

The Processor: Basic Pipeline



Caiwen Ding

Department of Computer Science and Engineering
University of Connecticut

CSE3666: Introduction to Computer Architecture

Outline

- Concept of pipeline
- Implementation of a 5-stage pipeline
- Pipeline Hazards

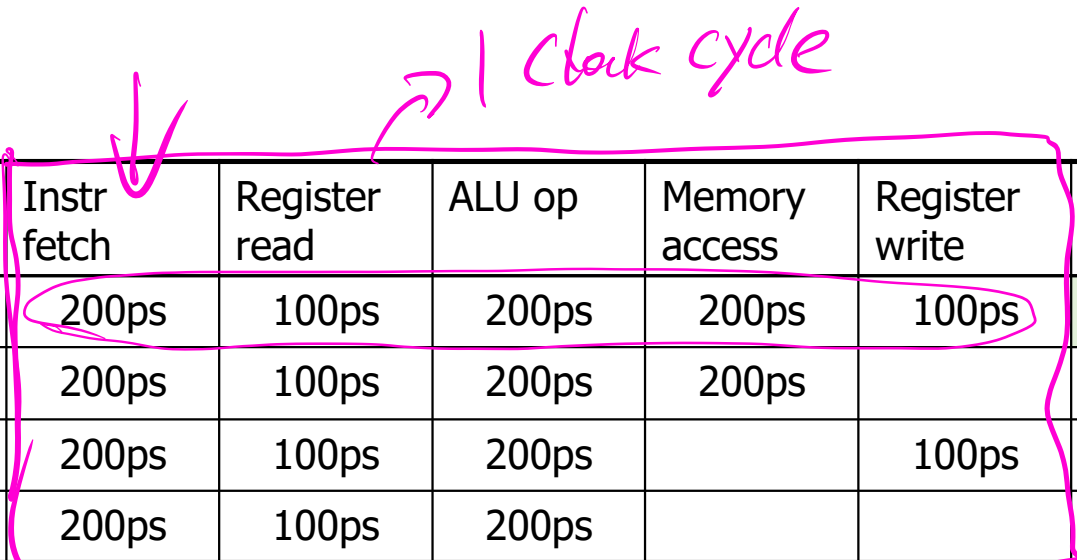
? { why?
What?
How?

Reading: Sections 4.6 and 4.7.

Skip discussions on hazards in Section 4.6, for now.

Review Clock Cycle Time of Single-Cycle Processor

- Assume time for stages is
 - 100ps for register read or write
 - Main control and register read can be done at the same time
 - 200ps for other stages




Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100ps	200ps	200ps	100ps	800ps
sw	200ps	100ps	200ps	200ps		700ps
R-format	200ps	100ps	200ps		100ps	600ps
beq	200ps	100ps	200ps			500ps

Performance Issues with Single-Cycle Implementation

- The cycle time is the same for all instructions
 - Not feasible to vary period for different instructions
- Longest delay determines clock period
 - **Critical path**: the load (LW) instruction

Instruction memory → Register file → ALU → Data memory → Register file



- Violates design principle
 - Making the common case fast
- How can we improve the performance ?

Five steps in RISC-V instruction execution

Observations in the single-cycle RISC-V execution.

The execution of an instruction has five important steps:

1. Fetch instruction from memory (IF)
2. Read register file and decode instructions (ID)
3. Use ALU to compare numbers or to compute results/addresses (EX)
4. Access data memory (MEM)
5. Write the result into register (WB)

Another Important Observation

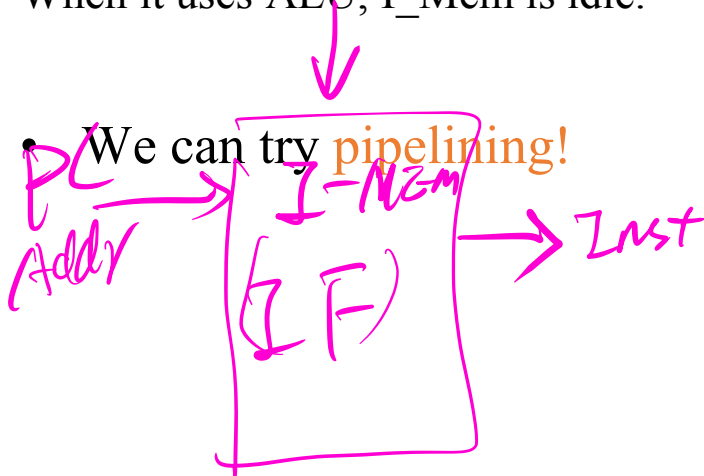
- An instruction does not need to do all stages at the same time.
 - A hardware module is idle in most part of a cycle.

Instruction memory → Register file → ALU → Data memory → Register file

Seq!

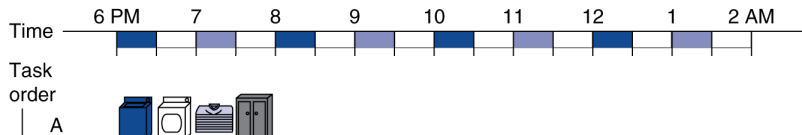
For example, an instruction only uses I-Mem at beginning of a cycle.

When it uses ALU, I_Mem is idle.



Pipelining Analogy

- 4-step laundry:
 - Place one dirty load of clothes in the **washer**.
 - When the washer is finished, place the wet load in the **dryer**.
 - When the dryer is finished, place the dry load on a table and **fold**.
 - When folding is finished, ask your roommate to **take out** the clothes.

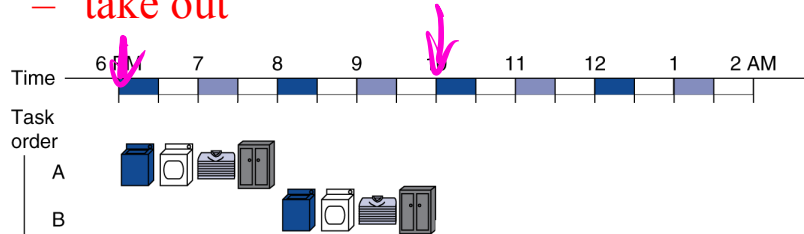


1 task: 4 steps (2 hours)

Pipelining Analogy

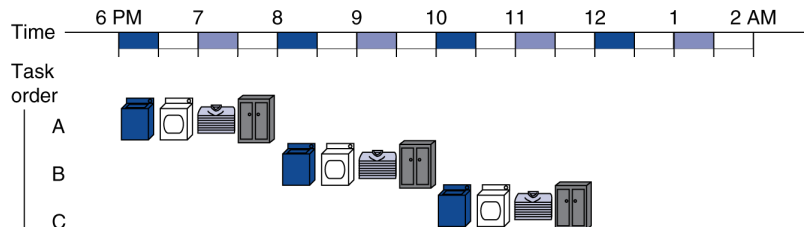
- Laundry: multiple tasks

- washer
- dryer
- fold
- take out



Pipelining Analogy

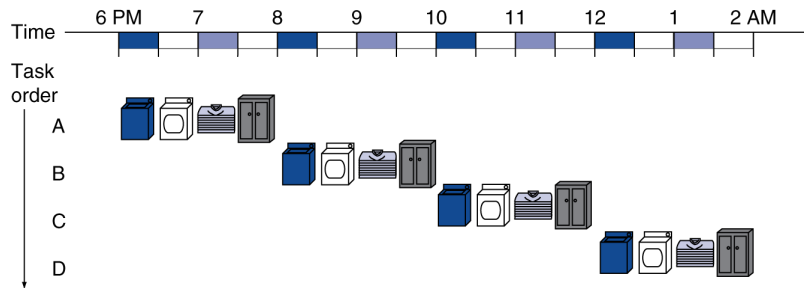
- Laundry: Multiple loads
 - washer
 - dryer
 - fold
 - take out



Pipelining Analogy

- Laundry: Multiple tasks

- washer
- dryer
- fold
- take out



4 tasks (6pm – 2am)

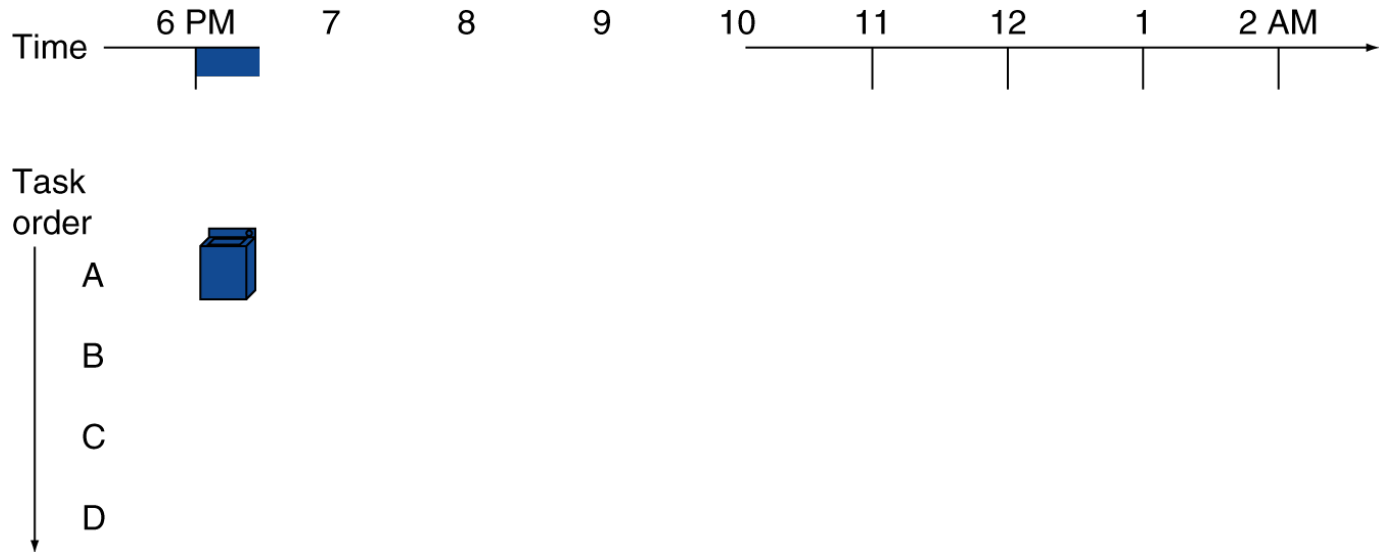
Question: how many “hardware” do we have?

1 set

Execution time of Non-stop n tasks:
 $4n$

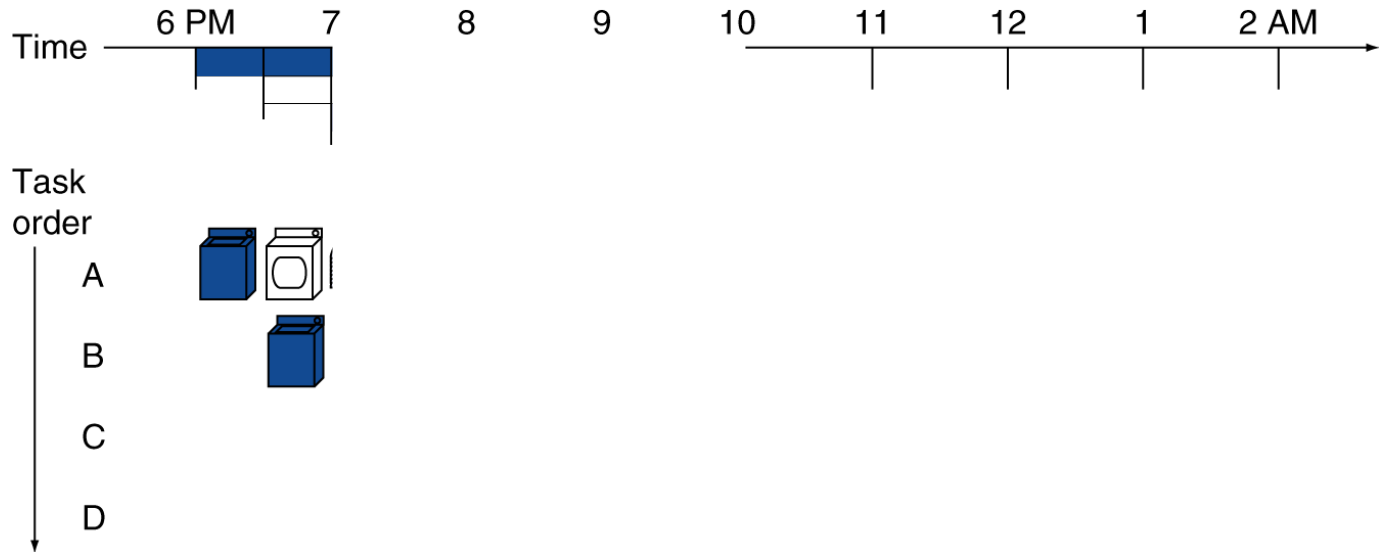
Pipelining Analogy

- Pipelined laundry: overlapping execution



Pipelining Analogy

- Pipelined laundry: overlapping execution

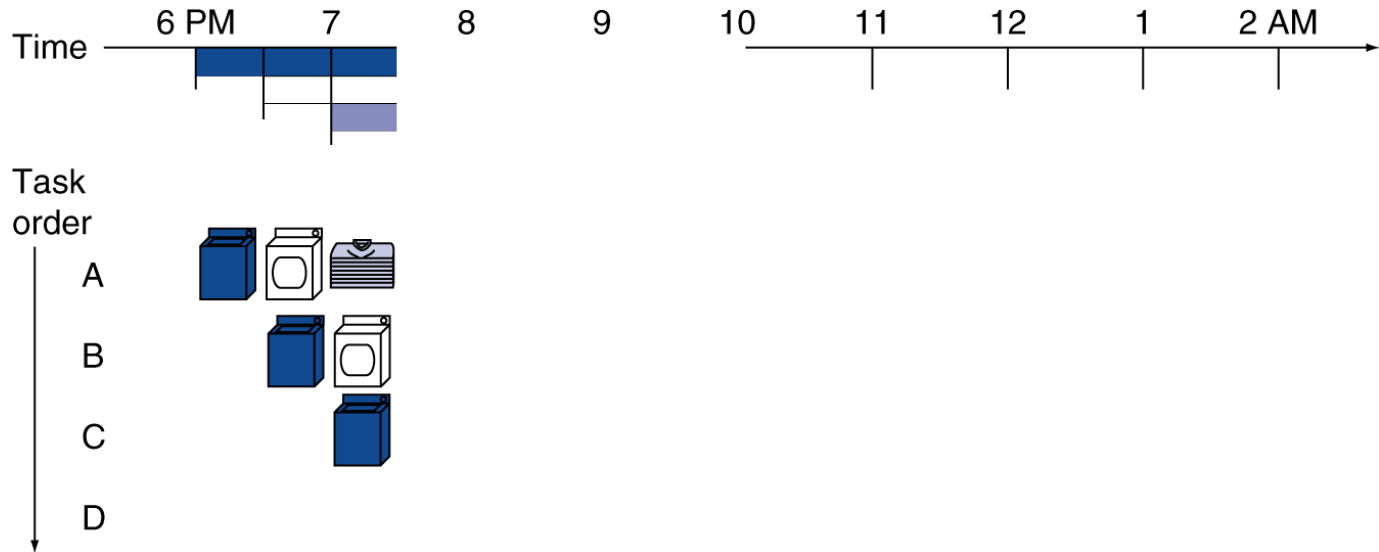


Question: how many “hardware” do we have?

