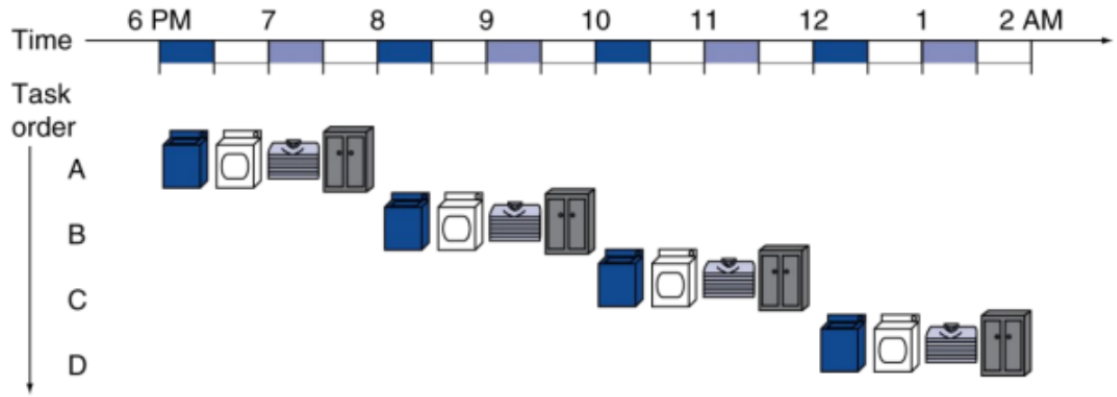
COM SCI M151B Week 3

Aidan Jan

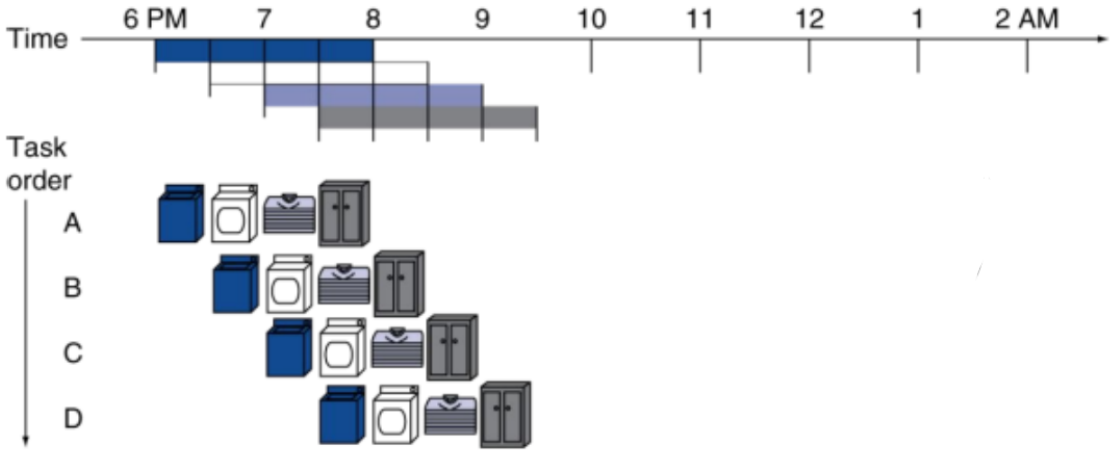
October 17, 2024

Pipelining

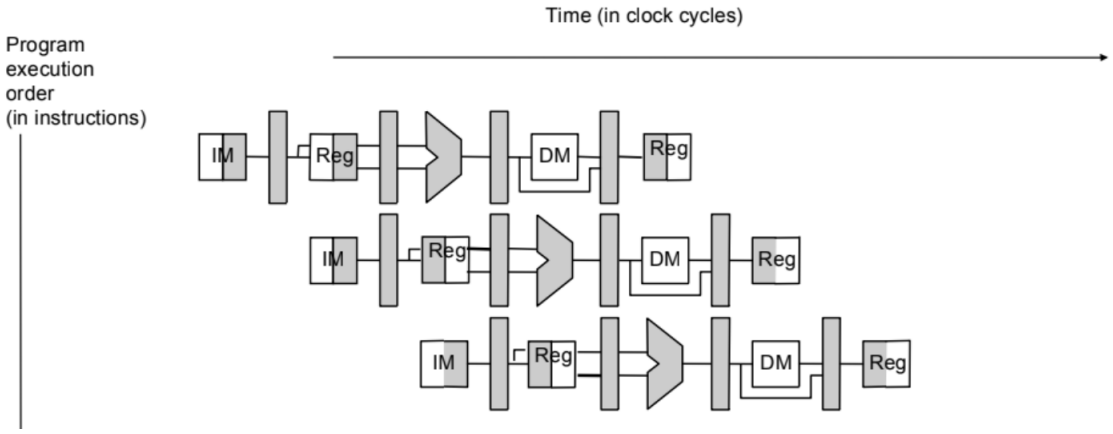
- Last week, we discussed that a single-cycle processor is simple and relatively to implement, but not the most efficient. Why isn't it the most efficient?
- **Pipelining** refers to parallelizing resources.
- Consider the following analogy: suppose you have four people, A, B, C, D, who have to do laundry.
- 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!

