Introduction to Pipelined CPU

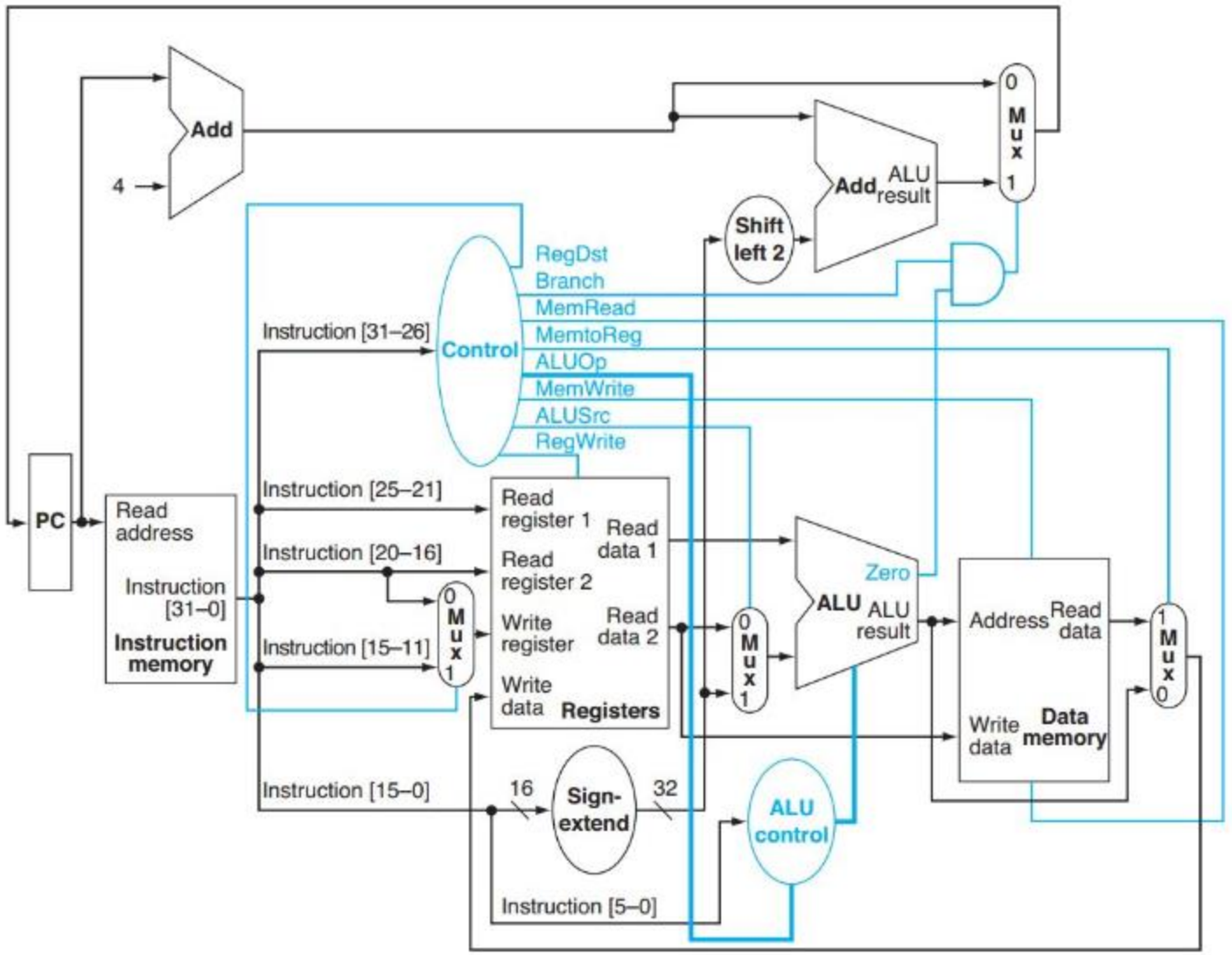
Important note about Lab 4 Report:

- **No need to test the instructions that you converted into binary!**
- **But make sure to provide binary conversions in the report**

Let's talk about the Single Cycle CPU

- All instructions take one cycle for execution
- If it is memory, branch ,arithmetic;
- Makes no difference

The goal in computer architecture (ISA) to be able to make instructions take less cycles.

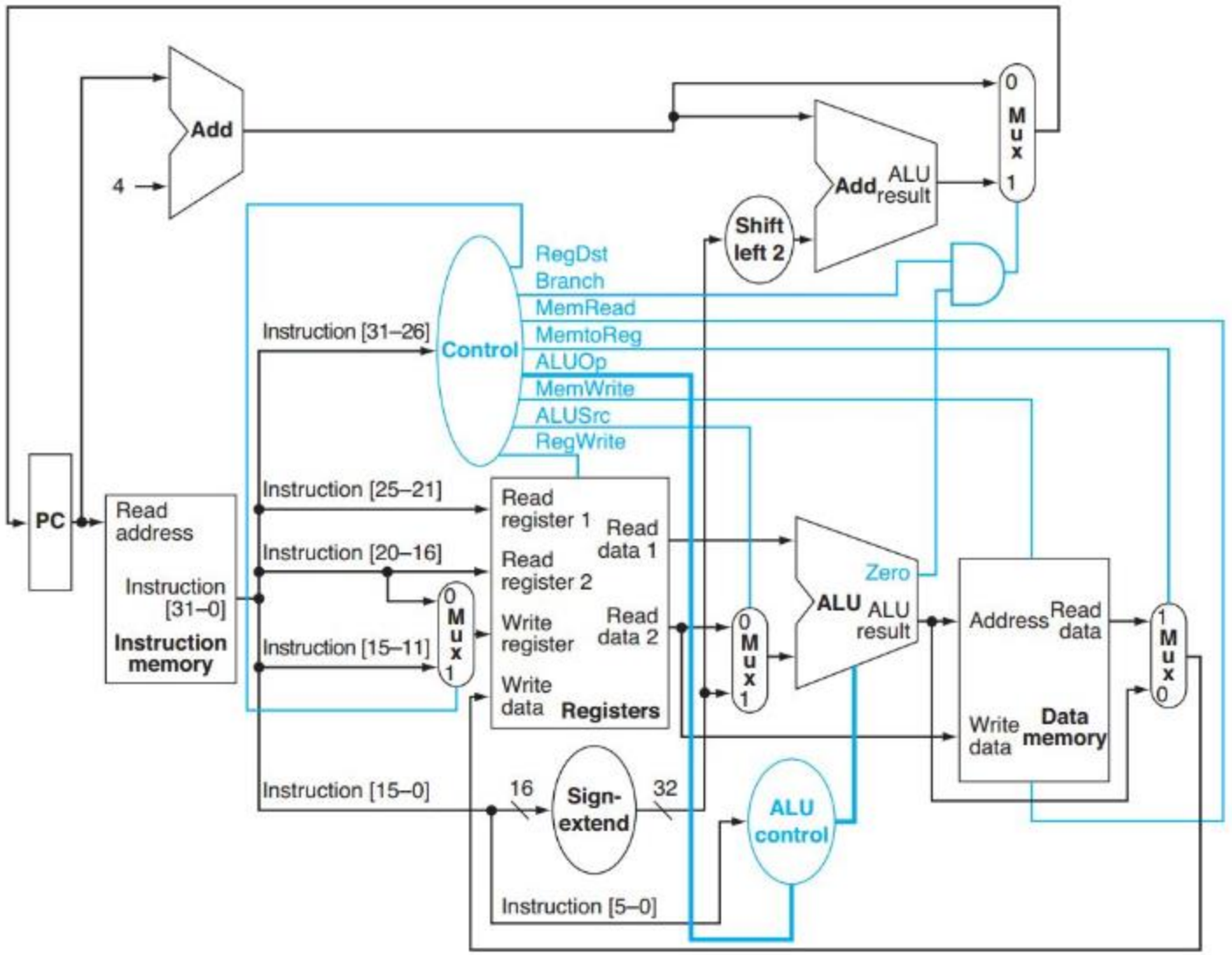


Let's talk about the Single Cycle CPU

This means the cycle time should be minimized as much as possible.

But the cycle time for single cycle datapath is the maximum of execution times for all instructions.

Might be hard or even impossible to reduce this time.