1-set

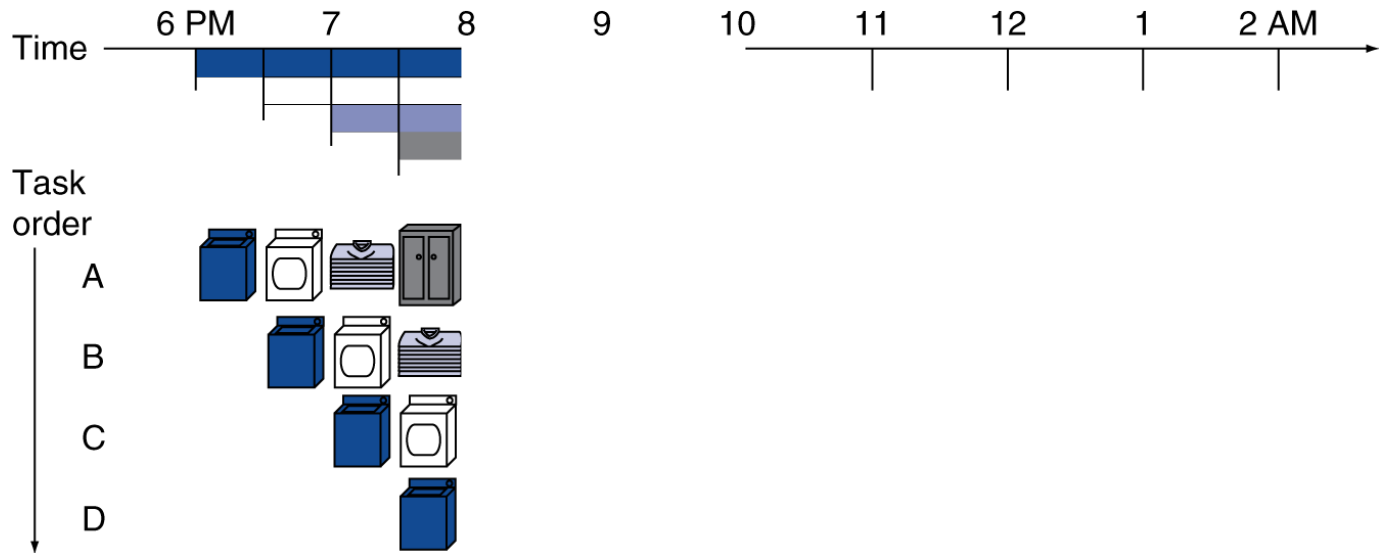
Pipelining Analogy

- Pipelined laundry: overlapping execution



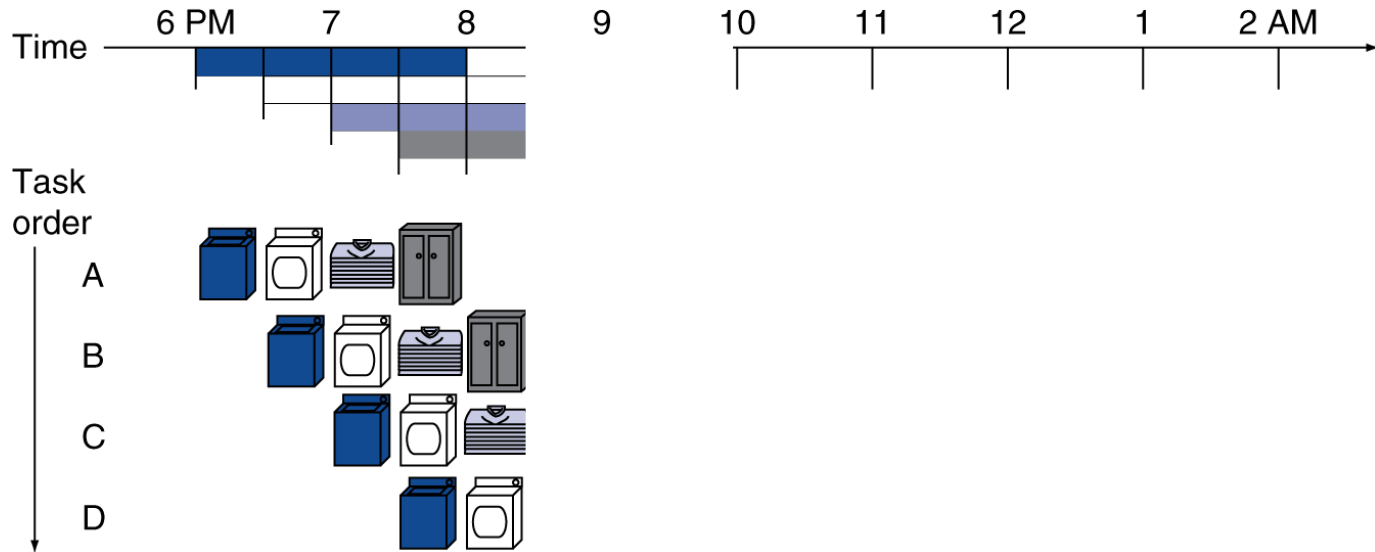
Pipelining Analogy

- Pipelined laundry: overlapping execution



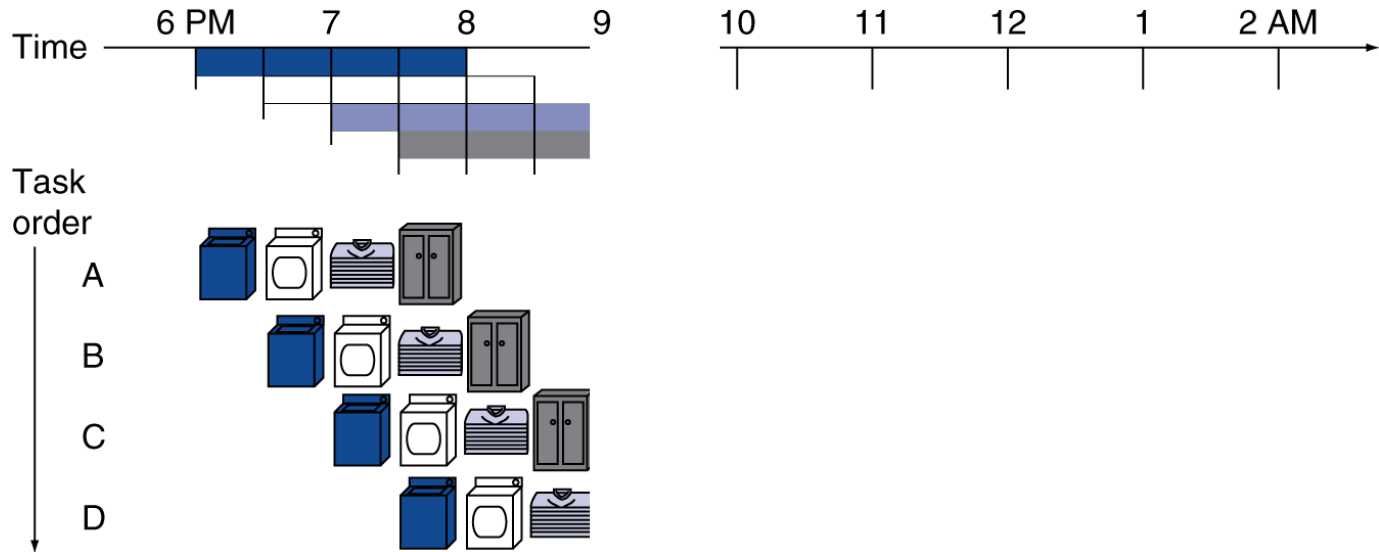
Pipelining Analogy

- Pipelined laundry: overlapping execution



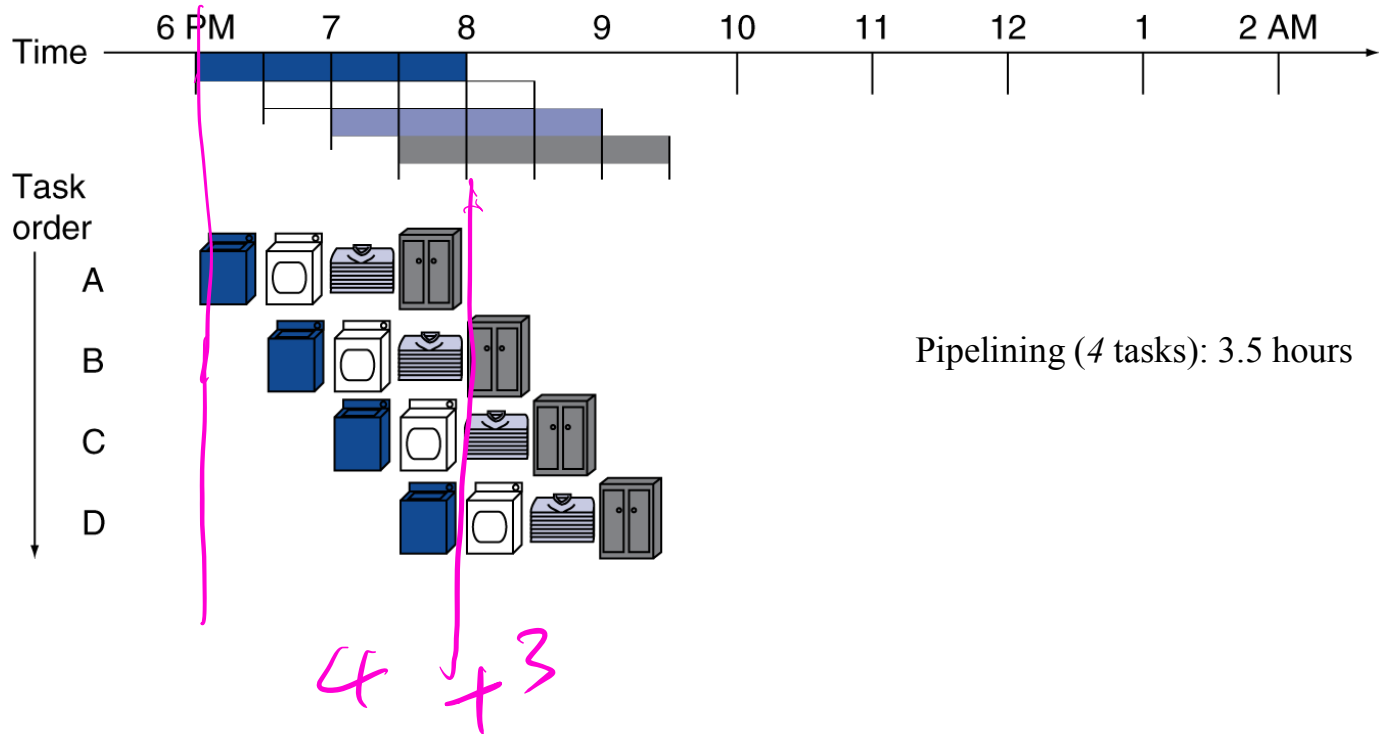
Pipelining Analogy

- Pipelined laundry: overlapping execution



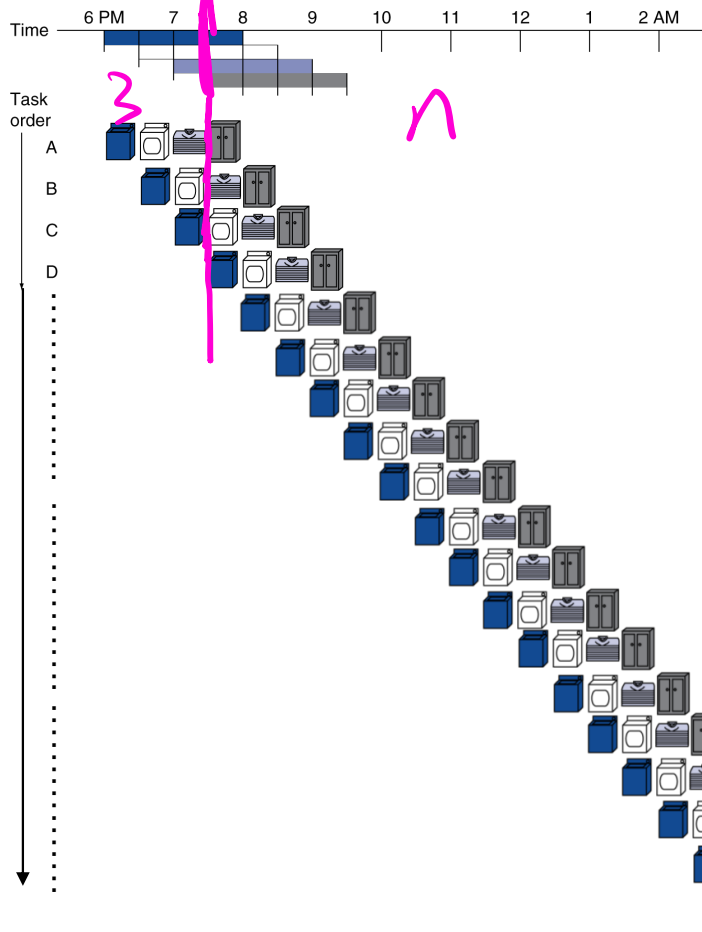
Pipelining Analogy

- Pipelined laundry: overlapping execution



Pipelining Analogy

- Pipelined laundry: overlapping execution

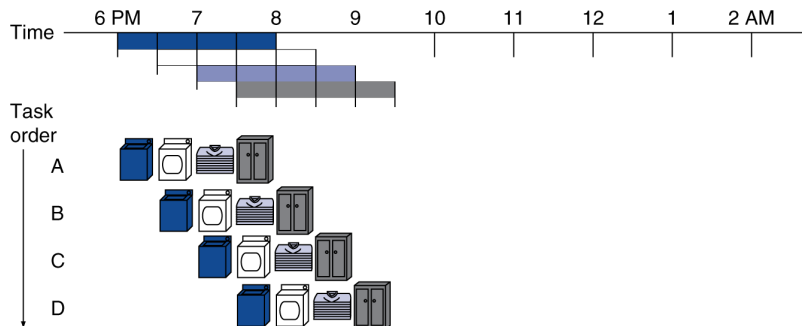
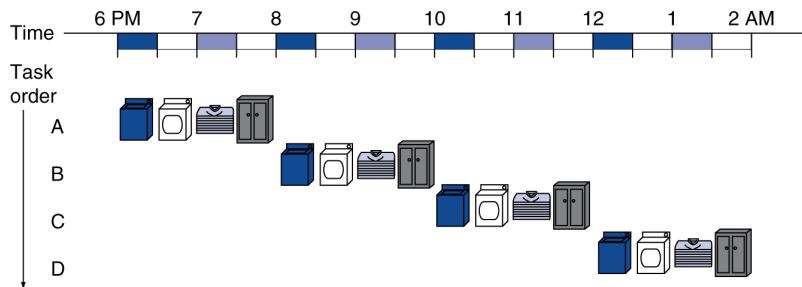


After 7:30pm, one **take out** per half hour (stage)

Execution time of pipelining
 n tasks: $n+3$

Pipelining Analogy

- Pipelined laundry: overlapping execution
 - Parallelism improves performance
 - Do you see the parallelism in the figures?



Speedup

$$= \frac{2n}{0.5n + 1.5} \approx 4$$

= number of stages

Handwritten notes in pink:

- 4+1
- $n+3$
- 4

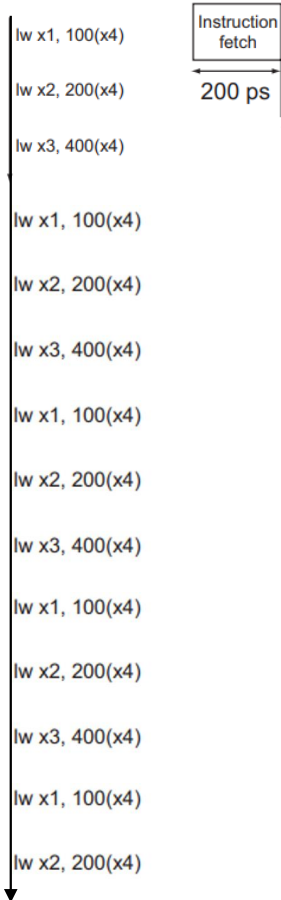
RISC-V Pipeline

Create a pipeline of five stages, one step per stage.

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

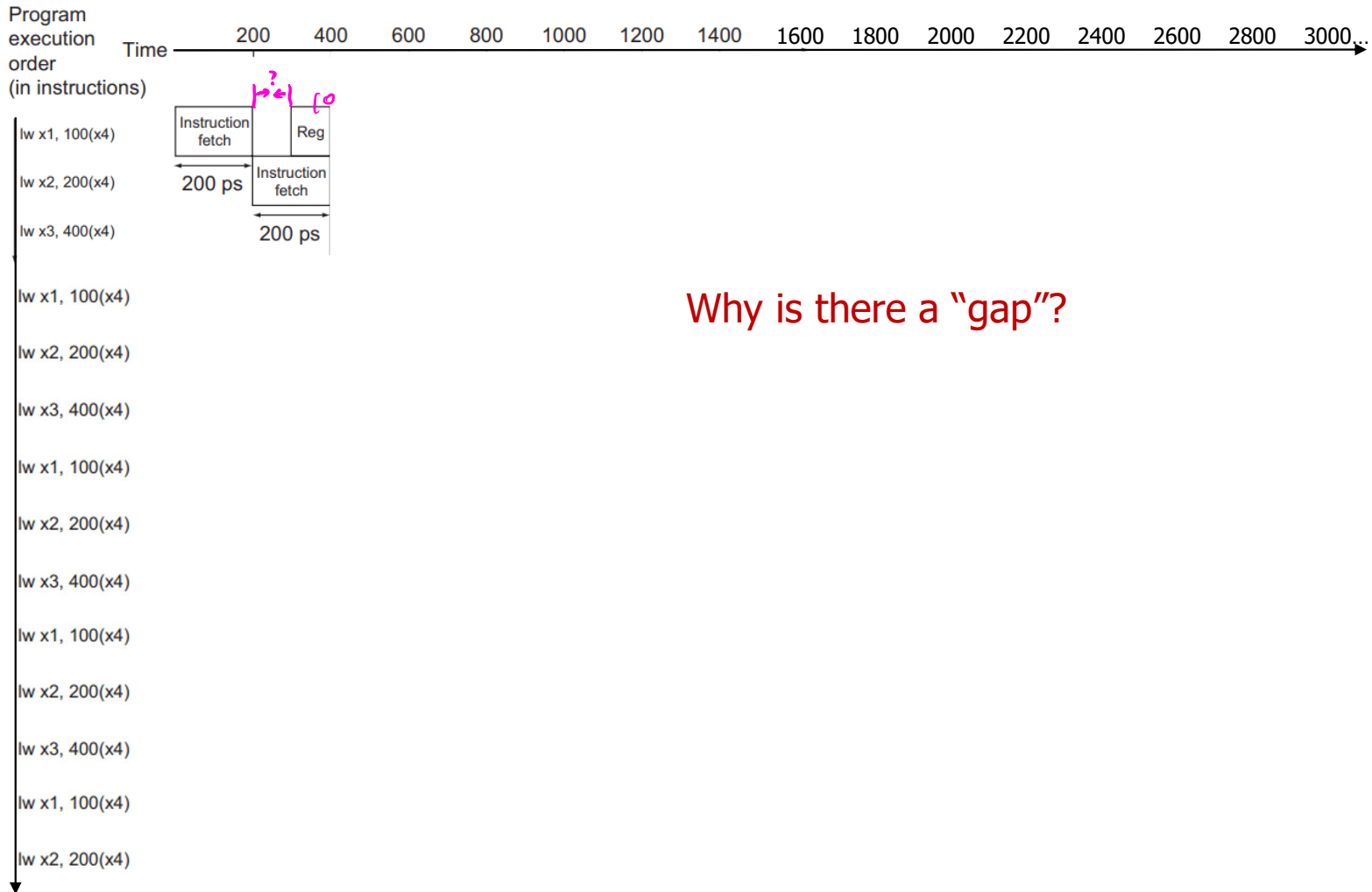
Pipelined ($T_c = 200\text{ps}$)

Time 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800 3000...



Pipeline Performance

Pipelined ($T_c = 200\text{ps}$)



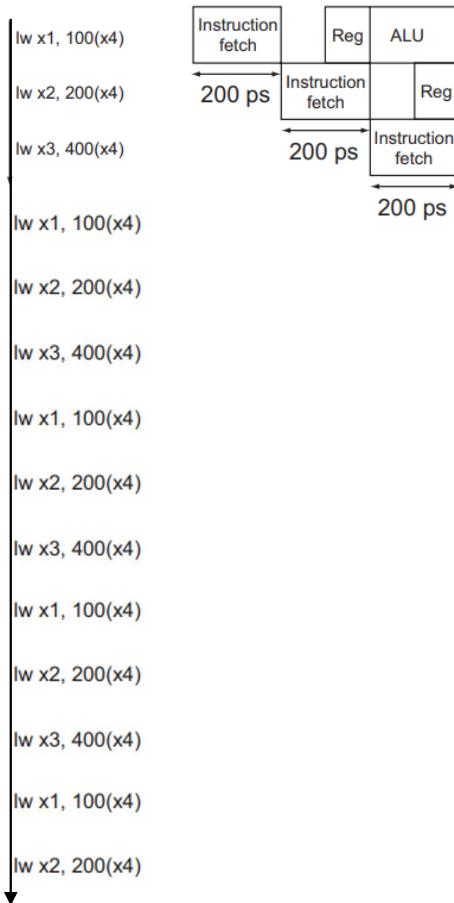
Why is there a "gap"?

Pipeline Performance

Pipelined ($T_c = 200\text{ps}$)

Program execution order (in instructions)

Time 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800 3000...



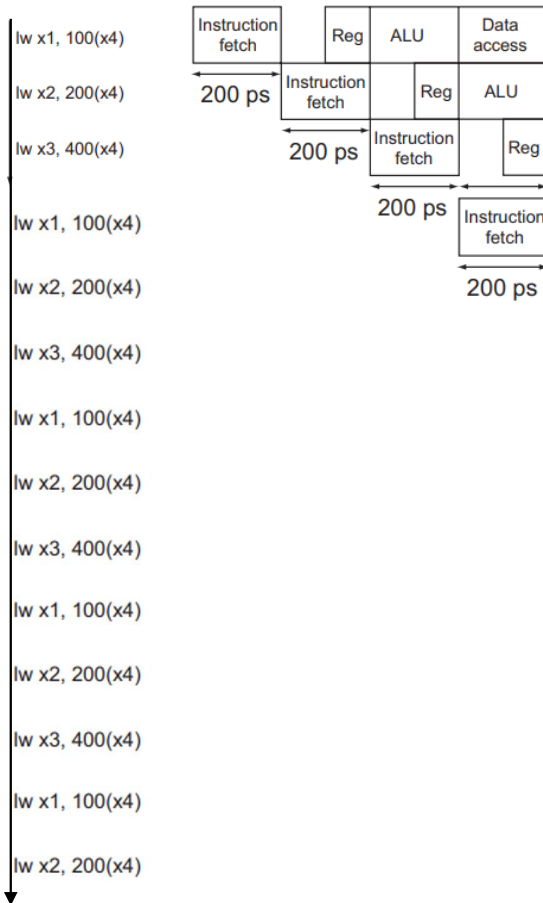
Why is there a "gap"?

Pipeline Performance

Pipelined ($T_c = 200\text{ps}$)

Program execution order (in instructions)

Time 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800 3000...



Pipelined ($T_c = 200\text{ps}$)



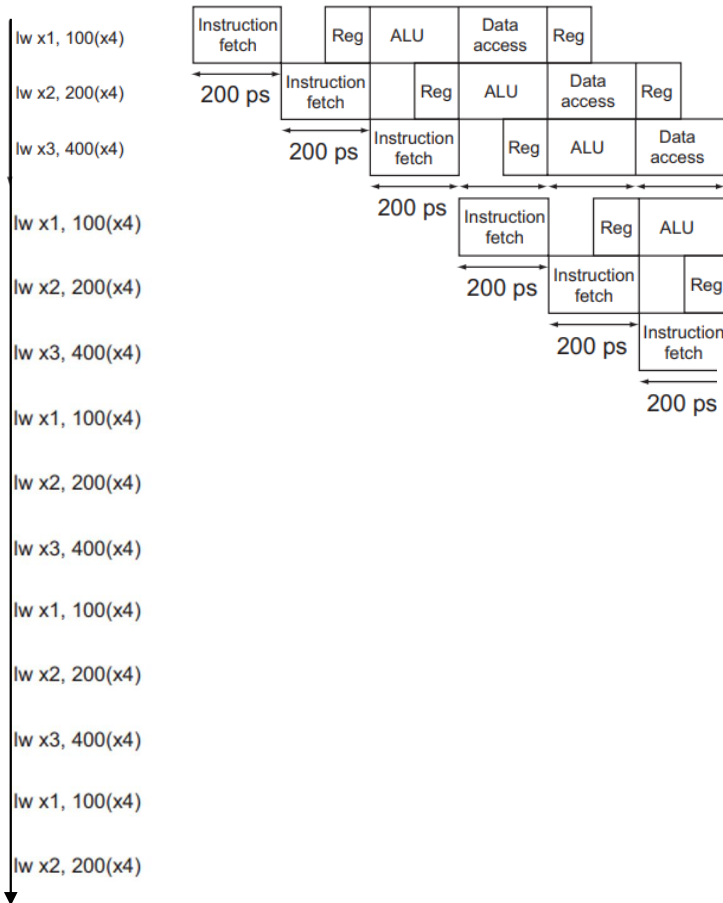
How many hardware components? e.g.,
Reg?

Pipeline Performance

Pipelined ($T_c = 200\text{ps}$)

Program execution order (in instructions)

Time 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800 3000...

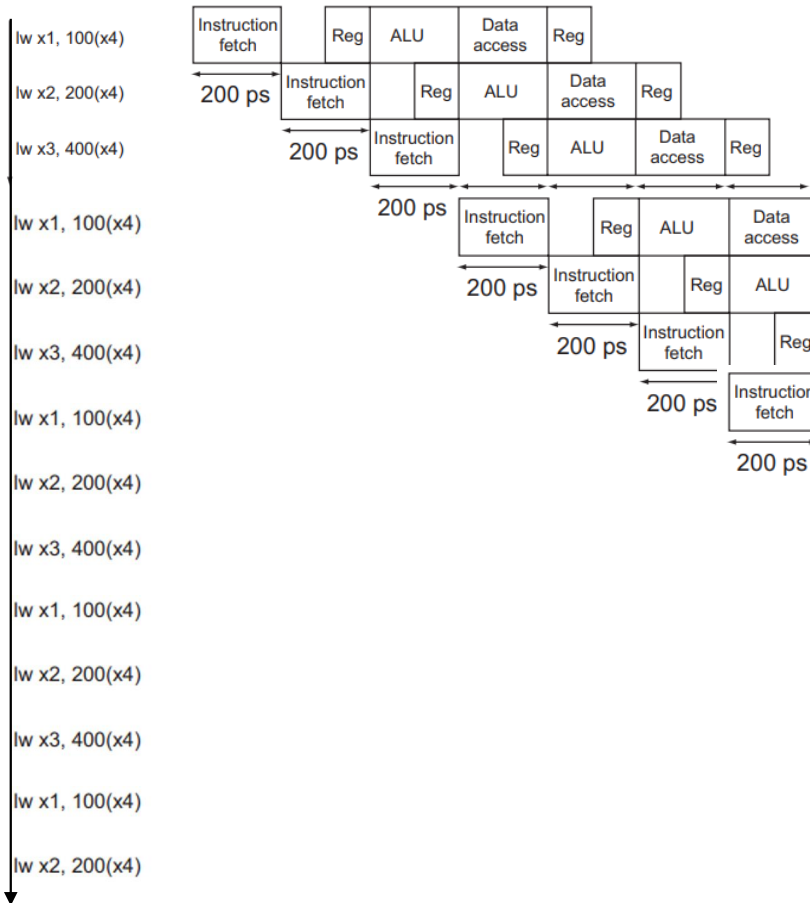


Pipeline Performance

Pipelined ($T_c = 200\text{ps}$)

Program execution order (in instructions)

Time 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800 3000...

