Name: Prathamesh Sai **Student ID: 19314123**

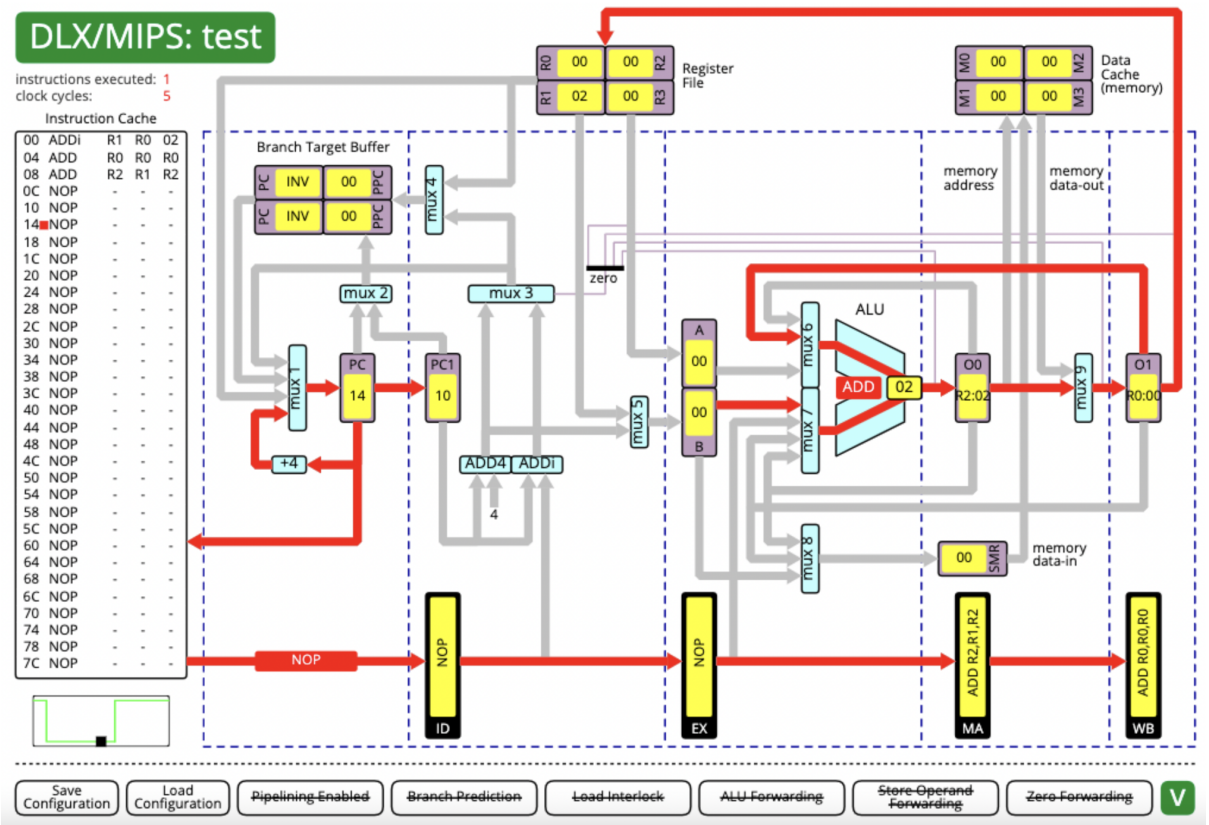
Computer Architecture II (CSU34021) Tutorial 4 Report

Question 1

1. O1 to Mux6

```
ADDi R1,R0,02
ADD  R0,R0,R0
ADD  R2,R1,R2
```

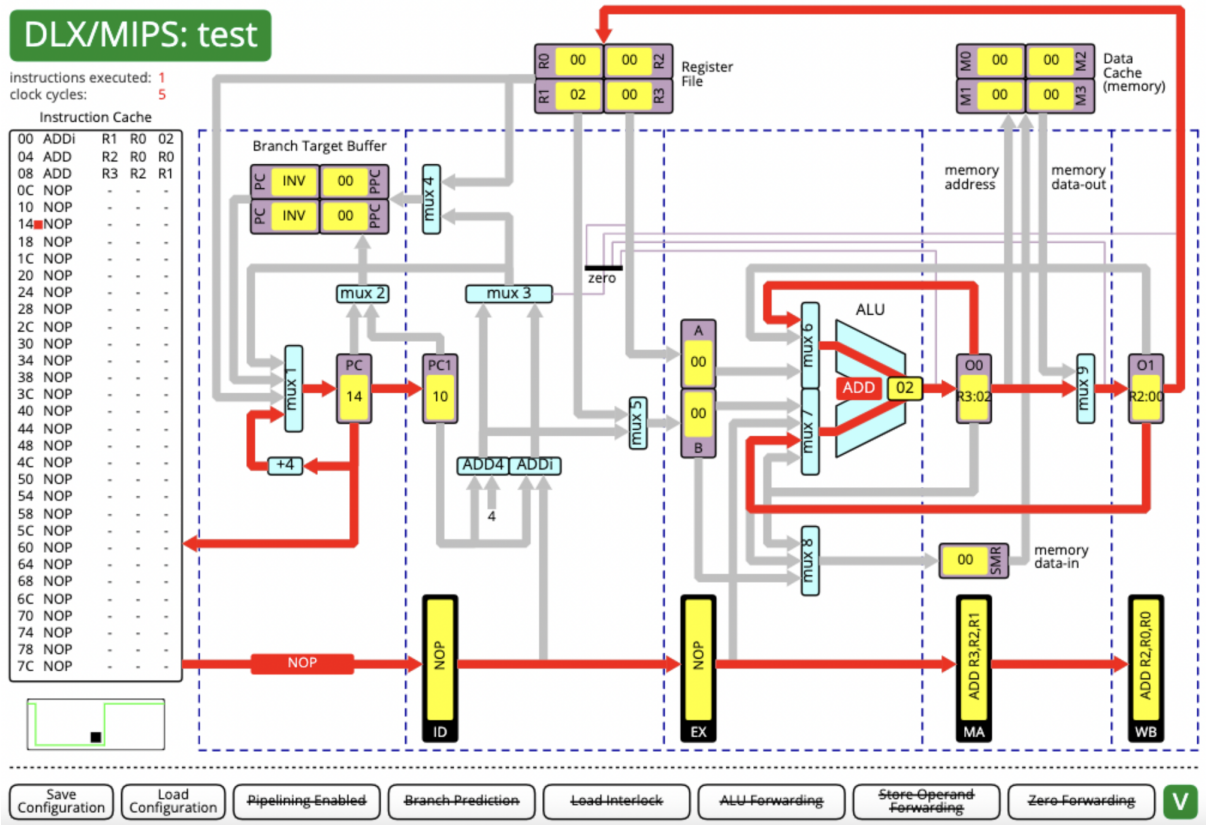
Screenshot:



2. O0 to Mux6 and O1 to Mux7 (simultaneously)

```
ADDi R1,R0,02
ADD  R2,R0,R0
ADD  R3,R2,R1
```

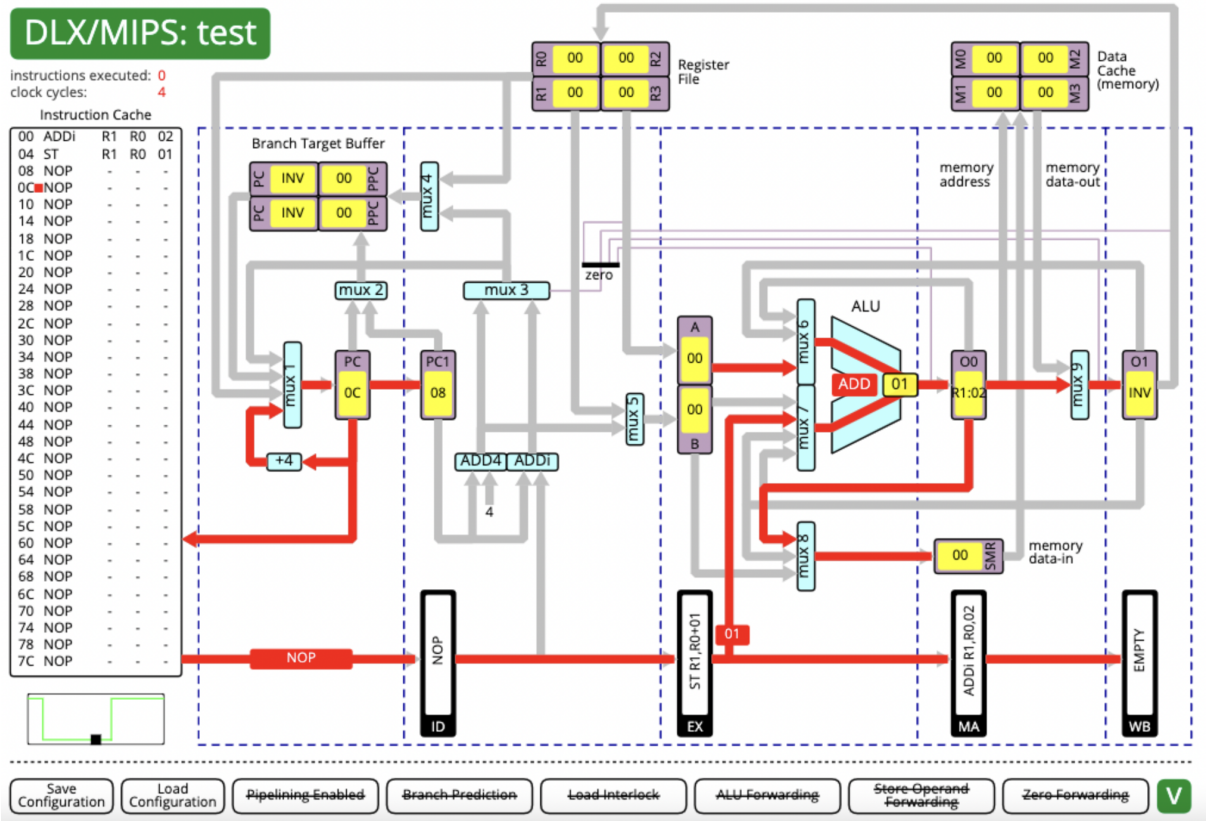
Screenshot:



3. O0 to Mux8

```
ADDi R1,R0,02
ST   R1,R0,R1
```

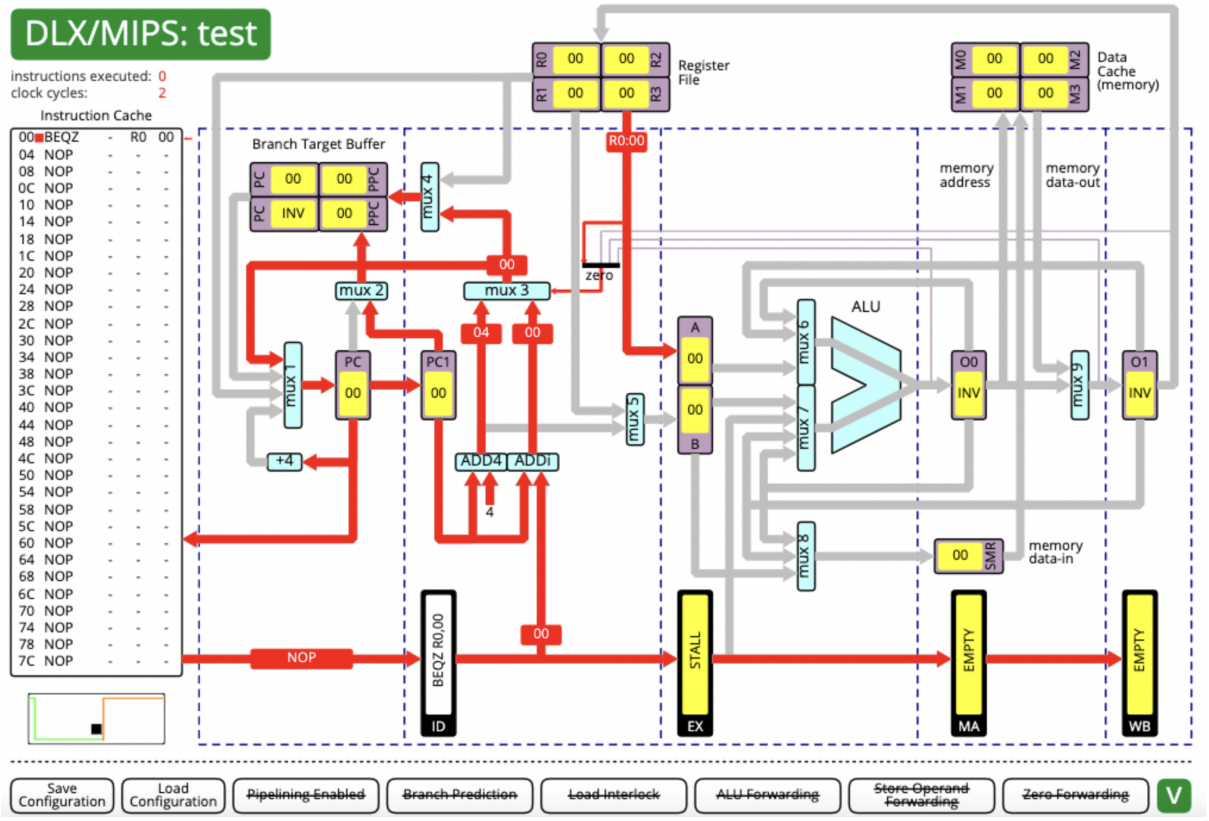
Screenshot:



4. ID to Mux3 to Mux1/Mux4

```
BQEZ -,R0,00
```

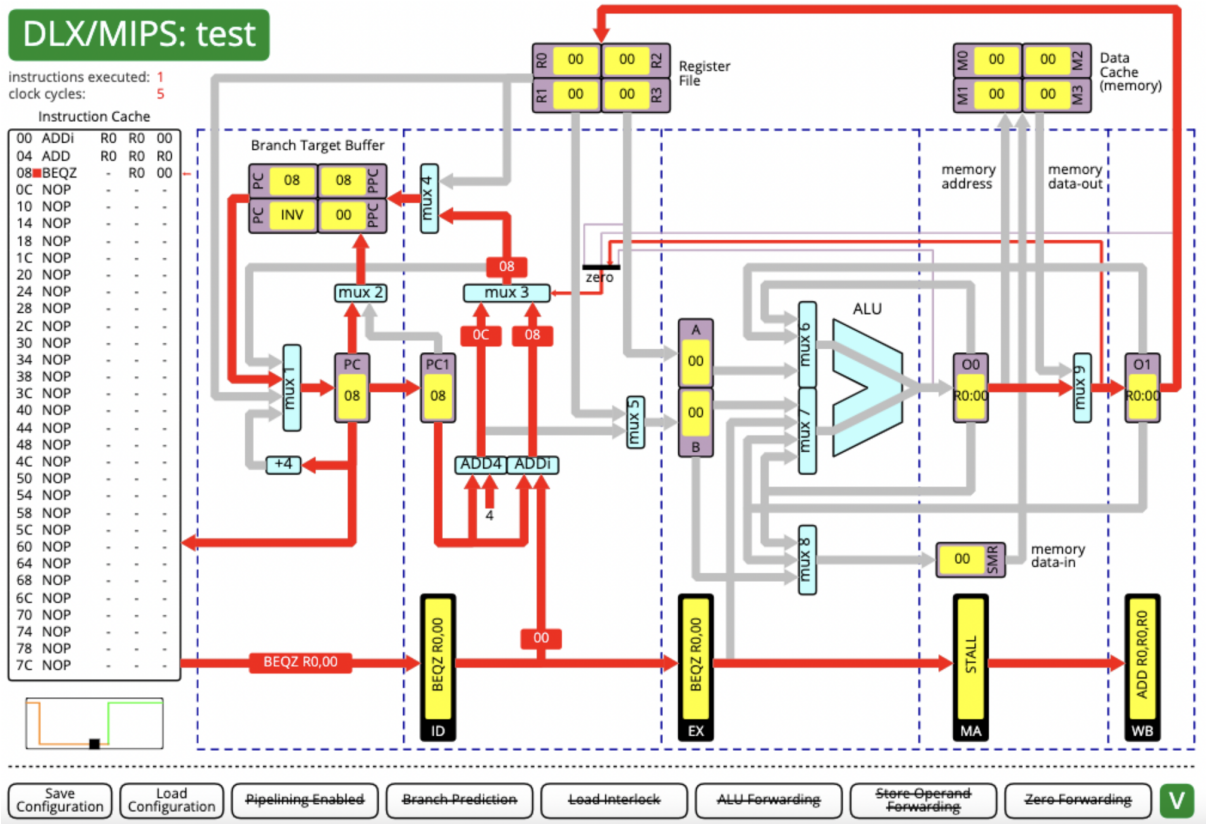
Screenshot:



5. Mux9 out to Zero detector

```
ADDi  R0,R0,00
ADD   R0,R0,R0
BQEZ  - ,R0,00
```

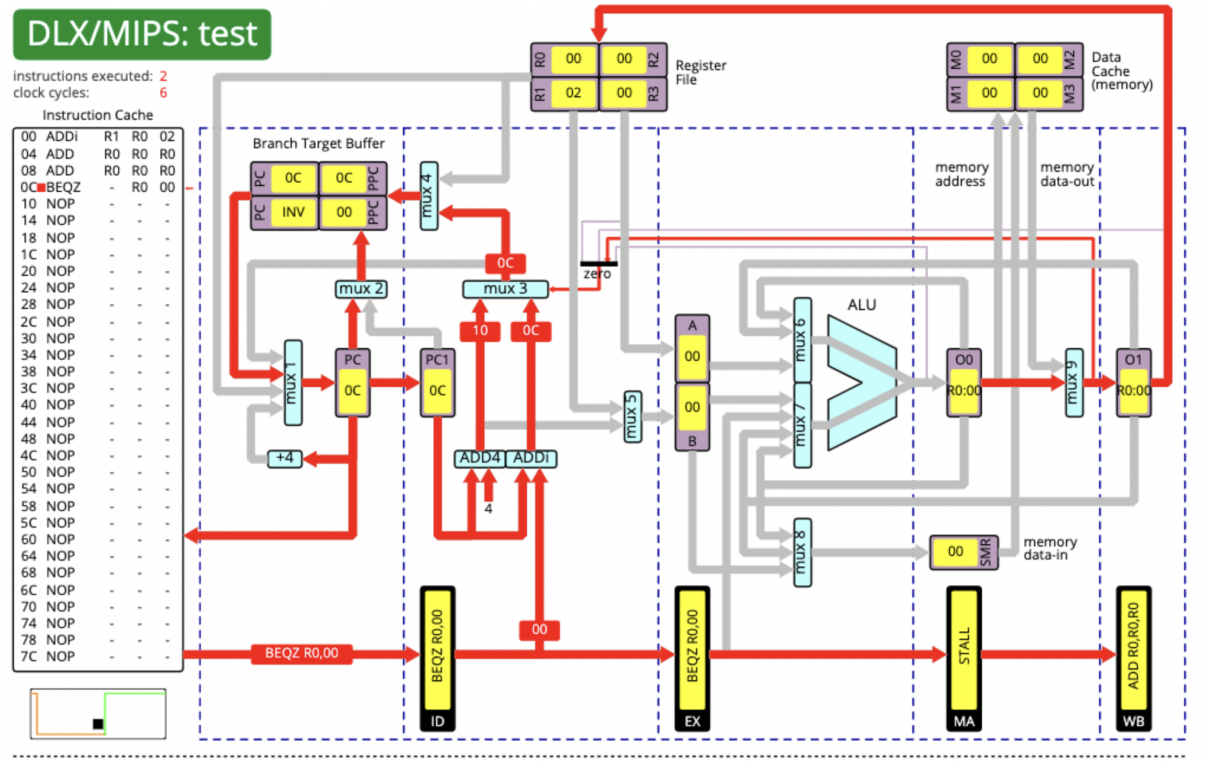
Screenshot:



6. Branch target buffer to Mux1

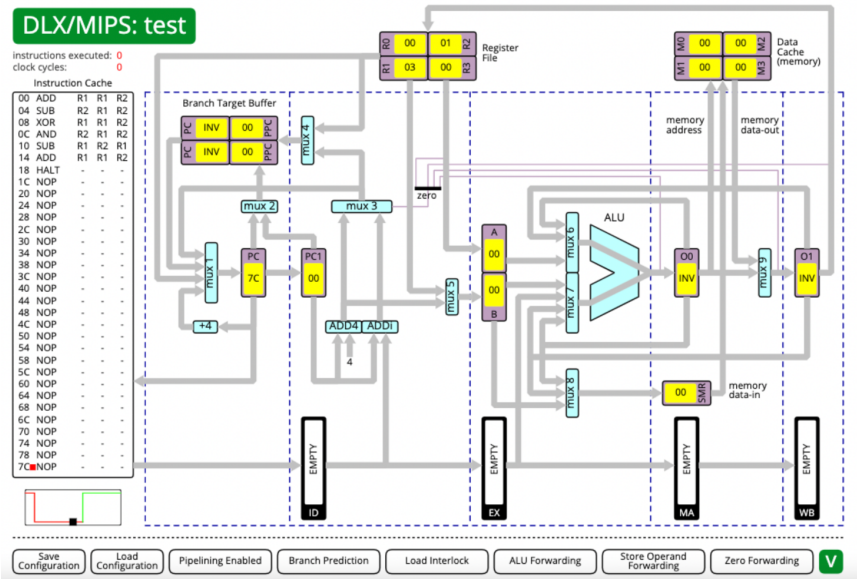
```
ADDi R1,R0,02
ADD  R0,R0,R0
ADD  R0,R0,R0
BQEZ - ,R0,00
```

Screenshot:



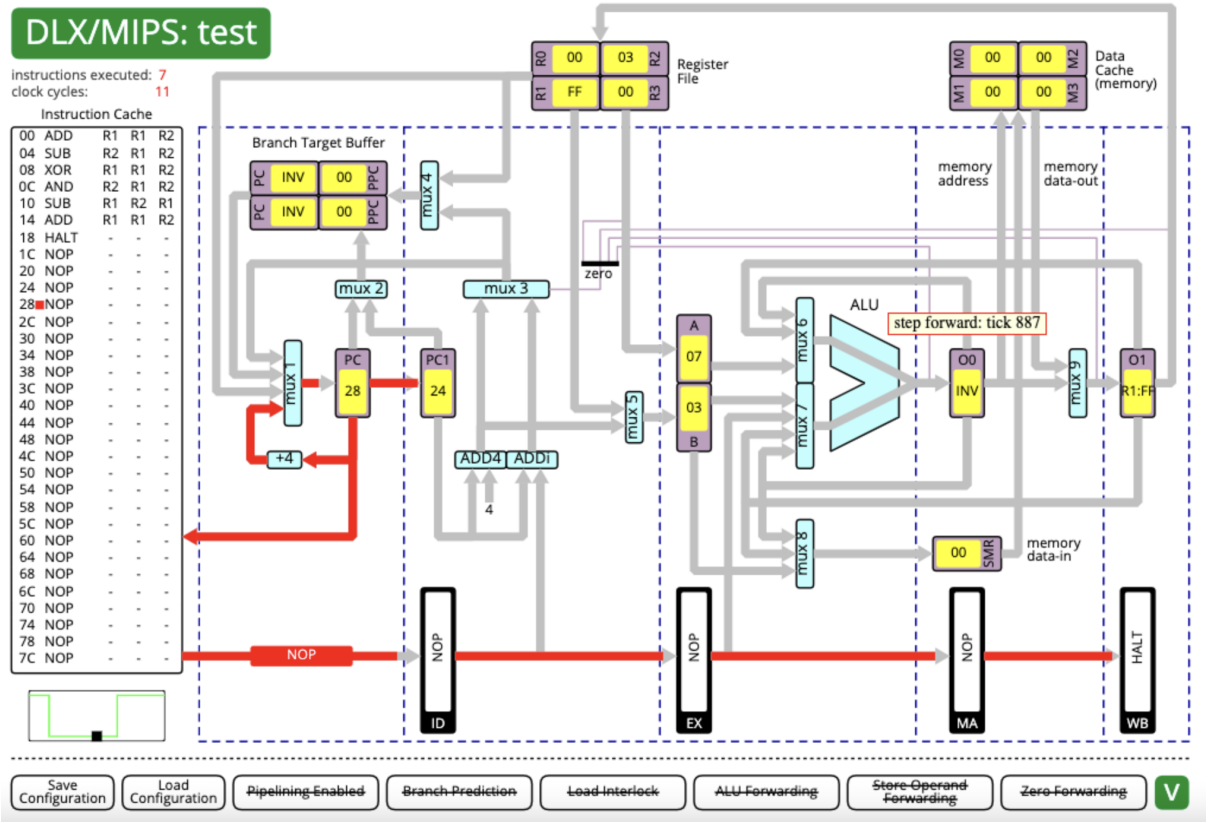
Name: Prathamesh Sai **Student ID: 19314123**

