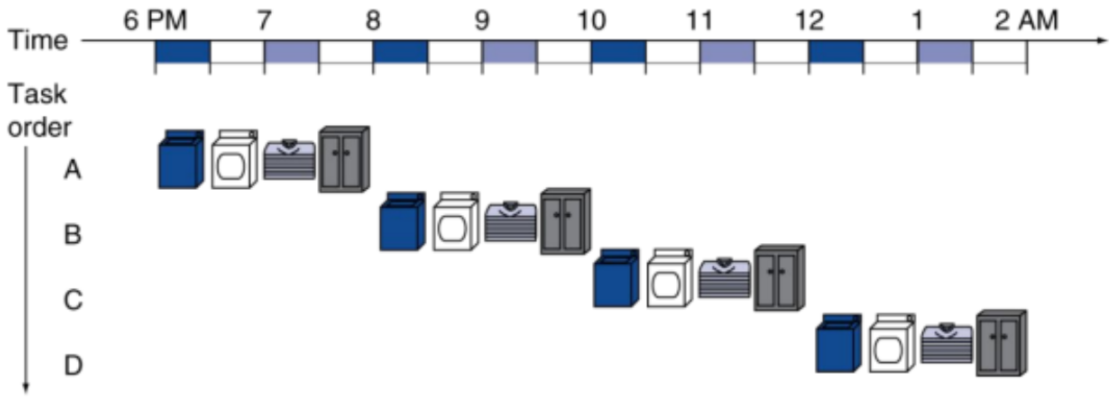
COM SCI M151B Week 3

Aidan Jan

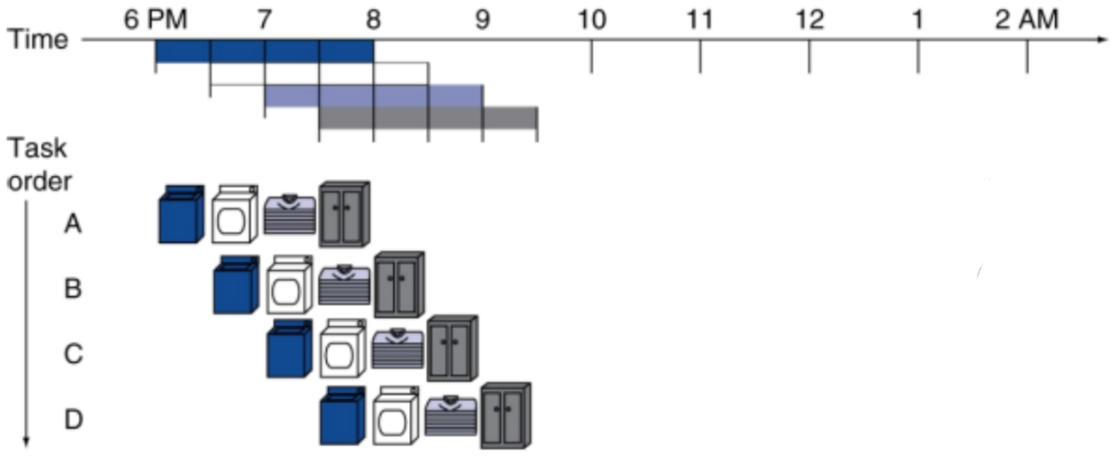
October 15, 2024

Pipelining

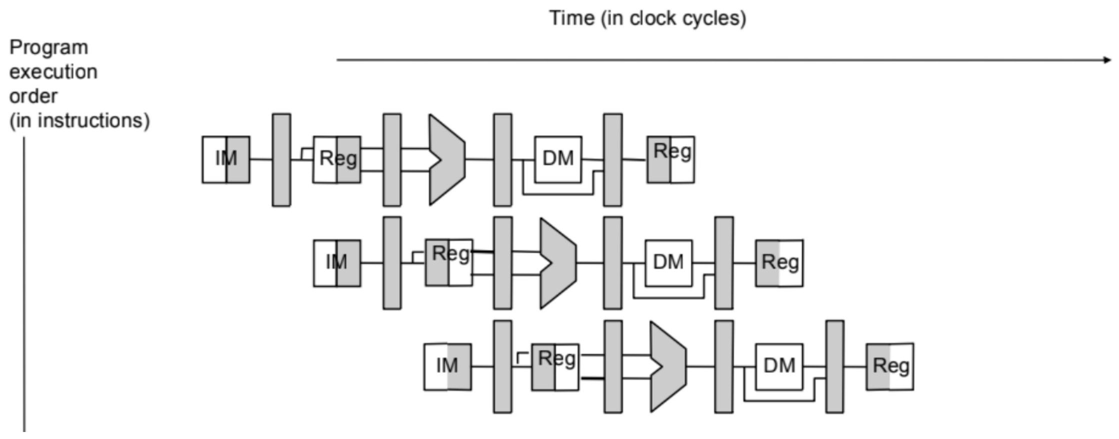
- Last week, we discussed that a single-cycle processor is simple and relatively to implement, but not the most efficient. Why isn't it the most efficient?
- **Pipelining** refers to parallelizing resources.
- Consider the following analogy: suppose you have four people, A, B, C, D, who have to do laundry.
- If one set of laundry takes 2 hours to complete, and the four people go one after another, the entire task takes 8 hours.