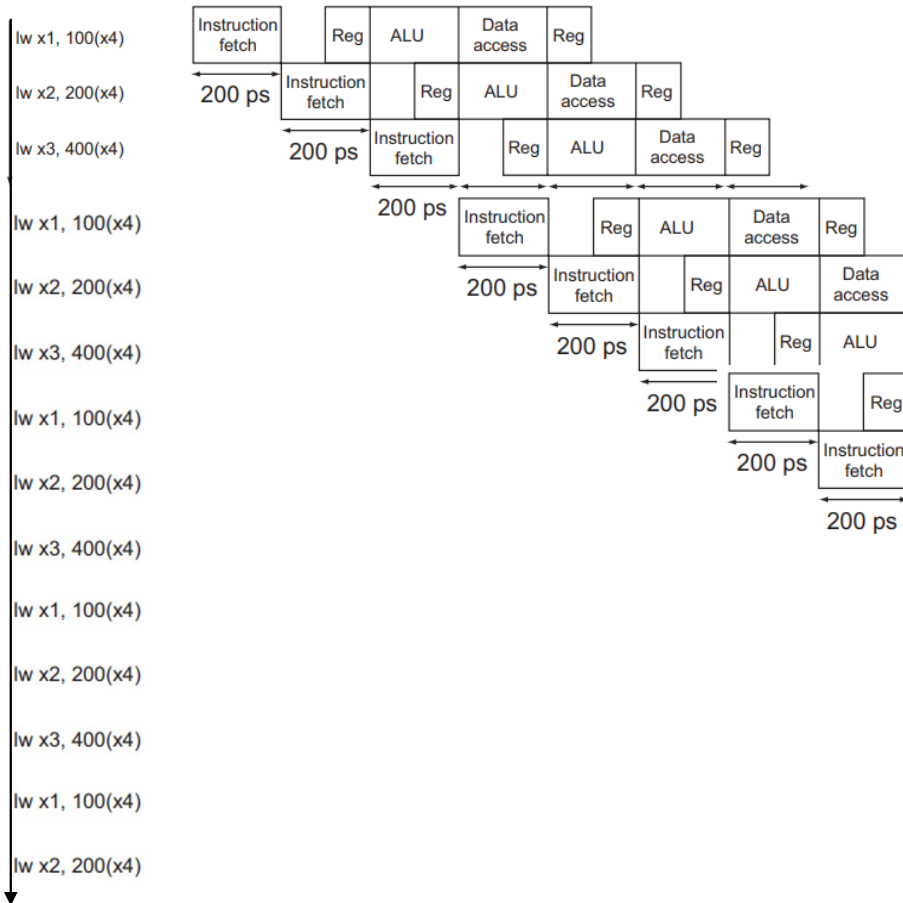


Pipeline Performance

Pipelined ($T_c = 200\text{ps}$)

Program execution order (in instructions)

Time 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800 3000...

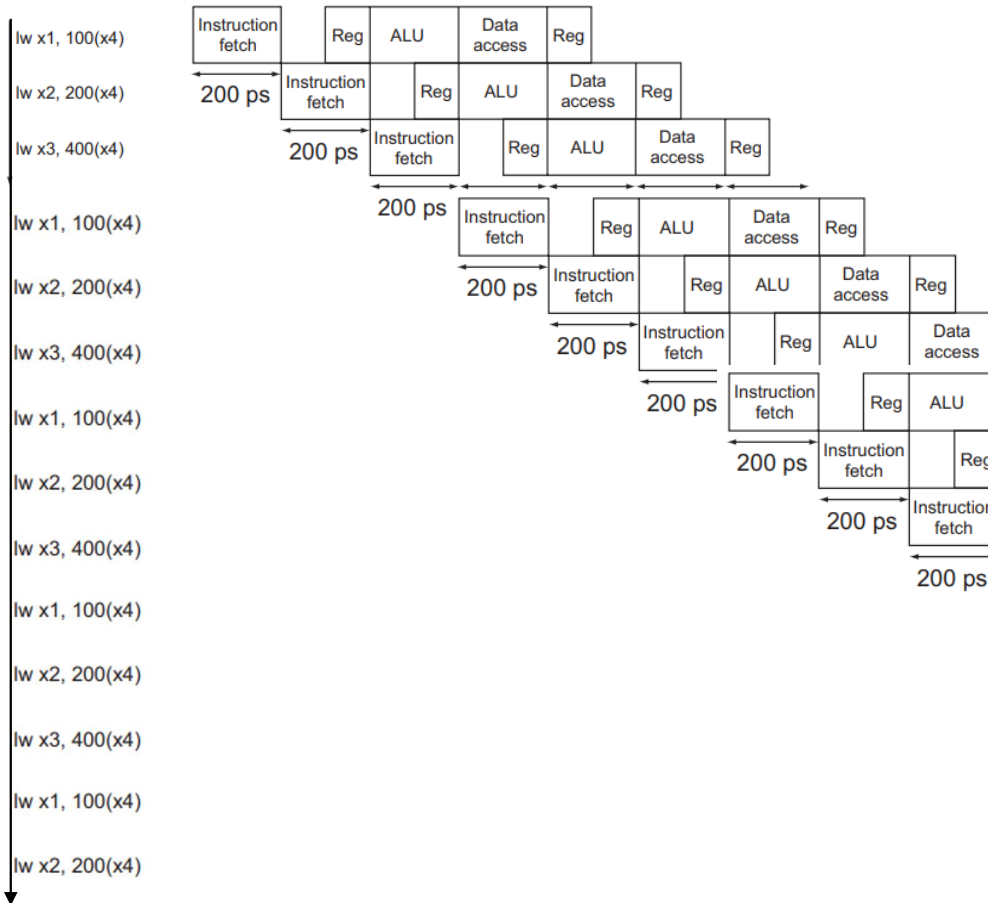


Pipeline Performance

Pipelined ($T_c = 200\text{ps}$)

Program execution order (in instructions)

Time 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800 3000

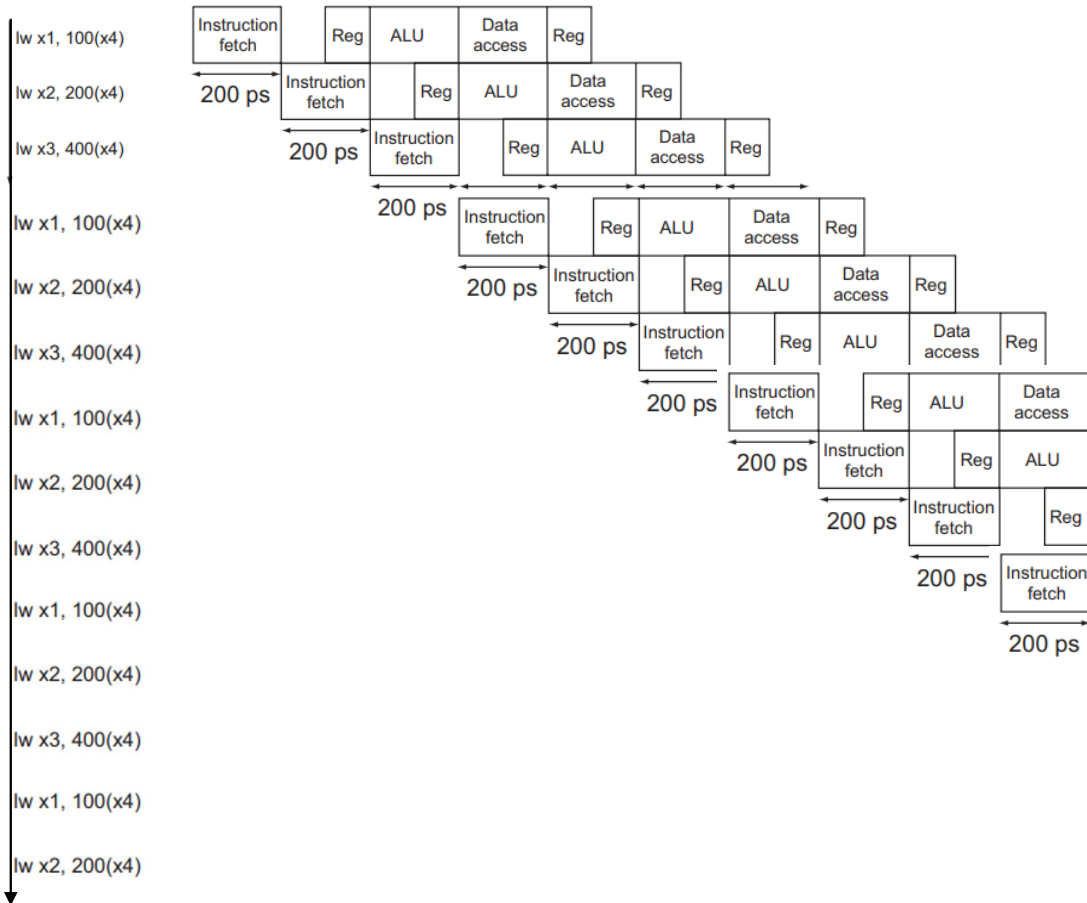


Pipeline Performance

Pipelined ($T_c = 200\text{ps}$)

Program execution order (in instructions)

Time 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800 3000...

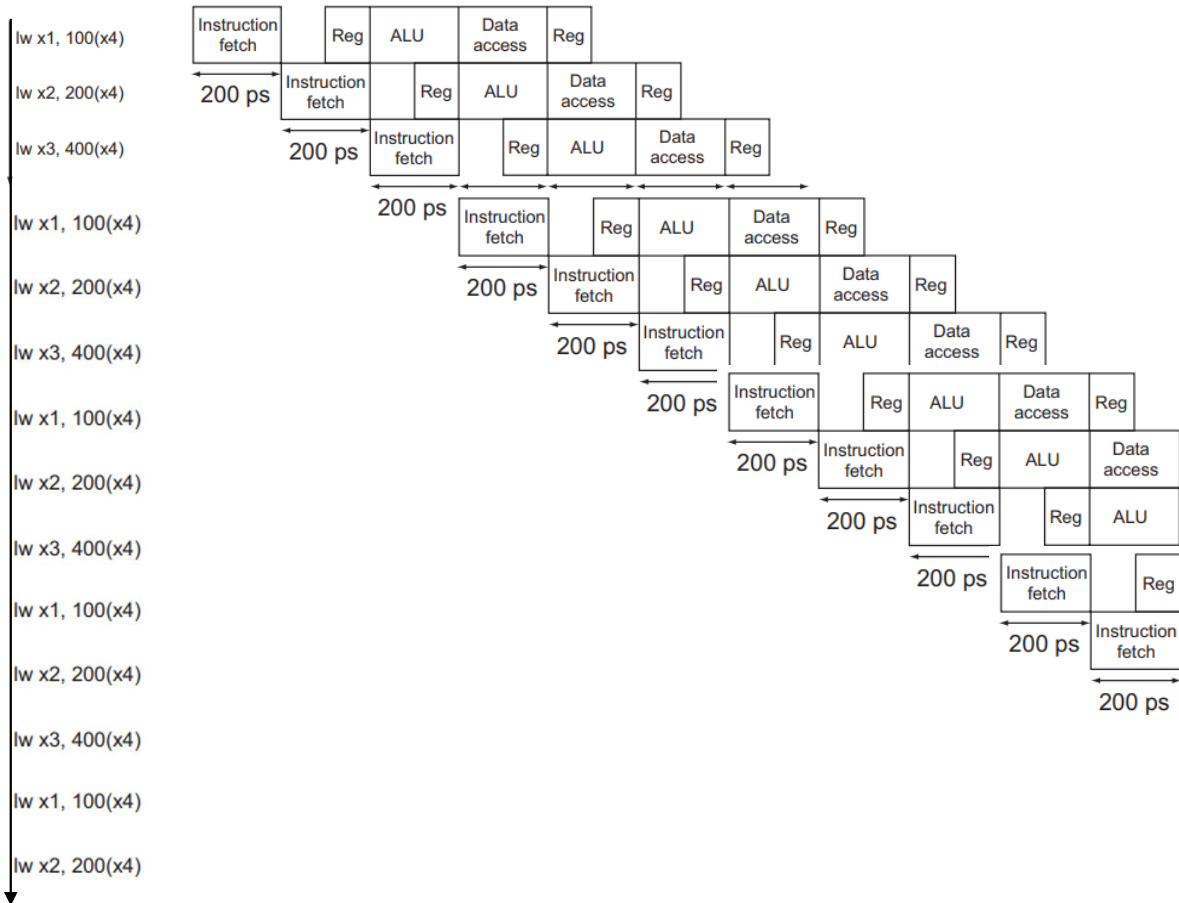


Pipeline Performance

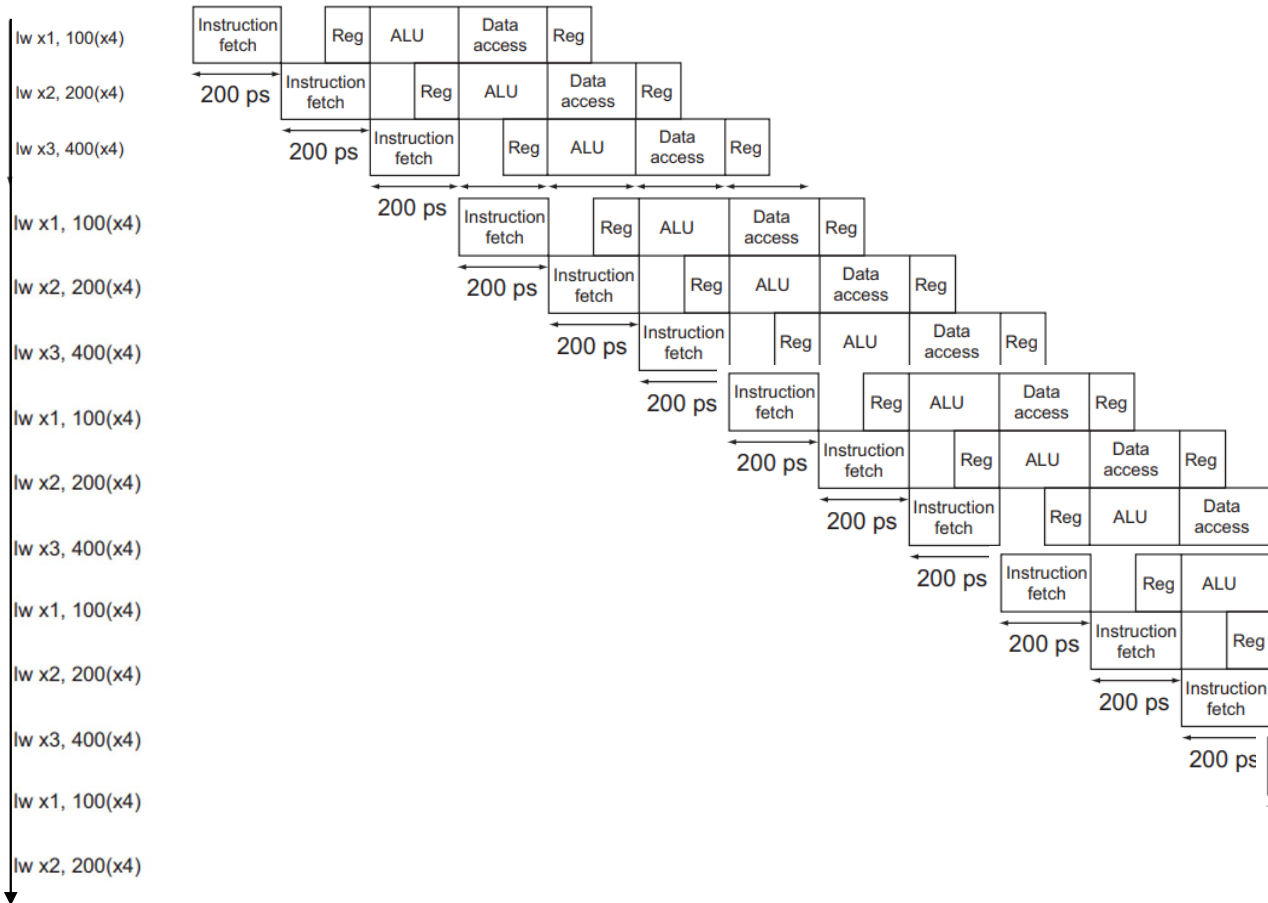
Pipelined ($T_c = 200\text{ps}$)

Program
execution
order
(in instructions)

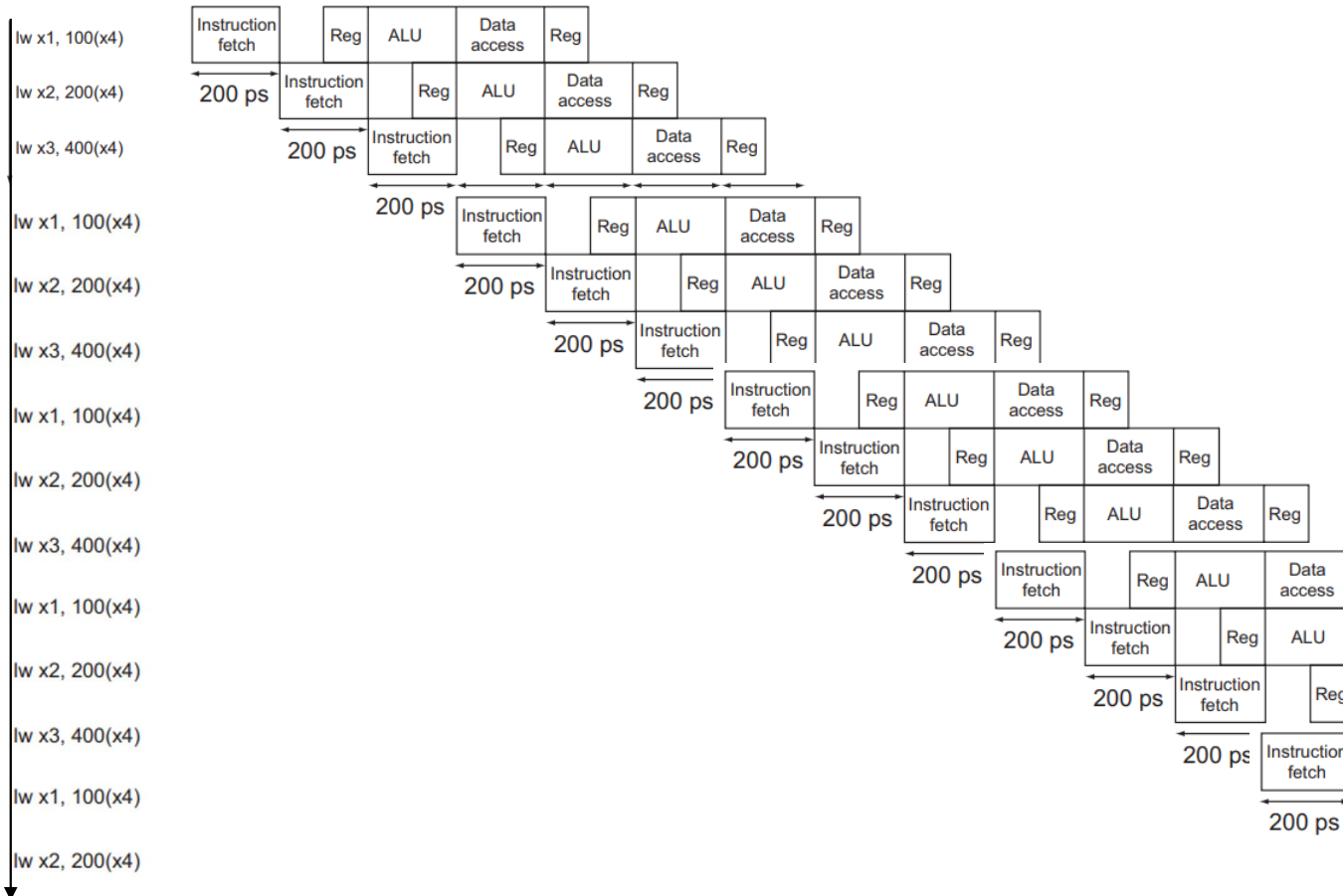
Time → 200 400 600 800 1000 1200 1400



Pipelined ($T_c = 200\text{ps}$)