Question 2



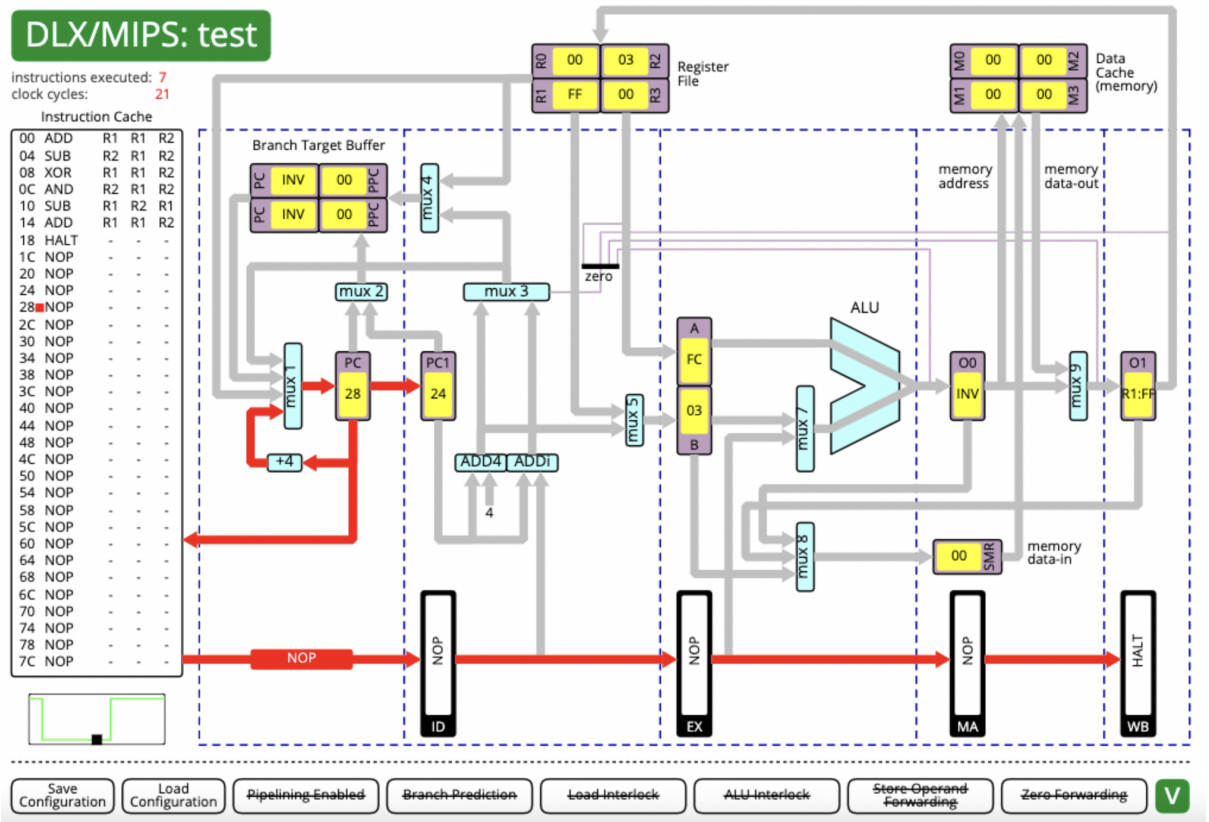
1. **ALU Forwarding is enabled**
   Resulting value in R1: Hex 0xFF (Correct)
   Resulting value in R2: Hex 0x03 (Correct)
   Number of clock cycles needed to execute: 11

Screenshot:



2. **ALU forwarding is disabled with CPU data dependency interlocks enabled**
   Resulting value in R1: Hex 0xFF (Correct)
   Resulting value in R2: Hex 0x03 (Correct)
   Number of clock cycles needed to execute: 21

Screenshot:



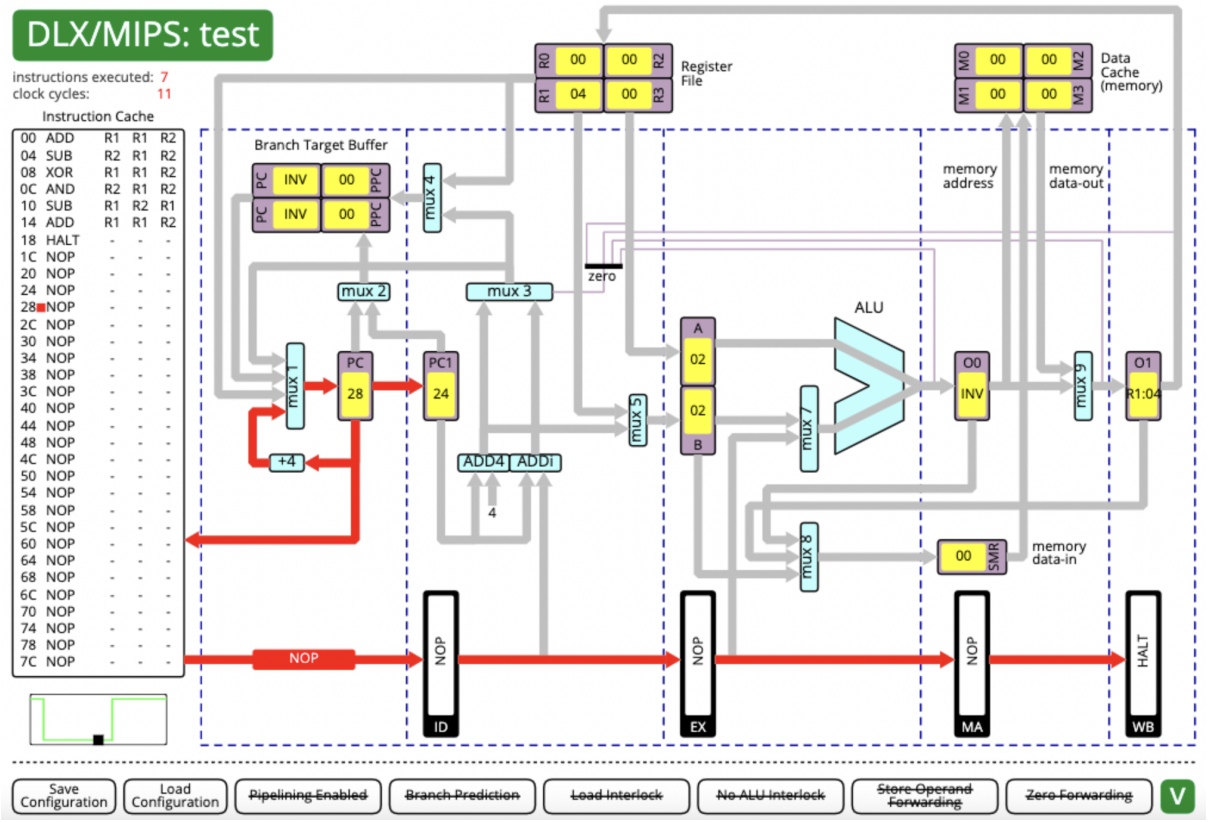
3. **ALU forwarding and CPU data dependency interlocks disabled**

