Pipelined MIPS

Why pipelining?

While a typical instruction takes 3-4 cycles (i.e. 3-4 CPI), a pipelined processor targets 1 CPI (and gets close to it).

How is it possible? By overlapping the execution of consecutive instructions ...
Study the Laundromat example from the book.
Example of pipelining

- Pipelining doesn’t help latency of single load, it helps throughput 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 “drain” it reduces speedup

Pipelining in a laundry
Washer takes 30 minutes
Dryer takes 40 minutes
Folding takes 20 minutes
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>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>IF ID EX</td>
</tr>
<tr>
<td>R-type</td>
<td>IF ID EX WB</td>
</tr>
<tr>
<td>sw</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>lw</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
Break datapath into 5 stages

- Each stage has its own functional units.
- Each stage can execute in 2ns
  - Just like the multi-cycle implementation

It shows the rough division of responsibilities.
The buffers between stages are not shown.
Since multiple memory operations overlap, we had to return to Harvard architecture!

How can the same adder perform IF and EX in cycle 3? We need an extra adder! Gradually we need to modify the data path for the multi-cycle implementation.
Uniformity is simplicity

- 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

<table>
<thead>
<tr>
<th>R-type</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>NOP</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>NOP</td>
<td>WB</td>
</tr>
<tr>
<td>sub</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>NOP</td>
<td>WB</td>
</tr>
<tr>
<td>lw</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>or</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>NOP</td>
<td>WB</td>
</tr>
<tr>
<td>lw</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

- Stores and Branches have NOP stages, too...

<table>
<thead>
<tr>
<th>store</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>NOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>NOP</td>
<td>NOP</td>
</tr>
</tbody>
</table>
**Speedup**

The steady state throughput is determined by the time $t$ needed by one stage.

The length of the pipeline determines the pipeline filling time.

If there are $k$ stages, and each stage takes $t$ time units, then the time needed to execute $N$ instructions is

$$k.t + (N-1).t$$

_Estimate the speedup when $N=5000$ and $k=5$_
Hazards in a pipeline

Hazards refer to conflicts in the execution of a pipeline. One example is the need for the same resource (like the same adder) in two concurrent actions. This is called structural hazard. To avoid it, we have to replicate resources. Here is another example:

\[
\text{lw } \$s1, 4(\$sp) \quad \text{IF ID EX MEM WB} \\
\text{add } \$s0, \$s1, \$s2 \quad \text{IF ID EX MEM WB}
\]

Notice the second instruction tries to read \$s1 before the first instruction complete the load! This is known as data hazard.
Avoiding data hazards

One solution is in insert bubbles (means delaying certain operation in the pipeline)

lw $s1, 4($sp)       IF  ID  EX  MEM  WB
add $s0, $s1, $s2    IF  nop  nop  nop  ID

Another solution may require some modification in the datapath, which will raise the hardware cost

Hazards slow down the instruction execution speed.
Control hazard

```
sub $s1, $t1, $t2      IF   ID   EX   MEM   WB
beq $s1, $zero L      IF   ID   EX   MEM
some instruction here  IF   ID   EX
```

Will the correct instruction be fetched?

There is no guarantee! The next instruction has to wait until the predicate ($s1=0) is resolved. Look at the tasks performed in the five steps – the predicate is evaluated in the EX step. Until then, the control unit will insert **nop (also called bubbles)** in the pipeline.
The Five Cycles of MIPS

(Instruction Fetch)
IR:= Memory[PC]; PC:= PC+4

(Instruction decode and Register fetch)
A:= Reg[IR[25:21]], B:=Reg[IR[20:16]]
ALUout := PC + sign-extend(IR[15:0])

(Execute|Memory address|Branch completion)
Memory refer: ALUout:= A+ IR[15:0]
R-type (ALU): ALUout:= A op B
Branch: if A=B then PC:= ALUout

(Memory access | R-type completion)
LW: MDR:= Memory[ALUout]
SW: Memory[ALUout]:= B
R-type: Reg[IR[15:11]]:= ALUout

(Write back)
LW: Reg[[20:16]]:= MDR
An alternative approach to deal with this is for the compiler (or the assembler) to insert **NOP instructions**, or reorder the instructions.