MICROPROCESSOR SYSTEMS (IAS0430)

Department of Computer Systems
Tallinn University of Technology

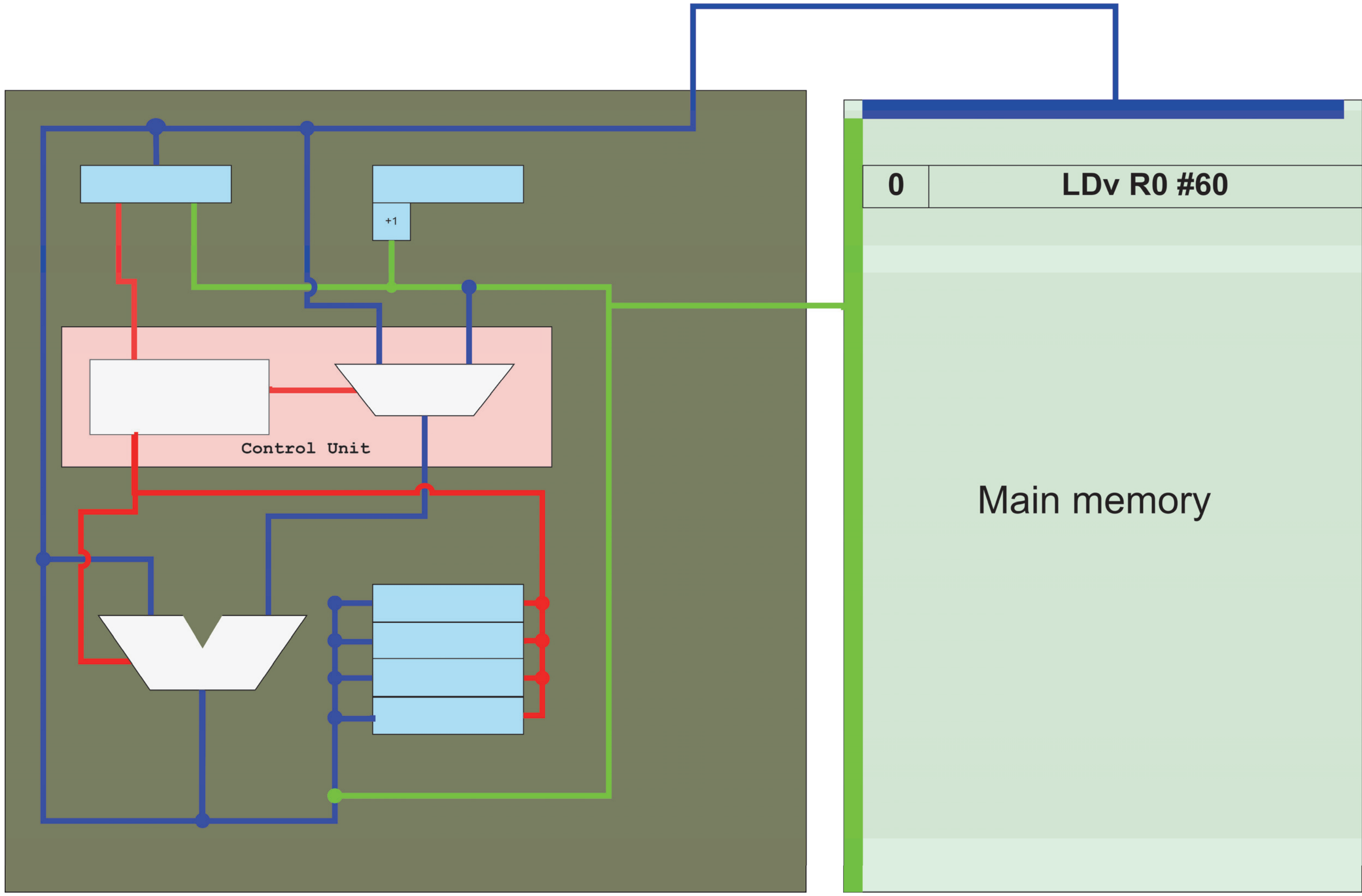
1.10.2021

THE PIPELINE

- The Pipeline is (mostly) a RISC based CPU feature.
  - The idea is to divide instructions into five stages of execution.
  - This will help executing multiple instructions at the same time.
    - Reducing execution time and increasing performance.
  - The instruction is divided into 5 stages as follows:
    - **Instruction Fetch (IF):** in this stage, the instruction is loaded into the Instruction register.
    - **Instruction Decode (ID):** in this stage the instruction is decided in the control unit decoder.
    - **Instruction Execute (EXE):** Execute the instruction.
    - **Memory Access (MEM):** access the memory (for single cycle instructions)
    - **Write Back (WB):** store result to register/memory address (for second cycle instructions)
  - **Why would we have a memory access and a write back?**

THE PIPELINE

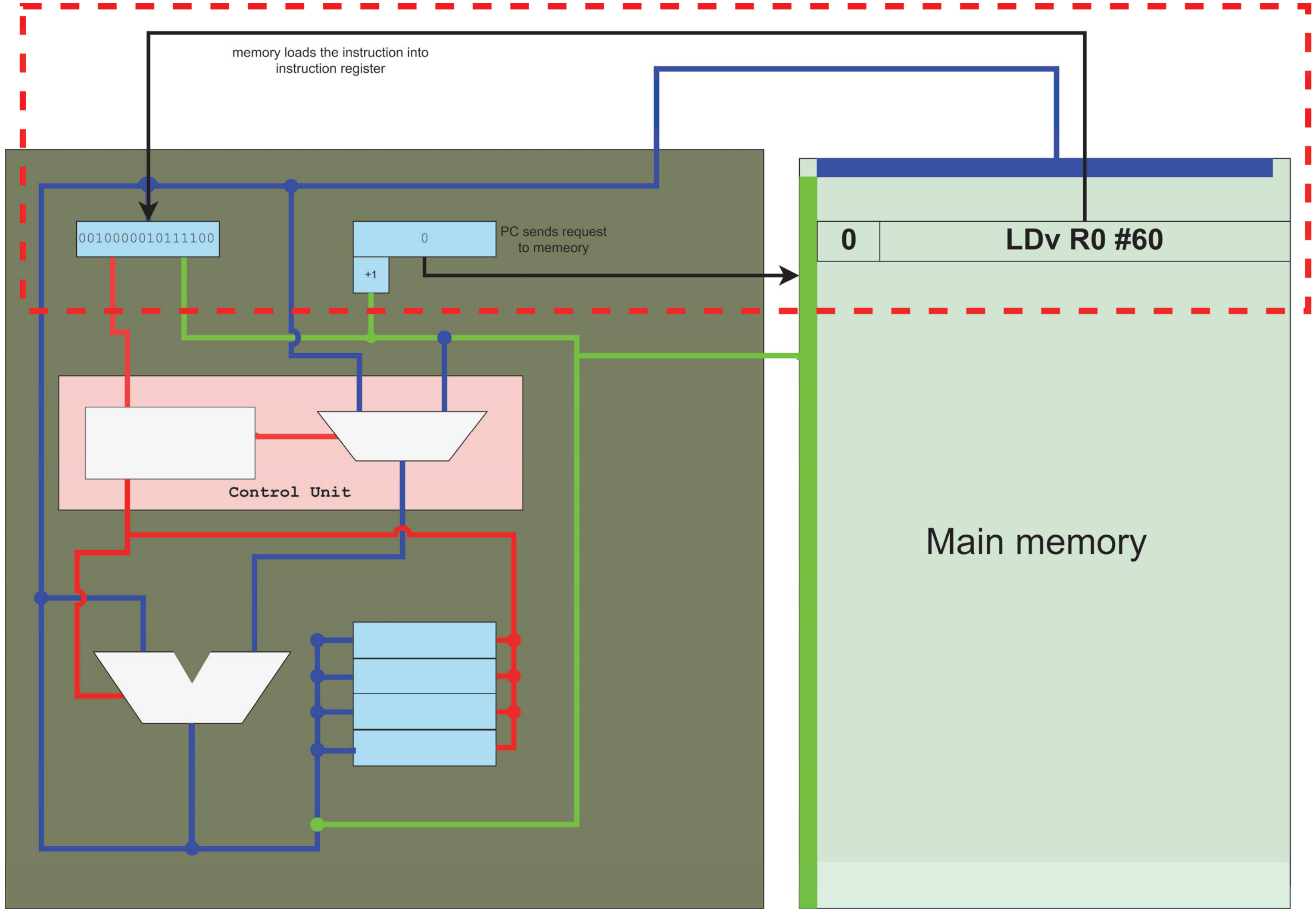
- Lets see how that works for a LD instruction.