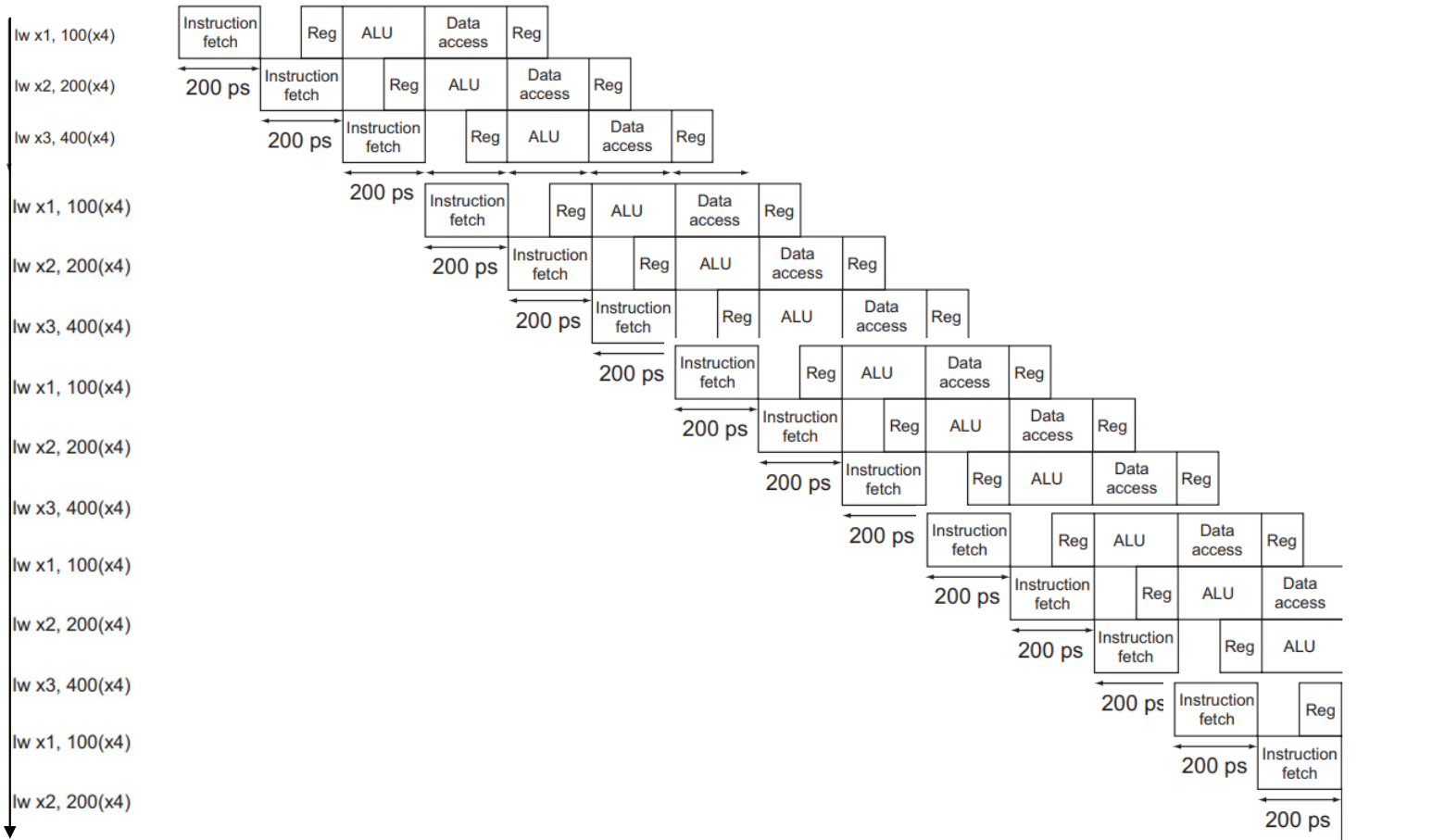


Pipelined ($T_c = 200\text{ps}$)



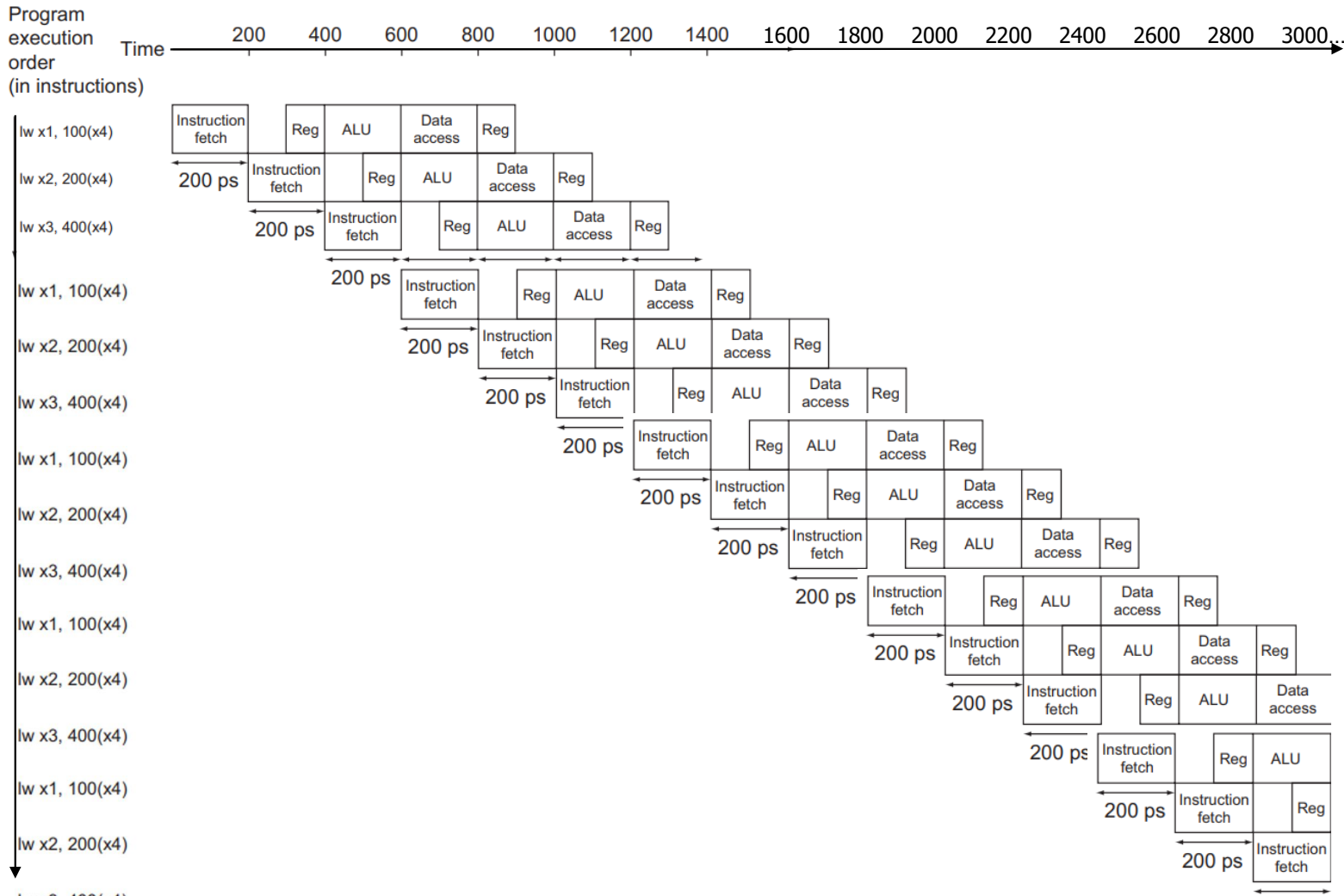
Pipelined ($T_c = 200\text{ps}$)

Time 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800 3000...

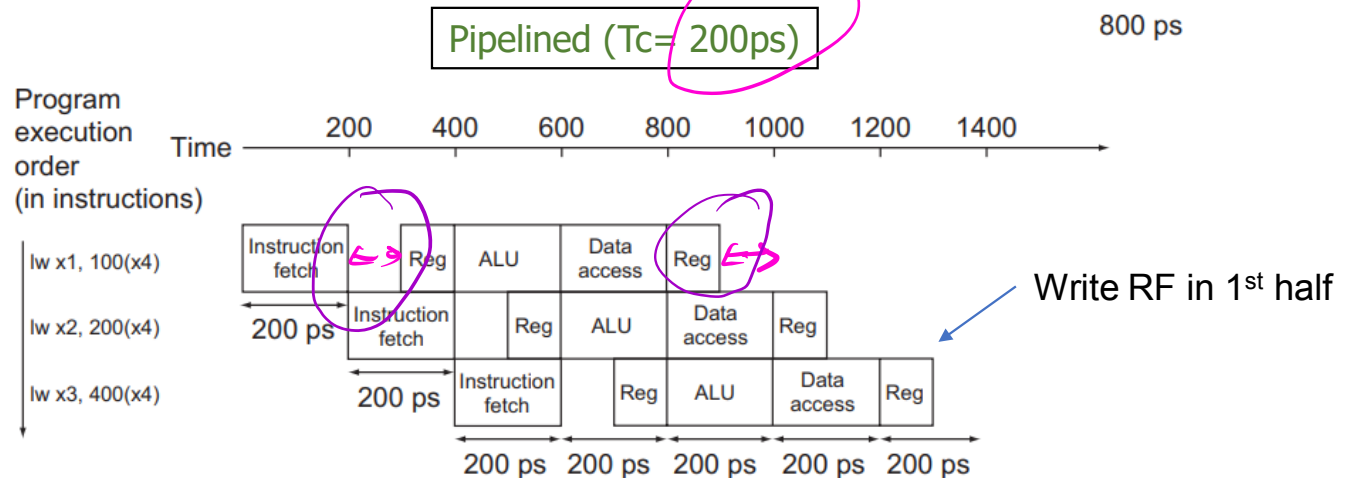
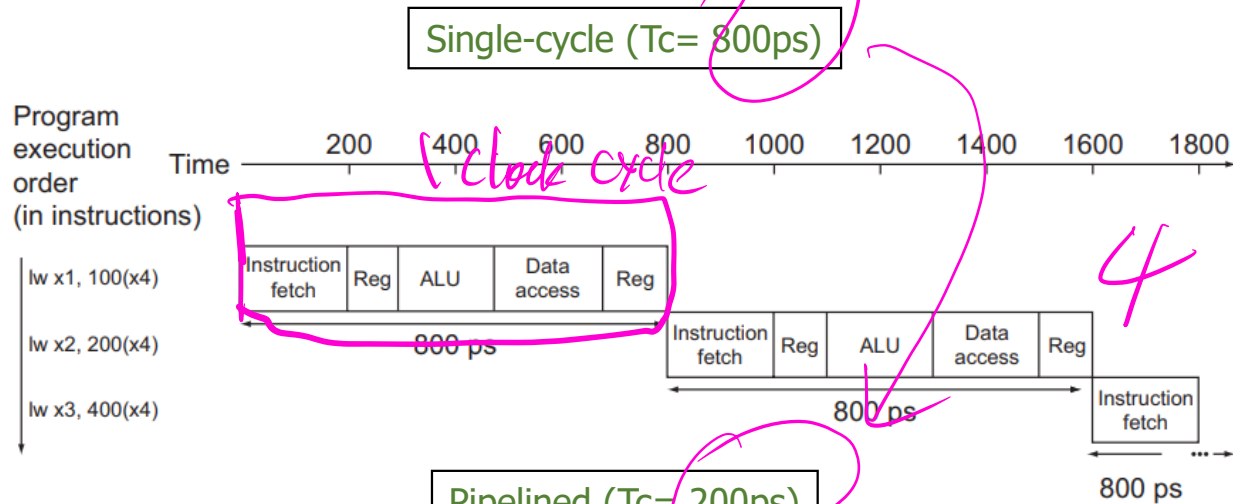


Pipeline Performance

Pipelined ($T_c = 200\text{ps}$)



Pipeline Performance



What is the speedup for a long instruction sequence?

Ideal Pipeline Speedup

Ideally,

$$\text{Time Between Instr}_{\text{pipelined}} = \frac{\text{Time Between Instr}_{\text{nonpipelined}}}{\text{Number Of Stages}}$$

$$\text{Speedup} = \frac{\text{Time Between Instr}_{\text{nonpipelined}}}{\text{Time Between Instr}_{\text{pipelined}}} = \text{Number of Stages}$$

Actual speedup is less than the ideal speedup, why?

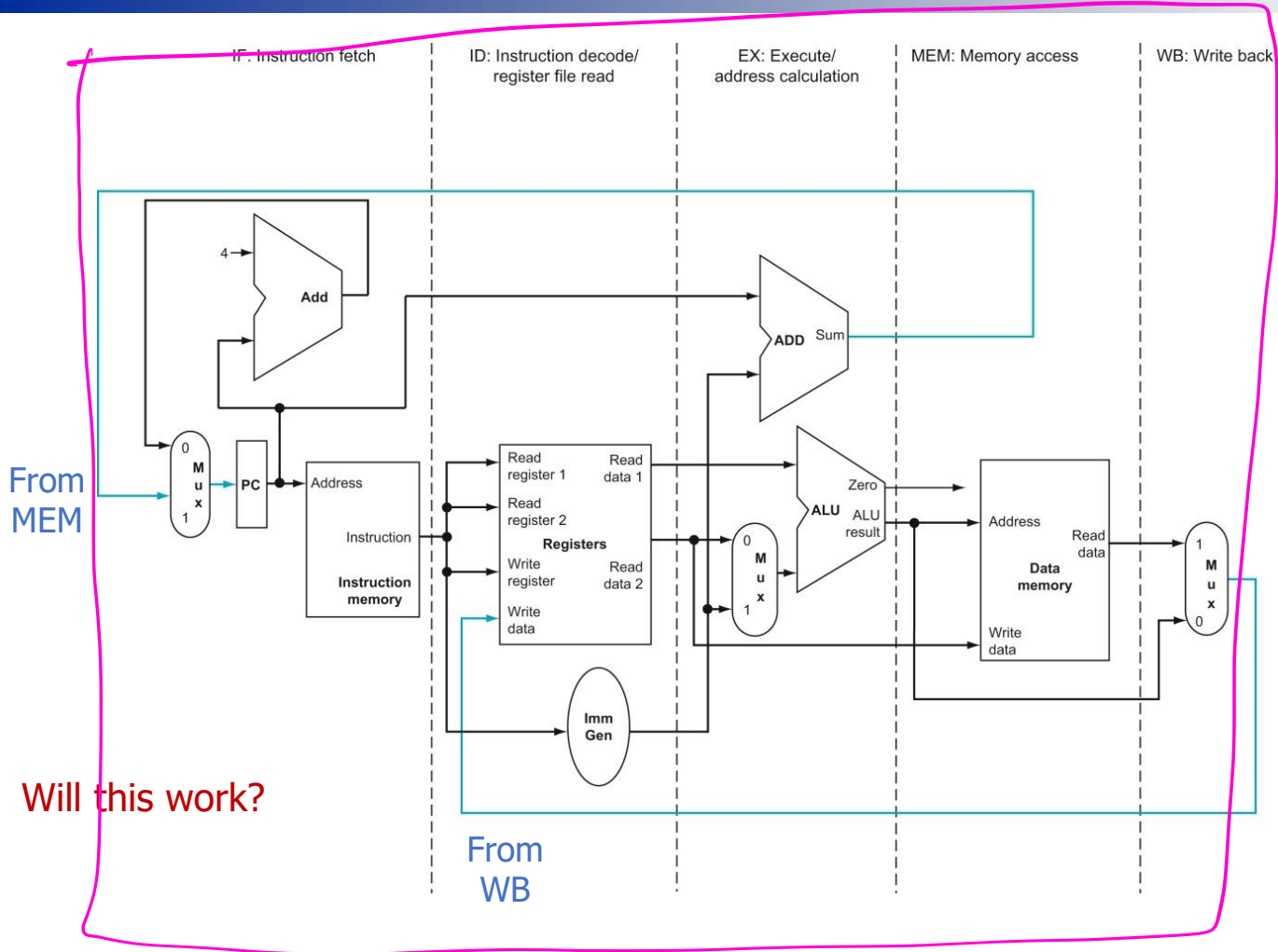
gap.

Pipeline Speedup

- Actual speedup is less than the ideal speedup
 - Pipeline stages are not balanced (gap)
 - Overhead in pipeline
 - Clock skew
 - Hazard
- Speedup due to increased throughput vs (latency)
 - Latency (time for executing each instruction) does not decrease, but increases

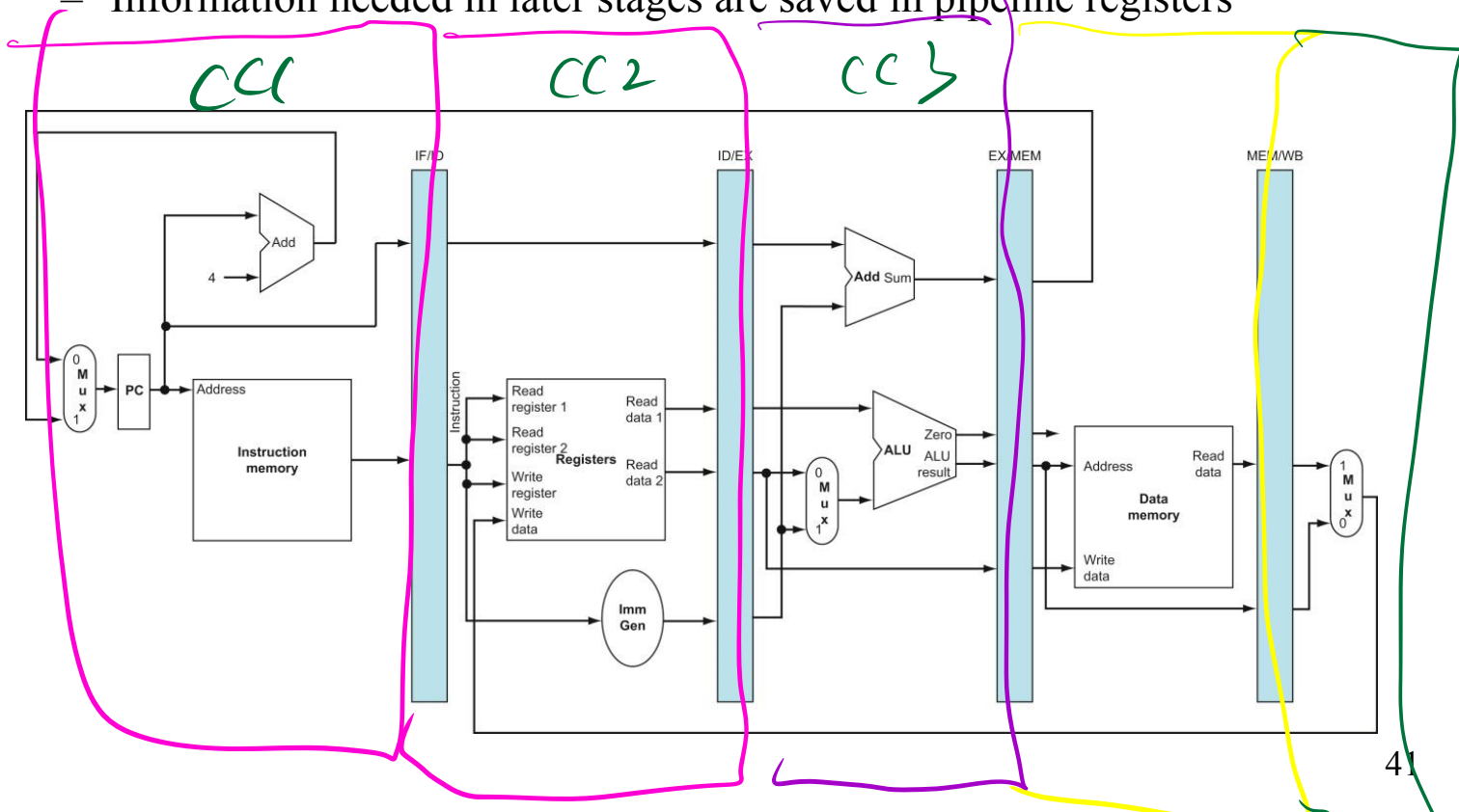
1 clock cycle → 5 inst - cycle

RISC-V Datapath Divided in Five Stages



Pipelined Datapath

- Add **pipeline registers** between pipeline stages to **isolate** them
 - Data stored in registers are stable for the cycle
 - Information needed in later stages are saved in pipeline registers



Two rules

No need to memorize!