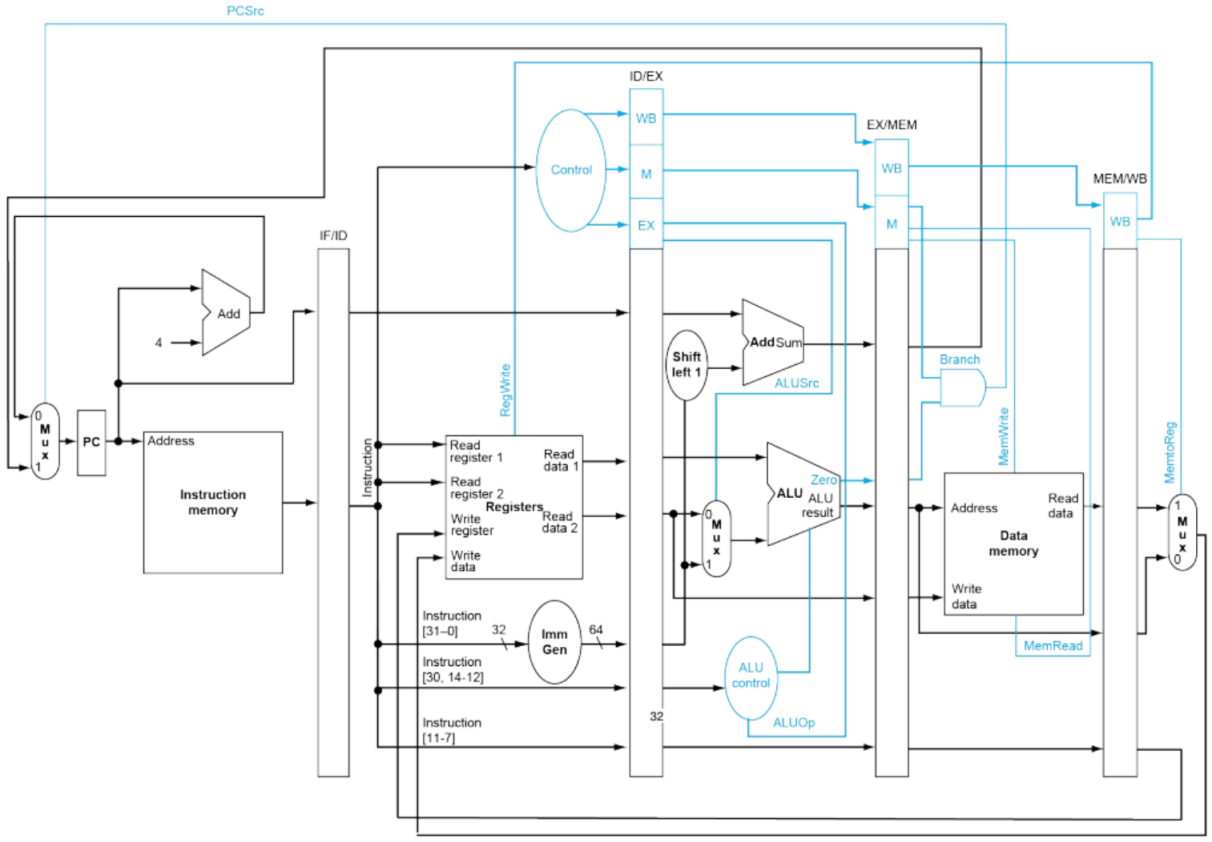


Single Cycle vs. Pipeline

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

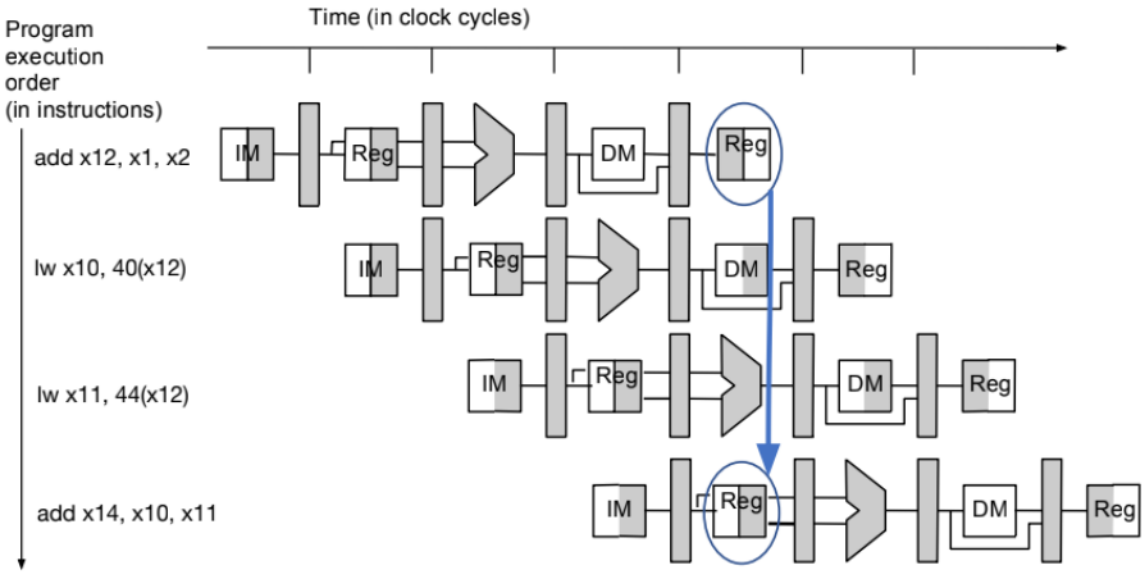
Pipelined Datapath + Controller



- 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:



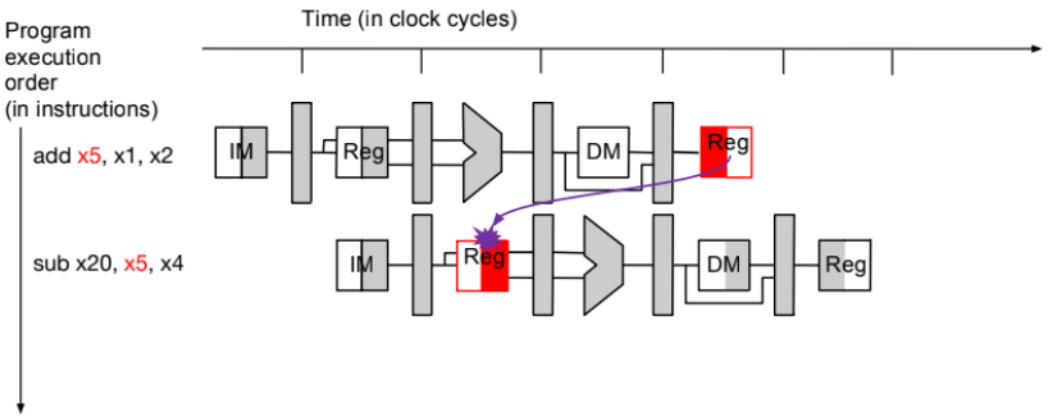
Do we read the old value or new value of the register?