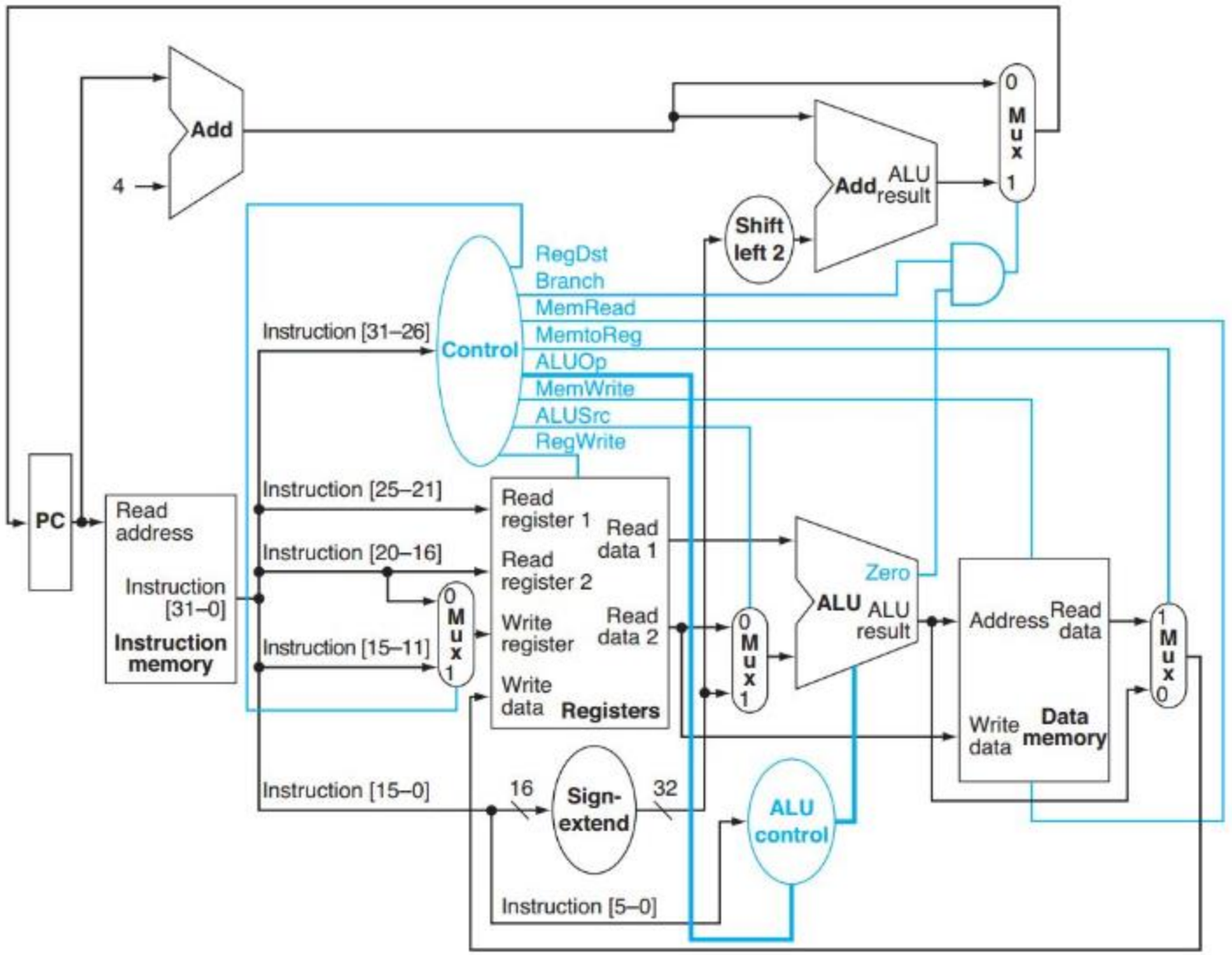


Let's talk about the Single Cycle CPU

This means the cycle time should be minimized as much as possible.

But the cycle time for single cycle datapath is the maximum of execution times for all instructions.

Might be hard or even impossible to reduce this time.

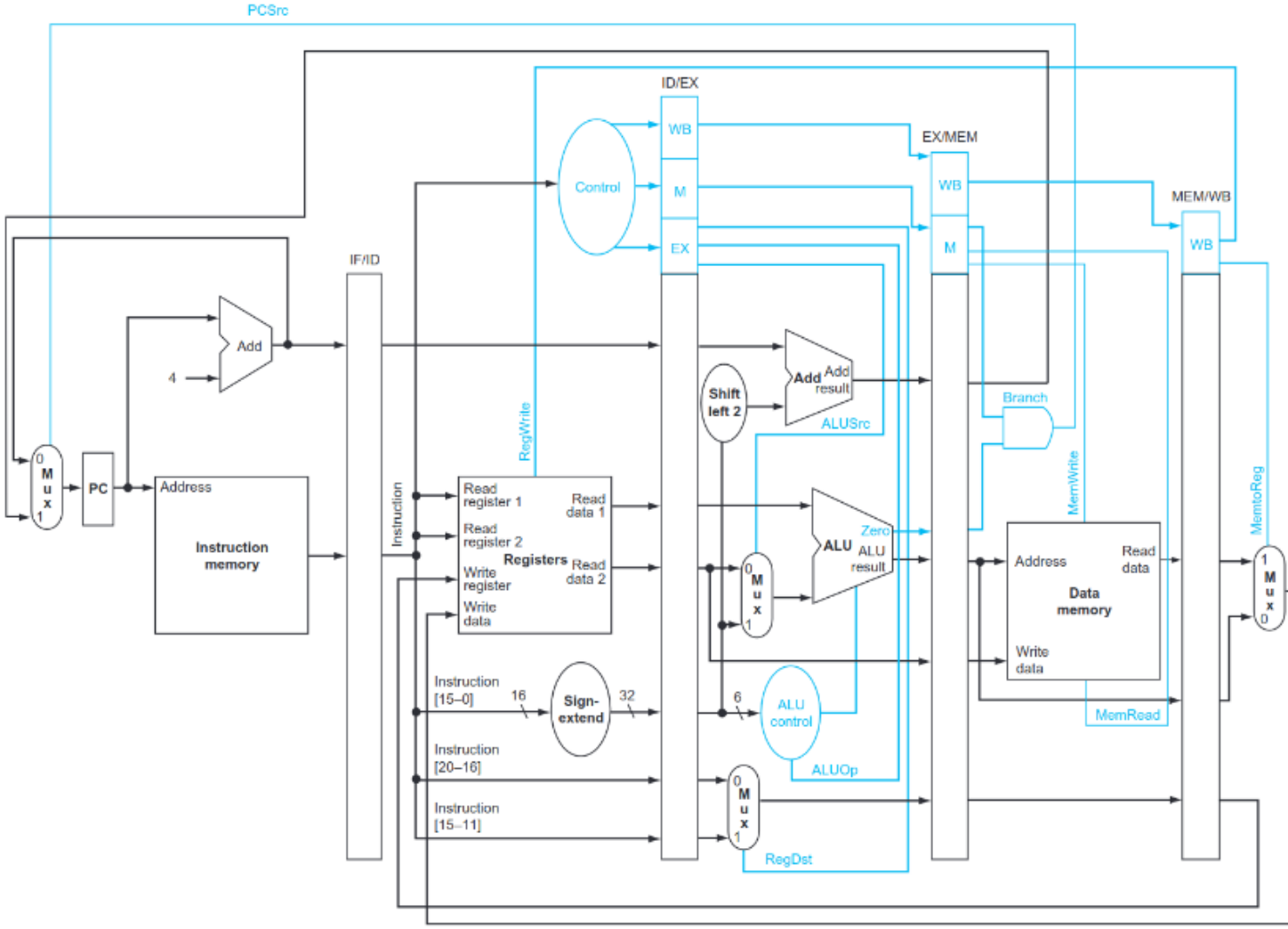


Maybe dividing the single cycle into finer granulatiy?

A better idea is to divide single cycle into more cycles and do less operations in one cycle instead of executing the whole instruction.

This will also allow the parallel execution of instructions and better resource utilization.