TALTECH TALLINN UNIVERSITY OF TECHNOLOGY

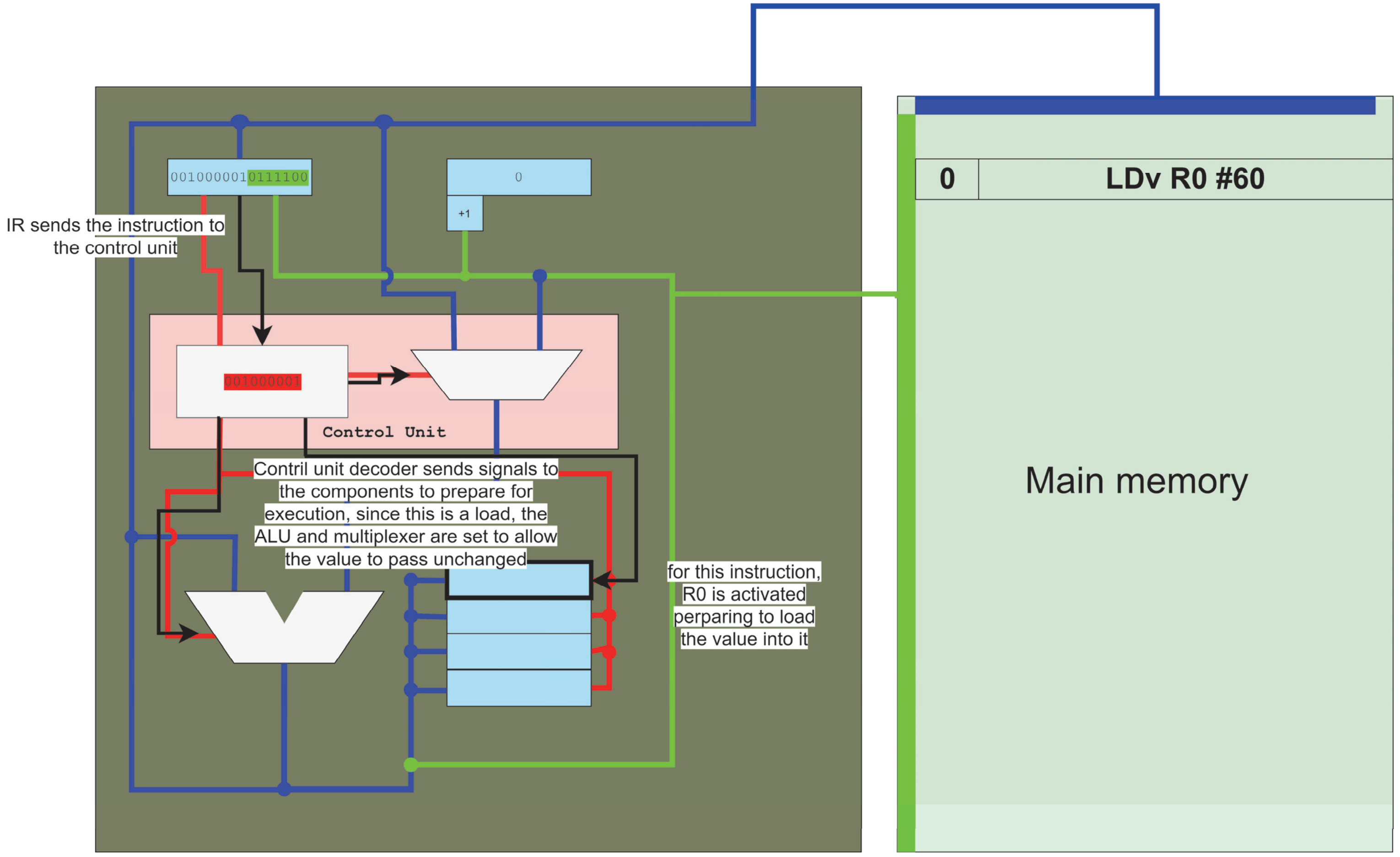
THE PIPELINE

- **Instruction Fetch (IF):** in this stage, the instruction is loaded into the Instruction register.


memory loads the instruction into instruction register
0010000010111100
0
+1
PC sends request to memeory
0
LDv R0 #60
Control Unit
Main memory

THE PIPELINE

- **Instruction Decode (ID):** in this stage the instruction is decided in the control unit decoder.



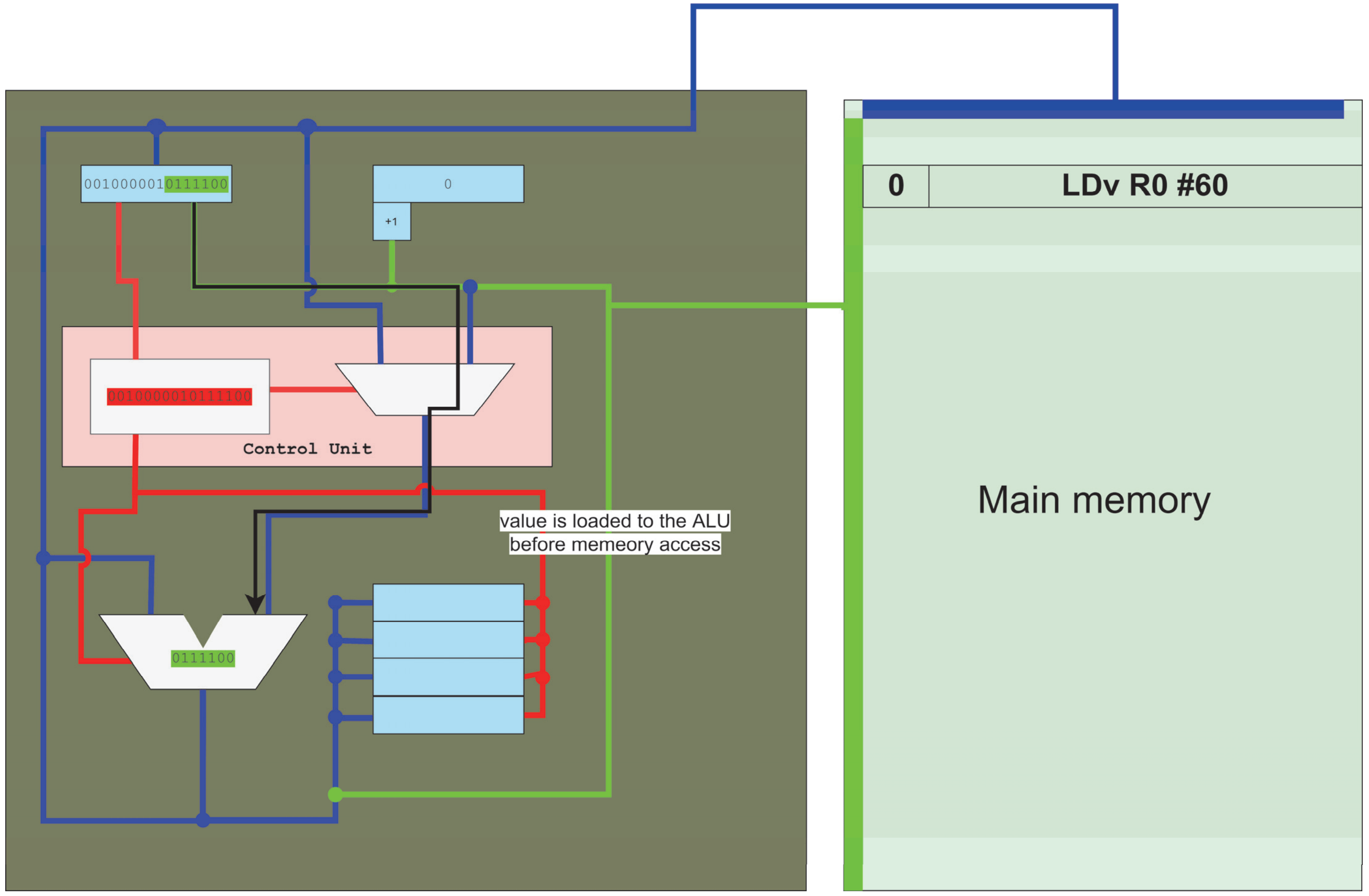
IR sends the instruction to the control unit