Resulting value in R1: Hex 0x04 (Incorrect)

Resulting value in R2: Hex 0x00 (Incorrect)

Number of clock cycles needed to execute: 11

Screenshot:



Explain in detail why the results and number of clock cycles are different.

The results in part 1 and part 2 are the same correct answer. They are both correct because we use the updated versions of each register while giving appropriate time for any updates to propagate. This results in both part 1 and part 2 having the same result, which is correct.

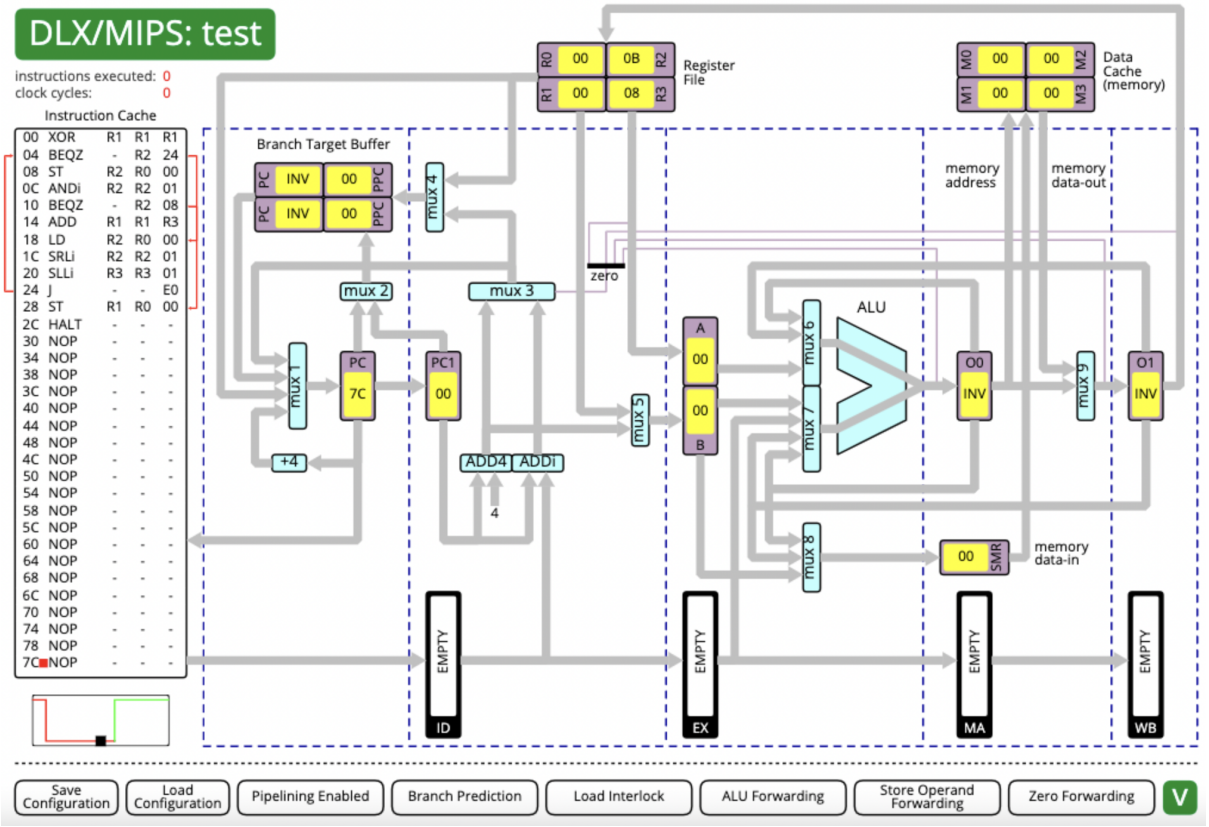
With ALU forwarding and CPU data dependency interlocks disabled, it takes the same number of cycles as part 1, but this time we do not care if the updates have been propagated or not. This makes us get the wrong answer in part 3, since we executed ALU operations although the results of the previous ALU operation did not have enough time to update the registers.

The number of clock cycles between part 1 and part 2 are different because in part 1, we are using ALU forwarding. Therefore, with the help of O0 and O1, we can access the result of the ALU operation immediately after it is done. Hence, even before the values in the following instruction is written back, we are still able to get them. This in turn will prevent pipeline stalls/data hazards.

When using ALU interlock in part 2, we don't have the same functionality. Therefore, we must wait for the ALU operation to be written back before we are able to continue to the next operation every time an instruction is executed. Hence, we use more cycles to execute the same instructions as in part 1 because of the pipeline stalls that occur as we wait for the write back to finish.

