Five Classic Components of a Computer

- Compiler
- Interface
- Control
- Datapath
- Processor
- Memory
- Input
- Output

Evaluating performance
Pipelining

- analogy: laundry
  - wash – dry – fold – put away
- analogy: industrial assembly line
- sequential vs. pipelined execution
- latency vs. throughput
  - startup latency for individual operation vs.
  - overall latency for sequence of operations
Pipelining
Typical Instruction Cycle

- **IF:** instruction fetch
- **ID:** instruction decode
  - Sometimes referred to as register read
  - load register values into ALU
- **EX:** execute
- **MEM:** memory access
  - access memory
- **WB:** write back
  - write results back to registers
Pipelined Performance

- clock cycle is 200ps = 5GHz, single-cycle model
- Consider the following program. What is the total CPU time with pipelining?

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</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>
</tbody>
</table>

lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
Pipeline Performance

Program execution order (in instructions)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $1, 100($)</td>
<td>800</td>
</tr>
<tr>
<td>lw $2, 200($)</td>
<td>200</td>
</tr>
<tr>
<td>lw $3, 300($)</td>
<td>200</td>
</tr>
</tbody>
</table>
Pipeline Speedup

- if all stages are balanced
  - i.e., all take the same time
  - time between instructions\textsubscript{pipelined} =
    time between instructions\textsubscript{serial} / \# of stages
- if stages are not balanced, speedup is less
- speedup due to increased throughput
  - latency for each instruction unchanged
  - (maybe) even slowed down a bit
MIPS Instruction Set for Pipelining

- constant length instructions (fetch, decode)
- few instruction formats, source fields same
  - register operands can be fetched while decoding
- memory operands only in load and store
  - one memory access per instruction
  - compute address during execute
  - no separate stage needed
Pipeline Hazards

- instructions are not completely independent
- hazard: condition that blocks pipelined flow

- structural: combination of instruction types
  - resource is busy
- data: dependency between instructions
  - need to wait for data read/write
- control: dependency between instructions
  - control depends on previous instruction
Structural Hazard

- instruction fetch vs. load/store

- solution: more resources
  - different memories (details later)

- if hazard cannot be resolved
  - introduce wait stages: called stall or bubble
Data Hazard

- `add $s0, $t0, $t1`
- `sub $t2, $s0, $t3`

=> reorder instructions, if possible
Forwarding / Bypassing

- use result when it is computed
  - don't wait for it to be stored in register
  - requires extra connections in the data path
Load-Use Data Hazard

- can't always avoid stalls by forwarding
  - if value not computed when needed
  - can't forward backward in time
- Stall = Bubble
Example: Avoiding Stalls

\[ a = b + e; \]
\[ c = b + f; \]

\[
\begin{align*}
\text{lw} & \quad \text{\$t1, 0\text{(\$t0)}} \\
\text{lw} & \quad \text{\$t2, 4\text{(\$t0)}} \\
\text{add} & \quad \text{\$t3, \$t1,\$t2} \\
\text{sw} & \quad \text{\$t3, 12\text{(\$t0)}} \\
\text{lw} & \quad \text{\$t4, 8\text{(\$t0)}} \\
\text{add} & \quad \text{\$t5, \$t1,\$t4} \\
\text{sw} & \quad \text{\$t5, 16\text{(\$t0)}} \\
\text{lw} & \quad \text{\$t1, 0\text{(\$t0)}} \\
\text{lw} & \quad \text{\$t2, 4\text{(\$t0)}} \\
\text{lw} & \quad \text{\$t4, 8\text{(\$t0)}} \\
\text{add} & \quad \text{\$t3, \$t1,\$t2} \\
\text{sw} & \quad \text{\$t3, 12\text{(\$t0)}} \\
\text{add} & \quad \text{\$t5, \$t1,\$t4} \\
\text{sw} & \quad \text{\$t5, 16\text{(\$t0)}} 
\end{align*}
\]