- However, if the second person starts as soon as the first is done with the washer, etc., then the process takes much less time.



- Notice that each individual load of laundry did not get faster; but the entire task does.
- This is like executing one instruction per cycle!

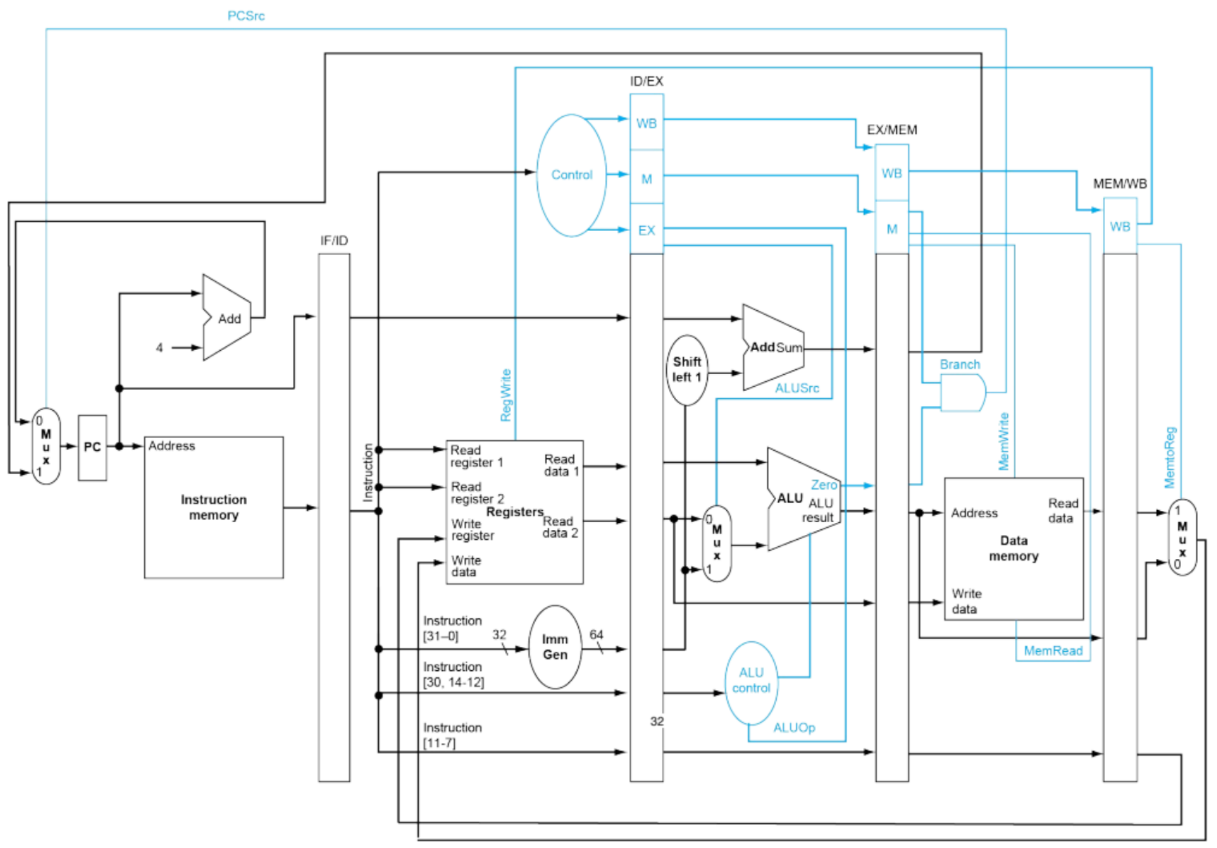


Single Cycle vs. Pipeline

Instr	Instr fetch	Register read	ALU	Memory access	Register write	Total time
lw	200ps	100 ps	**200ps**	**200ps**	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R/I-type	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- Remember that the processor's clock speed is the time it takes for the longest procedure to execute.