Contril unit decoder sends signals to the components to prepare for execution, since this is a load, the ALU and multiplexer are set to allow the value to pass unchanged

for this instruction, R0 is activated perparing to load the value into it

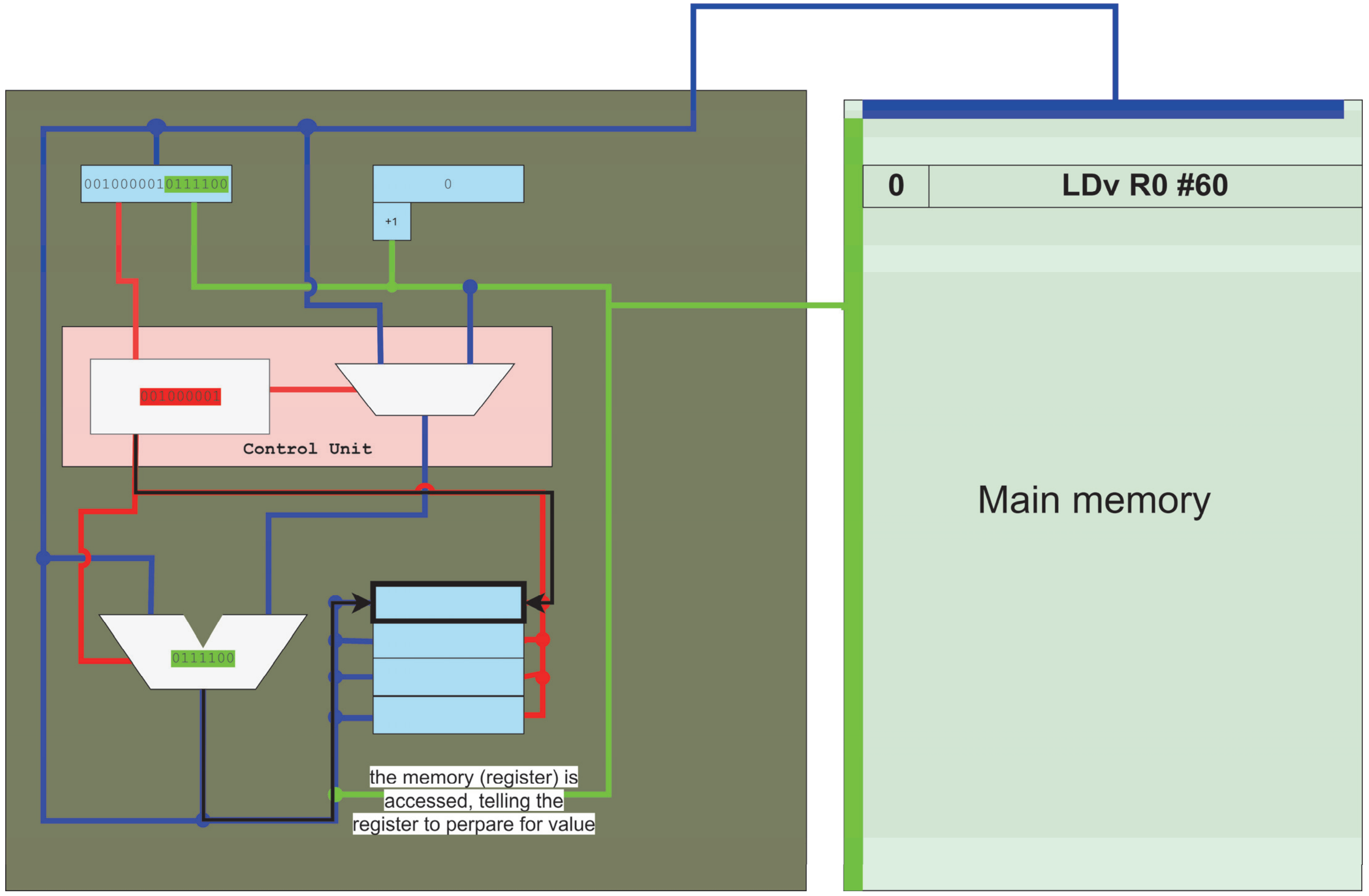
THE PIPELINE

- **Instruction Execute (EXE):** Execute the instruction.



THE PIPELINE

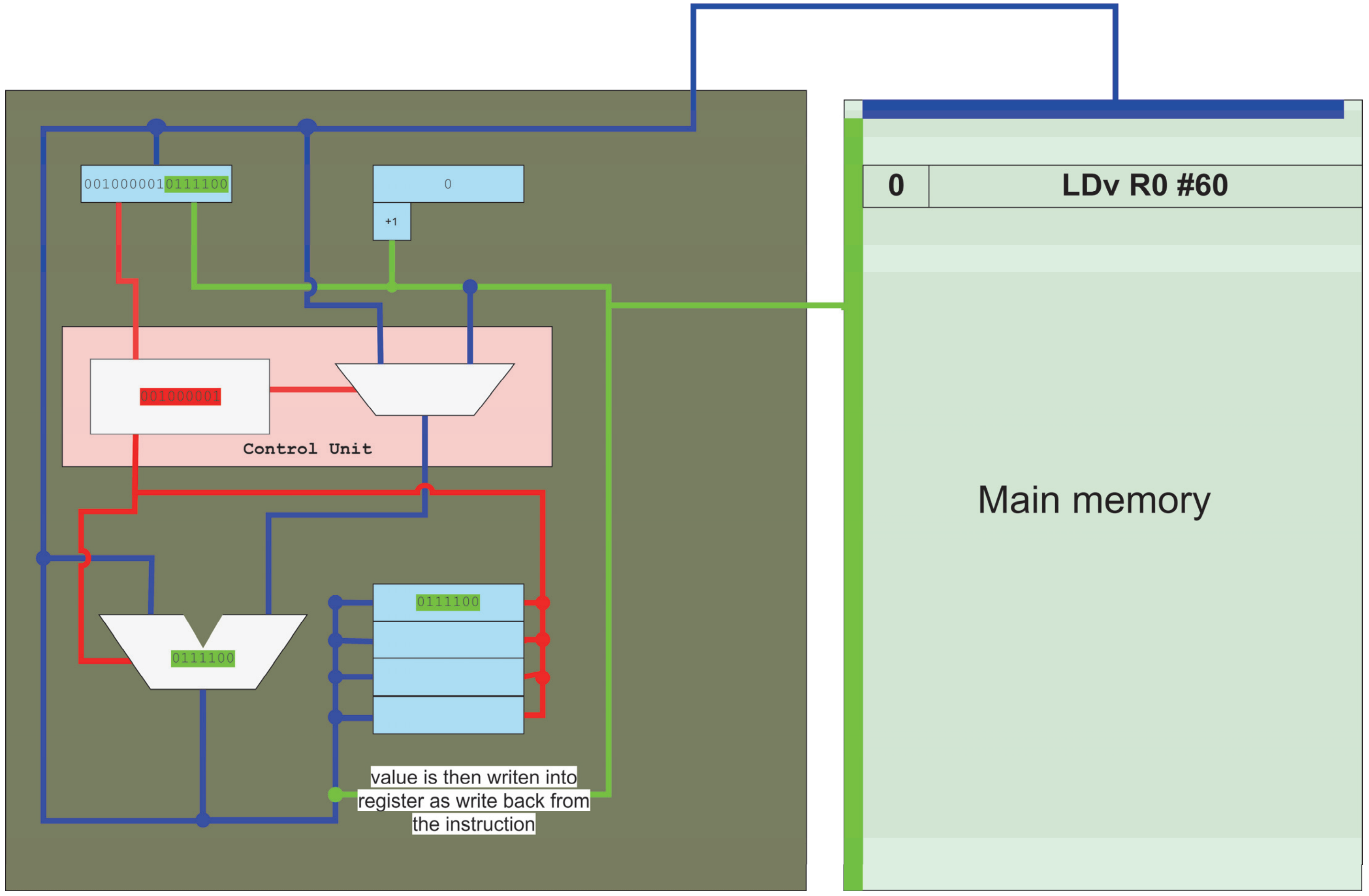
- **Memory Access (MEM):** access the memory (for single cycle instructions)