Parallel execution of instructions: ILP – instruction level parallelism

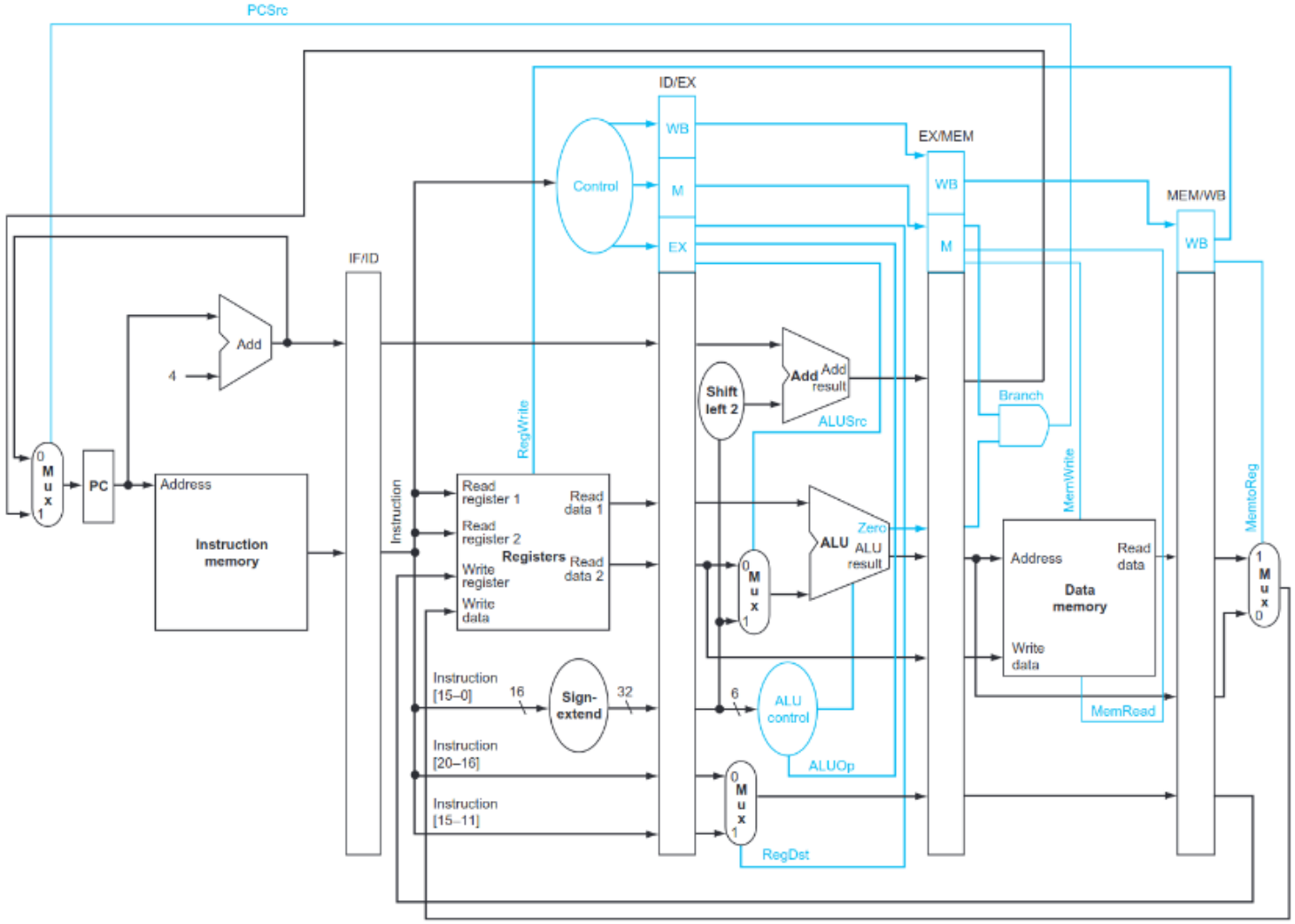


This idea is similar to assembly line in car production, or sandwich making procedure in Subway.

Instruction Pipelining

Stages in this pipeline:

1. Instruction Fetch
2. Instruction Decode
3. Execution
4. Memory
5. Writeback

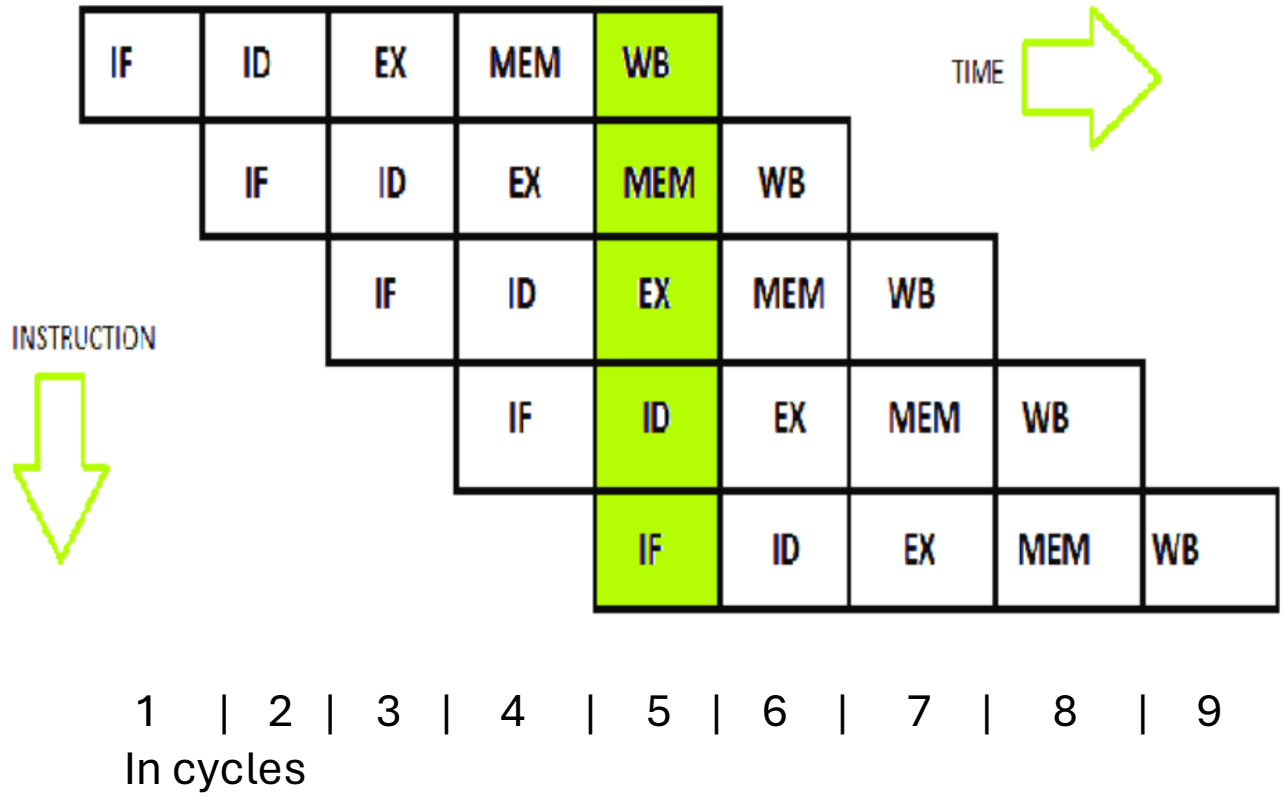


Simple execution model in pipelined model

Each stage is involved in each cycle (except for first several ones)

After first 4 cycles, we will achieve 5 instructions per cycle.
This is the optimal speed. But there are some dependency problems.

What are those?



Dependency Issues

This is called hazard in the datapath. Why do we have hazards?

1. Data hazards:
   - The needed data is not ready (not written back yet)!

   Solution: Forward the data from write back stage maybe?
   Or Stall the pipeline if data is not ready

2. Control hazards:
   - This is related to branch instructions. Not sure/ don't know which instruction is next when we need to fetch it!

   Solution: Predict the branch output or Stall the pipeline!

3. Structural hazards:
   - No available resource -> can't do the instruction!

   Solution: Add more hardware:)

Examples for Hazards

How to prevent hazard here?

Consider the following set of instructions in a 5-stage pipeline.
Operands are read in ID.
MEM is memory Write for result; RW is Register Write for result

ADD **R3**, R6, R5 - Results to be written in R3
SUB R4, **R3**, R5 - R3 has one of the operand
OR R6, **R3**, R7 - R3 has one of the operand
AND R8, **R3**, R7 - R3 has one of the operand
XOR R12, **R3**, R10 - R3 has one of the operand

R3 is accessed in READ mode; expect the result of ADD to be available in **R3**

But result of ADD written in R3 at t5

	t1	t2	t3	t4	t5	t6	t7	t8	t9
ADD R3, R6, R5	IF	ID	IE	MEM	RW R3	--	--	--	--
SUB R4, R3, R5	--	IF	ID R3	IE	MEM	RW	--	--	--
OR R6, R3, R7	--	--	IF	ID R3	IE	MEM	RW	--	--
AND R8, R3, R9	--	--	--	IF	ID R3	IE	MEM	RW	--
XOR R10, R3, R11	--	--	--	--	IF	ID R3	IE	MEM	RW