- Without pipelining, the clock speed would have to be 800 ps, since that instruction takes the longest.
- With pipelining, the clock speed can be 200 ps, since the specific part takes the longest and the rest can be pipelined.
- CPI remains at 1

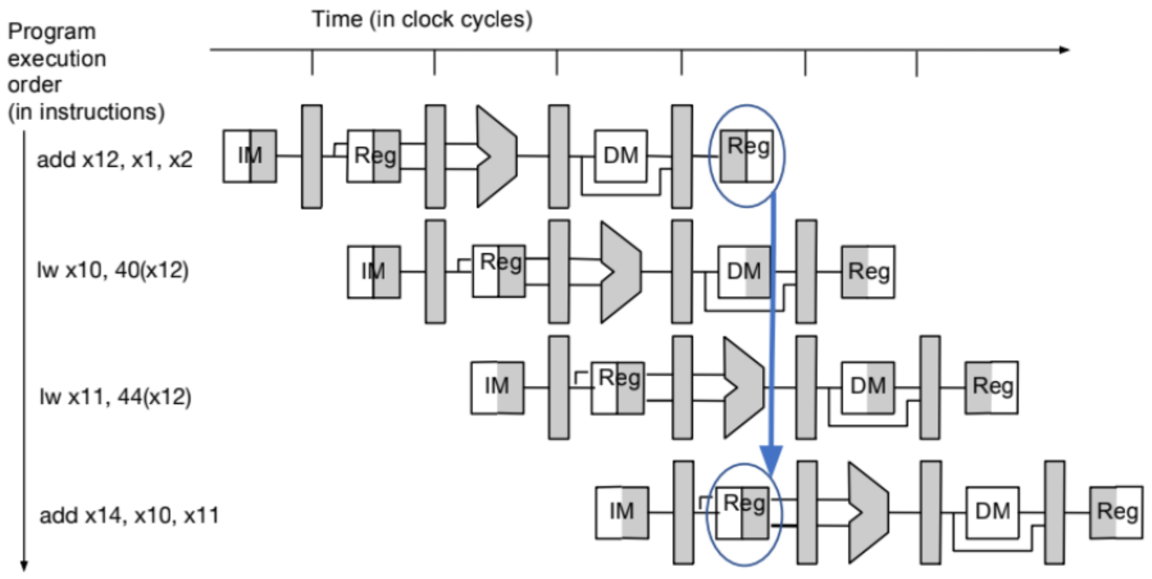
Pipelined Datapath + Controller



- We divide the original datapath and controller into 5 sections. (The fifth section is writing to registers.)
- This lets us decrease the clock cycle since each section can be pipelined.

Half-Cycle

Since reading from registers and writing to registers happen at different times, we get an issue:



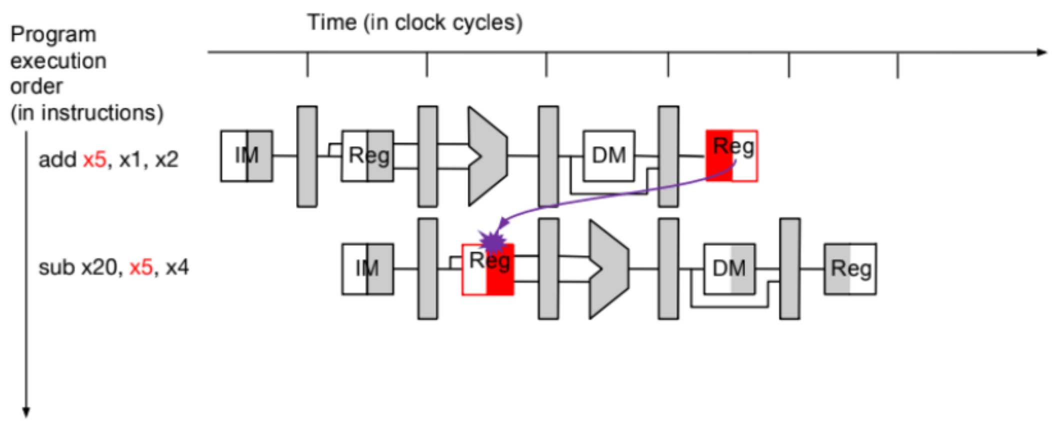
Do we read the old value or new value of the register?