the memory (register) is accessed, telling the register to perpare for value

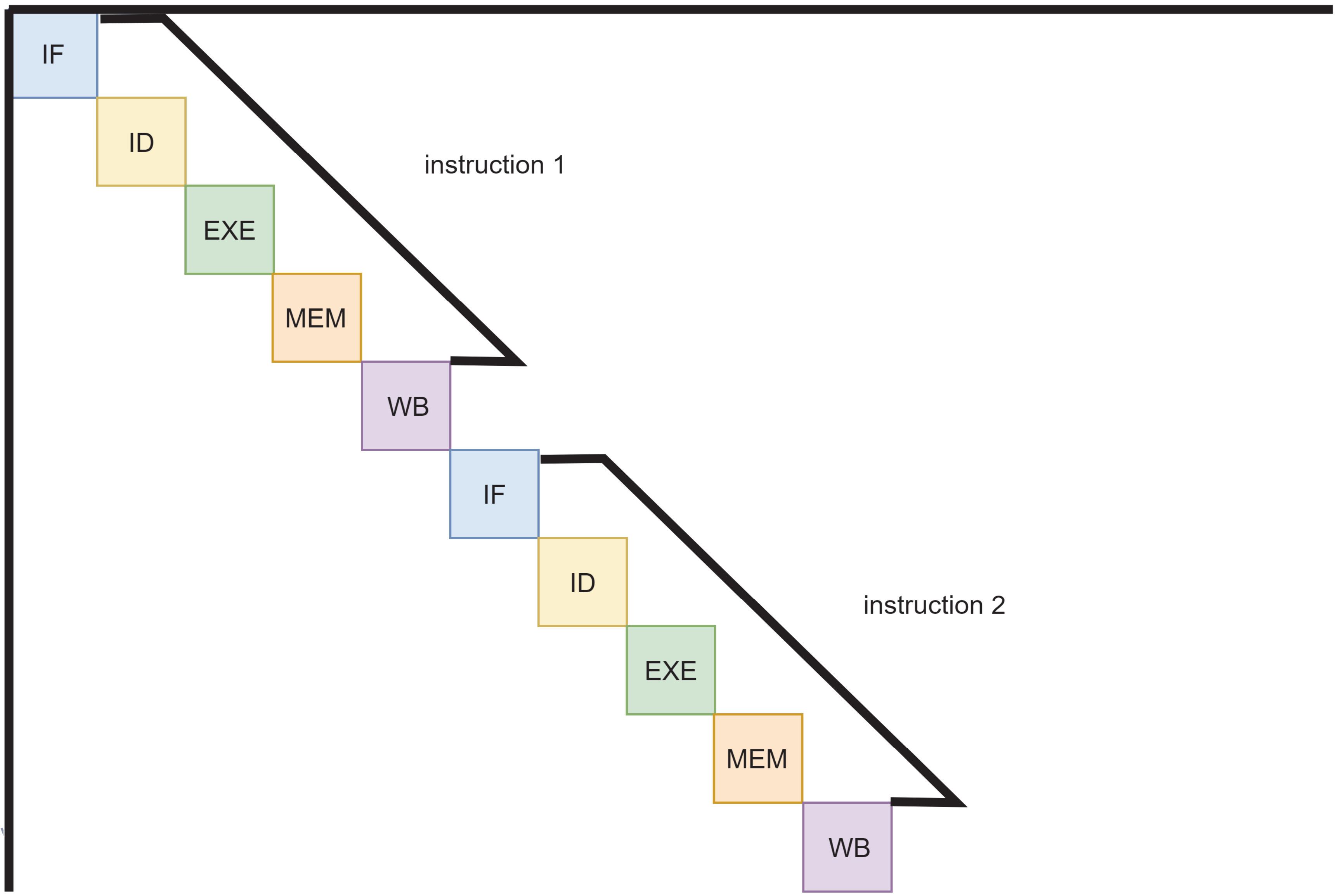
THE PIPELINE

- **Write Back (WB):** store result to register/memory address (for second cycle instructions)



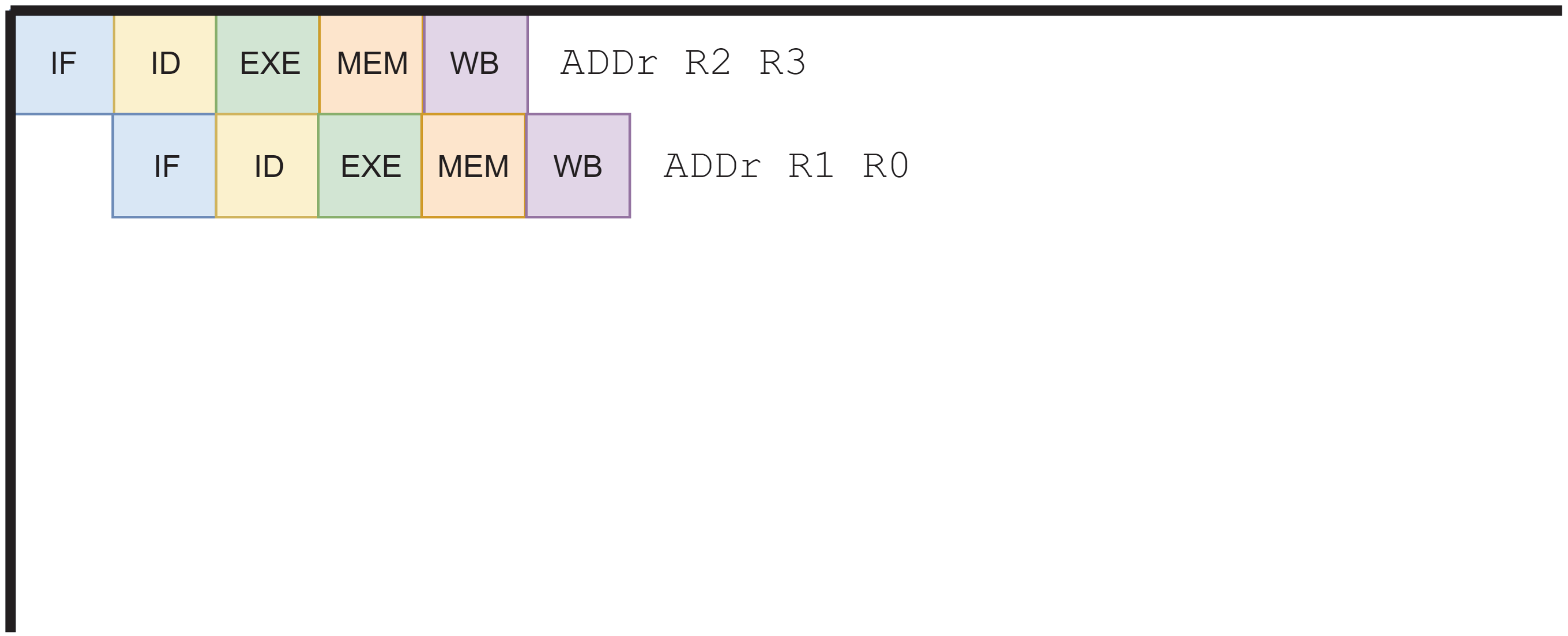
NO PIPELINE

- The stages of execution will take 1 cycle each.



PIPELINE

- The stages of execution will take 1 pipeline cycle each.
- Using the pipeline, it will significantly reduce the time to execute those instructions.