Note when each instruction is accessing R3

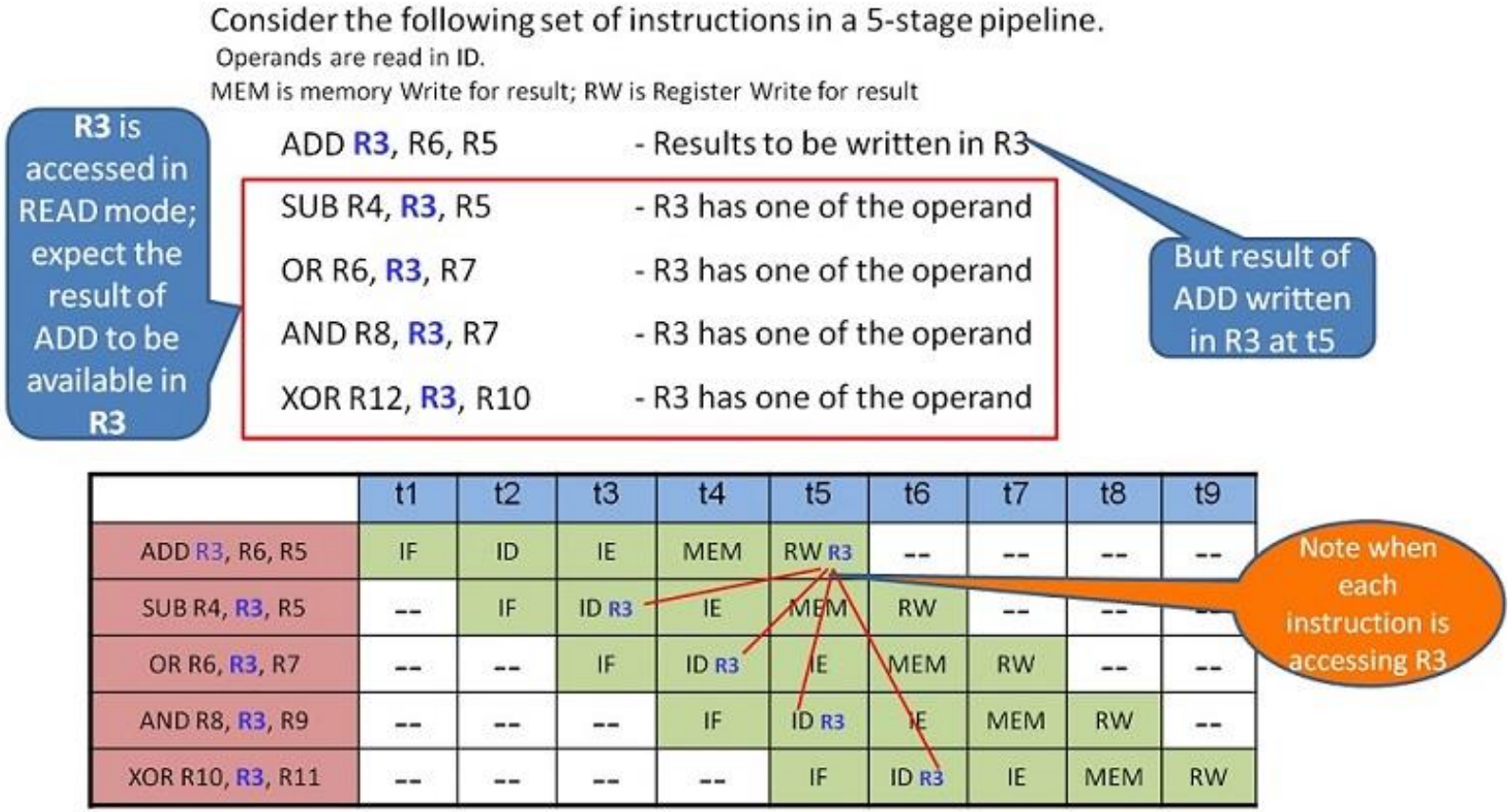
Examples for Hazards

Solution:

1. Insert stalls/NOPs before SUB instruction

ADD R3, R6, R5
NOP
NOP
NOP
SUB R4, R3, R5

In this case, SUB instruction will start at t5.

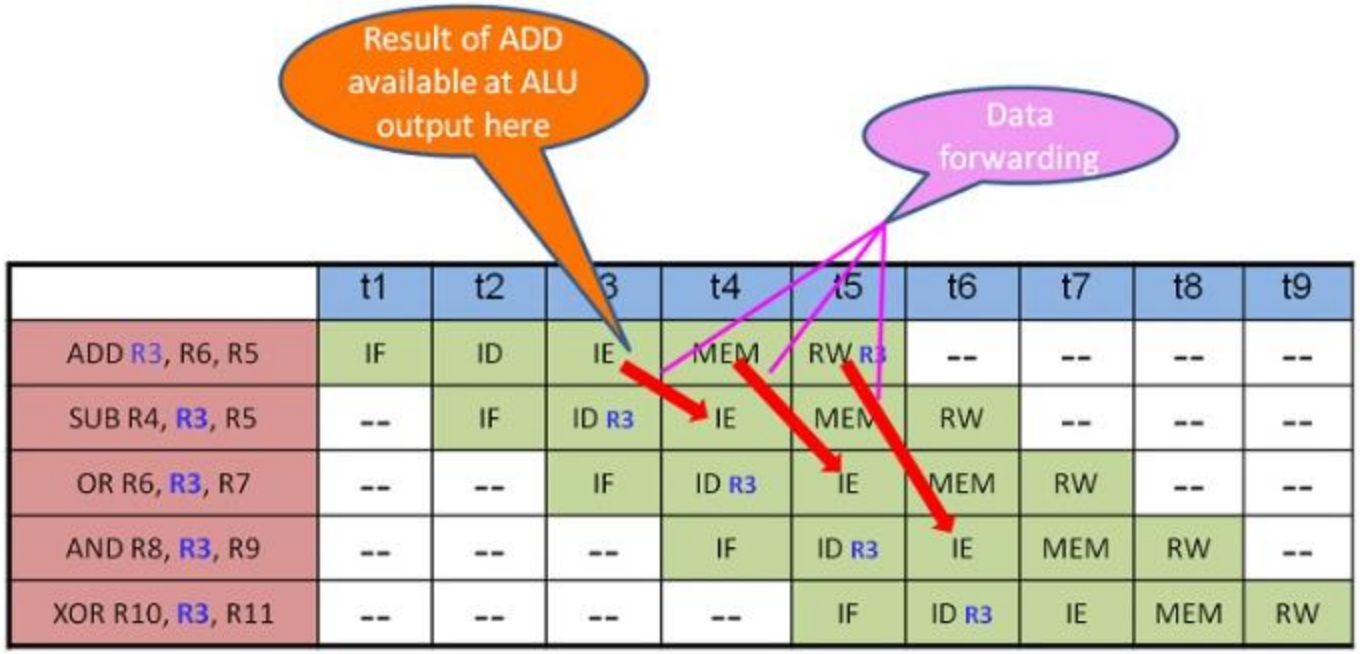


	t1	t2	t3	t4	t5	t6	t7	t8	t9
ADD R3, R6, R5	IF	ID	IE	MEM	RW R3	--	--	--	--
SUB R4, R3, R5	--	IF	ID R3	IE	MEM	RW	--	--	--
OR R6, R3, R7	--	--	IF	ID R3	IE	MEM	RW	--	--
AND R8, R3, R9	--	--	--	IF	ID R3	IE	MEM	RW	--
XOR R10, R3, R11	--	--	--	--	IF	ID R3	IE	MEM	RW

Examples for Hazards

Solution:

2. Forward the result of execution stage in instruction 1 to other instructions.



	t1	t2	t3	t4	t5	t6	t7	t8	t9
ADD R3, R6, R5	IF	ID	IE	MEM	RW R3	--	--	--	--
SUB R4, R3, R5	--	IF	ID R3	IE	MEM	RW	--	--	--
OR R6, R3, R7	--	--	IF	ID R3	IE	MEM	RW	--	--
AND R8, R3, R9	--	--	--	IF	ID R3	IE	MEM	RW	--
XOR R10, R3, R11	--	--	--	--	IF	ID R3	IE	MEM	RW

Some Information about Data hazards

Data Hazards classification

Data hazards are classified into three categories based on the order of READ or WRITE operation on the register and as follows:

- **RAW (Read after Write) [Flow/True data dependency]**

This is a case where an instruction uses data produced by a previous one. Example

```
ADD R0, R1, R2
SUB R4, R3, R0
```