- In general, we follow the following two rules
 - Do not use the signal generated in the same stage
 - To reduce the cycle time
 - Results are saved in pipeline registers and passed to later stage
 - Use a signal as early as possible
 - We do not need pass it to later stages

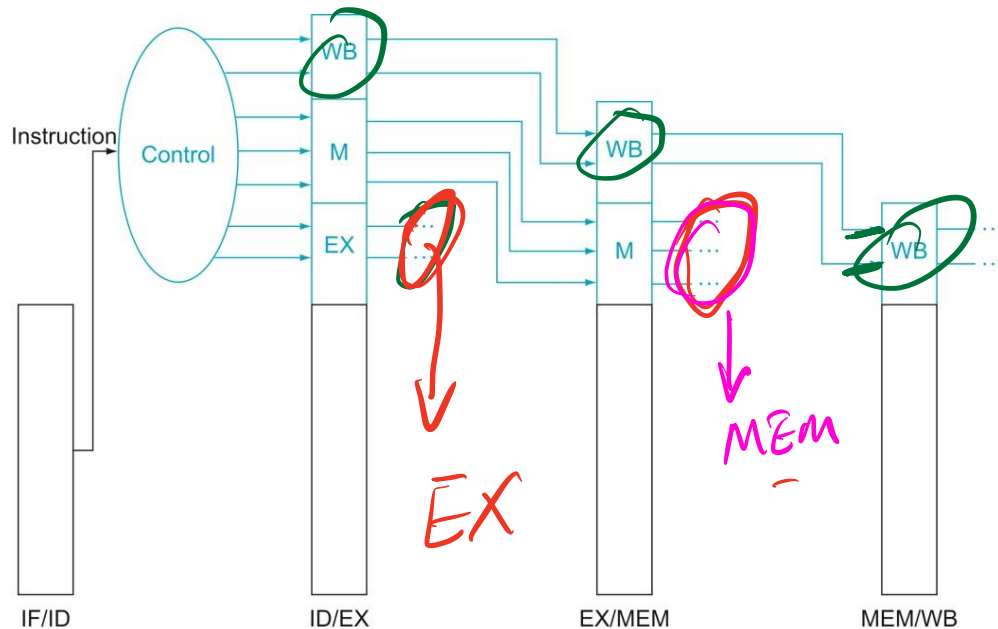
Control Signals in Pipeline

- Control signals derived from instruction, as in single-cycle implementation
- All control signals are generated in ID and passed to later stages through the **pipeline (also called state) registers**
 - 2 used in EX, 3 used in Mem, and 2 in WB

	EX Stage		MEM Stage			WB Stage	
	ALUOp	ALUSrc	Branch	Mem Read	Mem Write	Reg Write	Mem toReg
R	10	0	0	0	0	1	0
LW	00	1	0	1	0	1	1
SW	00	1	0	0	1	0	X
BEQ	01	0	1	0	0	0	X

Control Signals in Pipeline Registers

- Control signals are generated in ID and passed to later stages



RISC-V Pipeline

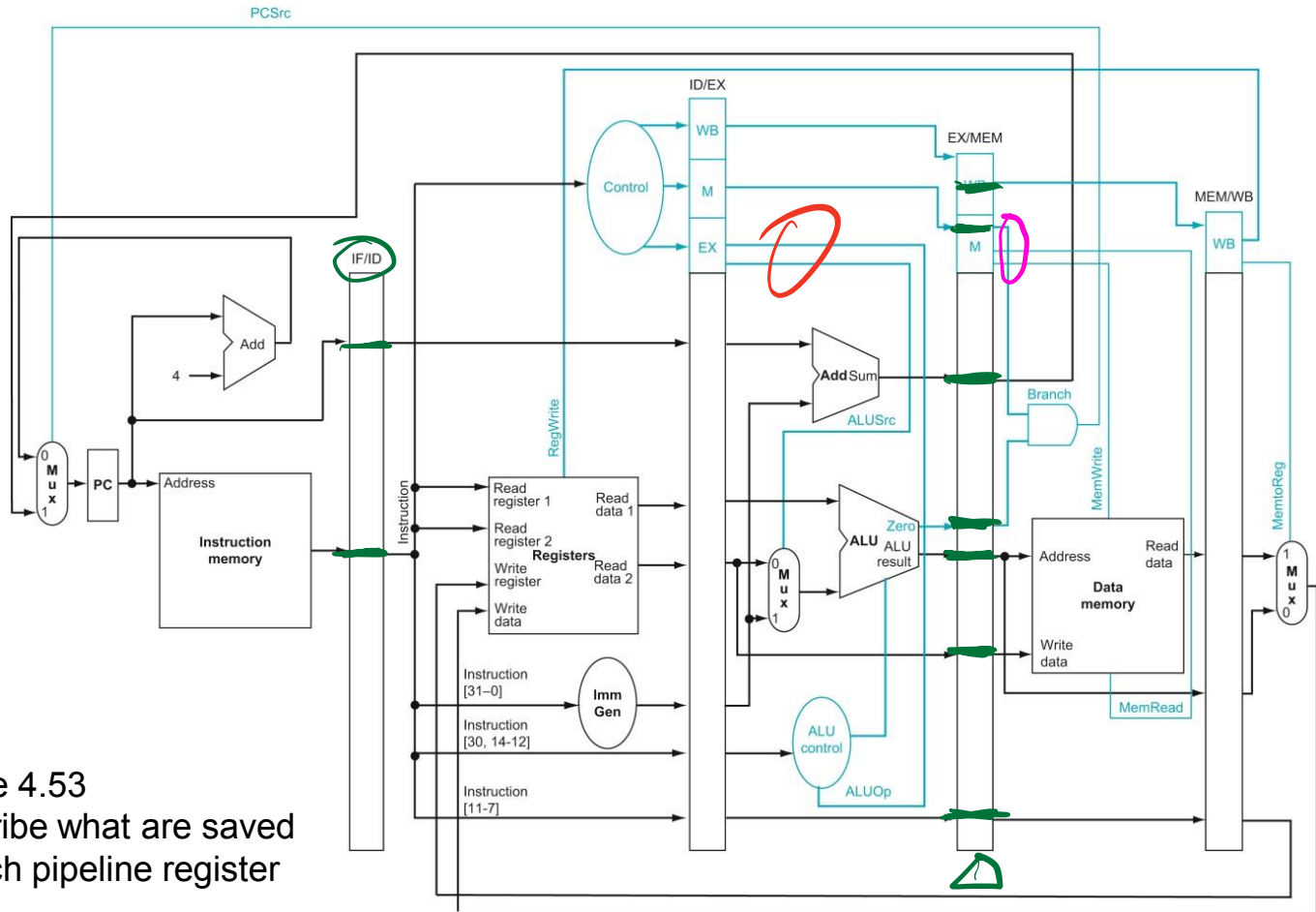


Figure 4.53
Describe what are saved
in each pipeline register

Explain why they need to be saved


Information saved into pipeline registers

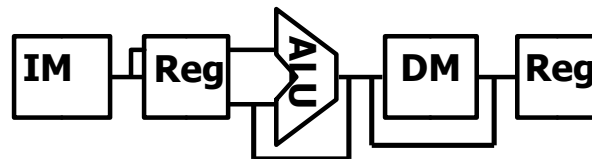
Any info needed in a later stage must be passed to that stage via pipeline registers
Study the diagram and check if any signal is missing

Do not memorize!

- IF/ID
 - PC, Instruction
- ID/EX
 - PC
 - Read data 1, Read data 2, imm16, funct3, and rd
 - Control signals
- EX/MEM
 - Read data 2, rd, MemRead, MemWrite, Branch, RegWrite, and MemtoReg
 - ALU result and Zero, Branch target address, Write register
- MEM/WB
 - ALU result, rd, RegWrite, and MemtoReg
 - Mem read data

Pipeline Diagrams

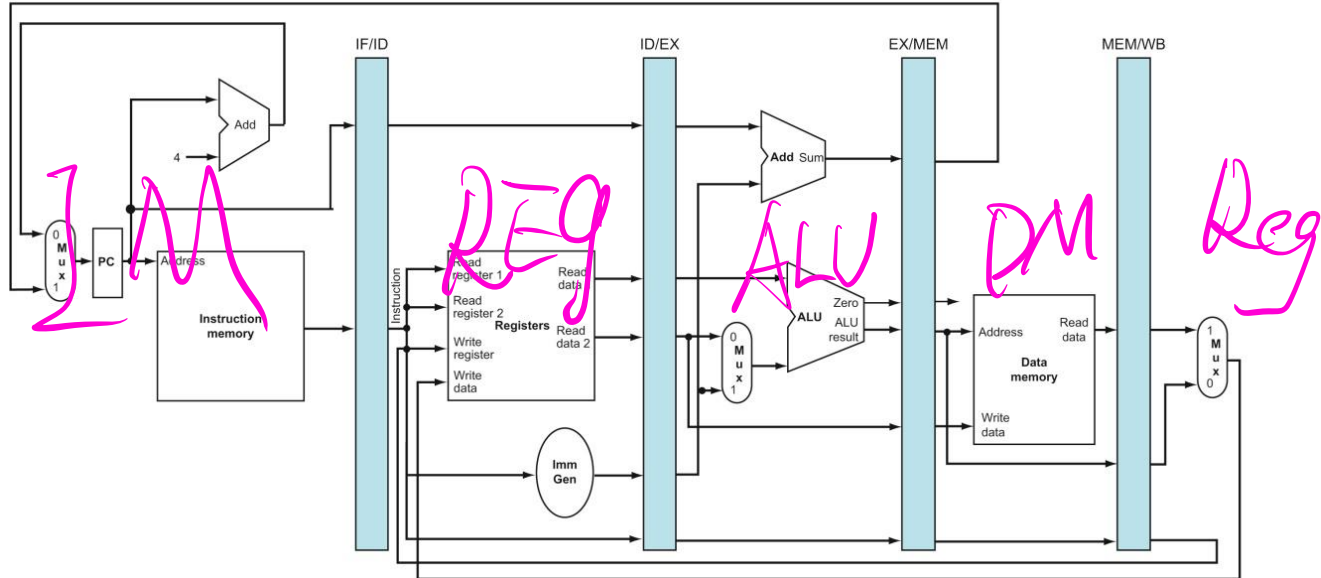
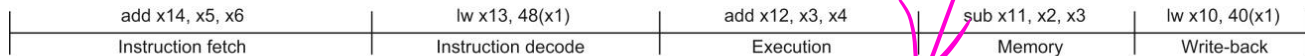
- Can help with answering questions like:
 - How many cycles does it take to execute this code? *RISC-V*
 - What is the ALU doing during cycle 4?
 - Is there a hazard, why does it occur, and how can it be fixed?
- Two types of diagrams
 - “Single-clock-cycle” pipeline diagram
 - Shows pipeline usage in a single cycle
 - Highlight resources used
 - “Multi-clock-cycle” pipeline diagram  What we mainly use
 - Showing how instructions are executed over time



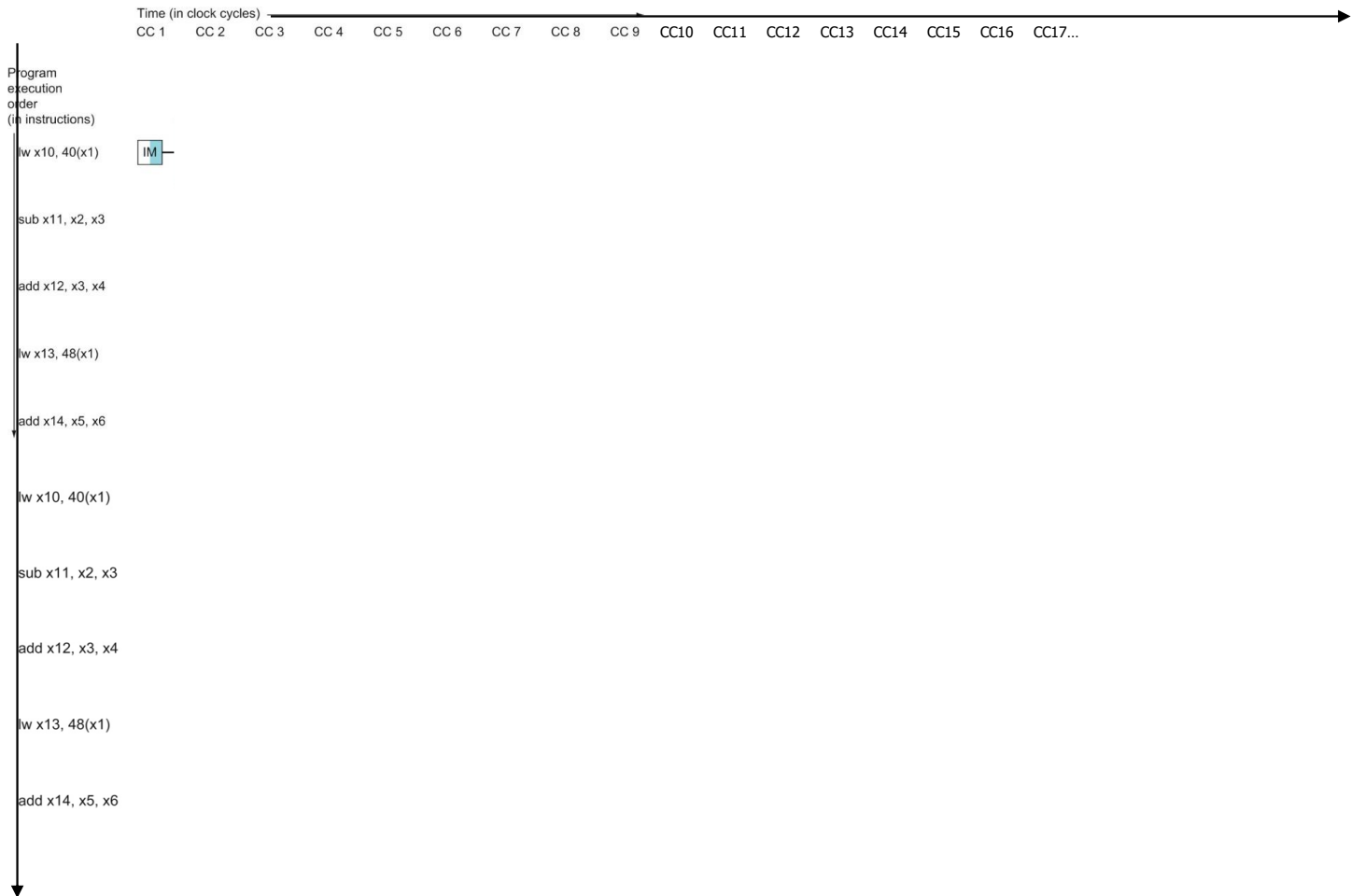
✓ exam

Single-Cycle Pipeline Diagram

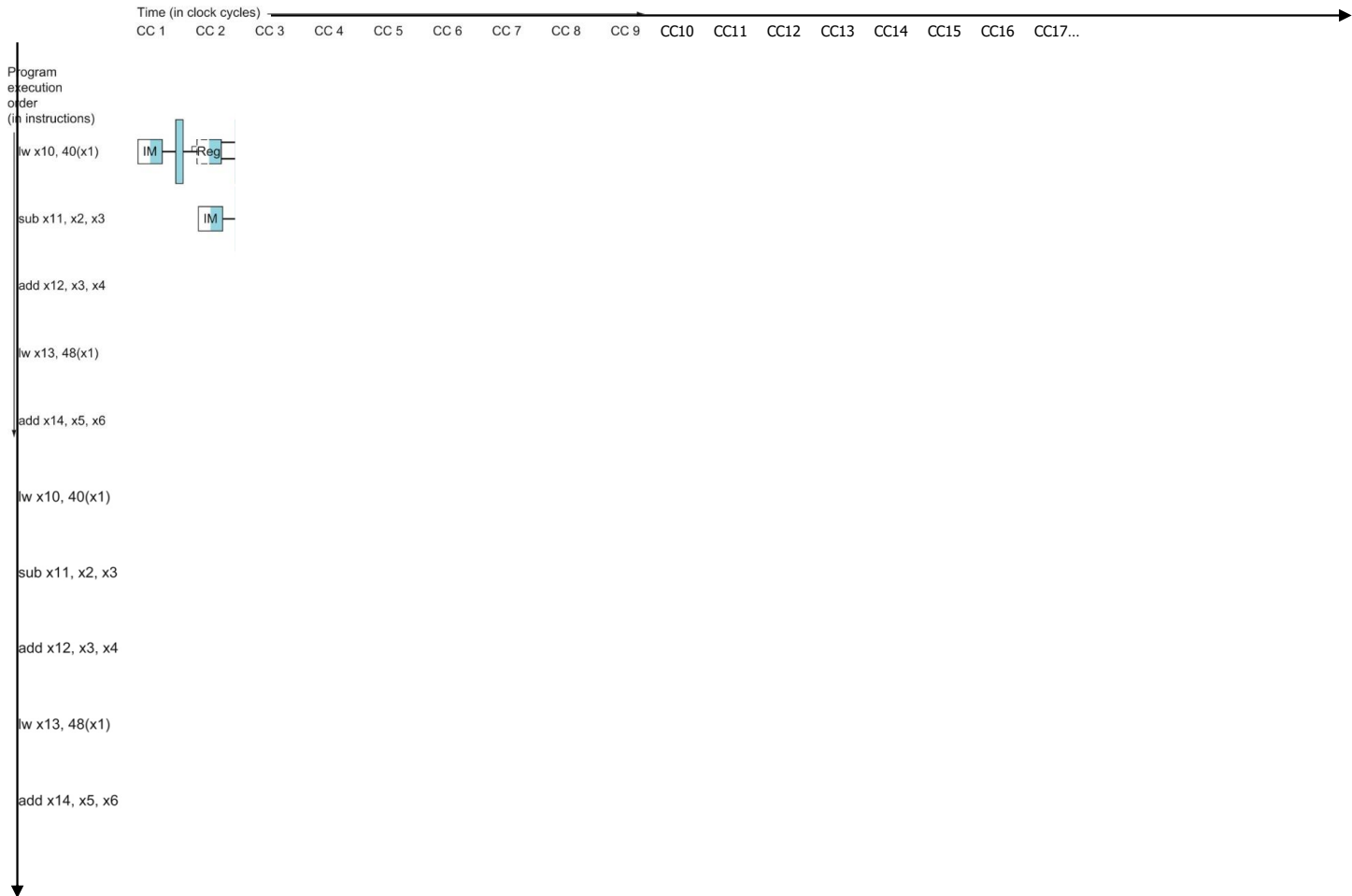
- State of pipeline in a given cycle
 - Instructions flow from left to right
 - Each pipeline stage has all the signals for the instruction in that stage