- 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
```



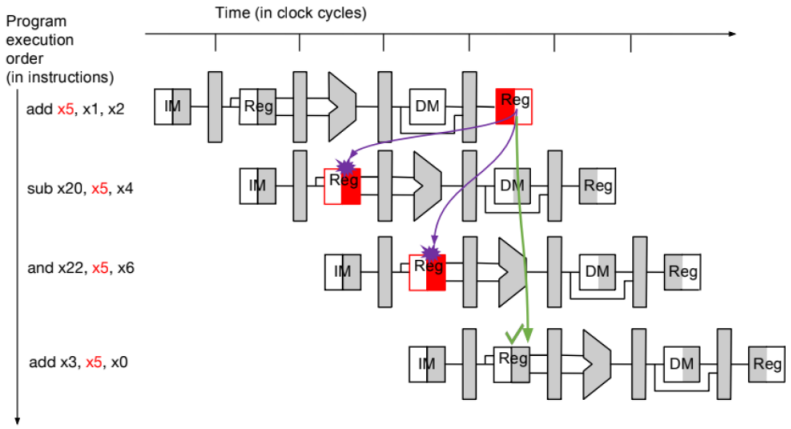
- There are situations in pipelining when the next instruction cannot execute in the following clock cycle. **These events are called hazards.**
- There are **three** different types of hazards.
  1. Data hazard
  2. Control hazard
  3. Structural hazard

Data Hazard

- Read after write (RAW)
- Writing to a register (rd) and using it (rs1 or rs2) before 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?

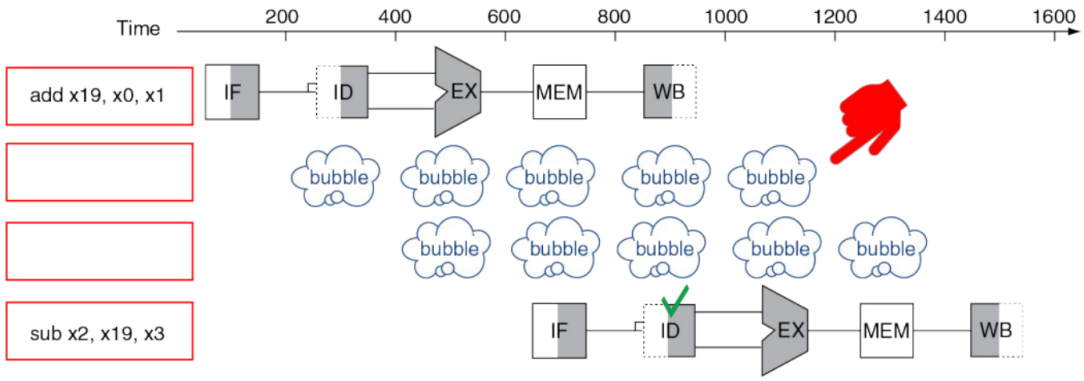


What about other relations/dependencies?