- The answer is the new value, because programmers assume the instructions execute sequentially.
- The purpose is to abstract concepts like instruction timing.
- As a result, we divide into half-cycles. Every read (from registers or memory) happen in the second half-cycle, and all the writes (registers and memory) happen in the first half-cycle.

Hazards

Consider this scenario:

```
add x5, x1, x2
sub x20, x5, x4
```



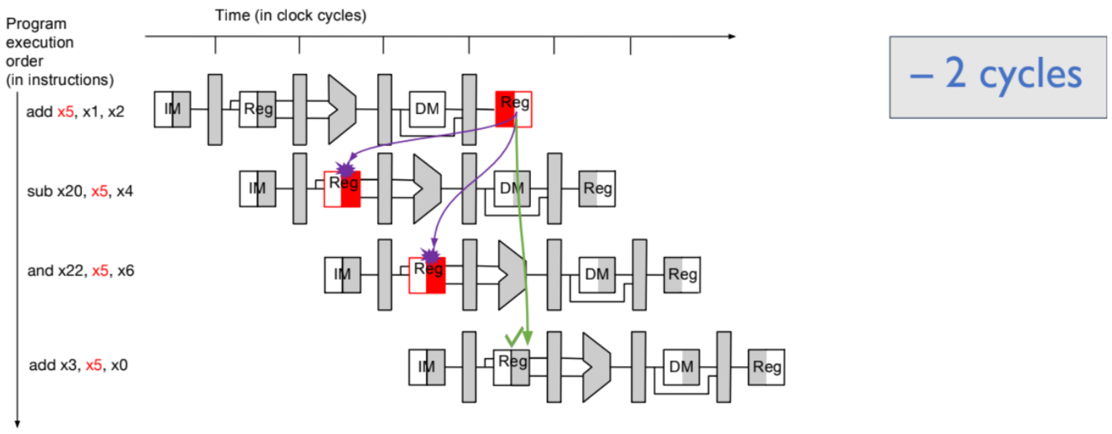
- There are situations in pipelining when the next instruction cannot execute in the following clock cycle. **These events are called hazards.**
- There are **three** different types of hazards.
  1. Data hazard
  2. Control hazard
  3. Structural hazard

Data Hazard

- Read after write (RAW)
- Writing to a register (rd) and using it (rs1 or rs2) <u>before</u> the writing is finished (i.e., *rd* reaches to the *WB stage*).

```
add x5, x1, x2
sub x20, x5, x4
```

- For how many cycles we might have RAW hazard?

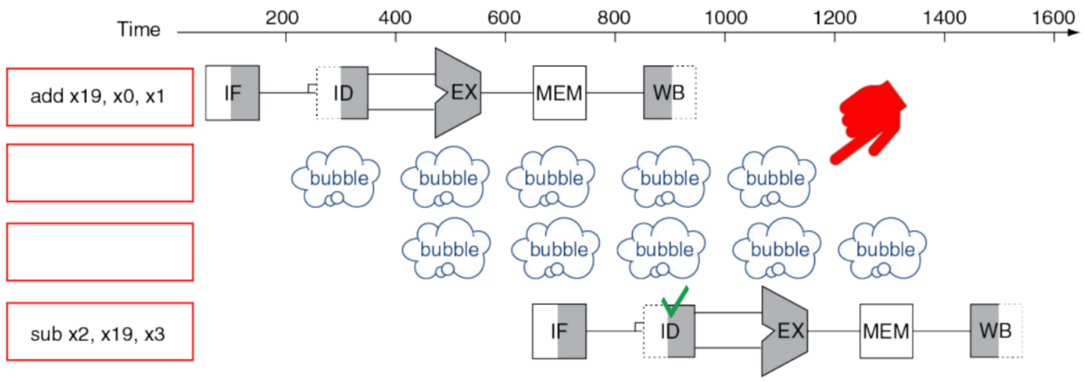


What about other relations/dependencies?