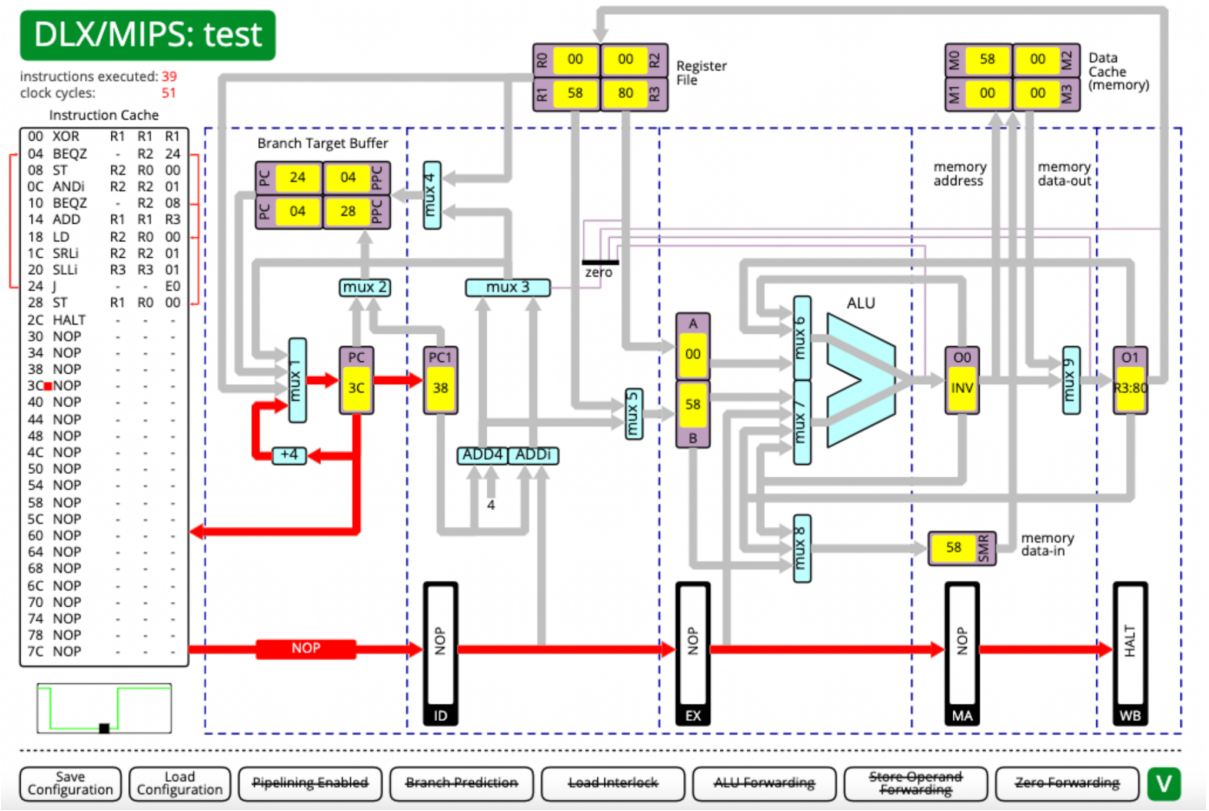
Question 3

1. How many instructions are executed and how many clock cycles are needed to execute this program until it halts? Explain in detail why these two numbers are not equal and account for each stall cycle.



How many instructions are executed: 39
how many clock cycles are needed: 51

Screenshot:



Explain in detail why these two numbers are not equal and account for each stall cycle:

Pipeline stalls result in these two numbers not being equal. We have stalls at the beginning of execution before the pipeline has been filled, and the number of stalls is proportional to the number of stages in the pipeline. We have 4 stages in this example, so 4 stalls occur here.

Another reason for stalls is when the branch instructions execute and the branch prediction is incorrect. In our program this happens a total of 4 times between all branch instructions, hence this is an extra 4 pipeline stalls.

Lastly, there is a data dependency between LD R2, R0, 00 and SRLi R2, R2, 01 since the load instruction will take two cycles to put a value into O1. These instructions are executed a total of 4 times meaning that accounts for 4 extra stalls.

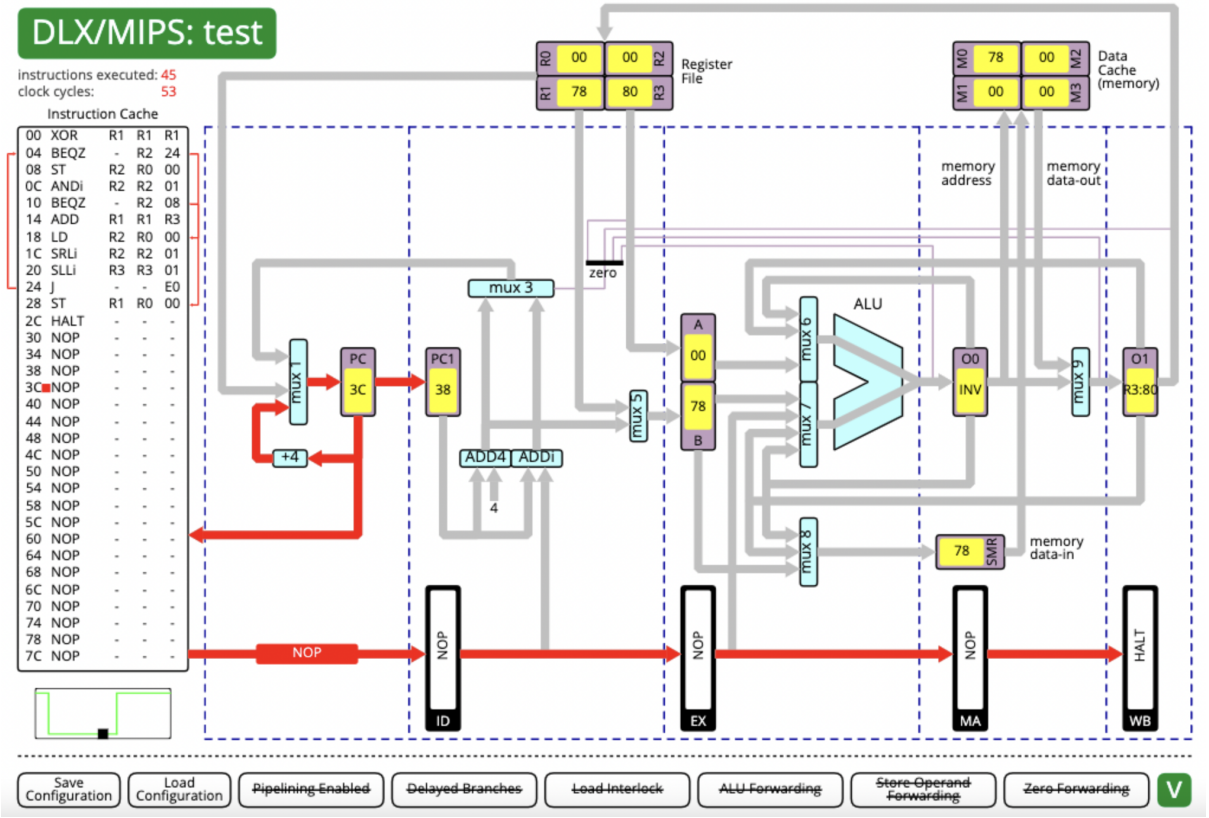
In total, there are (4+4+4) 12 stalls, which makes sense because the difference between the cycles and instructions executed is also 12 (51-39=12), and we can conclude that this accounts for the difference between the instructions and clock cycles.