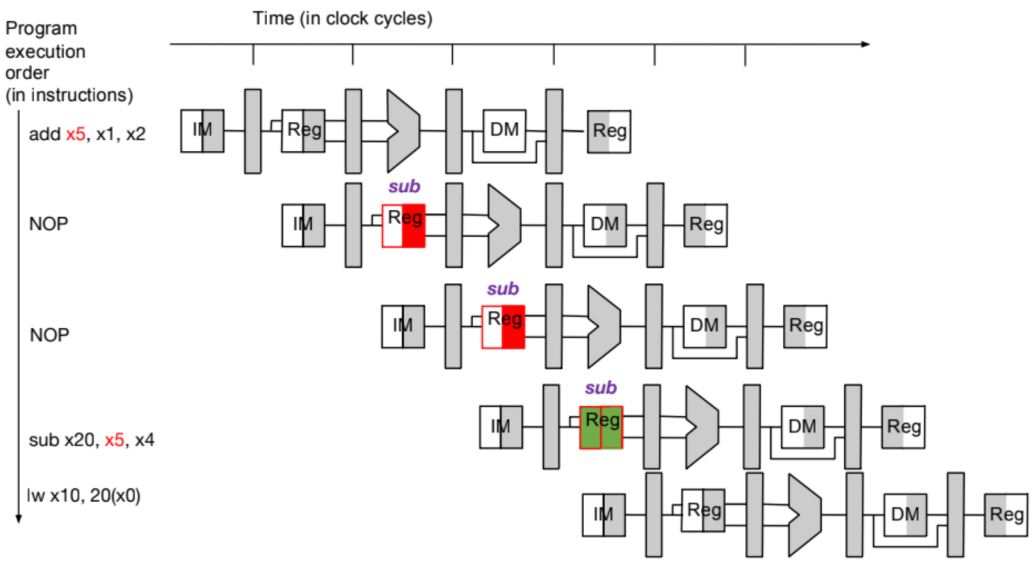
- **write after write (WAW)** - this is called *output dependency*. For our pipelined processor, *this won't be an issue* since instructions are executed ***in order***.
- **write after read (WAR)** - this is called *false dependency*. For our pipelined processor *this also won't be an issue for the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

How to solve RAW hazard?

- Stop the processor until the first instruction finishes!

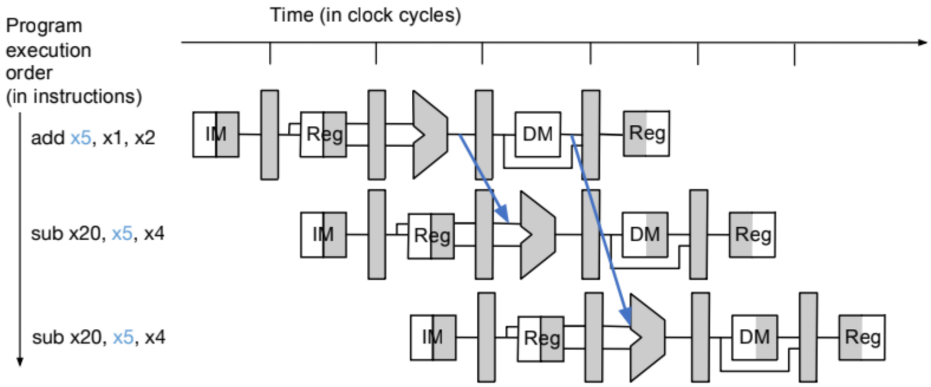


- Stopping (a part of) the pipeline is called a "stall".
- To achieve stall, we can **prevent** pipeline registers from updating (only those stages that we want to freeze/stall)
- Since some pipeline registers are not updated, we need to send *something* to the next stage.
  - Sending *arbitrary* values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



Forwarding (Bypassing)

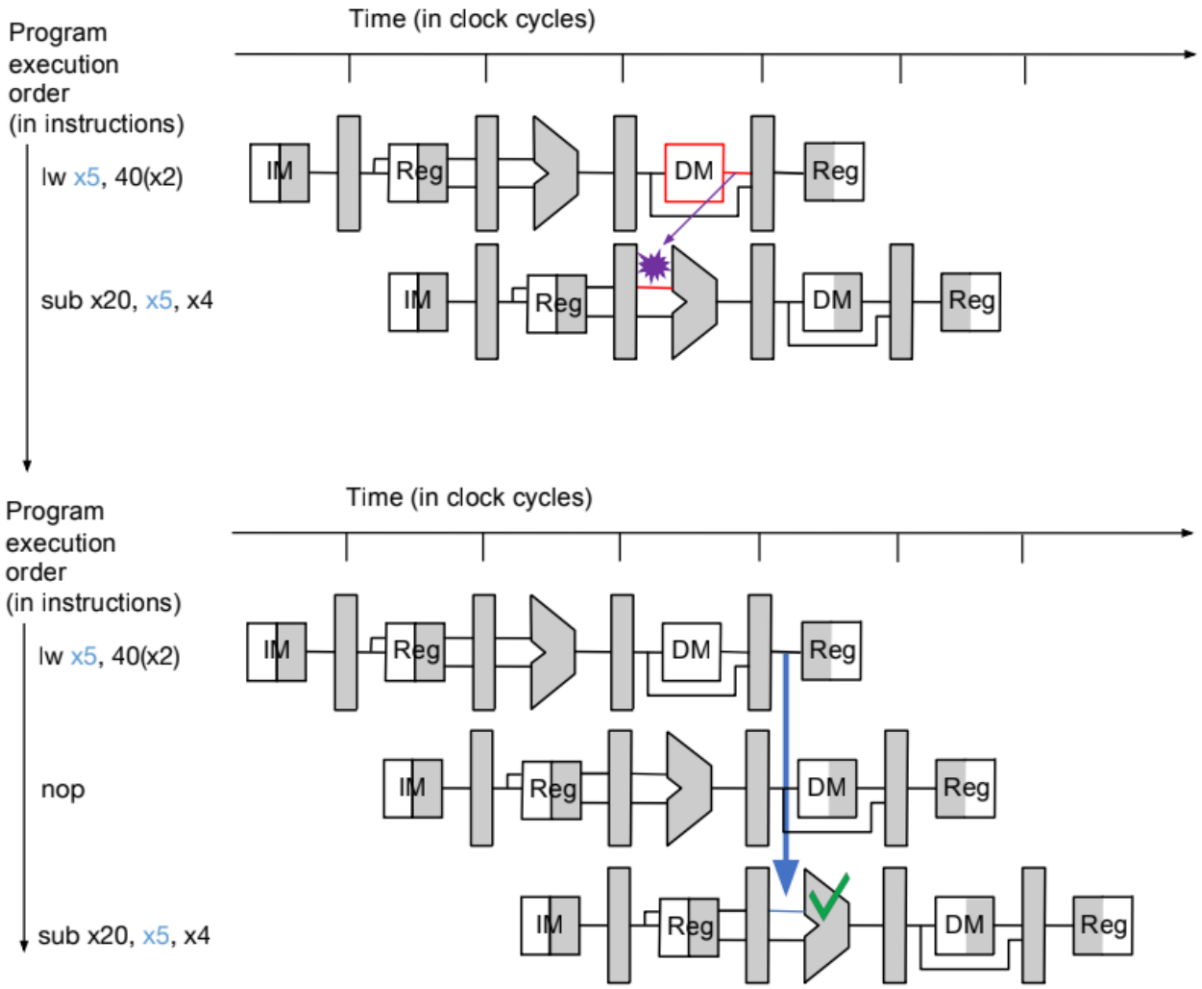
- Stalling can hurt performance if RAW often happens (which it may).
