Pipelining

Fundamentals of Pipeline Math

- Time for a single stage = `T`
- Time for `S` stages = `ST` (simple enough...)
- Time *between* initiations = `T` (not `ST` as in the multi-cycle approach that we've discussed so far – i.e. where we do 1 instruction right after the next.) (i.e. a new instruction starts each CC.)
- Idea: Improve **throughput** ... or how often something comes out of the datapath. For example:
  - Assume we have a non-pipelined, multi-cycle datapath where each instruction takes 4 CCs on average. Assume our clock rate is x. How many seconds does it take to finish N instructions?
    * N Instructions $\times \frac{4CCs}{Instruction} \times \frac{xs}{CC}$ = $4N$
  - Now, with a pipelined version, we would have:
    * N Instructions $\times \frac{1CCs}{Instruction} \times \frac{xs}{CC}$ = N

How long will it take or N initiations to finish?

- $N(T) + (S-1)(T)$
  - The `N(T)` part refers to the fact that something finishes every 'T' time units. If there are 'N' items in the pipeline, `N(T)` is one component of the execution time – and it takes N units of time T for all instructions to exit.
  - The `(S-1)(T)` part refers to the fact that we need to fill up the pipeline. I.e. if there are S stages, than (S-1) time units will be spent filling up those stages. No "useful work" will be output during this time. **(see simple trace with stages in row and time in column)**
- Example:
  - 4 loads, 4 stages, 40 minutes/stage (i.e. laundry!)
  - Pipelined: (4)(40) + (4-1)(40) = 160 + 120 = 280
  - Nonpipelined: (4)(4)(40) = 640!

Throughput

- Can divide $N(T) + (S-1)(T)$ by N: $\rightarrow \frac{N(T)+(S-1)(T)}{N} \rightarrow$ As N gets large, equation goes to T. Thus, time per initiation is T.
- As N gets large, the component on the right trends toward 0 and the NT component dominates. Ideally, you see a result produced every (shorted) clock cycle.
- Thus, despite the slight increase in overhead, the pipelined version produces results faster.

Speedup

- Ideally, the speedup one sees is equal to the number of pipe stages...
  - Assume/recall that with the multi-cycle approach, we tried to balance the amount of work per cycle. We try to do the same thing here by balancing the amount of work per stage.
  - Assume that each cycle takes τ time units. If there are 4 steps, then the total time spent is 4τ.
  - The time for 1000 pieces of data/instruction is thus $4\tau \times 1000$.
  - Now, what about the time for a pipelined version?
  - We can actually use the forumla: $NT + (S-1)T$. Thus, we get: $(1000)\tau + (4-1)\tau = 1003\tau$.
  - Therefore, the speedup is essentially equal to the number of stages: $\frac{4000\tau}{1003\tau}$ is essentially 4 (the number of stages).
- (more in the slides)

Part B – Pipelining Hazards

We need to worry about 3 things that can (a) affect pipelining performance (e.g. that can prevent an instruction from finishing each CC) and (b) affect pipelining correctness (e.g. that can prevent registers from changing to the state they logically should).

Structural Hazards:
Example:

	1	2	3	4	5	6	7	8	9	10
lw $1, 0($2)	F	D	E	M	**W**					
add $3, $4, $5		F	D	E	M	W				
sub $6, $7, $8			F	D	E	M	W			
or $9, $10, $11				F	**D**	E	M	W		
and $12, $13, $14					…	…	…	…		

In Cycle 5, we need to support the writing and reading of a register simultaneously.

If there is no support for simultaneous reading/writing, a structural hazard occurs – and the or instruction would have to wait. More specifically:

	1	2	3	4	5	6	7	8	9	10
lw $1, 0($2)	F	D	E	M	W					
add $3, $4, $5		F	D	E	M	**W**				
sub $6, $7, $8			F	D	E	M	W			
or $9, $10, $11				"Bubble"	F	**D**	E	M	W	
and $12, $13, $14					…	…	…	…		

(but then the problem just repeats itself…)

More formally:
- The simplest way to resolve a structural hazard is to add HW
  - In the above example, this would mean more register "ports"
- Basically, structural hazards arise from resource conflicts
  - HW can't support all combinations of instructions going through the pipeline
- Sometimes its actually better to stall instead of adding more HW
  - E.g. if the combination occurs rarely.

Data Hazards:
Let's look at another sequence of instructions:

	1	2	3	4	5	6	7	8
lw $1, 0($2)	F	D	E	M	W	**$1 available**		
add $3, $1, $4		F	D **$1 needed**	E	M	W		
sub $5, $1, $6			F	D **$1 needed**	E	M	W	
or $7, $1, $8				F	D **$1 needed**	E	M	W

This is a data hazard:

- Data hazards arise from *dependencies* between instructions
- Here, add, sub, and or all depend on lw

How do we fix it?

Option 1: Wait.

	1	2	3	4	5	6	7	8	9	10	11	12
lw $1, 0($2)	F	D	E	M	W							
add $3, $1, $4		F	---	---	---	D	E	M	W			
sub $5, $1, $6						F	D	E	M	W		
or $7, $1, $8							F	D	E	M	W	

Idea: Stall the pipeline; wait for the result we need to be produced.

Option 2: (Read data when it is being written – slightly better)

	1	2	3	4	5	6	7	8	9	10	11	12
lw $1, 0($2)	F	D	E	M	W							
add $3, $1, $4		F	---	---	D	E	M	W				
sub $5, $1, $6					F	D	E	M	W			
or $7, $1, $8						F	D	E	M	W		

Hint:

- How do you know you have the table right?
- (Look down a column – there should *never* be two of the same letter/stage)

Option 3: (Use data as soon as its available – even better)

Looking at the chart associated with Option 2, it's clear that the data we want to use – e.g. that will be put in $1 is "available" before it goes to the register file. Let's use it as soon as it is produced. (This is more obvious with a slightly different instruction mix so note the change below…)

	1	2	3	4	5	6	7	8	9
add $1, $10, $9	F	D	E **$1 data available here**	M	W				
add $3, $1, $4		F	D	E	M	W			
sub $5, $1, $6			F	D	E	M	W		
or $7, $1, $8				F	D	E	M	W	

(This is called *forwarding*. We'll talk more about it later, but it can be easily implemented by feeding back the output of the ALU back to the input multiplexors.)

Control Hazards:

What if we have:

```
        beq   $5, $6, target
        add   $1, $2, $3
        …
Target: add   $1, $5, $6
```

What's the problem?

- If \$5 == \$6, then \$1 should = \$5 + \$6
- If \$5 != \$6, then \$1 should = \$2 + \$3
- We need to make sure we put the right value into \$1 – otherwise our program will be incorrect.

There's another problem too…

- To finish 1 instruction each CC, we need to start 1 instruction each C
- At first glance, the only way to ensure logical correctness is to wait until the branch outcome is decided – so we'll have to stall our pipeline every time we get a branch.
    - (But about 1 of every 6 instructions is a branch…)

More formally:

- This is a <u>control</u> <u>hazard</u>. It arises from a change in program control flow.
- Which instruction do we start down the pipeline?
- If we start the wrong instruction, can we fix? Should we guess???