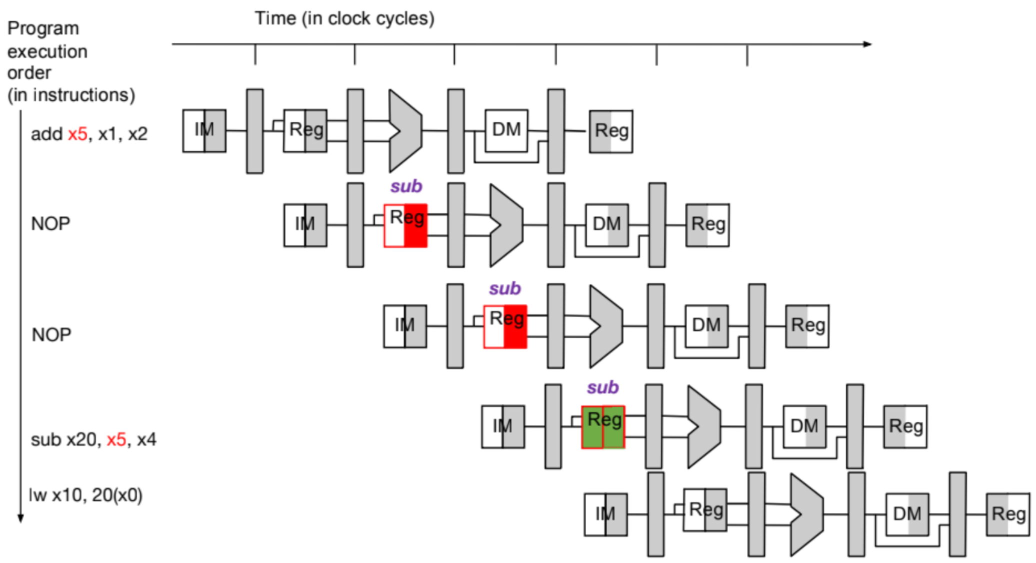
- **write after write (WAW)** - this is called *output dependency*. For our pipelined processor, *this won't be an issue* since instructions are executed ***in order***.
- **write after read (WAR)** - this is called *false dependency*. For our pipelined processor *this also won't be an issue for the same reason* (i.e., in-order execution).

How to solve RAW hazard?

- Stop the processor until the first instruction finishes!

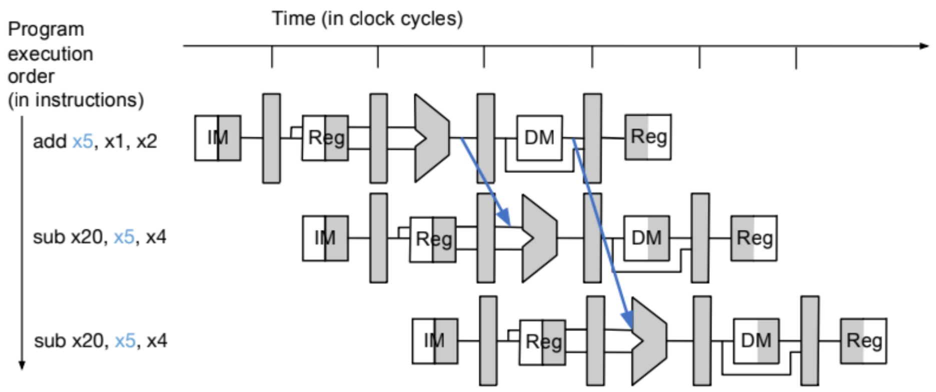


- Stopping (a part of) the pipeline is called a "stall".
- To achieve stall, we can **prevent** pipeline registers from updating (only those stages that we want to freeze/stall)
- Since some pipeline registers are not updated, we need to send *something* to the next stage.
  - Sending *arbitrary* values could overwrite data and cause problems.
  - We can send **NOP** which acts as a bubble (i.e., does nothing).