- **WAR (Write after Read) [Anti-Data dependency]**

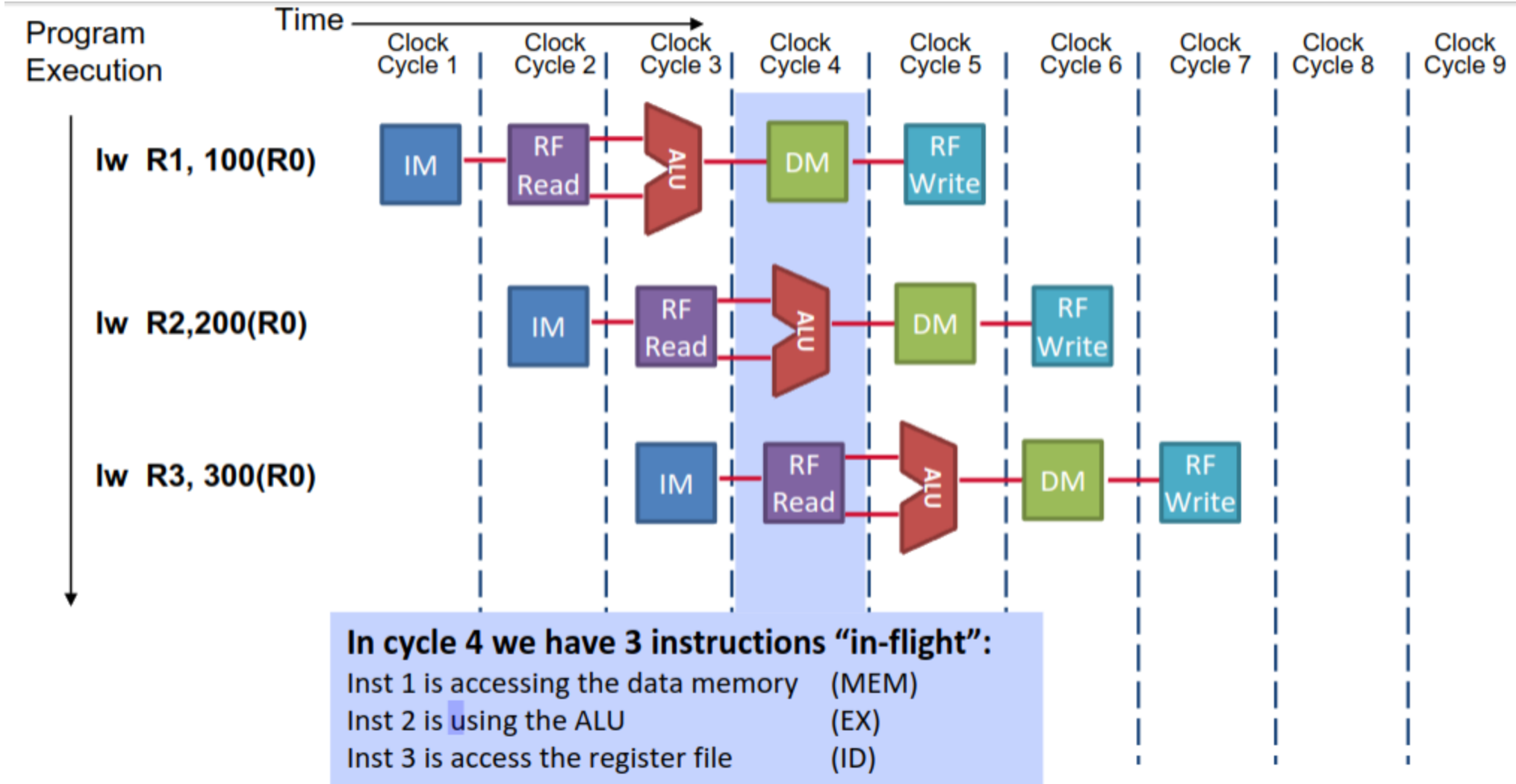
This is a case where the second instruction writes onto register before the first instruction reads. This is rare in a simple pipeline structure. However, in some machines with complex and special instructions case, WAR can happen.

```
ADD R2, R1, R0
SUB R0, R3, R4
```

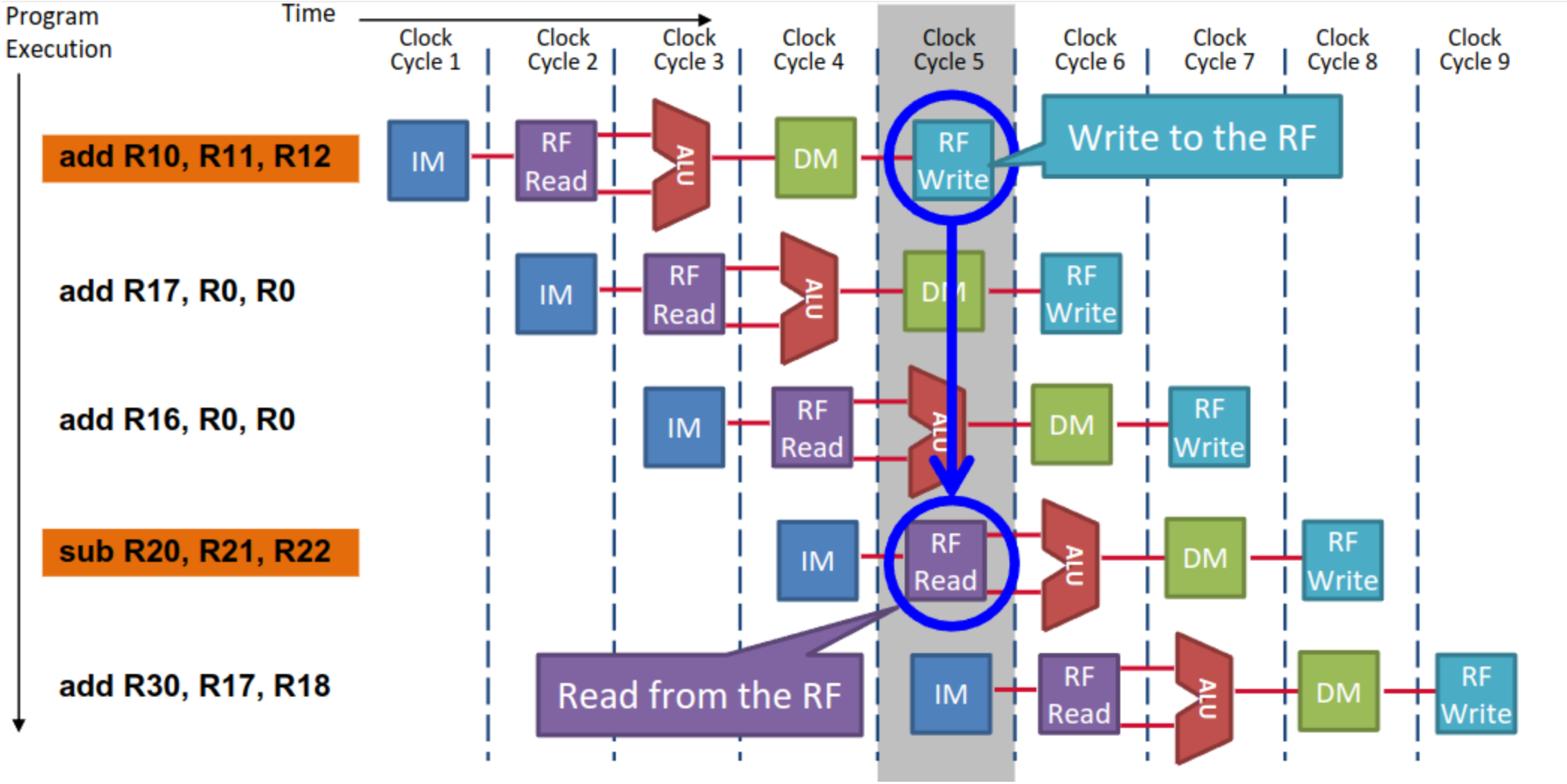
- **WAW (Write after Write) [Output data dependency]**

This is a case where two parallel instructions write the same register and must do it in the order in which they were issued.

```
ADD R0, R1, R2
SUB R0, R4, R5
```

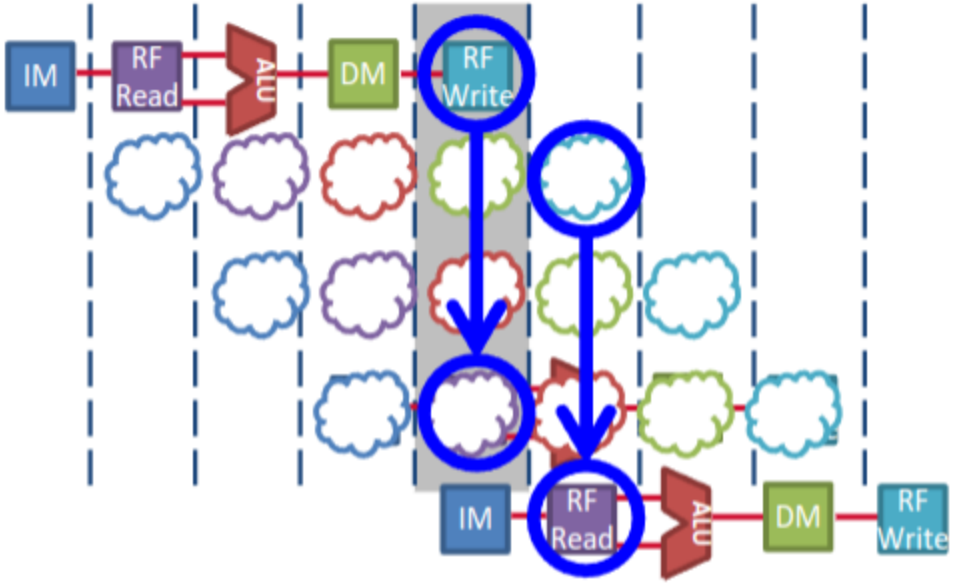

Program Execution
Time
Clock Cycle 1
Clock Cycle 2
Clock Cycle 3
Clock Cycle 4
Clock Cycle 5
Clock Cycle 6
Clock Cycle 7
Clock Cycle 8
Clock Cycle 9
lw R1, 100(R0)
lw R2,200(R0)
lw R3, 300(R0)
IM
RF Read
ALU
DM
RF Write
In cycle 4 we have 3 instructions "in-flight":
Inst 1 is accessing the data memory (MEM)
Inst 2 is using the ALU (EX)
Inst 3 is access the register file (ID)