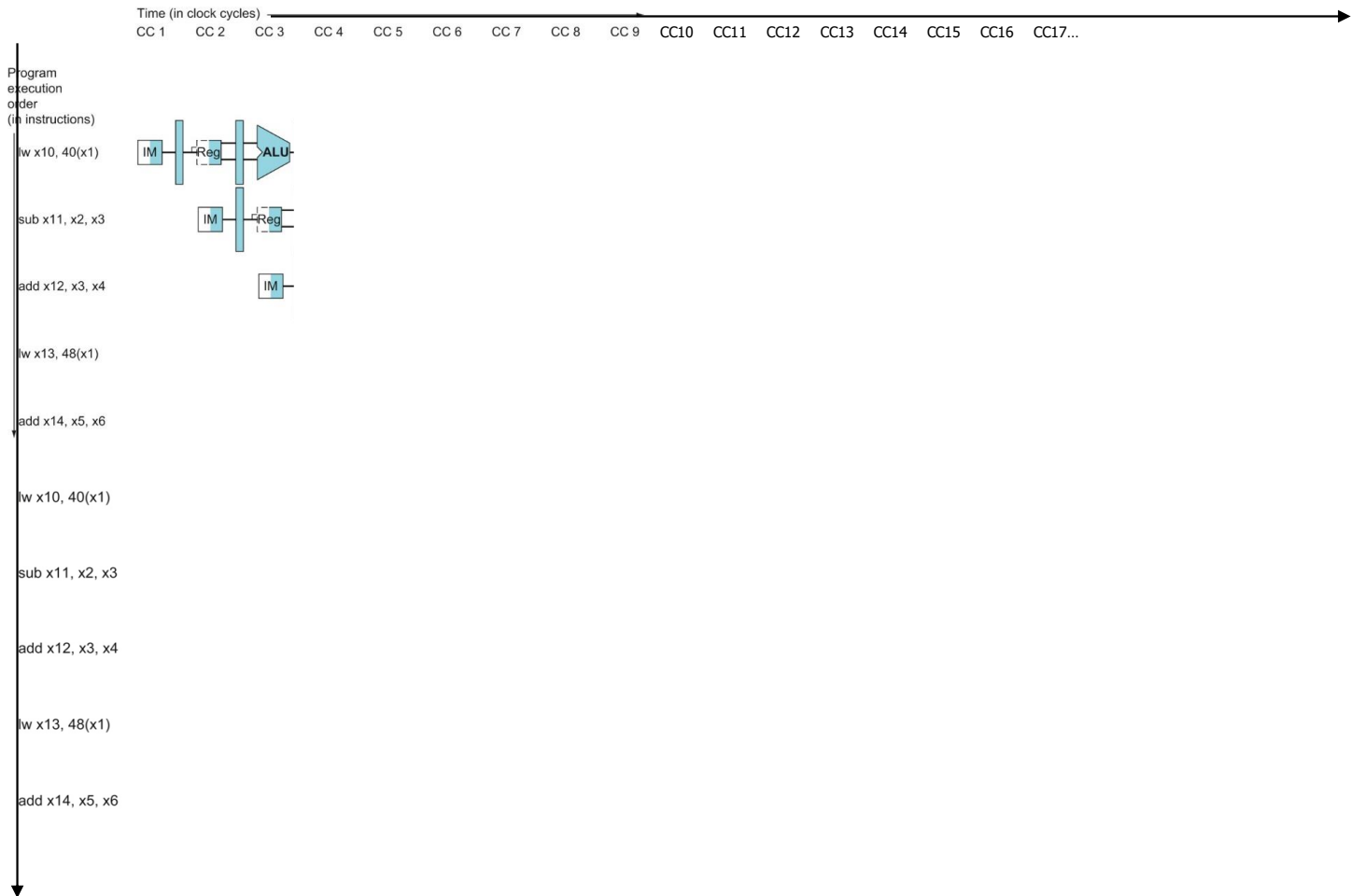
Multi-Cycle Pipeline Diagram



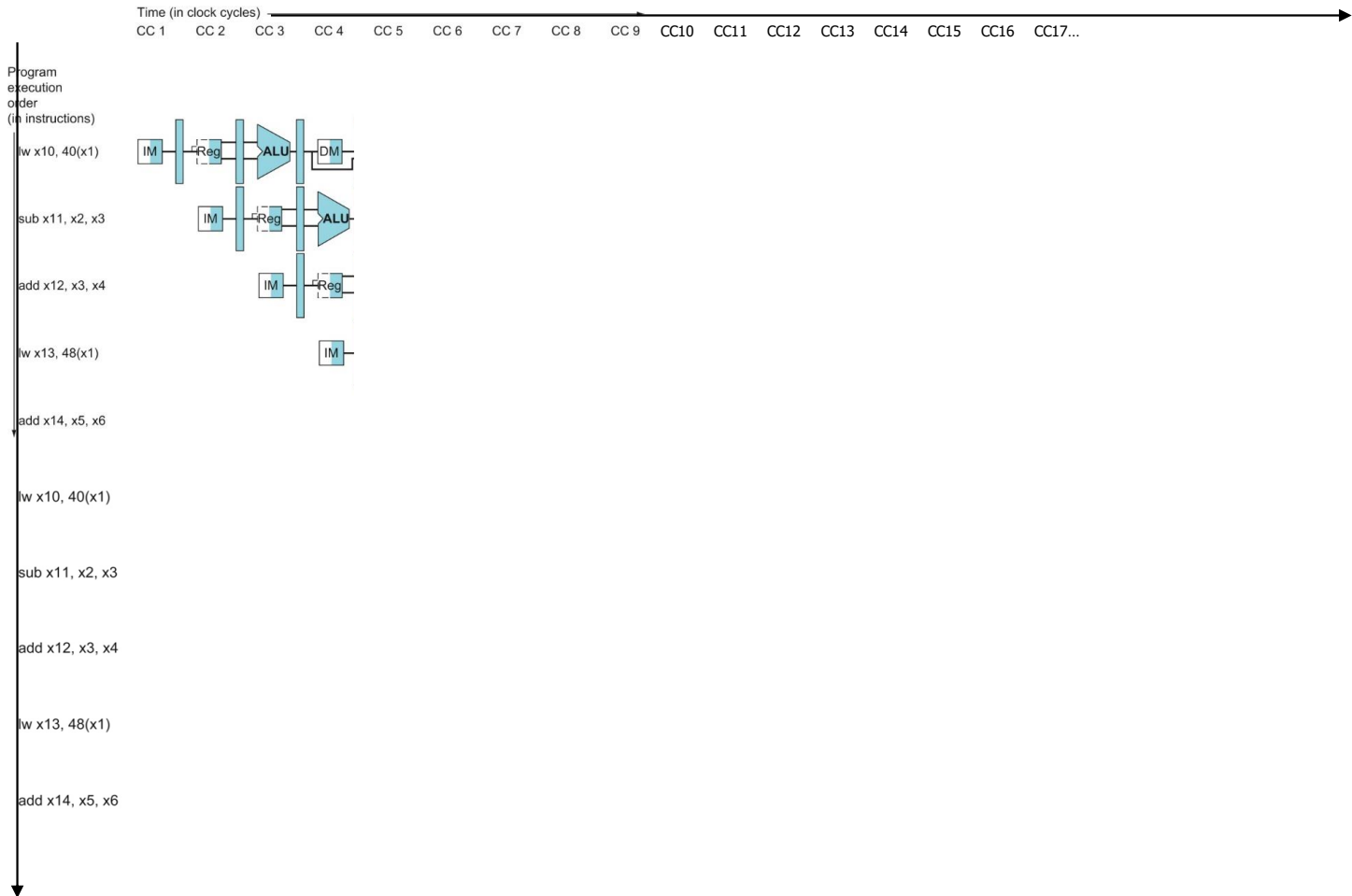
Multi-Cycle Pipeline Diagram



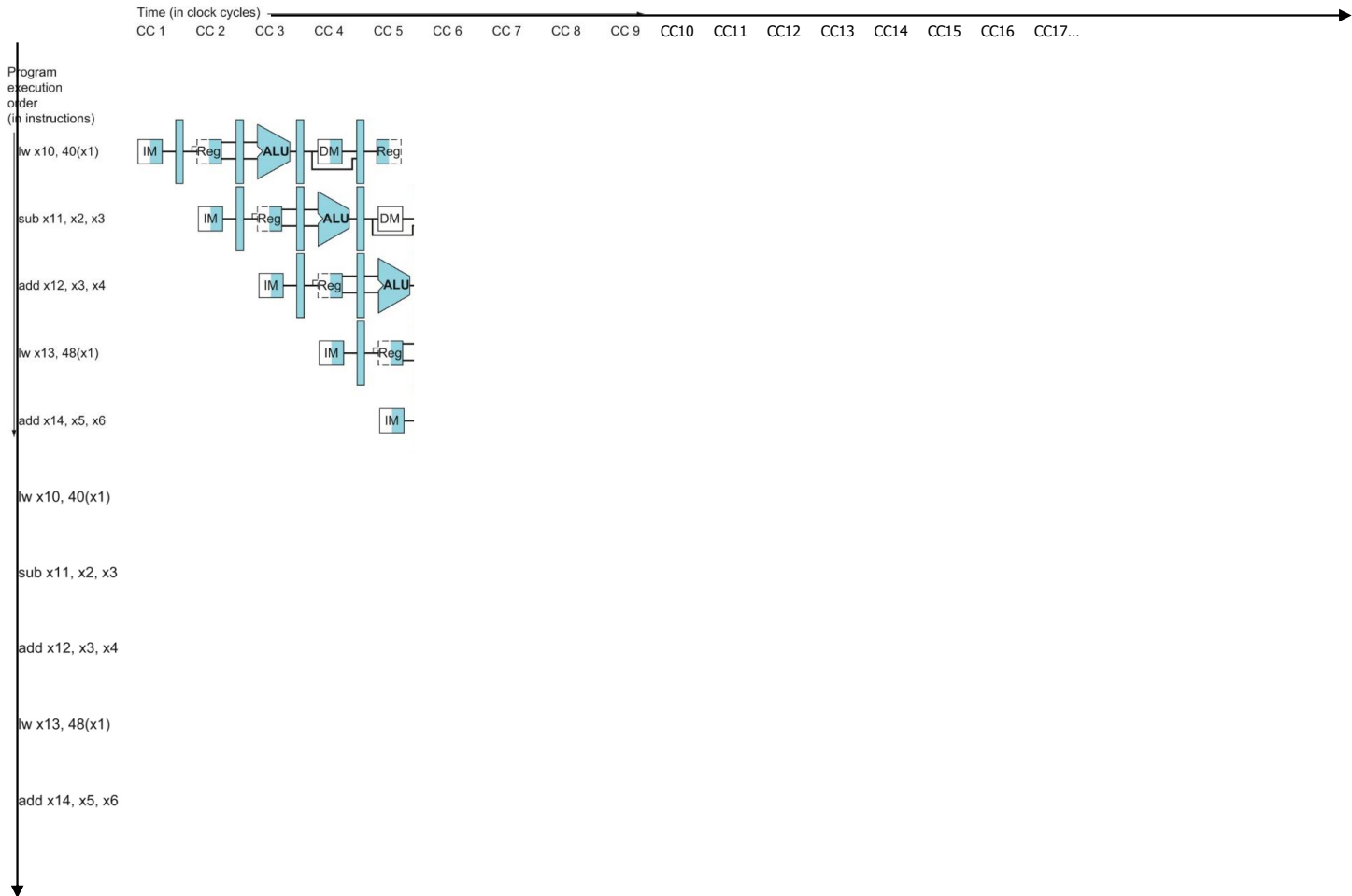
Multi-Cycle Pipeline Diagram



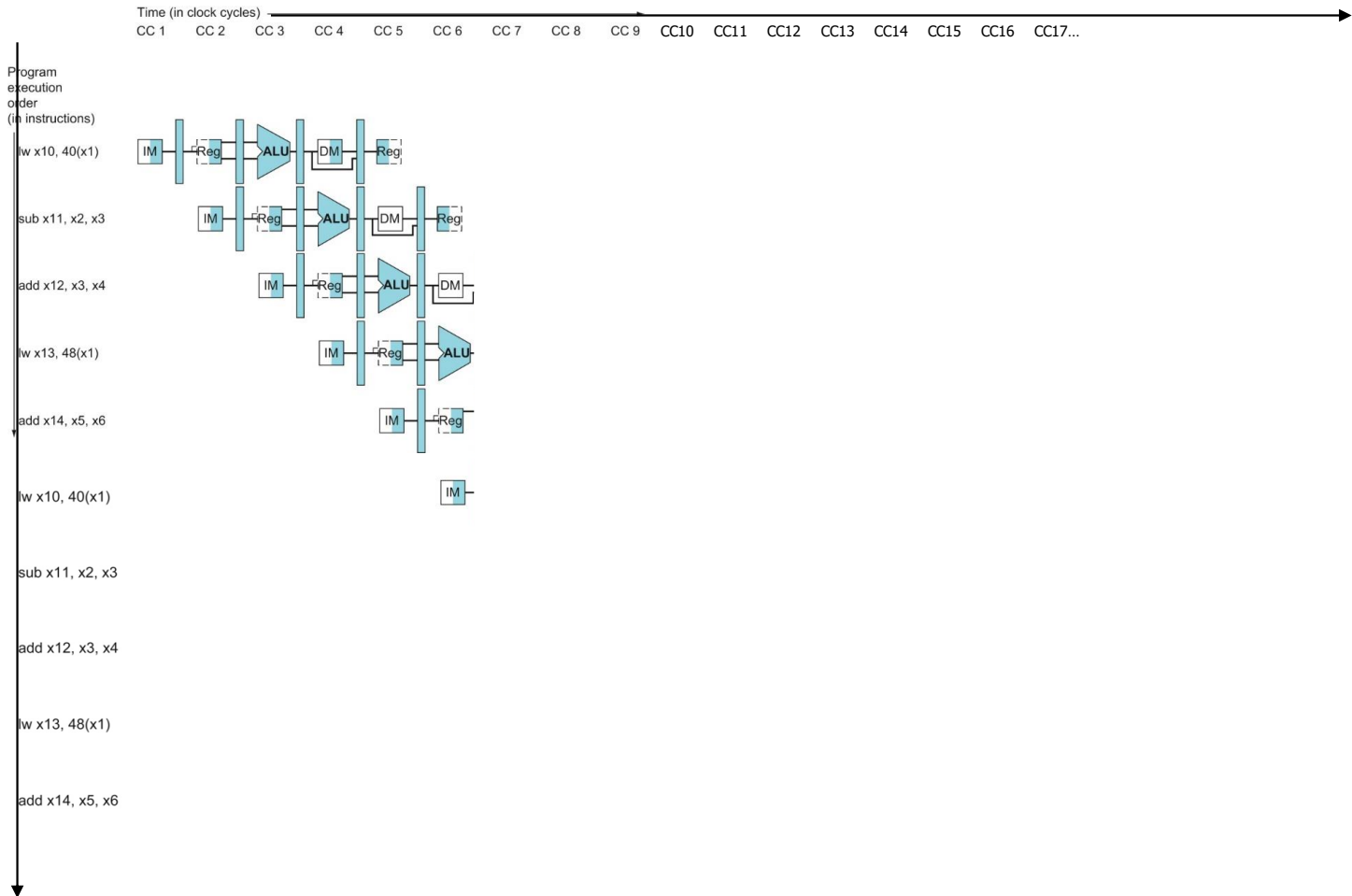
Multi-Cycle Pipeline Diagram



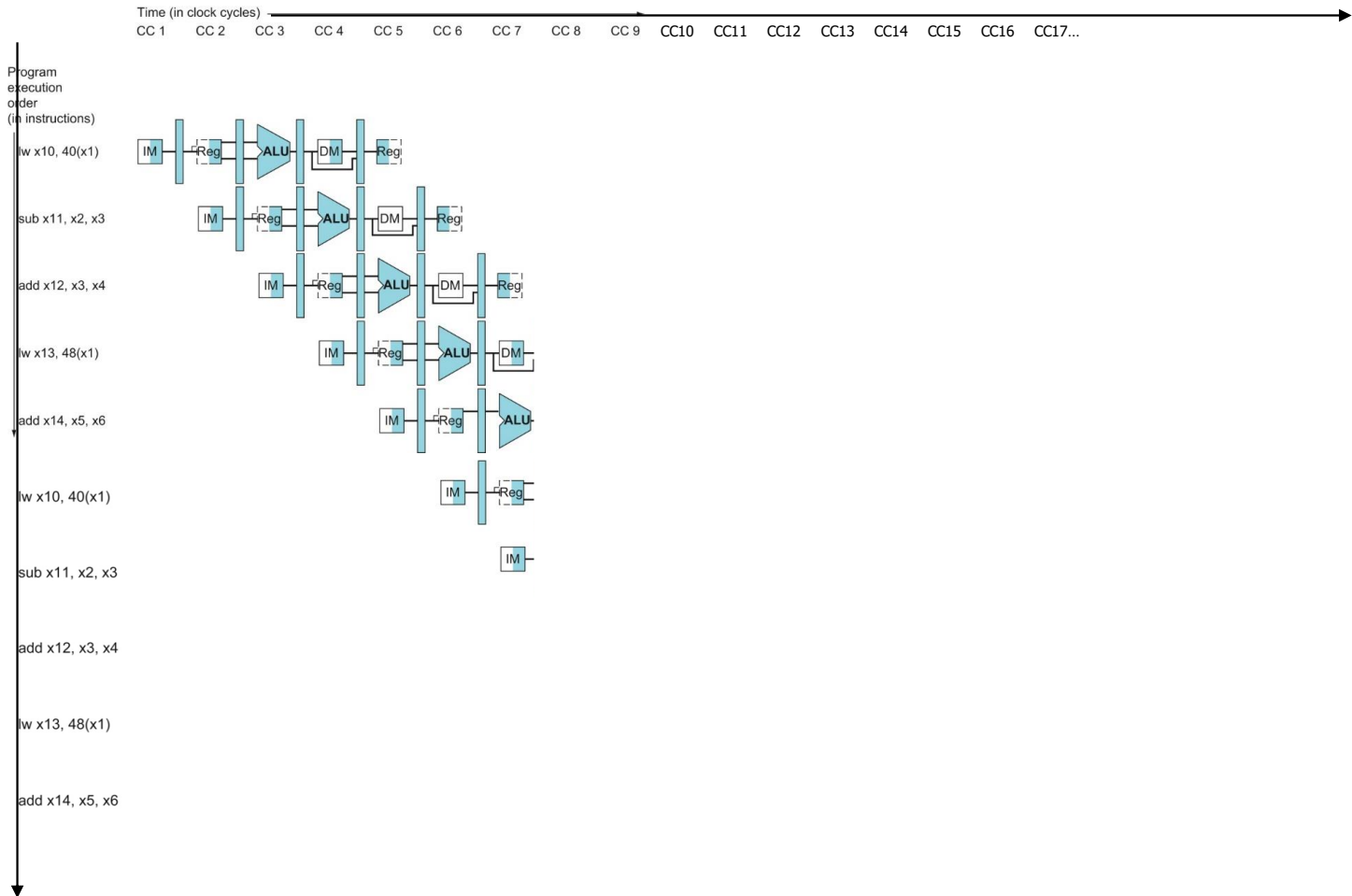
Multi-Cycle Pipeline Diagram



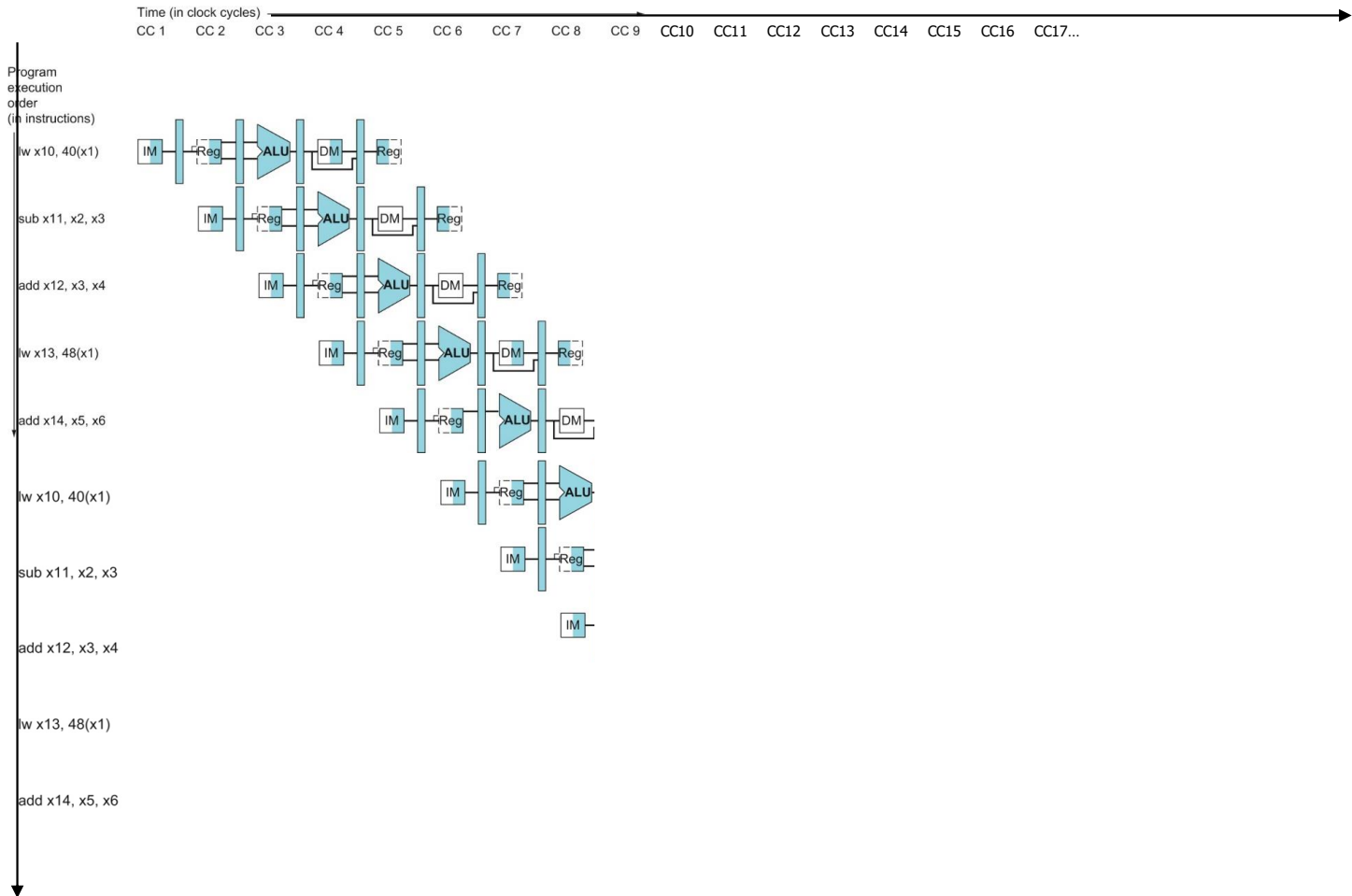
Multi-Cycle Pipeline Diagram



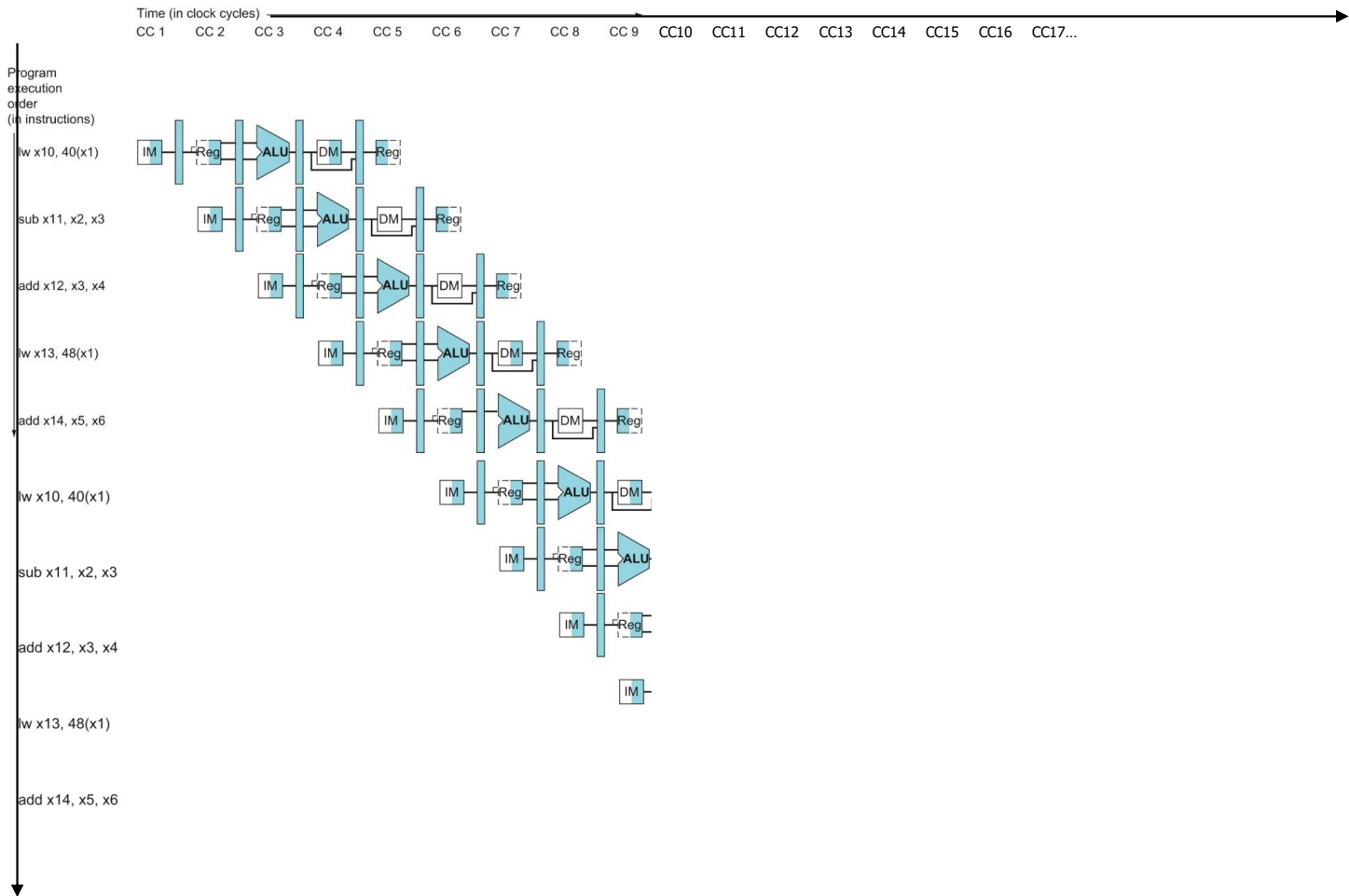
Multi-Cycle Pipeline Diagram



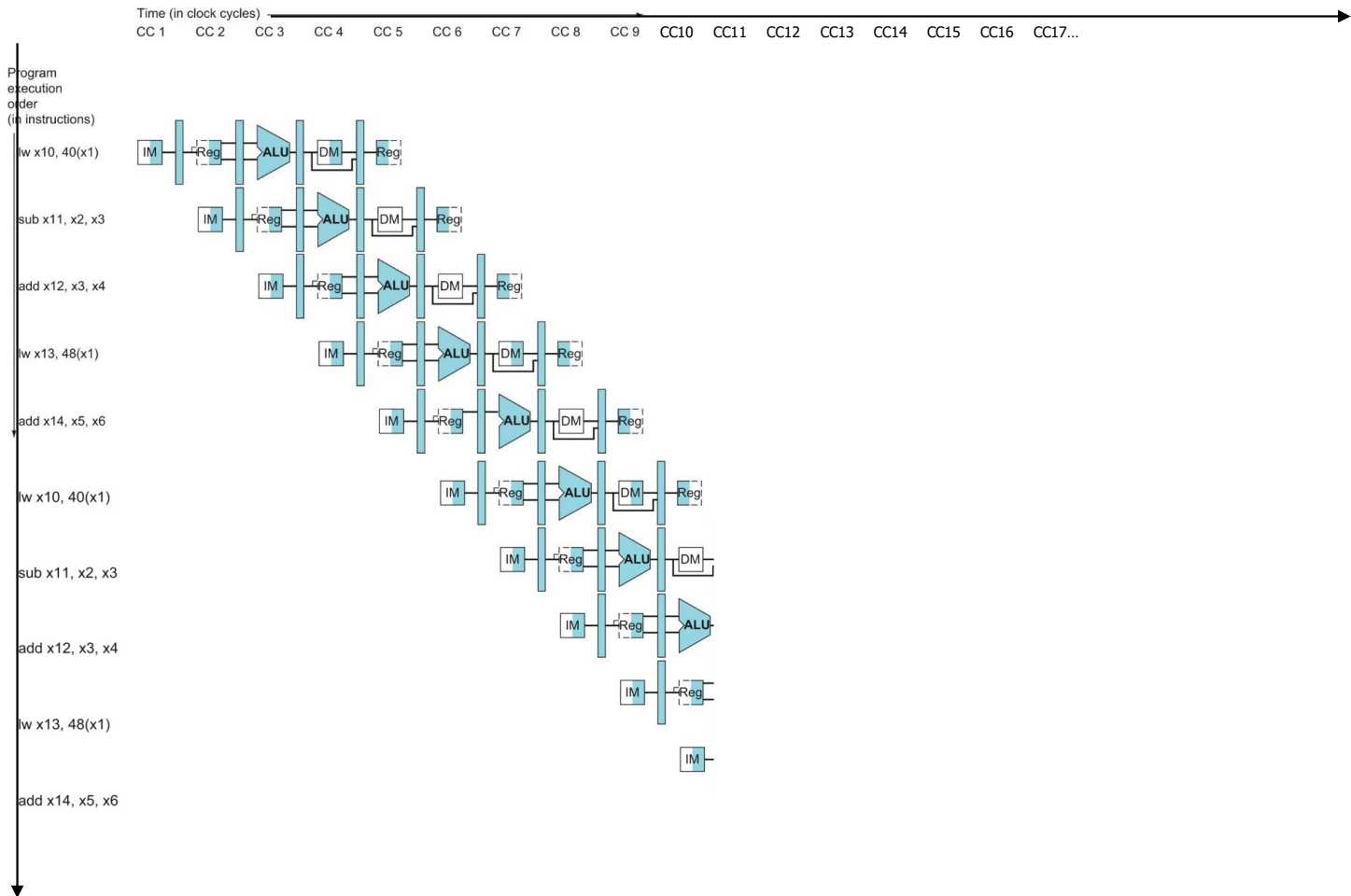
Multi-Cycle Pipeline Diagram



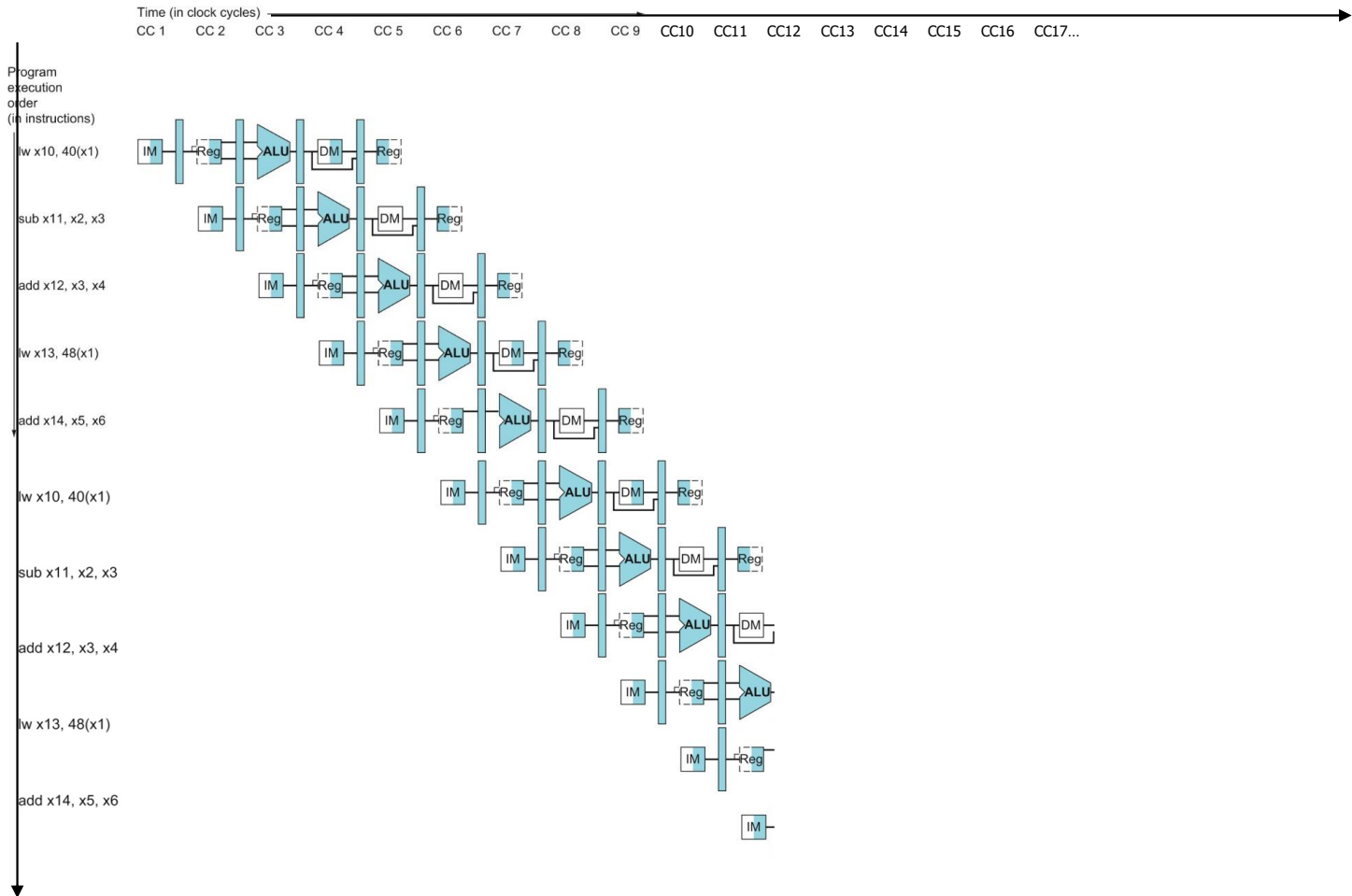
Multi-Cycle Pipeline Diagram



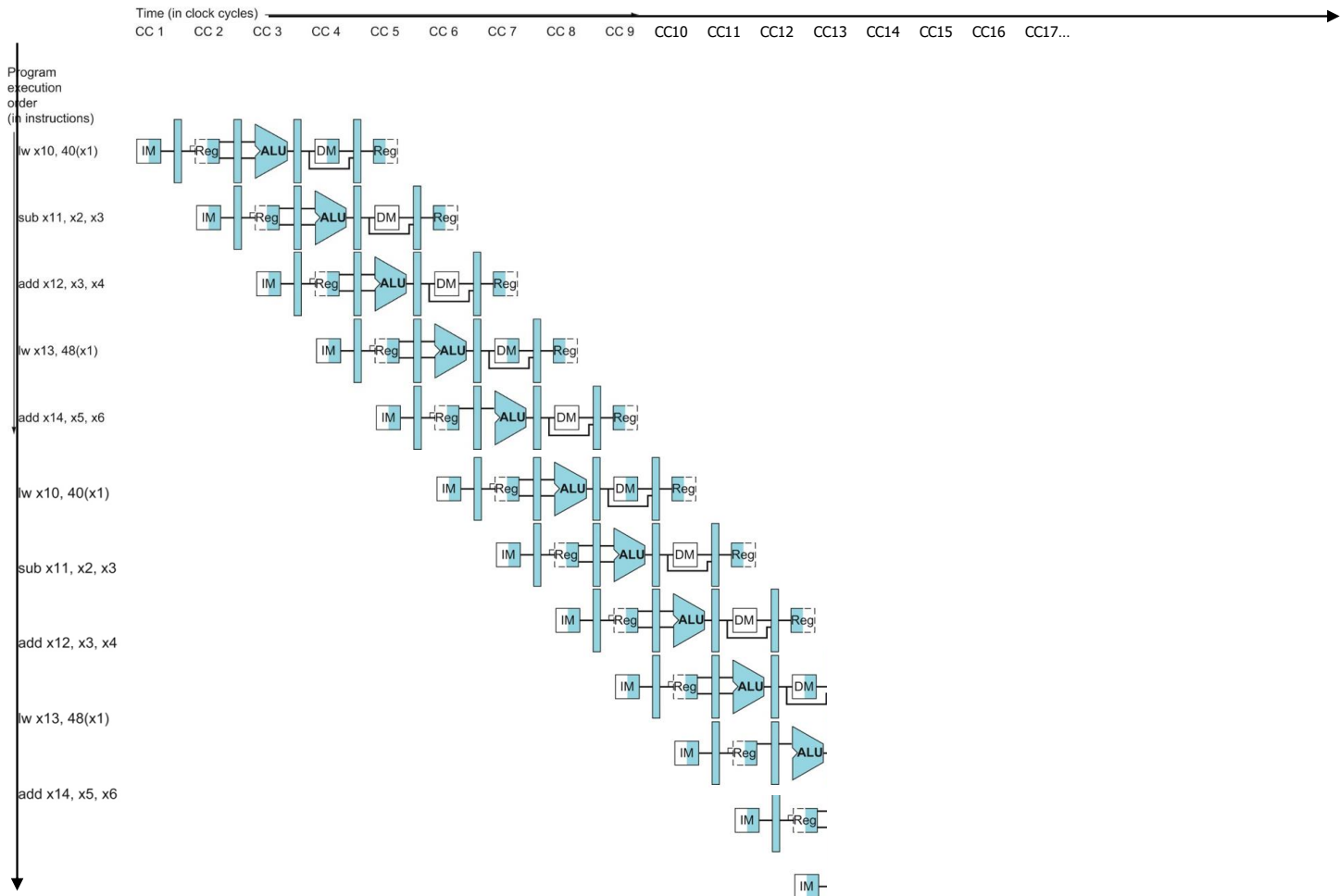
Multi-Cycle Pipeline Diagram



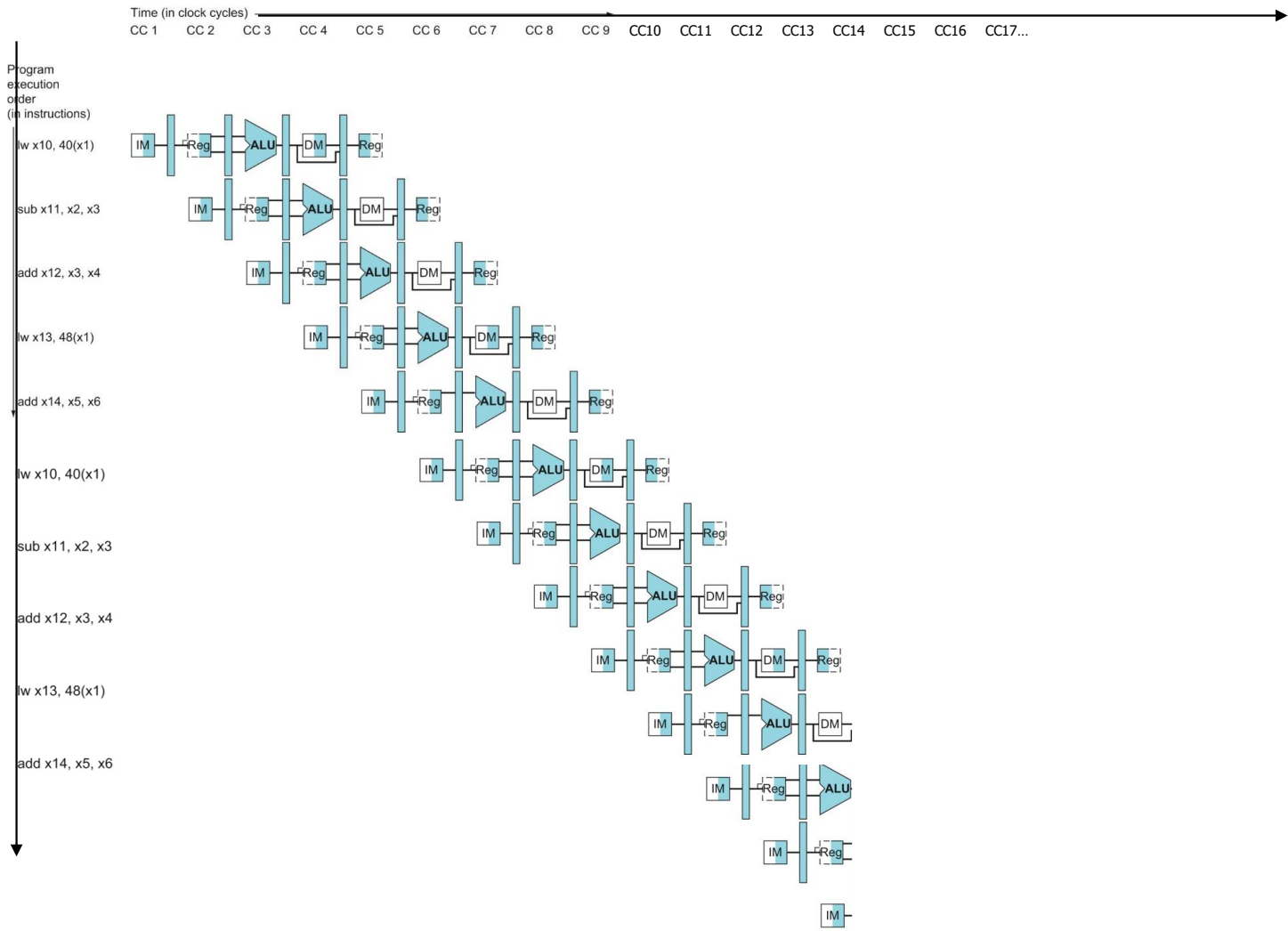
Multi-Cycle Pipeline Diagram



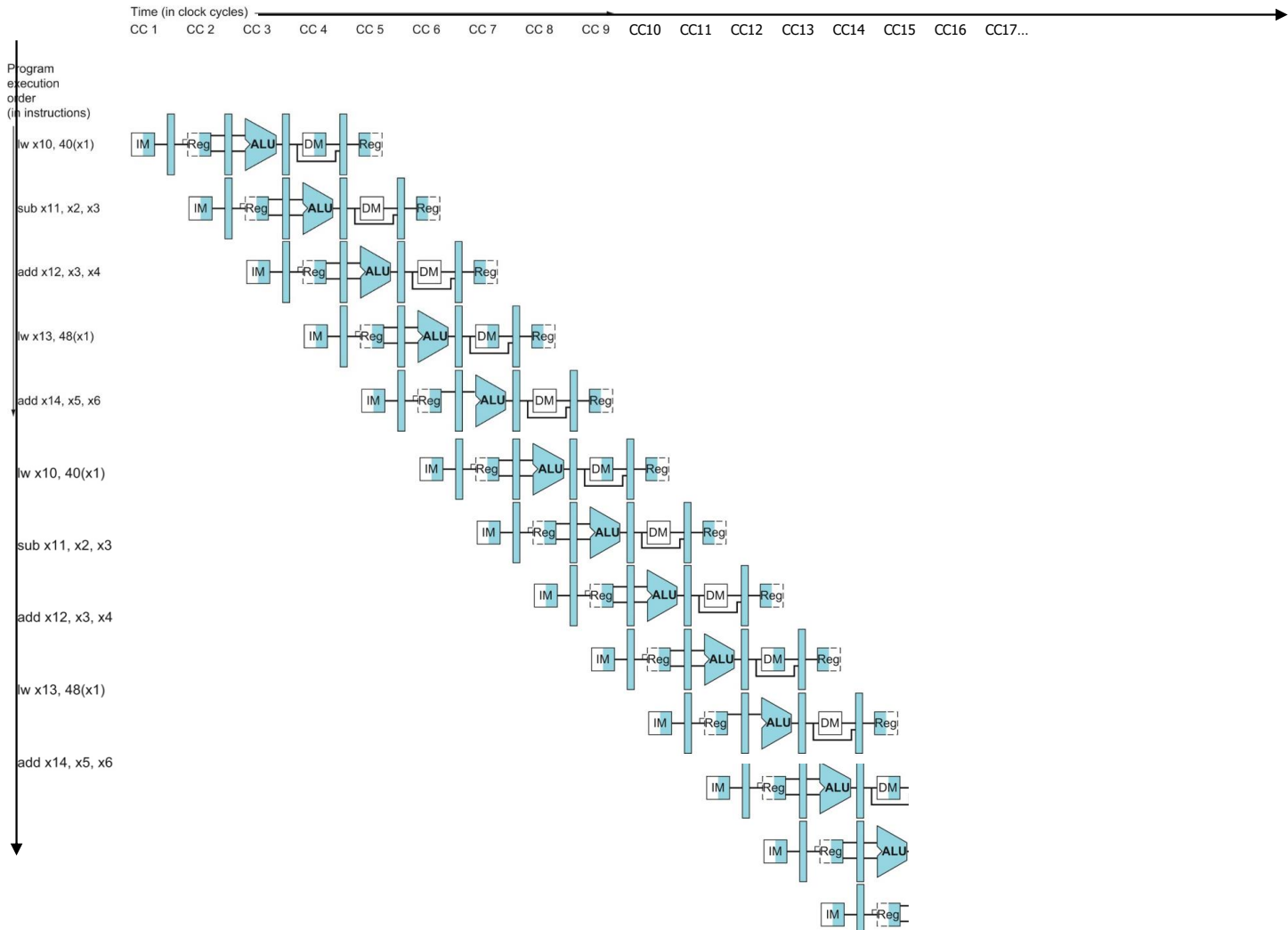
Multi-Cycle Pipeline Diagram



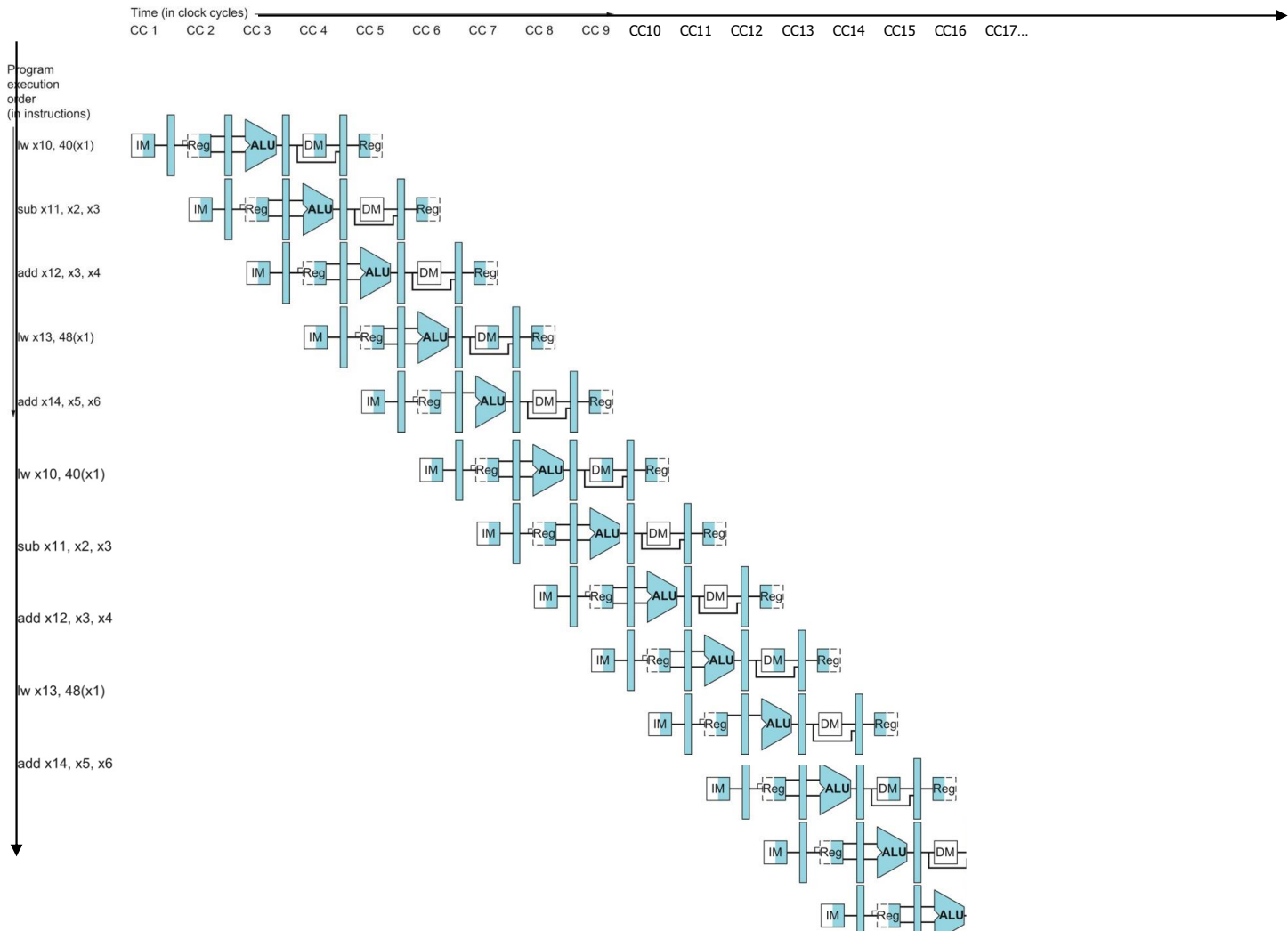
Multi-Cycle Pipeline Diagram



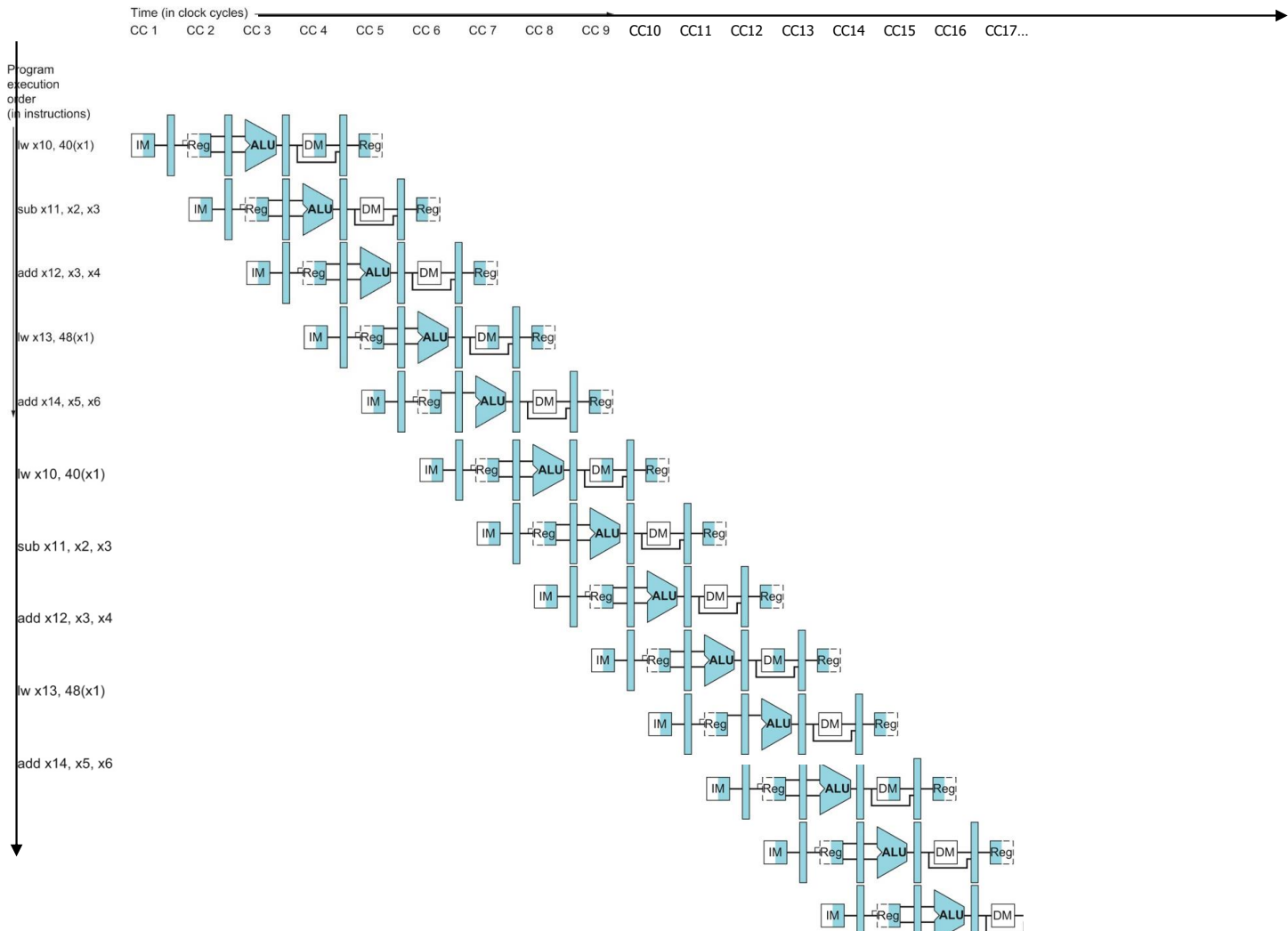
Multi-Cycle Pipeline Diagram



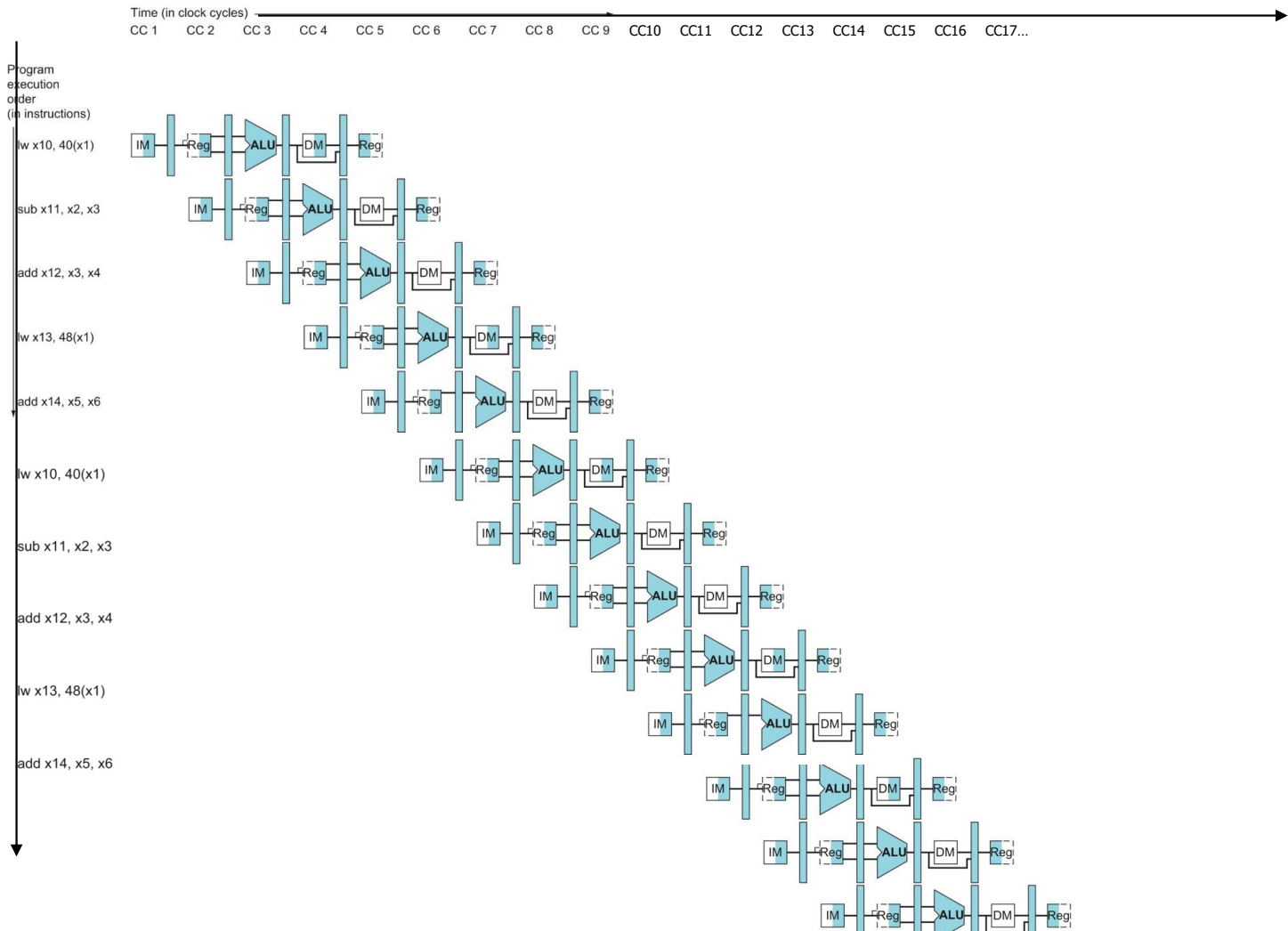
Multi-Cycle Pipeline Diagram



Multi-Cycle Pipeline Diagram

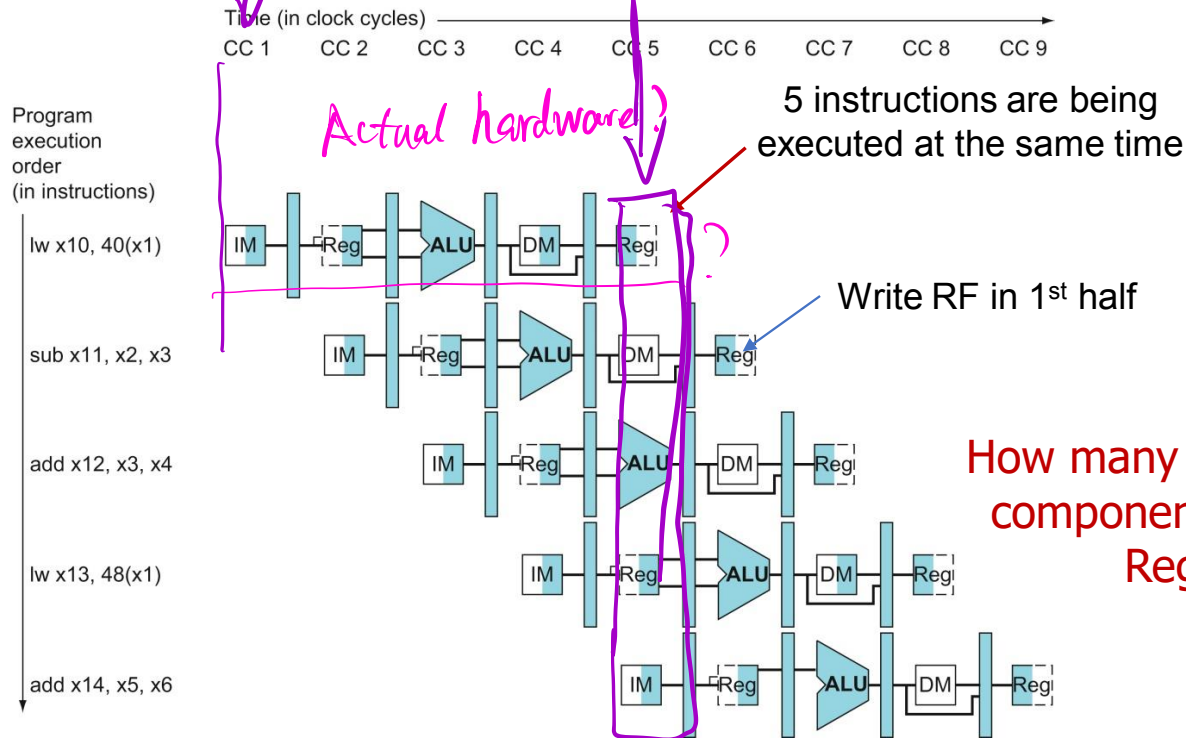


Multi-Cycle Pipeline Diagram