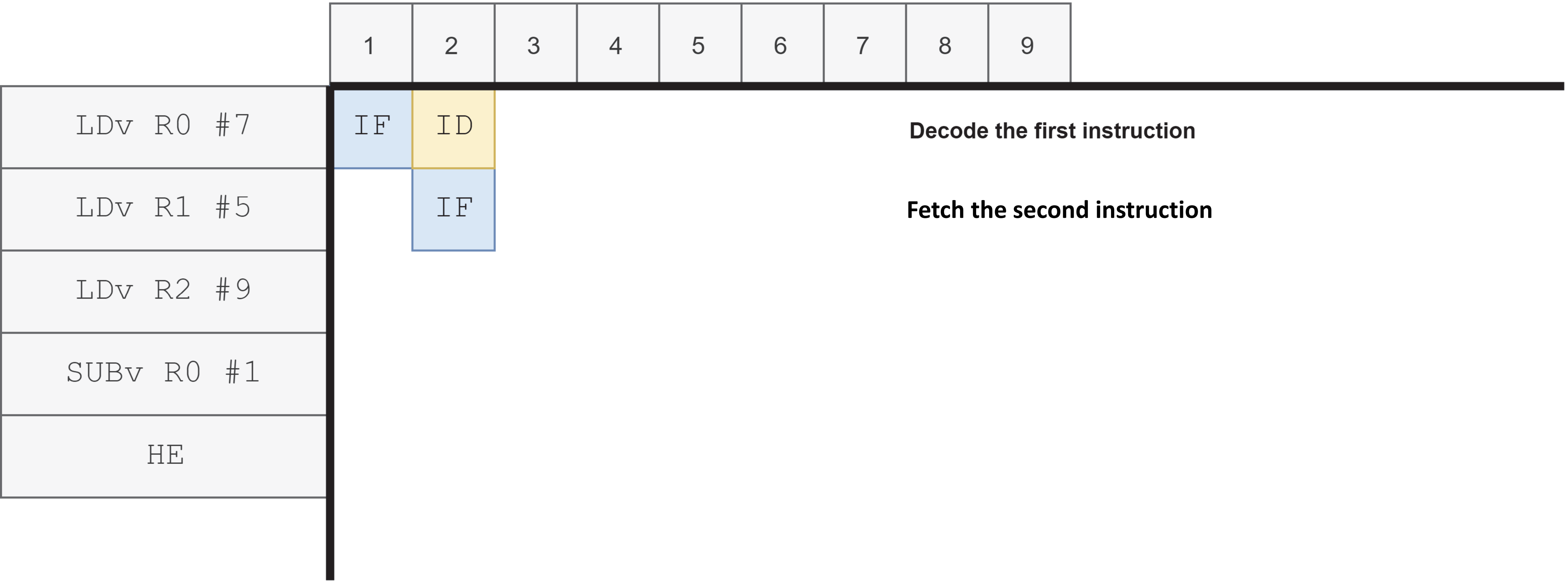
	1	2	3	4	5	6	7	8	9
LDv R0 #7									
LDv R1 #5									
LDv R2 #9									
SUBv R0 #1									
HE									

- Suppose we want to execute the above instructions.

	1	2	3	4	5	6	7	8	9
LDv R0 #7	IF								
LDv R1 #5									
LDv R2 #9									
SUBv R0 #1									
HE									

Fetch the first instruction

- Suppose we want to execute the above instructions.



- Suppose we want to execute the above instructions.

	1	2	3	4	5	6	7	8	9
LDv R0 #7	IF	ID	EXE						
LDv R1 #5		IF	ID						
LDv R2 #9			IF						
SUBv R0 #1									
HE									

Execute the first instruction

Decode the second instruction

Fetch the third instruction

- Suppose we want to execute the above instructions.

	1	2	3	4	5	6	7	8	9
LDv R0 #7	IF	ID	EXE	MEM					
LDv R1 #5		IF	ID	EXE					
LDv R2 #9			IF	ID					
SUBv R0 #1				IF					
HE									

Access memory for the first instruction

Execute the second instruction

Decode the third instruction

Fetch the fourth instruction

- Suppose we want to execute the above instructions.

	1	2	3	4	5	6	7	8	9	
LDv R0 #7	IF	ID	EXE	MEM	WB					**Write back for the first instruction**
LDv R1 #5		IF	ID	EXE	MEM					**Access memory for the second instruction**
LDv R2 #9			IF	ID	EXE					**Execute the third instruction**
SUBv R0 #1				IF	ID					**Decode the fourth instruction**
HE					IF					**Fetch the last instruction**

- Suppose we want to execute the above instructions.

	1	2	3	4	5	6	7	8	9
LDv R0 #7	IF	ID	EXE	MEM	WB				
LDv R1 #5		IF	ID	EXE	MEM	WB			
LDv R2 #9			IF	ID	EXE	MEM	WB		
SUBv R0 #1				IF	ID	EXE	MEM	WB	
HE					IF	ID	EXE	MEM	WB

- Then we finish executing the rest of the stages

THE PIPELINE

- **This is not perfect though….**
- This can have problems:
  - Sometimes data is not ready for new instructions to use it.
  - Sometimes different instructions try to use the same CPU resources.
    - E.g:
      - **Instruction A** is adding 3 and 5 and putting it in Reg**X**
      - **Instruction B** is adding Reg**X** with 6.
        - But, **instruction A** only writes back to Reg**X** at 5th pipeline cycle, and **instruction B** will need data in register Reg**X** at the 3rd pipeline cycle!!
        - This will cause a data hazard.
      - Also, **instruction B** will require to access Reg**X** while **instruction A** is executing an addition that requires Reg**X** to be allocated for it.
        - This will cause a structural hazard.

THE PIPELINE

- **This is not perfect though….**
- This can have problems:
  - Sometimes data is not ready for new instructions to use it.
  - Sometimes different instructions try to use the same CPU resources.
    - E.g:
      - **Instruction A** is adding 3 and 5 and putting it in Reg**X**
      - **Instruction B** is adding Reg**X** with 6.
        - But, **instruction A** only writes back to Reg**X** at 5th pipeline cycle, and **instruction B** will need data in register Reg**X** at the 3rd pipeline cycle!!
        - This will cause a data hazard.
      - Also, **instruction B** will require to access Reg**X** while **instruction A** is executing an addition that requires Reg**X** to be allocated for it.
        - This will cause a structural hazard.
      - Let us see how that works!!

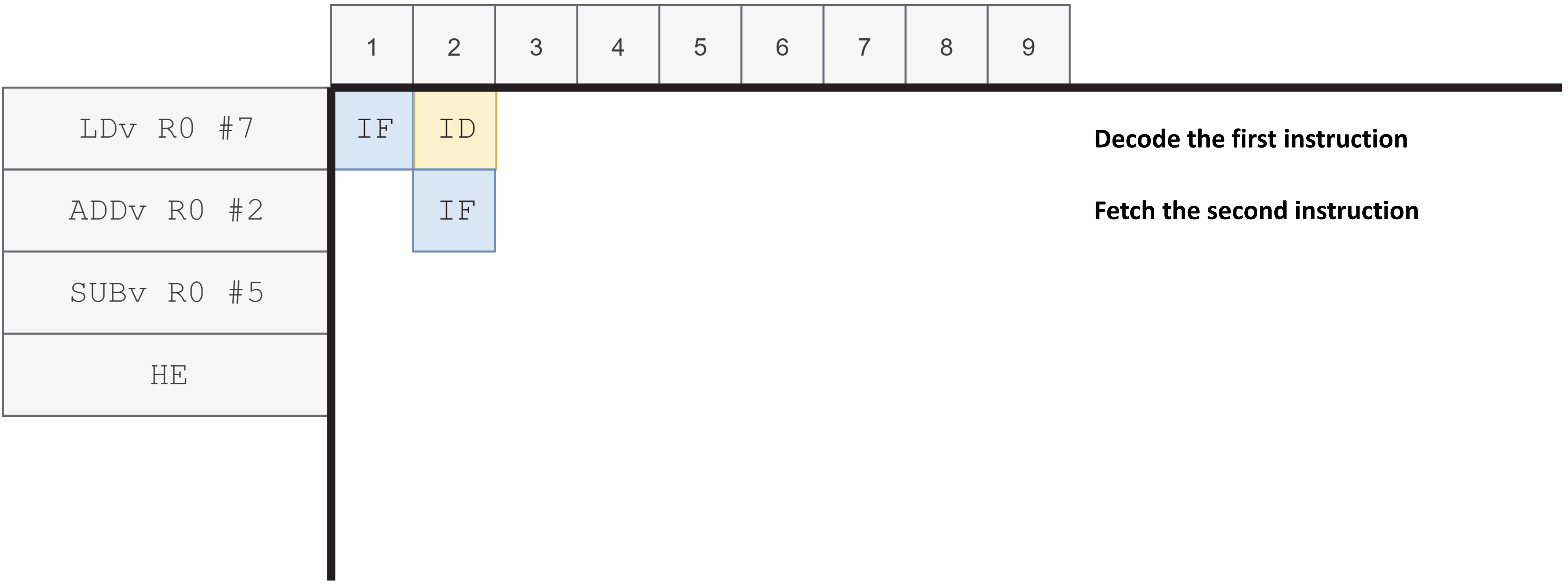
	1	2	3	4	5	6	7	8	9
`LDv R0 #7`									
`ADDv R0 #2`									
`SUBv R0 #5`									
`HE`									