Pipelining: interference



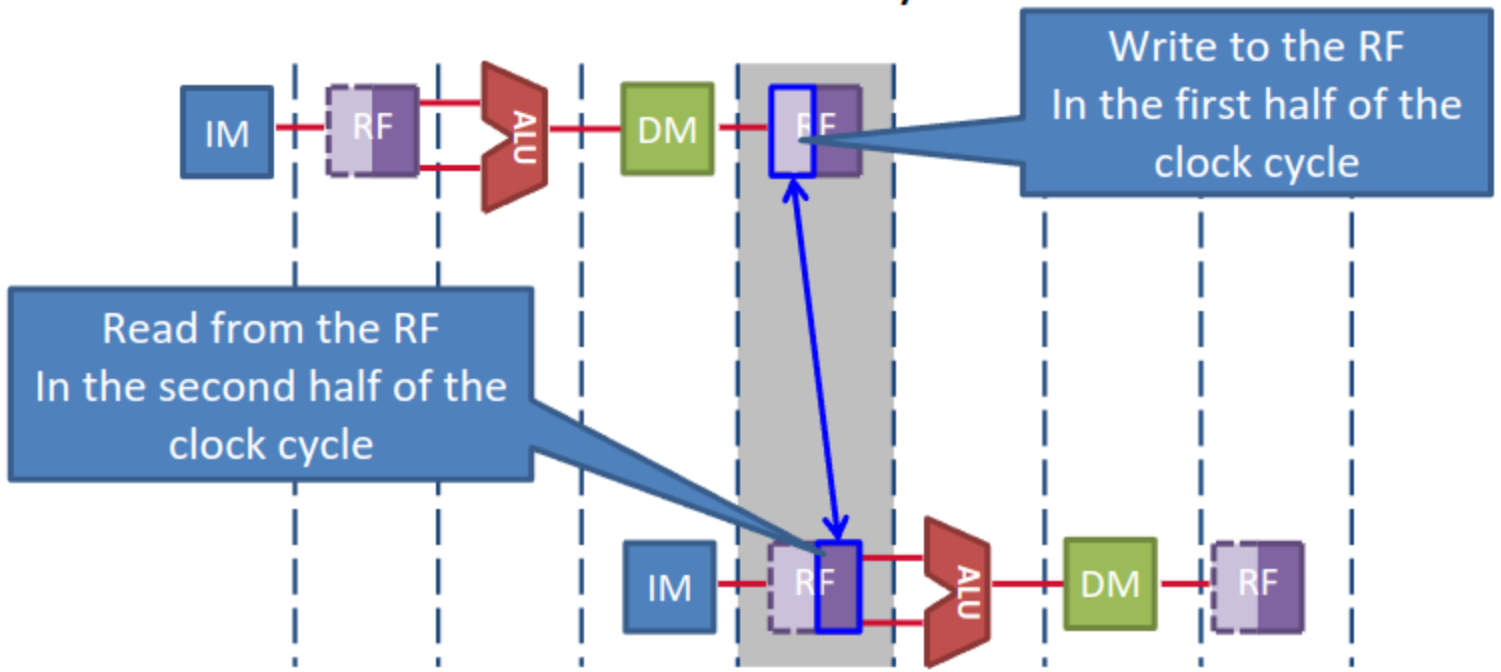
Instructions interfering in the pipeline

- **Problem: both instructions want the same resource**
  - One wants to **write**
  - One wants to **read**
  - But we only have **one register file**
- **How can we fix this?**
  - Delay the reading instruction
    - Bad for performance
  - Never write conflicting instructions?
    - Bad for performance
  - Or we could cheat…

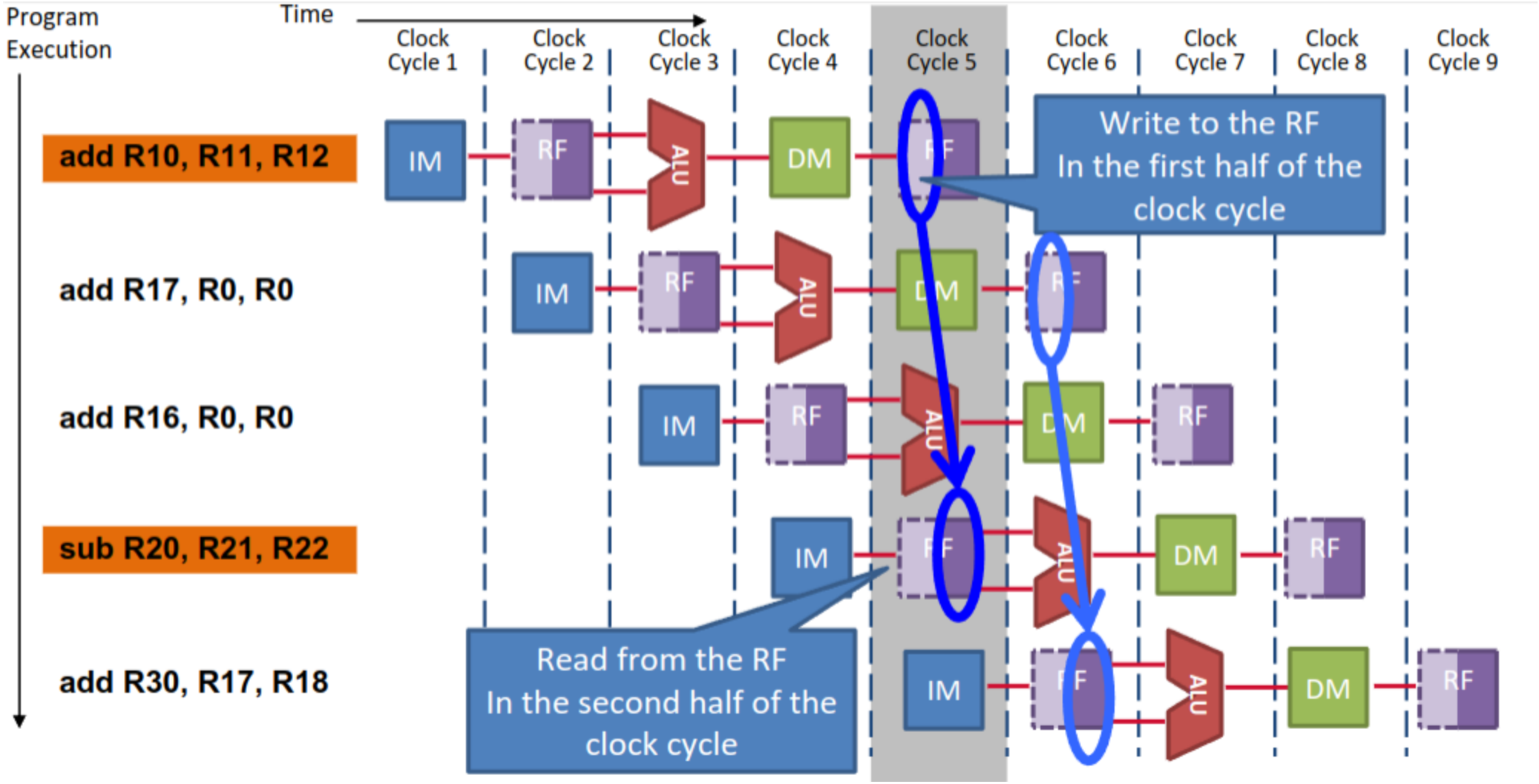


Fixing the register file (special case)

- The problem is that we want to **read** and **write** at the same time
- We can build a **double-pumped register file** that:
  - **Writes** in the 1st half of the clock cycle
  - **Reads** in the 2nd half of the clock cycle