2. **What changes do you need to make to the program so that it is still a valid multiplication program?:**

The program currently is not a valid multiplication program because we are using delayed branching. This means that instruction directly after the branch is executed anyway, we can take that into account and make use of it, and the instruction following the branch is said to be in the delay slot. To make the program a valid multiplication program, we need to add NOP instructions such as ADD R0,R0,R0 right after every BEQZ instruction. This means that after every branch, the instruction directly after is executed anyways, but we do not break the logic of the program by doing so, as we simply executed a dummy instruction. This will make the program a valid multiplication program.

How many cycles are now needed to execute this program until it halts: 53

Screenshot:



Explain in detail why this number differs from your answer to part (i).

This is because we are now using delayed branching and not using branching prediction. We have more instructions executed with this than with branch prediction. This is because now every time we hit a branch instruction, it will always cause the pipeline to stall unless the program just continues if the branch is not taken. With delayed branching, we define that the instruction directly after the branch is executed anyway, we can take that into account and make use of it, and the instruction following the branch is said to be in the delay slot. In our example. We have an extra 2 stalls compared to part 1, which increased the number of cycles executed to 53.

3.

Identify a data dependency in the program which results in a necessary stalling of the pipeline.

There is a data dependency between LD R2, R0, 00 and SRLi R2, R2, 01 since the load instruction will take two cycles to put a value into O1. These instructions are executed a total of 6 times meaning that accounts for 6 extra stalls.

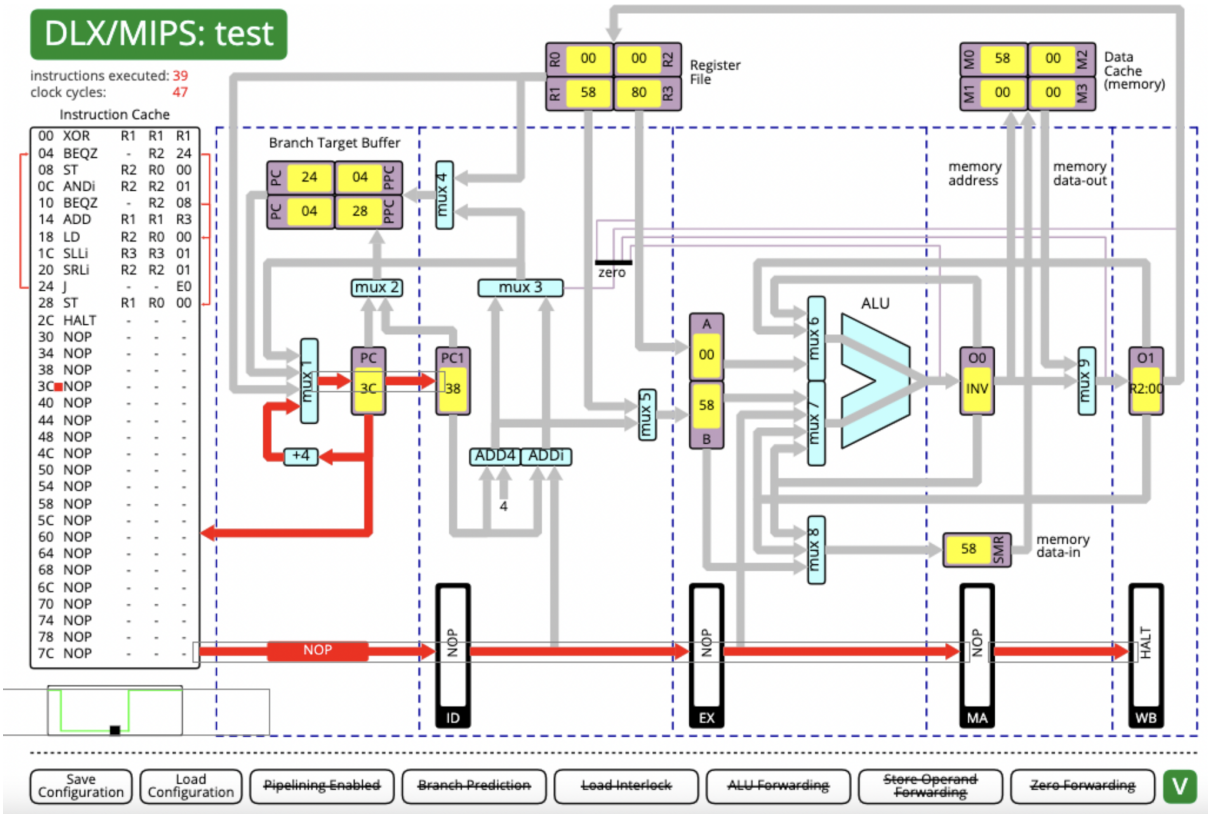
Remove the data dependency in a way that the pipeline is not stalled and correctness of the program stands

By swapping the shift instructions SRLi and SLLi, we can remove 6 data stalls (load hazards) that occur as a result of the data dependency between LD R2, R0, 00 and SRLi R2, R2, 01. This will therefore reduce the instructions executed and the clock cycles.

```
XOR  R1,R1,R1
BEQZ - ,R2,24
ST   R2,R0,00
ANDi R2,R2,01
BEQZ - ,R2,01
ADD  R1,R1,R3
LD   R2,R0,00
SLLi R3,R3,01
SRLi R2,R2,01
J    - , -,E0
ST   R1,R0,00
HALT - , -, -
```

How many clock cycles does the new program need for completion: 47

Screenshot:



What is the reason for a different number as compared to the original clock cycle count?:

The original cycle count was 53, but now the cycle count is 47. There is a difference of 6 cycles between them. This is because we remove 6 data stalls by swapping the shift instructions and removing the data dependency since SLLi R3, R3, 01 does not need R2 as mentioned above. This reduces the cycle count from 53-6-47, meaning our program can come to a halt in less cycles, meaning it is more efficient.