- Lets try to run this program here

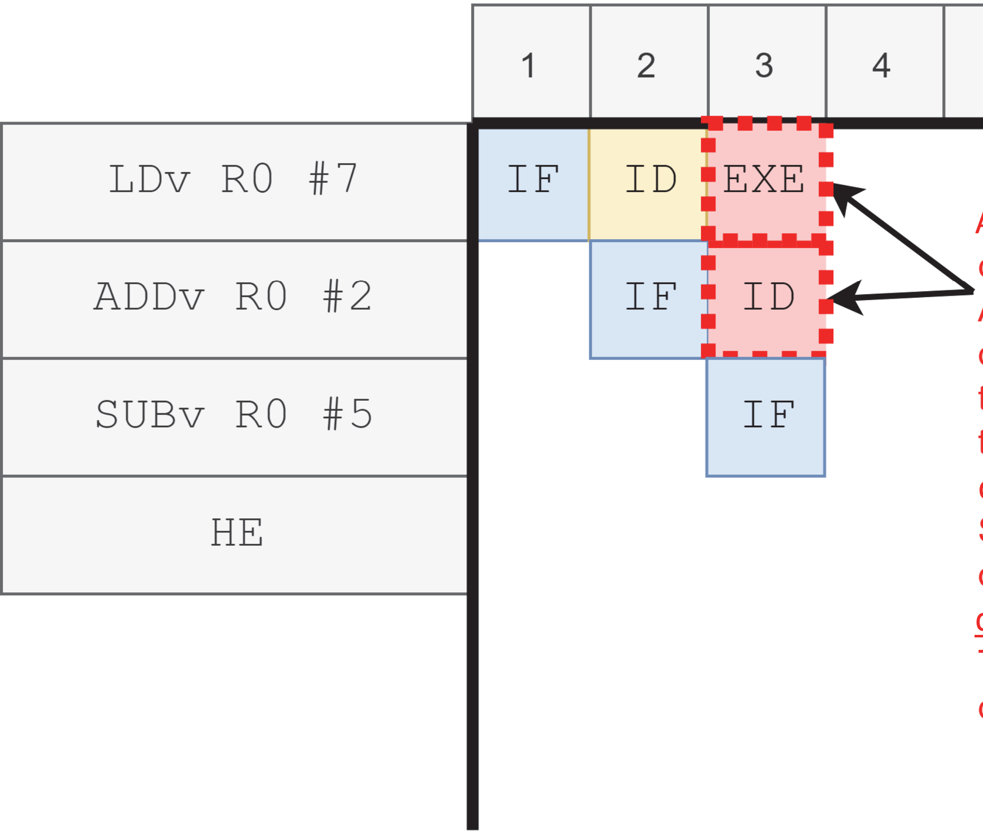
	1	2	3	4	5	6	7	8	9
LDv R0 #7	IF								
ADDv R0 #2									
SUBv R0 #5									
HE									

Fetch the first instruction

- Lets try to run this program here



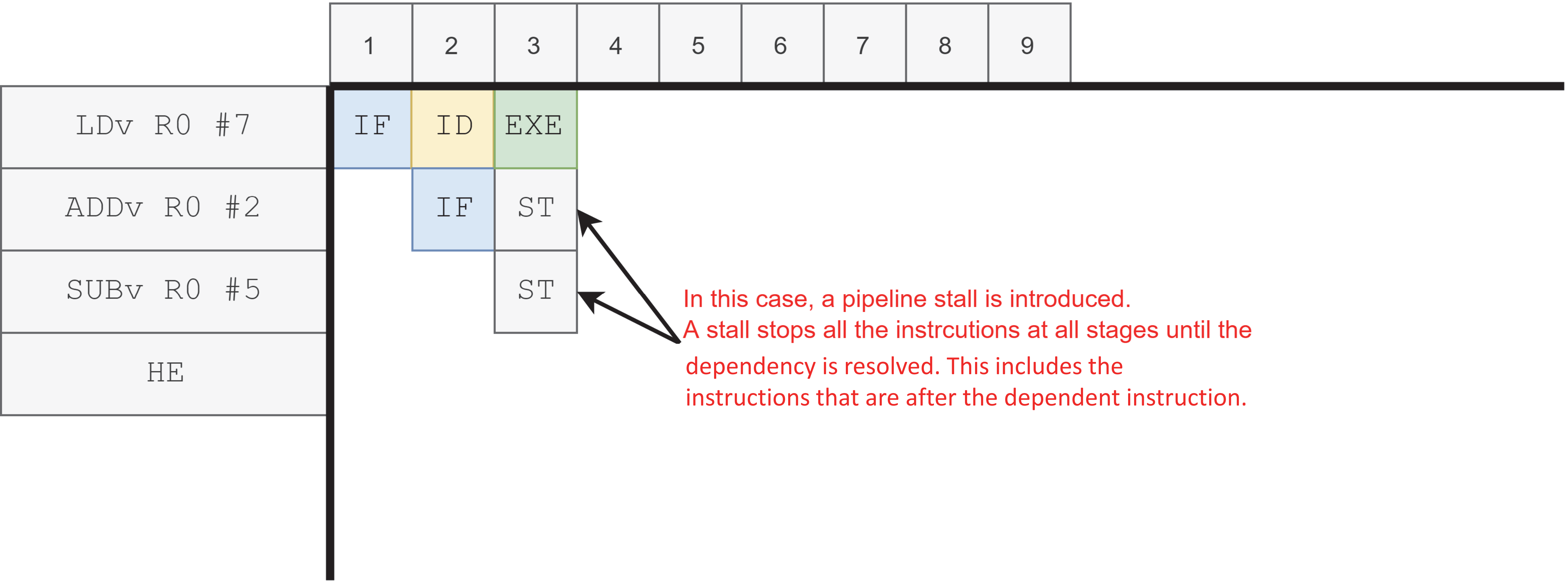
- Lets try to run this program here



At this point, the LDv instruction has not yet completed its operation on R0. Meaning that the ADDv instruction can not start executing until the data is actually written into the R0. The decoding of the ADDv instruction needs to have the data from the LDv instruction, meaning that we also can not decode the ADDv instruction either.
Since the ADDv uses data loaded by LDv, ADDv is dependent on LDv. This is called "instruction dependency".
The ADDv instrcution can not be executed until the data from LDv is written into the R0 register.