Now this works



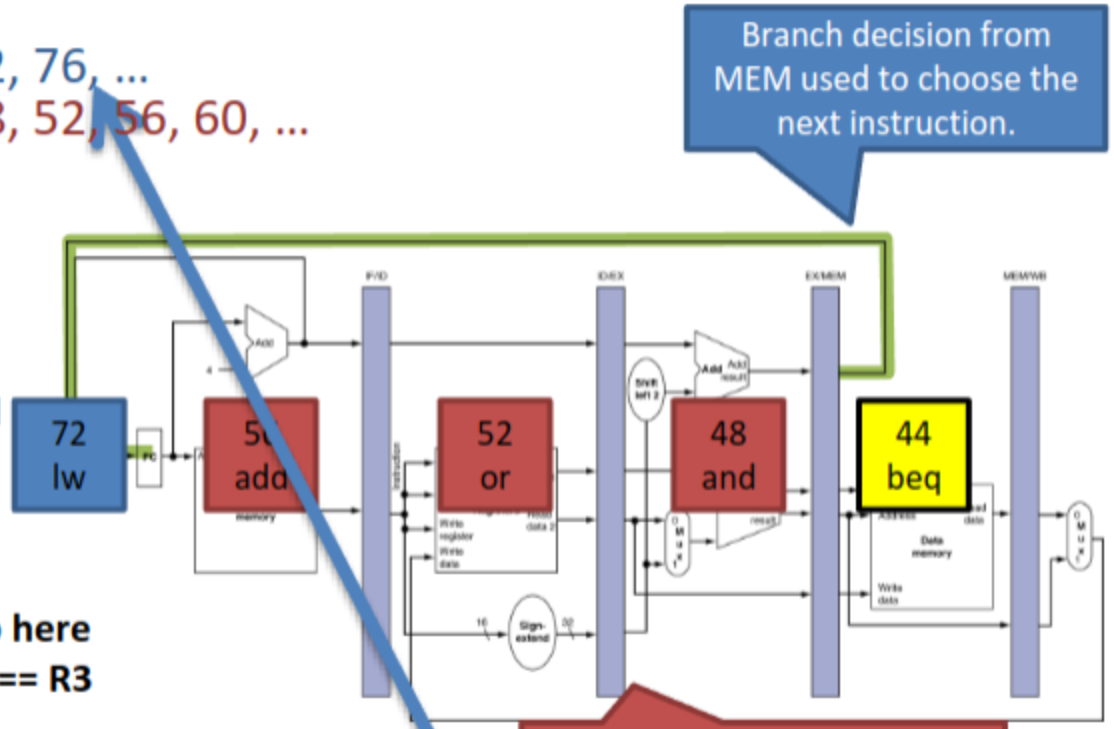
Control hazards: branches

- Let's look at the following code:
  - If branch is taken: 40, 44, 72, 76, …
  - If branch is not taken: 40, 44, 48, 52, 56, 60, …

Addr	Instruction
40	add R30, R30, R30
44	beq R1, R3, 6
48	and R12, R2, R5
52	or R13, R6, R2
56	add R14, R2, R2
60	…
72	lw R4, 50(R7)
76	…

Execute all of these if R1 != R3

Jump here if R1 == R3



Branch decision from MEM used to choose the next instruction.

Which instructions are already in the pipeline?

48, 52, 56: **They've already started executing before we decided to take the branch!**

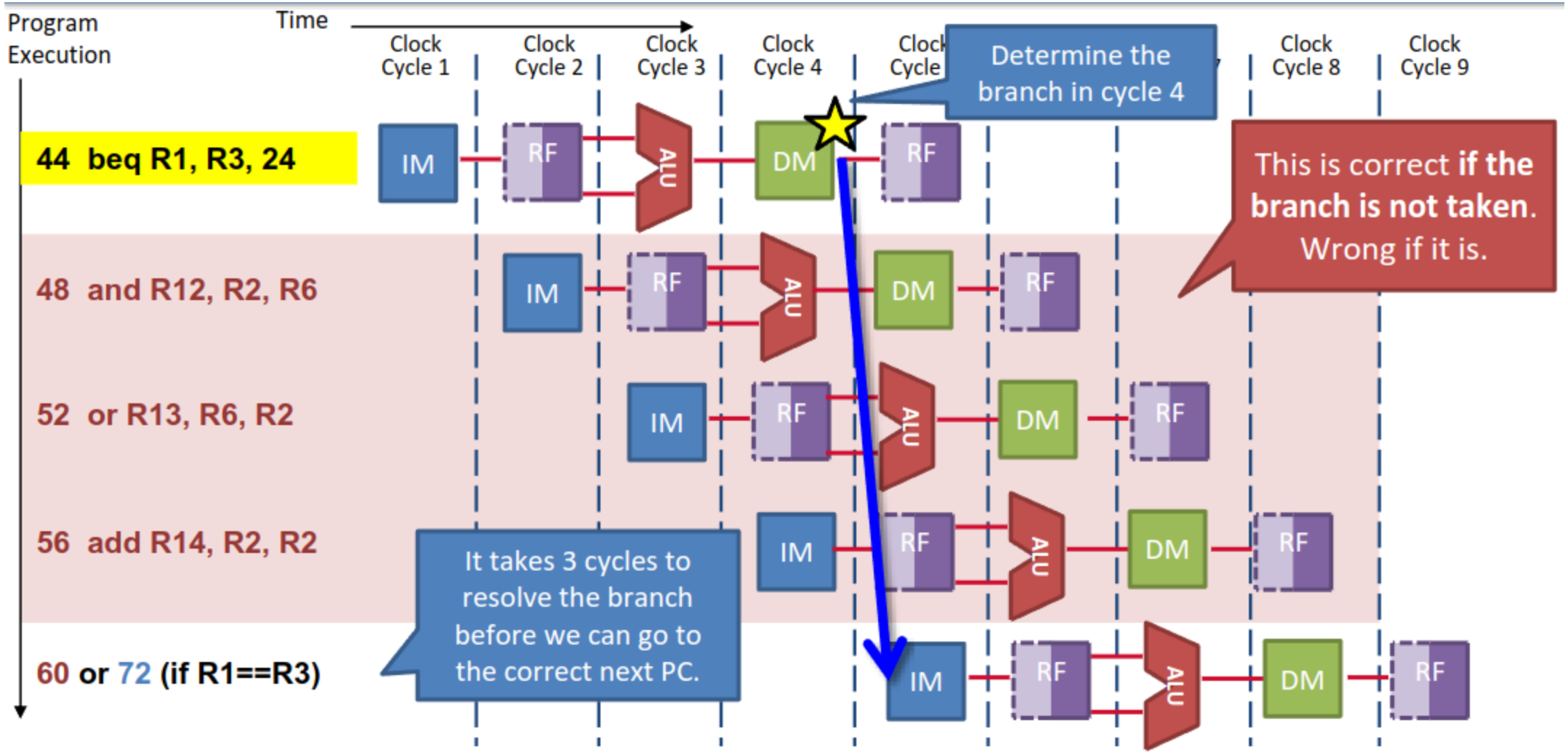
Q: In which stage do we determine if the branch is taken?

