Pipelining

**Intro**

**Single cycle pros and cons**

**Con**  Inefficient: clock must be timed to accommodate the slowest instruction.

<table>
<thead>
<tr>
<th>Clock</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Con**  May be wasteful of area since some functional units (e.g., adders) must be duplicated since they cannot be shared during a clock cycle.

**Big Pro**  Simple and easy to understand

**Optimizations next: pipelining, caches**

**Laundry**

1. Wash
2. Dry
3. Fold
4. Stow
Wellesley is a pipeline. (4 classes / 7 years)

Pipelining and Performance

- **Latency** = time from start to finish for one unit of work
- **Throughput** = number of units of work completed per unit time
- **Stage** = one well-defined step toward completing a unit of work

**Pipelining** = run all stages at same time, on different units of work
- like assembly line
- single-unit latency unchanged: single unit of work completes no faster with pipelining
- throughput improved: more units of work may be completed per unit time
- many-unit latency improved: large sequences of work may complete faster with pipelining

**Instruction Pipelining**

- **Instruction latency** = time to fully execute one instruction
- **Instruction throughput** = instructions executed per unit time
- **Instruction pipeline stage** = well-defined step in executing an instruction

**Instruction pipelining** = run all stages at same time, on different instructions
- single-instruction latency unchanged: single instruction executes no faster (maybe even slower)
- throughput improved: more instructions may be executed per unit time
- many-unit latency improved: large sequences of instructions execute faster
Ideal speedup is number of stages in the pipeline. Do we achieve this?

### A 5-stage MIPS pipeline

<table>
<thead>
<tr>
<th>Stage</th>
<th>lw</th>
<th>sw</th>
<th>R-type</th>
<th>branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Fetch instruction from memory</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>2. Read registers, decode instruction</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>3. ALU: calculate result or address</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>4. Access data memory</td>
<td>200</td>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Write result into register</td>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total time: 800 700 600 500

*Times for one particular implementation, in picoseconds.

### Pipelining and Performance

**Ideal** for long sequence of instructions:
- All stages take same time.
- Stage management is free.
- Nearby instructions are independent.

\[
\text{Time between instructions}_{\text{pipeline}} = \frac{\text{Duration of one instruction}}{\text{Number of stages}}
\]

**Reality:**
- Different stages take different amounts of time.
- Managing stages introduces extra time overhead on each stage.
- Some instructions depend on neighbors’ results.

**In other words:** time between instructions = unit time / throughput
- No pipelining: throughput = unit time / instruction latency
- Pipelining: throughput ≤ stages * unit time / pipelined instruction latency


**Practical limits**

Ideal 5-stage clock cycle time = 800 ps / 5 = 160 ps.

Reality: 200 ps

Stages are imperfectly balanced. Pipelining adds overhead. Clock cycle time is higher, throughput is lower.

Speed-up:

\[
\frac{2400 - 1400}{2400} = 42\% 
\]

Normalized time:

\[
\frac{1400}{2400} = 0.58x 
\]

Clock cycle ratio: 800 / 200 = 4. Shouldn’t it be 4x as fast? = 80% faster? Why not?

---

**MIPS: designed to be pipelined**

Fixed-length instructions: fetch and decode are simple.

Few instruction formats:
read registers *while* decoding.

Memory operands only appear in loads and stores:
ALU calculates address, memory access can follow immediately.

Operands must be aligned in memory:
one instruction requires at most one data memory access.

---

**Building a MIPS Pipeline**

Our initial MIPS pipeline assumes all sequences of instructions can be pipelined safely without modification.

We later consider *hazards*, the cases where this assumption is false.
**Pipeline registers**

Store results of stage at this cycle to be used as inputs to next stage on next cycle.

**Instruction fetch:**

\[
lw \ $t1, 100($s0)\]

**Instruction decode and register read**

\[
lw \ $t1, 100($s0)\]

**Execute ALU op / calculate address**

\[
lw \ $t1, 100($s0)\]
Memory access, write back

lw $t1, 100($s0)

Broken write back: lw $t1, 100($s0)
Which instruction’s destination register # is used?

Fixed write back: lw $t1, 100($s0)
Get destination register # from instruction in WB stage.

To remember WB instruction’s dest., pass it along at each stage.

Pipelined control
Remember control lines for later stages. (As with write-back destination.)
Pipeline Hazards and Stalls

Hazard: when pipelining the next instruction is not safe

Simple fix: pipeline stall ("bubble")
Delay later instructions until hazard is resolved.

Hazards
- **Structural**: Single hardware unit needed simultaneously by 2 stages
- **Data**: Earlier instruction result unready for use by later instruction
  - Register data hazard – fix by forwarding
  - Load-use data hazard – fix by stalling or compile-time reordering
- **Control**: Branch resolution unknown
  - Branch hazard – fix by predicting and speculating

Register data hazards

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

Hazard a guess at what might go wrong.
Forwarding resolves simple register data hazards.

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID=EX.RegisterRt)) then ForwardB = 10

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID=EX.RegisterRs)) then ForwardA = 10

Load-use data hazard

lw $2, 20($1)

Cannot forward.
Must stall to resolve load-use hazards.

Stall requires:
- Keep same PC.
- Keep same IF/ID register contents.
- Change ID/EX control register to NOP.

Guess not taken and keep going.

Control/branch hazards

If guessed right?
If guessed wrong?
Pipelined Branch

- What happens when a program branches, assuming the pipeline is optimized for branches that are not taken.

- Consider the sequence:
  
  36 sub $10, $4, $8  
  40 beq $1, $3, 7  
  44 and $12, $2, $5  
  48 or $13, $2, $6  
  52 add $14, $4, $2  
  56 s1t $15, $6, $7  
  ...  
  72 lw $4, 50($7)

Clock cycle 3

- Branch prediction (try not to stall!)

  **Hazard a guess**: branch probably will (not) be taken.
  
  - Always guess the same for all branches.
  - Guess statically for each branch (compile-time, use separate beq_likely, beq_unlikely instructions).
  - Guess dynamically for each branch (run-time):
    - Based on recent history of execution
    - ... classic research field in computer architecture, mostly done.

  and keeping going, but check, and fix things if you were wrong.