

# ECE154A — Discussion 08

---

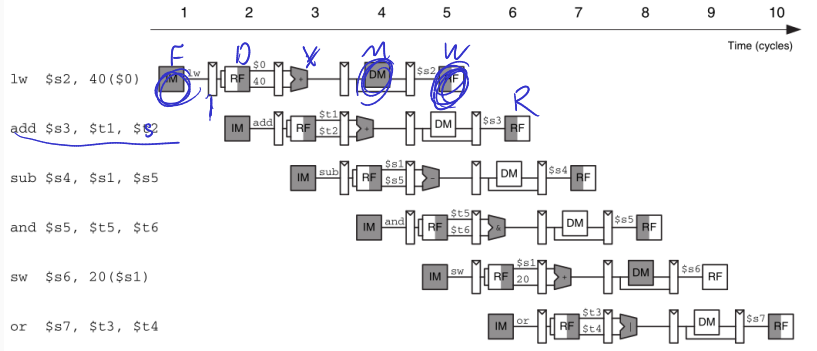
George Higgins Hutchinson

November 19, 2021

## Keep your eyes open for...

- Lab 5: due Nov 22
- HW5: due Nov 24

# Timing in a Pipeline



# Why pipeline?

- Smaller? Cheaper?
- Faster?
- Simpler to implement?
- Flexible (how hard to add complicated instructions?)

# Why pipeline?

- Smaller? Cheaper?

No! We have all the hardware from single-cycle, plus extra registers.

- Faster?

Depends what you care about. More latency (instructions have to go through register setup and  $\text{clk}-2\text{-}q$ ), but also more throughput (since you clock faster).

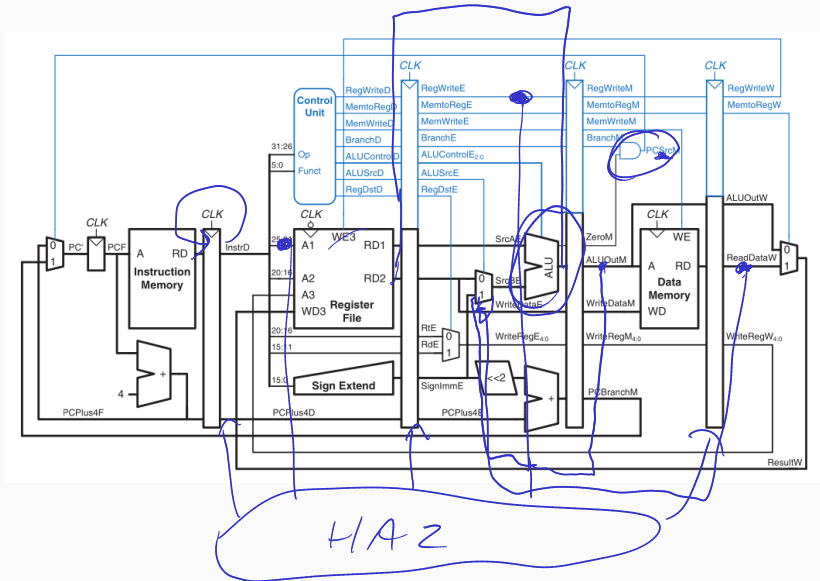
- Simpler to implement?

Probably not. New hazards require special management.

- Flexible (how hard to add complicated instructions?)

Probably harder, since all instructions need to share a schedule. (This was the motivation to develop early RISC architectures like MIPS!)

# Pipeline detail




# Hazards

- Multiple instructions now doing things at one time: what happens if they need things at the same time?
- Taxonomize: for any state element, might read or write in sequence

Read-after-read: same state, nothing to worry about.	Write-after-read: Remember that instructions are started in order and are same length. Can we get this?
Read-after-write: Next instruction might not have committed data yet ... what can we do?	Write-after-write: Instructions reach end in order. Sequence enforced naturally.

*add to e  
sw to ~*

# Hazards

- For RAW hazards, can also taxonomize on which state element is waiting.
  - Need PC register set to fetch next instruction. If not PC+4, *Control Hazard*.
  - Need program registers set to get correct operands for next instruction. If not, *Data Hazard*.
  - Multiple instructions need to share resource: *Structural Hazard*.
- 



# Forwarding



- For a lot of data hazards, the pipeline has the correct answer before “committing” it to regfile.
- Add a path to replace (mux) bad reads with correct data.
- Be careful not to introduce new critical path through your forwarding! (H&H do this by only reading from directly after registers)

## Stalling/Flushing

beg      mmm      some\_place  
add      to to to

- Pipeline barriers can inject a fake operation with no effect
- Need to make sure you save *all* state needed to resume in earlier steps.
- Can solve any problem, but costs IPC
- Branch/jump PC known by beginning of D stage. How many incorrect instructions in pipeline? Have any committed state?

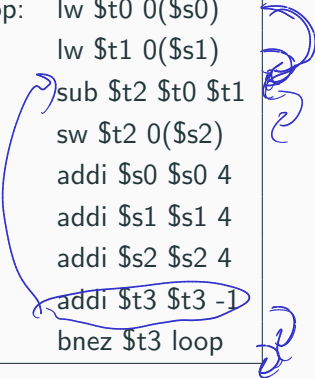
F D<sup>0</sup> (X) m w  
F D K I m w

# Hazard-aware ISAs

- Some early MIPS cores exposed a “branch-delay slot” (instruction right after branch gets executed whether or not branch is taken).
- How does this change code size? IPC? Hardware complexity?
- Explicit ISA hazard-awareness has fallen out out favor as timing gets less predictable, but compilers can still try to optimize around hazards.

## Pipeline practice

Consider the program:



```
loop:  lw $t0 0($s0)
      lw $t1 0($s1)
      sub $t2 $t0 $t1
      sw $t2 0($s2)
      addi $s0 $s0 4
      addi $s1 $s1 4
      addi $s2 $s2 4
      addi $t3 $t3 -1
      bnez $t3 loop
```

CPI of a loop iteration assuming only stalls? Assuming forwarding?

Re-order instructions to optimize CPI. Re-calculate.

## Pipeline practice

- If only stalling: branch penalty = 3, data hazard penalty = 4 - ReuseDistance

$$CPI = \frac{14 + 8}{8}$$

- If forward+stall: branch penalty = 1, load penalty = 1

$$CPI = \frac{2 + 8}{8} = 1.25$$

- Optimal order: move the independent addi instructions into data hazards (double-check immediates!)
- Note that there aren't quite enough to fill gaps if only stalling. End up with 2 + 3 bubbles if only stall, 3 if forwarding.