- EX
- MEM
- WB

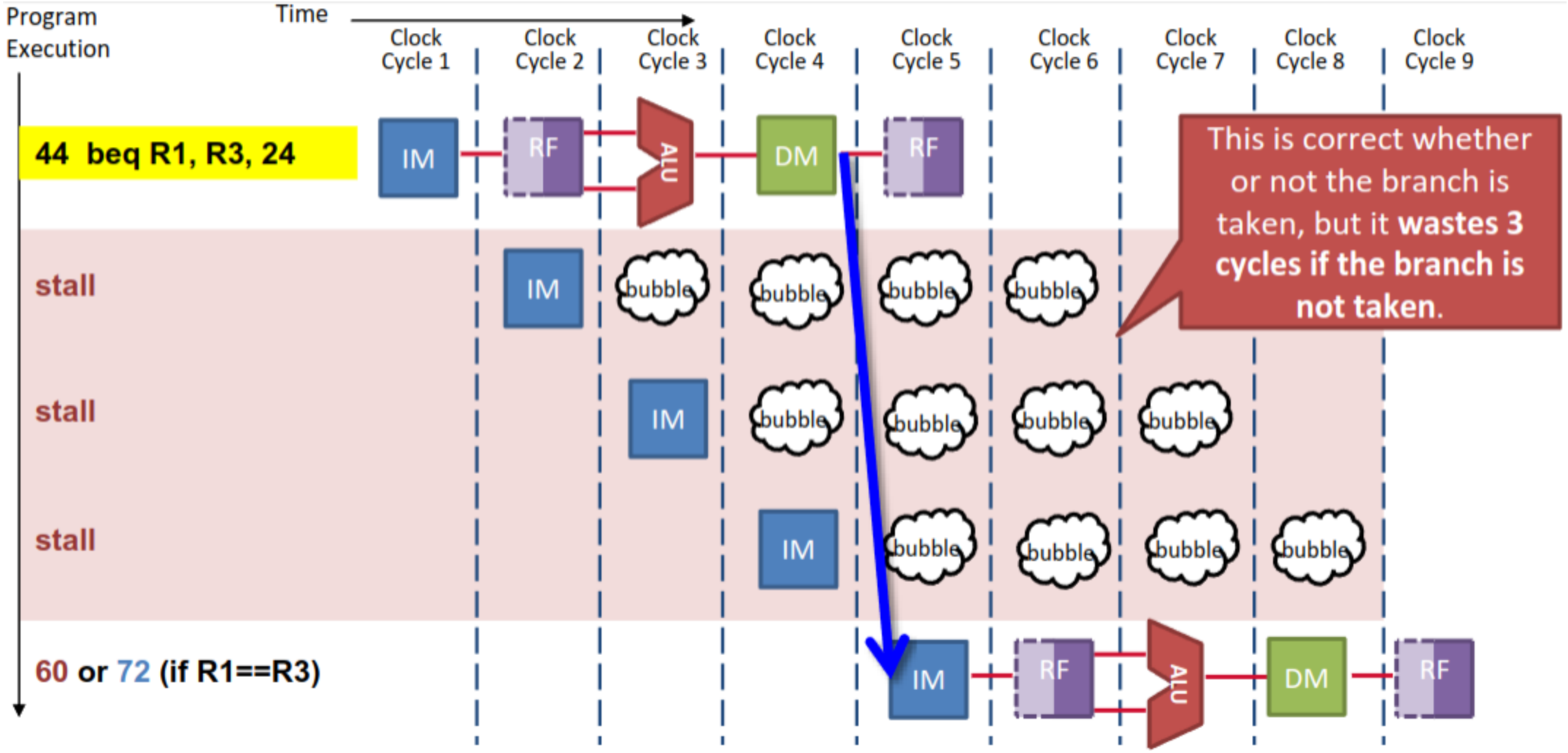
A: MEM

The comparison is done in EX, but we don't use the result until the next stage, MEM.

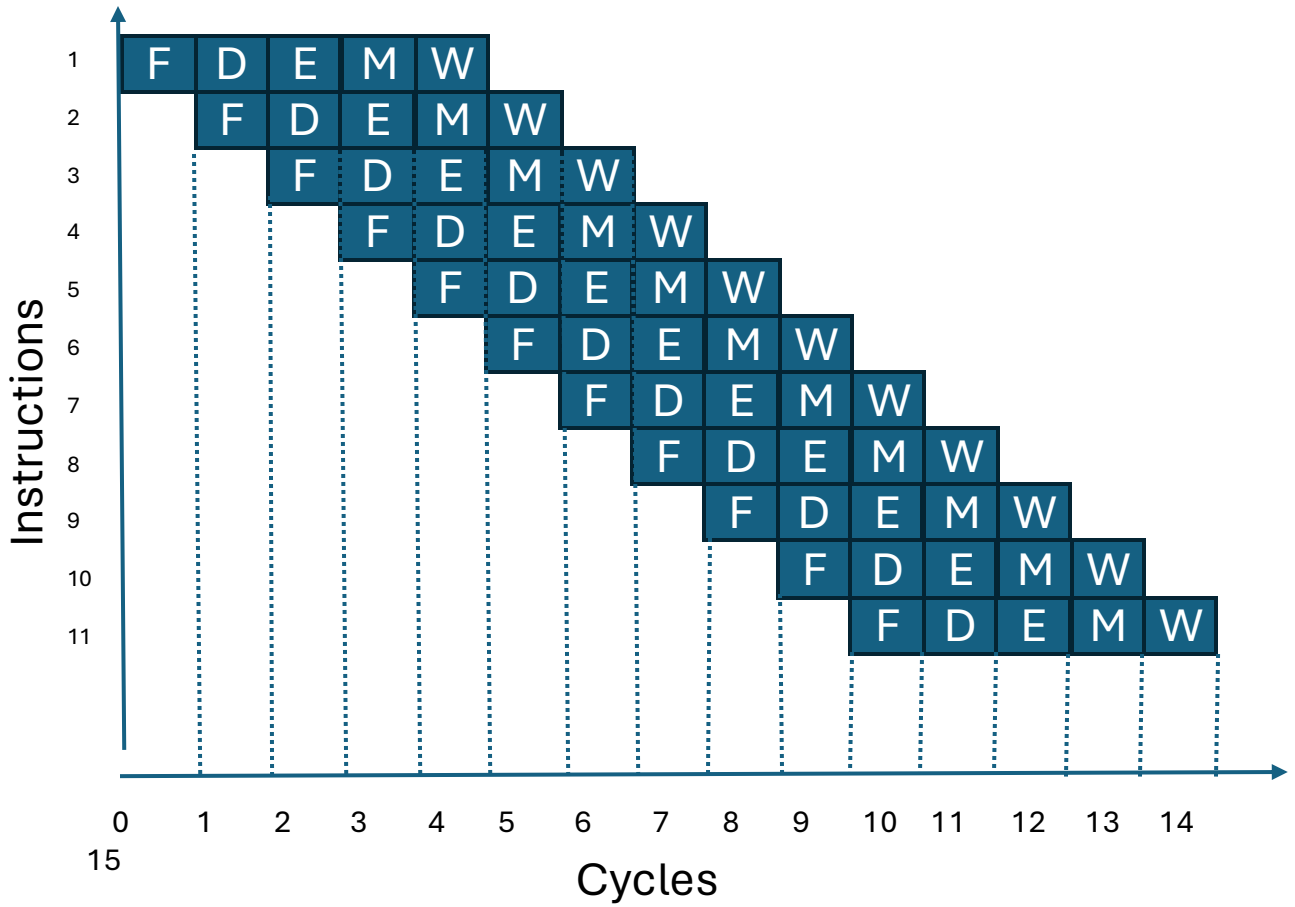
Branch hazard



Stall the pipeline?



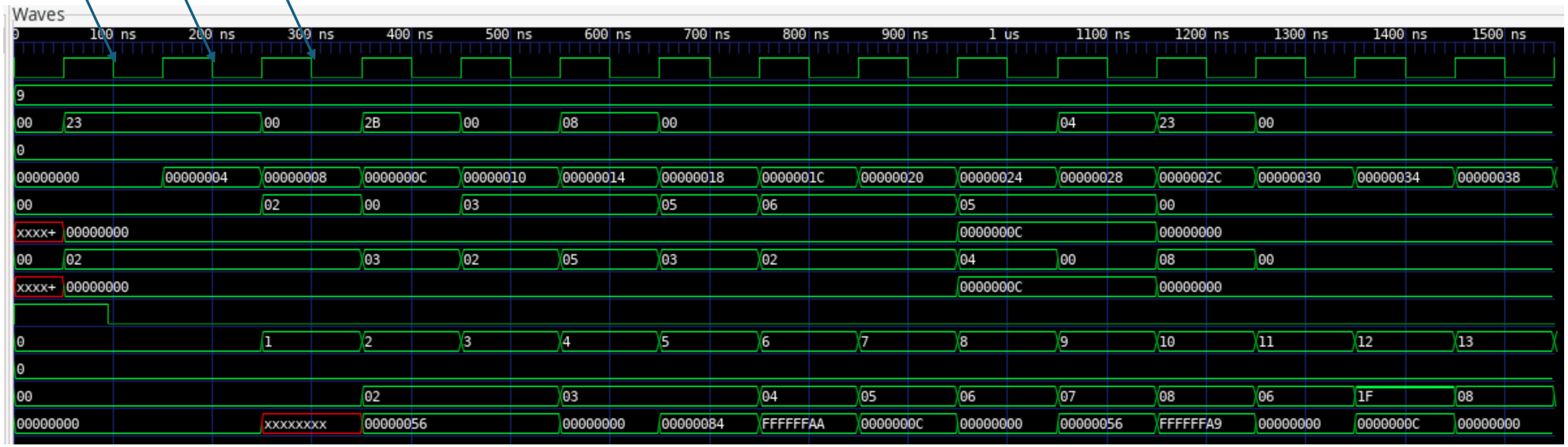
Why 15 cycles?



Execution stages:
F: Fetch
D: Decode
E: Execute
M: Memory
W: Write Back

Why 15 cycles?

Cycle 1 Cycle 2 Cycle 3



For lab 5:

- Analyze the code given in the github page
- Find out the hazards and suggest the solutions