Program execution order (in instructions)

<table>
<thead>
<tr>
<th>Time</th>
<th>200</th>
<th>400</th>
<th>600</th>
<th>800</th>
<th>1000</th>
<th>1200</th>
<th>1400</th>
<th>1600</th>
<th>1800</th>
</tr>
</thead>
</table>

- `lw $1, 100(0)`
- `lw $2, 200(0)`
- `lw $3, 300(0)`

Total time to execute 3 instructions would be 2400 ps.

Total time to execute N instructions would be $800N$ ps.
Pipelined Performance

Pipelined ($T_c = 200\text{ps}$)

Total time to execute these 3 instructions would be 1400 ps.

Speedup would be $2400/1400$ or about 1.7.

Total time to execute $N$ (similar) instructions would be $800 + 200N$ ps.

Speedup would be $800N/(800+200N)$ or about 4 for large $N$.
Pipeline Speedup

If all stages are balanced (i.e., all take the same time):

\[
\text{Time between instr completions}_{\text{pipelined}} = \frac{\text{Time between instr completions}_{\text{non-pipelined}}}{\text{Number of stages}}
\]

If not balanced, speedup is less

Speedup is due to increased throughput

- Latency (time for each instruction) does not decrease
- In fact…

Note: the goal here is to improve overall performance, which is often not the same as optimizing the performance of any particular operation.
MIPS32 ISA was designed for pipelining:

- 32-bit machine instructions (uniformity)
  - easier to fetch and decode in one cycle
  - vs x86: machine instructions vary from 1 to 17 bytes

Few, regular instruction formats
- can decode opcode and read registers in same clock cycle

Load/store addressing
- can calculate address in one pipeline stage…
  - … and access data memory in the next pipeline stage

Alignment requirements for memory operands
- 4-byte accesses must be at “word” addresses
- memory access takes only one clock cycle

QTP: what if we had to support:

\[ \text{add} \ 4(t0), \ 12(t1), \ -8(t2) \]
But… is there anything wrong with our thinking?
What about handling:

```
beq    $s0, $s1, exit
j      exit
lw    $s0, 12($s1)
add   $s3, $s0, $s1
add   $s4, $s0, $s5
```

Are there any other issues…?