Forwarding (Bypassing)

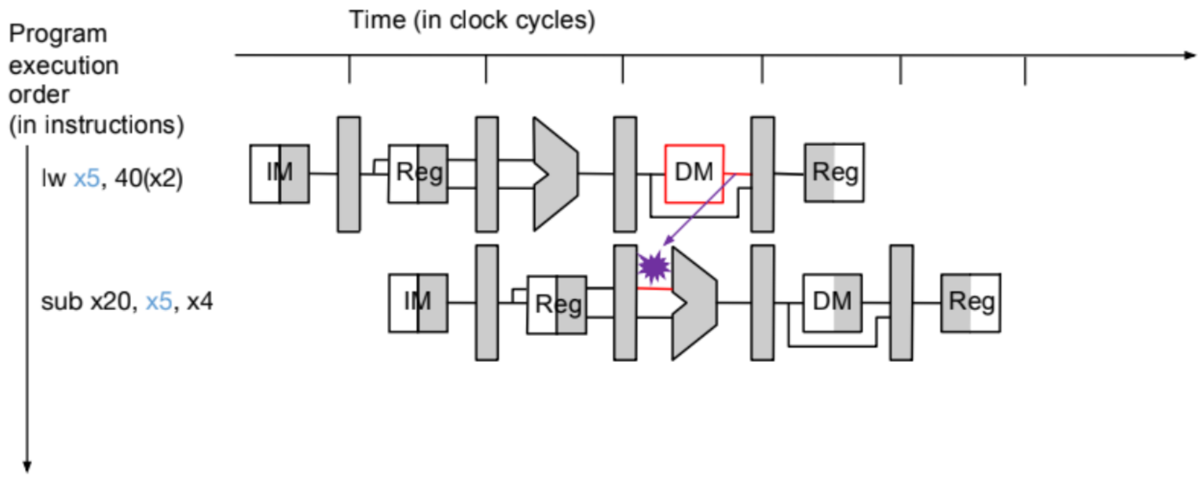
- Stalling can hurt performance if RAW often happens (which it may).
- We can prevent this by forwarding.
- Use the result when it is computed, don't wait for it to be stored in a register
- Requires extra connections in the datapath

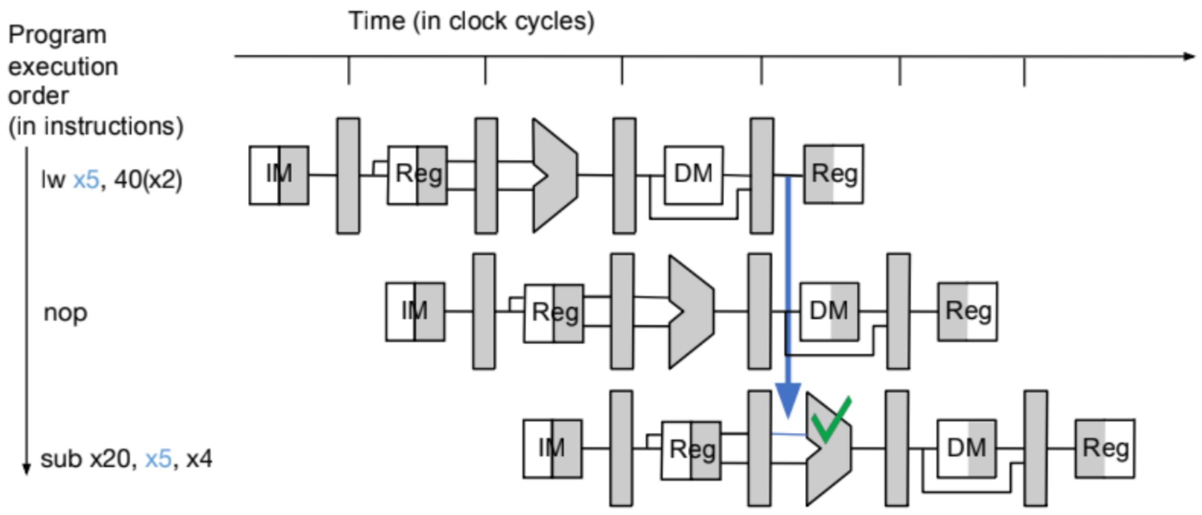


- Register to register (i.e., updating ALU operands)
- From ALU Result to ALU srcs
- From Mem to ALU srcs
- Can we always avoid stalling with this method?

Load-Use Data Hazard

- Unlike in *reg-to-reg* forwarding, in this case data is **not yet ready** in the Mem stage to be forwarded to EX stage.
- Half cycle!





- We can't avoid stalling completely. However, with forwarding, we stall for one less cycle.