- We can prevent this by forwarding.
- Use the result when it is computed, don't wait for it to be stored in a register
- Requires extra connections in the datapath



- 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!

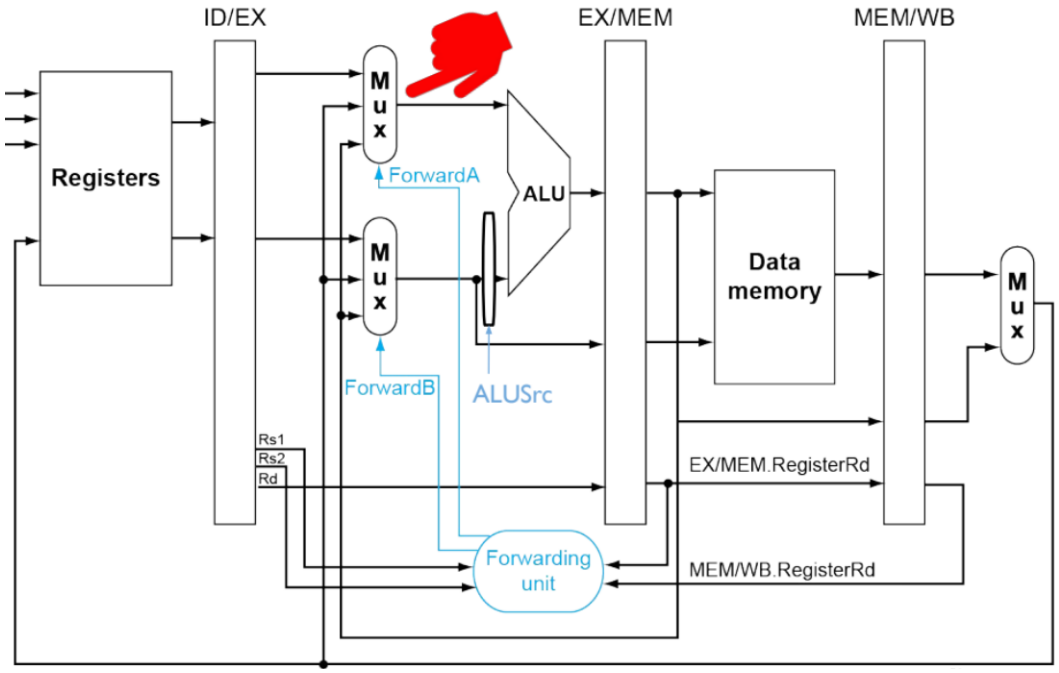


- We can't avoid stalling completely. However, with forwarding, we stall for one less cycle.

How to detect when to forward?