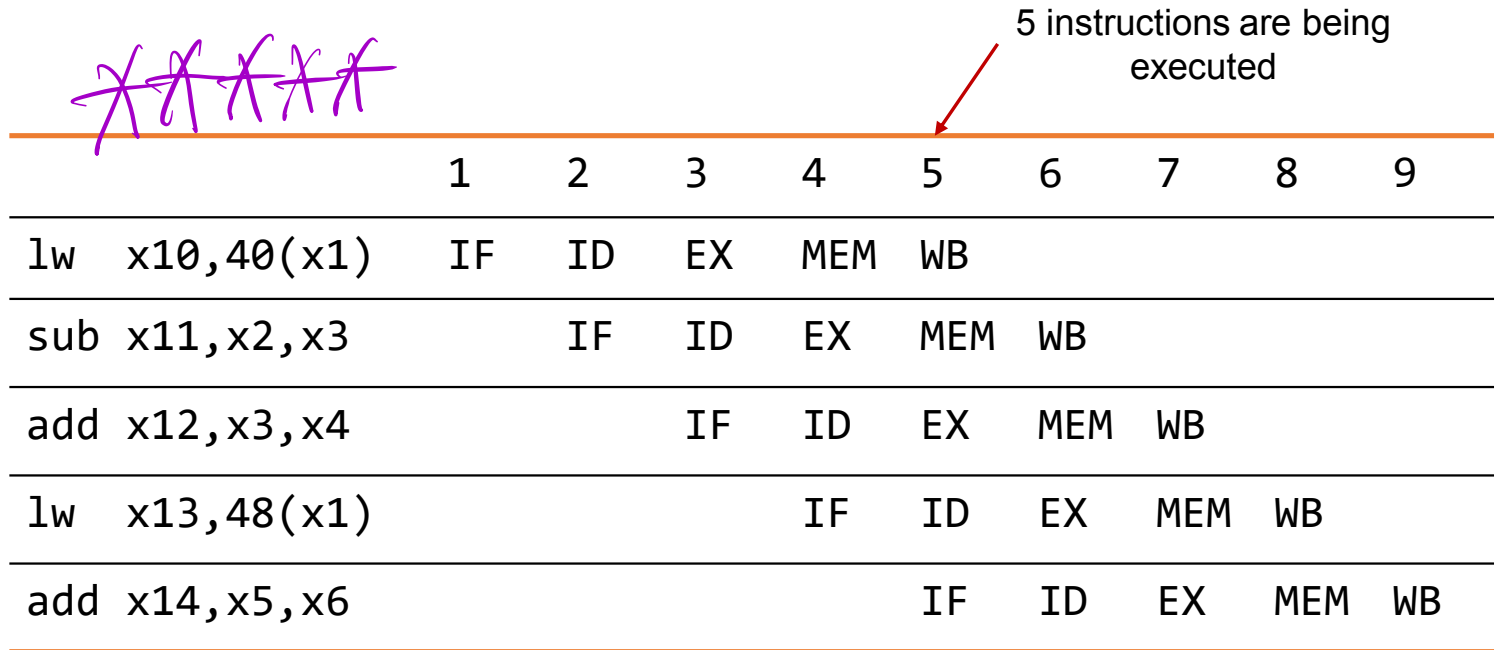
Multi-Cycle Pipeline Diagram

- Showing resource usage in multiple cycles
 - For each instruction, knows when a block is used
 - For each cycle, knows which block is doing what



Examples of Pipeline Diagram

- Use IF, ID, EX, MEM, WB to indicate pipe stages
 - Sometimes use EXE for EX and ME for MEM



One of the online chapters shows more detailed pipeline diagrams.

[Ch04_e2.pdf \(elsevier.com\)](#)

Troubles from Pipelining

- Pipeline Hazards

- **Structural hazards**: attempt to use the same resource by two different instructions at the same time

- **Data hazards**: attempt to use data before it is ready

- An instruction's source operand(s) are produced by a prior instruction still in the pipeline

- **Control hazards**: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated

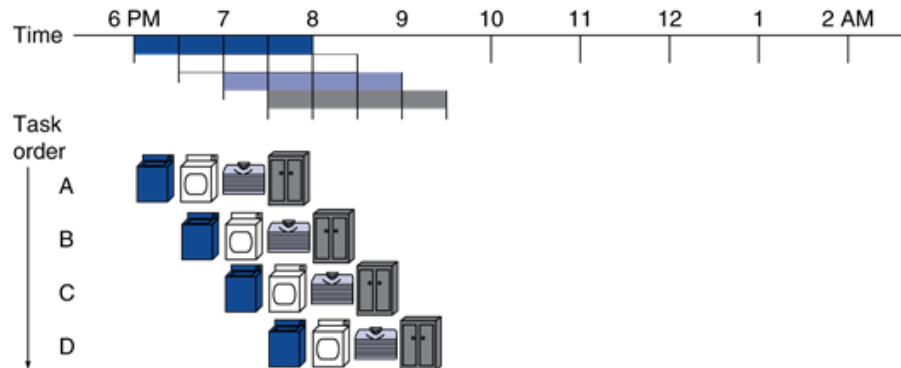
- branch and jump instructions, exceptions

Control unit

- Pipeline control must detect and take action to resolve hazards

Potential hazards in pipelined laundry example

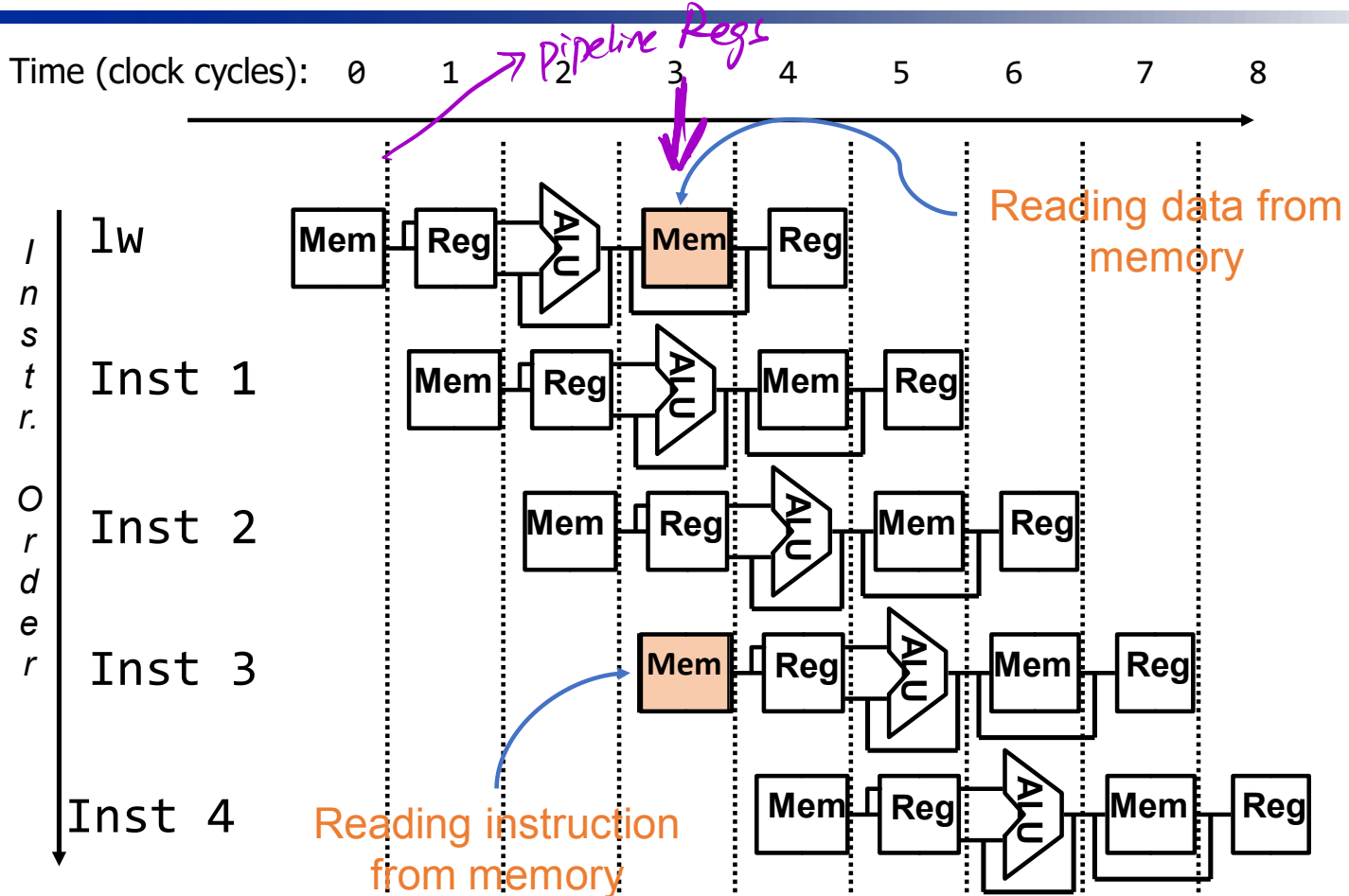
- What if you have a washer and dryer combo?
- What if you are the only person doing the laundry?



Dealing with Hazards

- We can usually resolve hazards by **waiting**
- We will find better ways to deal with hazards
- Dealing with structural hazards
 - Add more resources
 - Or share resources

A Single Memory Would Be a Structural Hazard



Fix: separate instr and data memories (I\$ and D\$)



Structural hazards in 5-stage pipeline

- Dealing with structural hazards
 - Add more resources (IM, DM)
 - Or Share resources (Reg)
- Memory
 - Add more resources: instruction memory and data memory
- Register
 - Share resources: write in the first half of a cycle and read in the second

Pipelining and ISA Design

- RISC-V ISA designed for pipelining
 - All instructions are 32-bits
 - Easier to fetch and decode in one cycle
 - c.f. x86: 1- to 17+ bytes instructions
 - Few and regular instruction formats
 - Can decode and read registers at the same time
 - Load/store addressing
 - Can calculate address in 3rd stage, access memory in 4th stage

Question

- In which stage are the control signals generated?
- A. IF
 - B. ID
 - C. EX
 - D. MEM
 - E. It depends on instruction types

Question

- In which stage is RegWrite used?
- A. IF
 - B. ID
 - C. EX
 - D. MEM
 - E. WB

Timing of writing to register file