- Lets try to run this program here

TALTECH TALLINN UNIVERSITY OF TECHNOLOGY

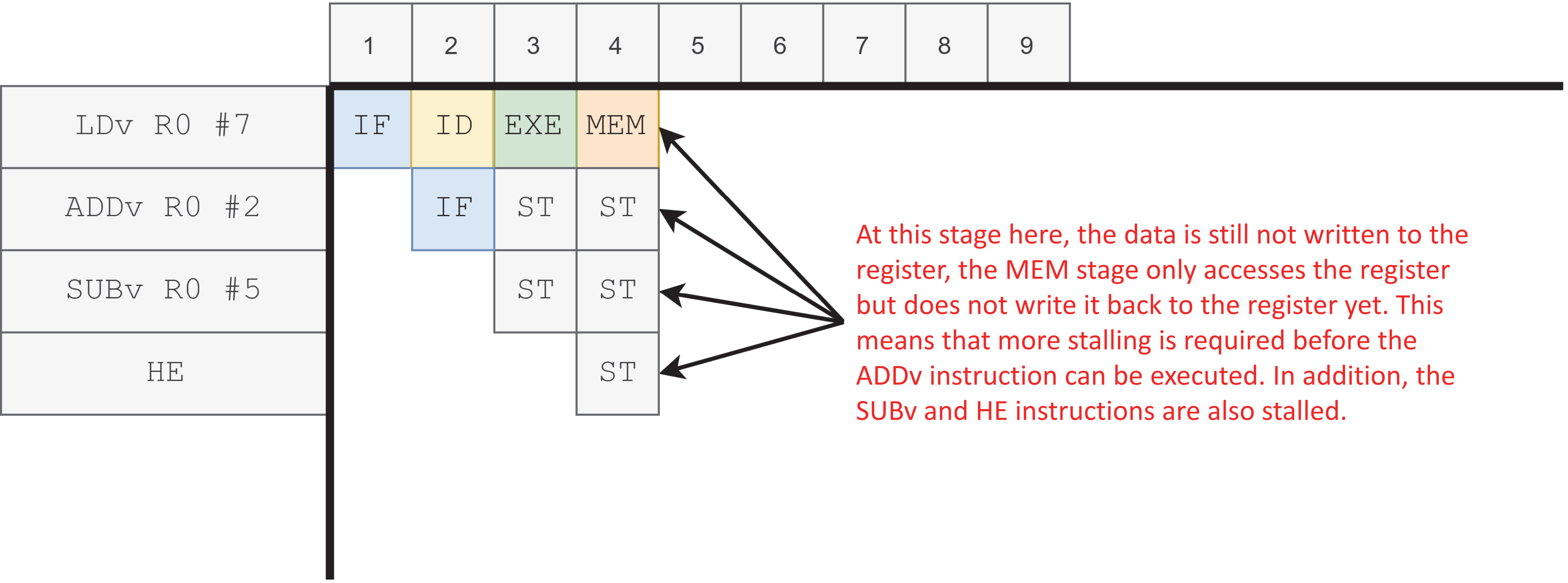


	1	2	3	4	5	6	7	8	9
LDv R0 #7	IF	ID	EXE						
ADDv R0 #2		IF	ST						
SUBv R0 #5			ST						
HE									

In this case, a pipeline stall is introduced.
A stall stops all the instrcutions at all stages until the dependency is resolved. This includes the instructions that are after the dependent instruction.

- Lets try to run this program here

TALLINN UNIVERSITY OF TECHNOLOGY



	1	2	3	4	5	6	7	8	9
LDv R0 #7	IF	ID	EXE	MEM					
ADDv R0 #2		IF	ST	ST					
SUBv R0 #5			ST	ST					
HE				ST					

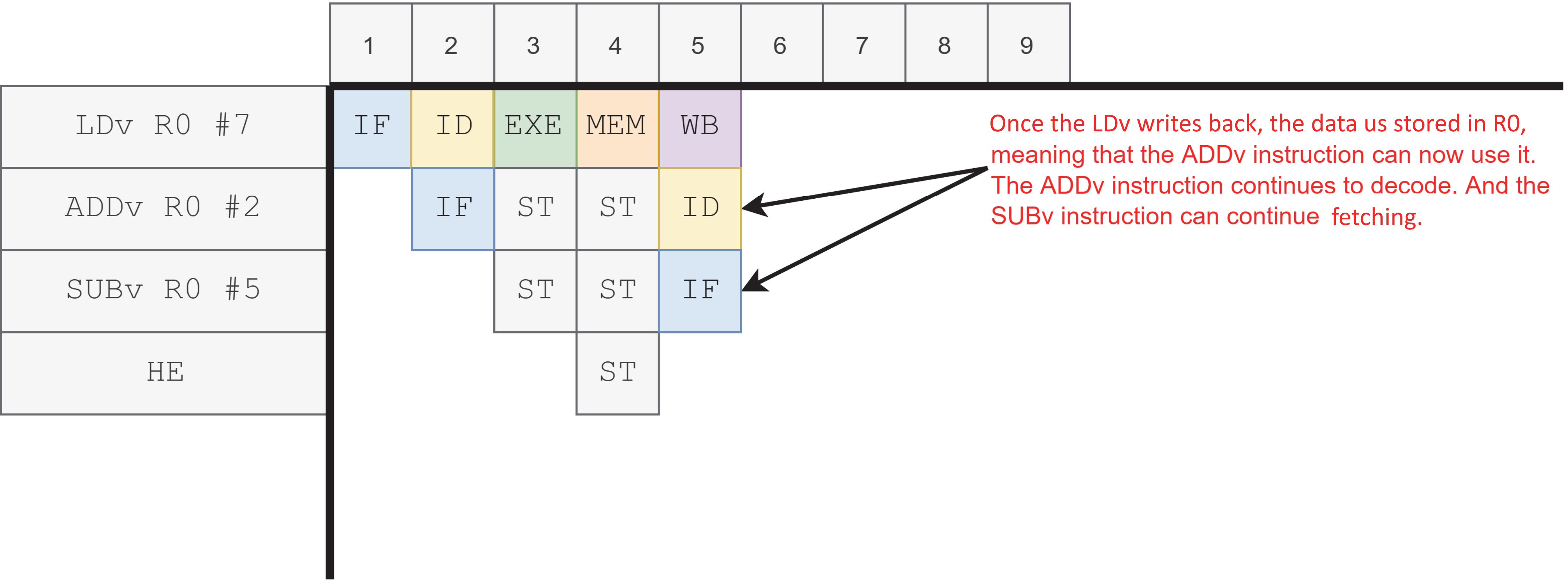
At this stage here, the data is still not written to the register, the MEM stage only accesses the register but does not write it back to the register yet. This means that more stalling is required before the ADDv instruction can be executed. In addition, the SUBv and HE instructions are also stalled.

- Lets try to run this program here

	1	2	3	4	5	6	7	8	9
LDv R0 #7	IF	ID	EXE	MEM					
ADDv R0 #2		IF	ST	ST					
SUBv R0 #5			ST	ST					
HE				ST					

This type of stall is called a DATA HAZARD. It occurs when an instruction tries to access data that is not yet in the register file.