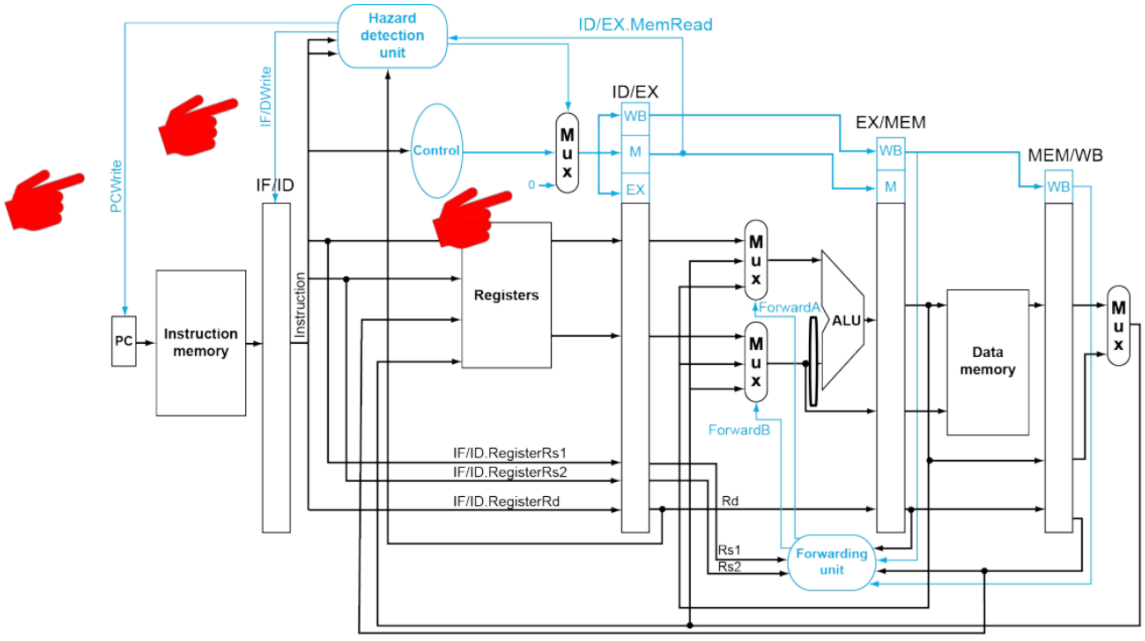
- Data hazards when:
- If the following happen, forward from EX/MEM pipeline reg
  - EX/MEM. Rd = ID/EX. Rs1
  - EX/MEM. Rd = ID/EX. Rs2
  - EX/MEM.RegWrite = 1
- EX = execution
- If the following happen, forward from MEM/WB pipeline reg
  - MEM/WB. Rd = ID/EX. Rs1
  - MEM/WB. Rd = ID/EX. Rs2
  - MEM/WB.RegWrite = 1

Datapath with RAW Forwarding



How to detect forward in Load-Use

- If the following happen, stall and insert bubble.
  - ID/EX. Rd = IF/ID.Rs1
  - ID/EX. Rd = IF/ID.Rs2
  - UD/EX.MemRead = 1
- (after inserting the bubble, the forwarding condition would be similar to that of RAW)
- Why ID/EX instead of EX/MEM?



What about the other hazards?

- **Structural:** one unit is busy
  - Shared resources (e.g., ALU)
  - Won't happen in our design (no sharing)
- **Control:** what happens when there is a branch instruction?

Control Hazards

- What instruction should be fetched after a branch?
- When the flow of instruction addresses is not known/expected, we have **control hazard**.
- `BEQ` in our design.
- This isn't an issue in single cycle because the next instruction is known by each cycle.
- With pipelining, you don't know the instruction until possibly multiple cycles later!

Resolving Control Hazards

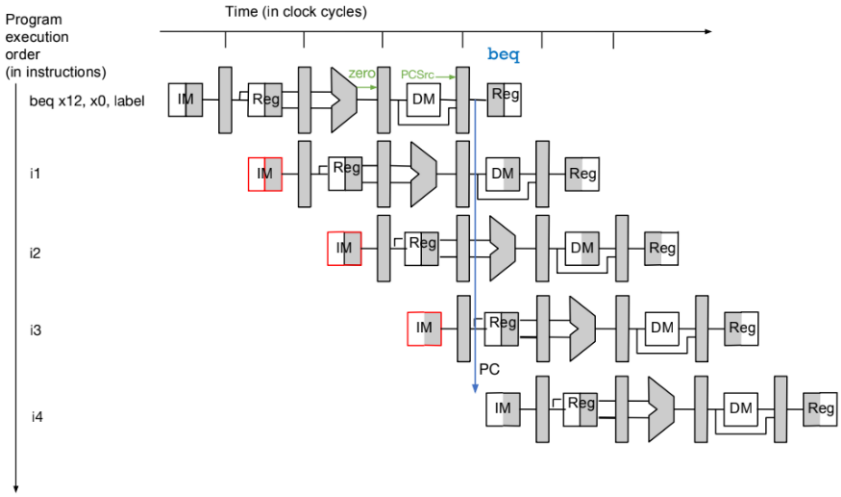
Answer: we can stall!

- For how many cycles? How?
- Any better solution(s)?

The **3 W's:**

- When the branch happens?
  - When to decode an undecoded instruction is a branch (e.g., `BEQ`)
  -
  - End of ID stage
- Where to branch to?
  - What is the target address of the branch (if taken) - i.e., pc + offset
  - End of EX or Beginning of Mem stage
- Whether to branch?
  - Predict if the branch is taken or not (for conditional)
  - End of Mem stage

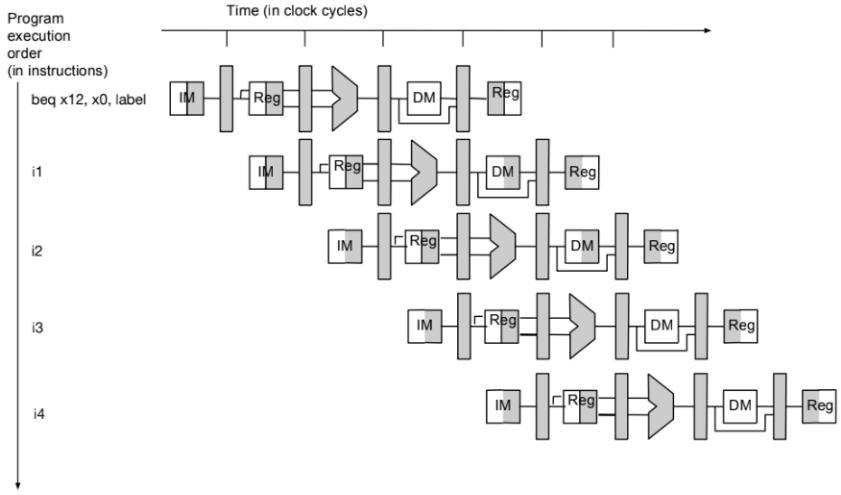
If we use stalling...



- 20% of instructions in a mix are control-flow instructions!
- We stall for 3 cycles (meaning a single control flow instruction takes 4 cycles)
- CPU Time $= 0.8 \times CPI \times CT + 0.2 \times CPI \times CT \times 4$
- This is a huge hit to performance!

Can we do better?

- How about **guessing?**
- Instead of waiting (stalling), let's pick one of the outcomes (i.e., {Taken, Not Taken}), and fetch the next instruction based on that!
- **Speculation:** An approach whereby the compiler or processor *guesses* the outcome of an instruction to remove it as a dependence in executing other instructions.
- For the sake of this example, let's always predict ***Not Taken!***



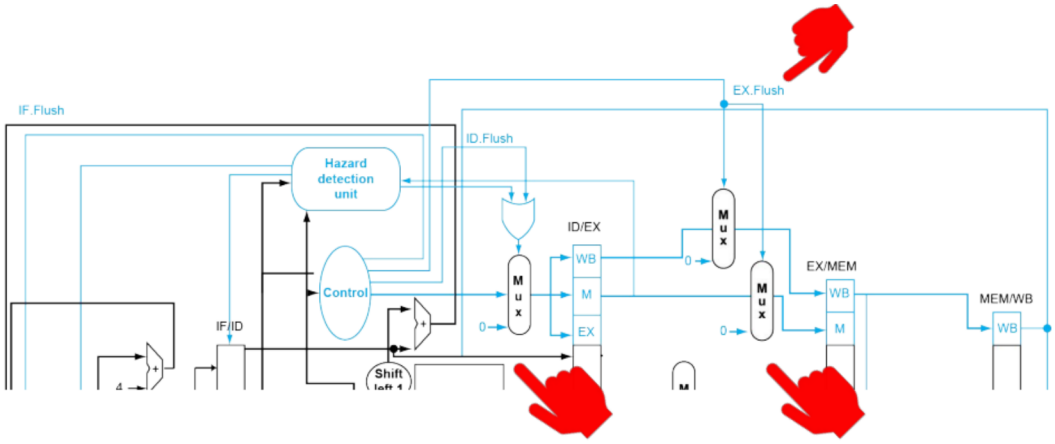
For any *not taken* branch, we don't have to stall at all!

Some stats:

- About 20% of instruction mix is control-flow
  - 50% of "forward" (i.e., if-then-else conditions) branches are taken.
  - 90% of "backward" (i.e., end of loops) branches are taken.
  - Overall, about 30% of all branch instructions are not-taken.
  - By predicting not-taken, we are fine in about 86% of the time (i.e., 80% non-branch + 30% accuracy for branch).

What would happen if the branch is **Taken**?

- This is a correctness issue!
- We have to clear those incorrectly fetched instructions. This is called a **"flush"**.



- If branch is Taken (i.e., Zero == 1), we need to flush.
- We need to flush FE, DE, and EX.
- At MEM we will realize that we need to flush.
- To *flush*, we have to connect Zero (in MEM) back to the controller. The controller then issues the Flush control signals.
- Referring to the program execution image above, we would need to flush instructions i1-i3.

Branch Miss Penalty

- If we make a wrong decision, we have to flush three instructions (i.e., same as stalling for three cycles).
- 70% of branches are Taken (i.e., our prediction accuracy - among branches - is only 30%).
- Another issue is the added complexity to the datapath!

Can we do better than this?

Resolving the Branch Sooner

- It turns out we can do better!
- Even when we make a wrong decision, at least flushing can happen sooner.
- This can be done as early as *Decode* stage! The "Where" and "Whether" out of the 3 W's needs to be changed.

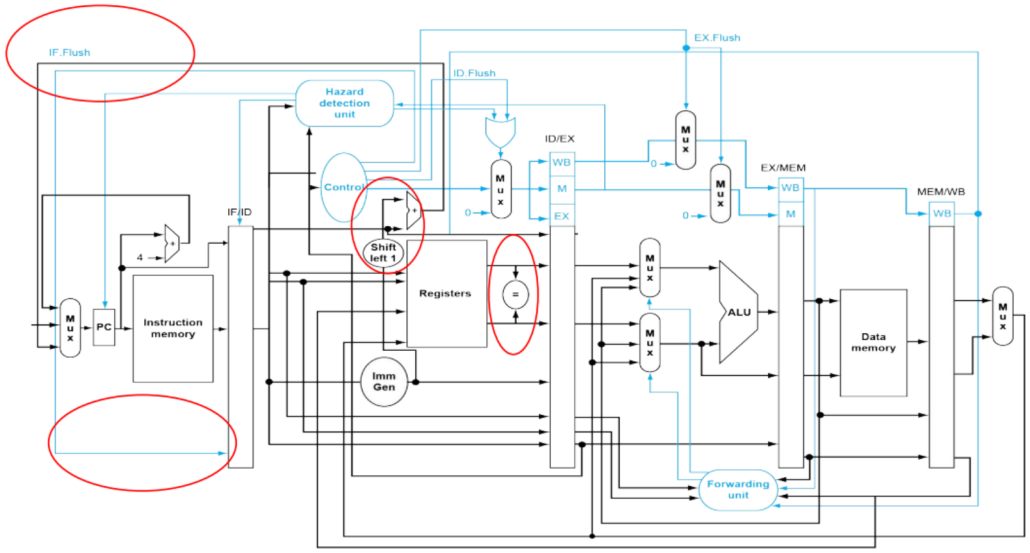
"Where"

- Offset is available in DE stage.