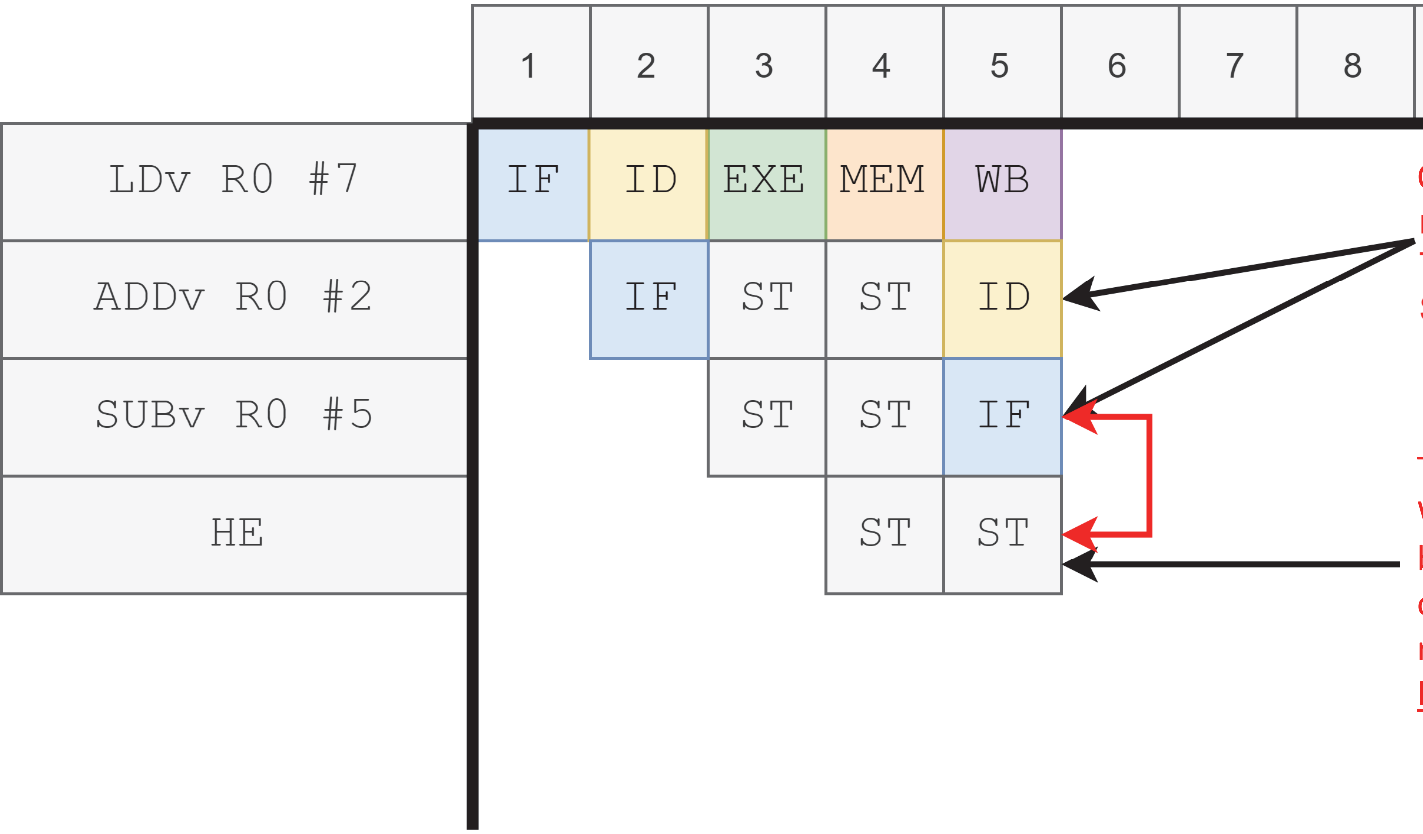
- Lets try to run this program here



	1	2	3	4	5	6	7	8	9
LDv R0 #7	IF	ID	EXE	MEM	WB				
ADDv R0 #2		IF	ST	ST	ID				
SUBv R0 #5			ST	ST	IF				
HE				ST					

Once the LDv writes back, the data us stored in R0, meaning that the ADDv instruction can now use it. The ADDv instruction continues to decode. And the SUBv instruction can continue fetching.

- Lets try to run this program here



	1	2	3	4	5	6	7	8	9
LDv R0 #7	IF	ID	EXE	MEM	WB				
ADDv R0 #2		IF	ST	ST	ID				
SUBv R0 #5			ST	ST	IF				
HE				ST	ST				

Once the LDv writes back, the data us stored in R0, meaning that the ADDv instruction can now use it. The ADDv instruction continues to decode. And the SUBv instruction can continue fetching.

The HE instruction is stalled again. This is because We can not fetch it while the SUBv instruction is being fetched. This is called a Structural Hazard. It occurs when two instructions try to access the same resource at the same time. The resource here is the Instruction Register.

- Lets try to run this program here

	1	2	3	4	5	6	7	8	9
LDv R0 #7	IF	ID	EXE	MEM	WB				
ADDv R0 #2		IF	ST	ST	ID	EXE			
SUBv R0 #5			ST	ST	IF	ID			
HE				ST	ST	IF			

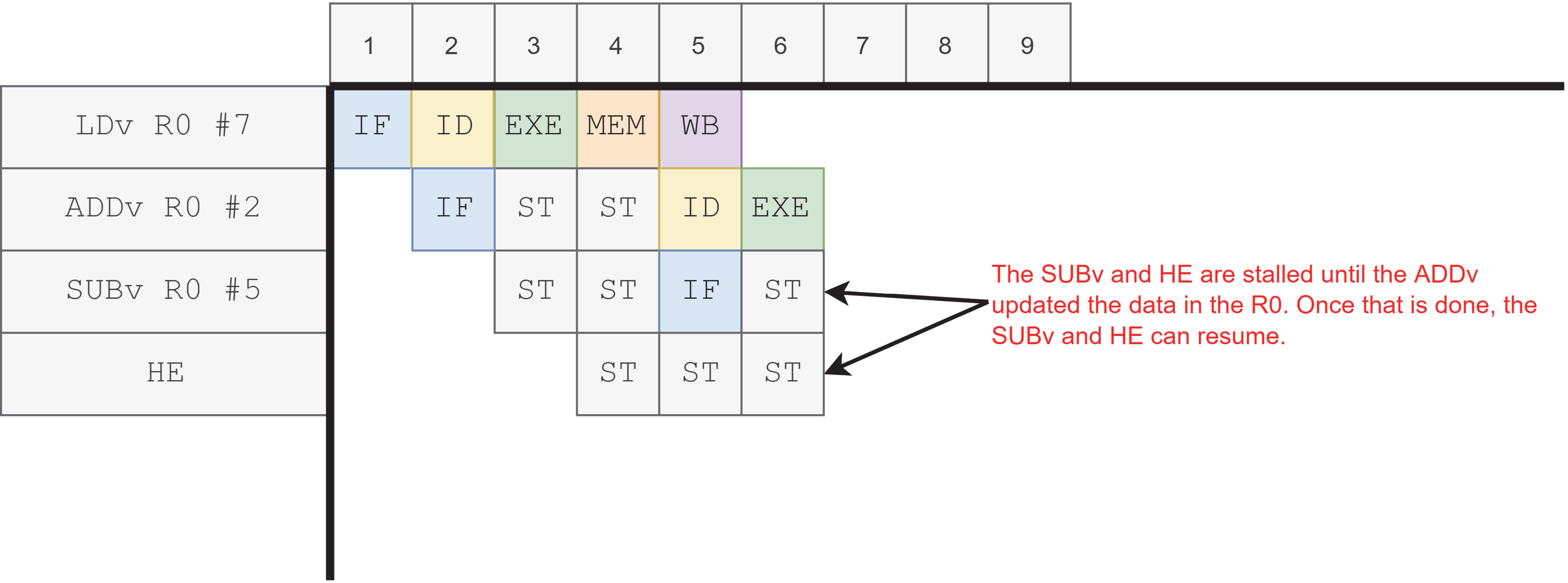
Can these stages be completed? How can this happen?

- Lets try to run this program here

	1	2	3	4	5	6	7	8	9
LDv R0 #7	IF	ID	EXE	MEM	WB				
ADDv R0 #2		IF	ST	ST	ID	EXE			
SUBv R0 #5			ST	ST	IF	ID			
HE				ST	ST	IF			

<u>Actually, it can not!</u>
Since the ADDv and the SUBv instructions try to perform operation on the same register (R0), the SUBv instruction will need ADDv to complete first before it can start executing.
<u>A stall is needed.</u>

- Lets try to run this program here



	1	2	3	4	5	6	7	8	9
LDv R0 #7	IF	ID	EXE	MEM	WB				
ADDv R0 #2		IF	ST	ST	ID	EXE			
SUBv R0 #5			ST	ST	IF	ST			
HE				ST	ST	ST			

The SUBv and HE are stalled until the ADDv updated the data in the R0. Once that is done, the SUBv and HE can resume.

- Lets try to run this program here

TALLINN UNIVERSITY OF TECHNOLOGY

	1	2	3	4	5	6	7	8	9	10	11	12
LDv R0 #7	IF	ID	EXE	MEM	WB							
ADDv R0 #2		IF	ST	ST	ID	EXE	MEM	WB				
SUBv R0 #5			ST	ST	IF	ST	ST	ID	EXE	MEM	WB	
HE				ST	ST	ST	ST	IF	ID	EXE	MEM	WB

- Lets try to run this program here