  - Move nextPC calculation to DE
  - Use ALU for PC + offset in DE stage

"Whether"

- rs1 and rs2 are also available in DE stage (for Zero).
  - Move the comparison between rs1 and rs2 after reading from the register file.

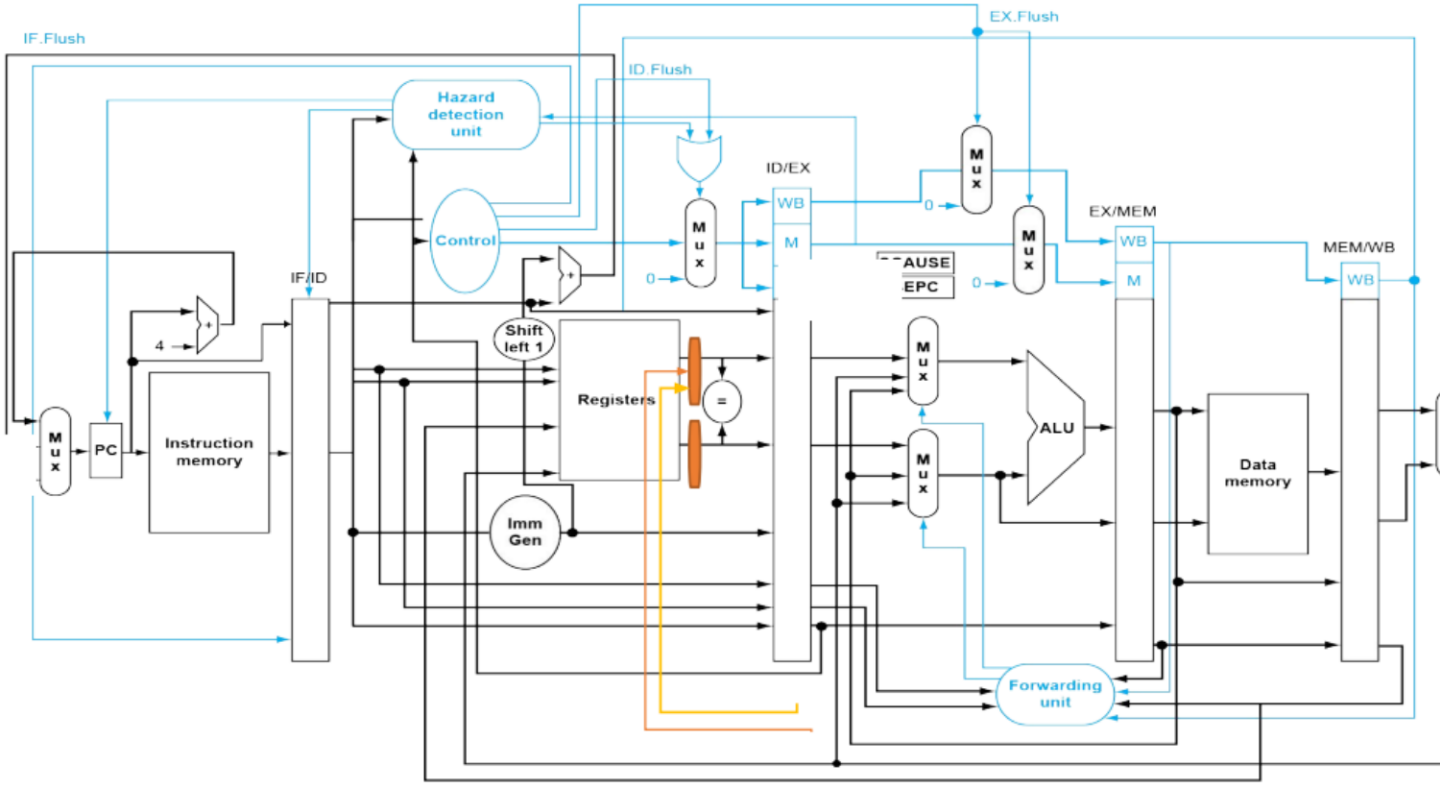
Resolving Branch in DE Stage



- New control signals are not shown.
  - The equal comparator next to the registers would output to the Control, if it detects branching.
  - This is how the instruction can be queued correctly.
- Only one instruction needs to be flushed!

This results in another problem: RAW Data Hazard?

- What if rs1 or rs2 are being written in the flushed instruction? You also have to revert the registers!



- Restore values from the buffer register.
- Forwarding condition?
  - Similar, only everything shift one cycle to the left.
    - check IF/ID.rs1 with EX/MEM or MEM/WB.rd AND RegWrite signal
- Hazard and Stalling condition?
  - For load-use hazard, we need to **stall** if needed.
  - Similar, again only shifted one cycle to the left.

Role of the Compiler

- Reordering the instruction
  - **Delay Slot:** moving an independent instruction between the branch and the next instruction.
- Increasing the chance of not-taken
- Loop unrolling
- ...