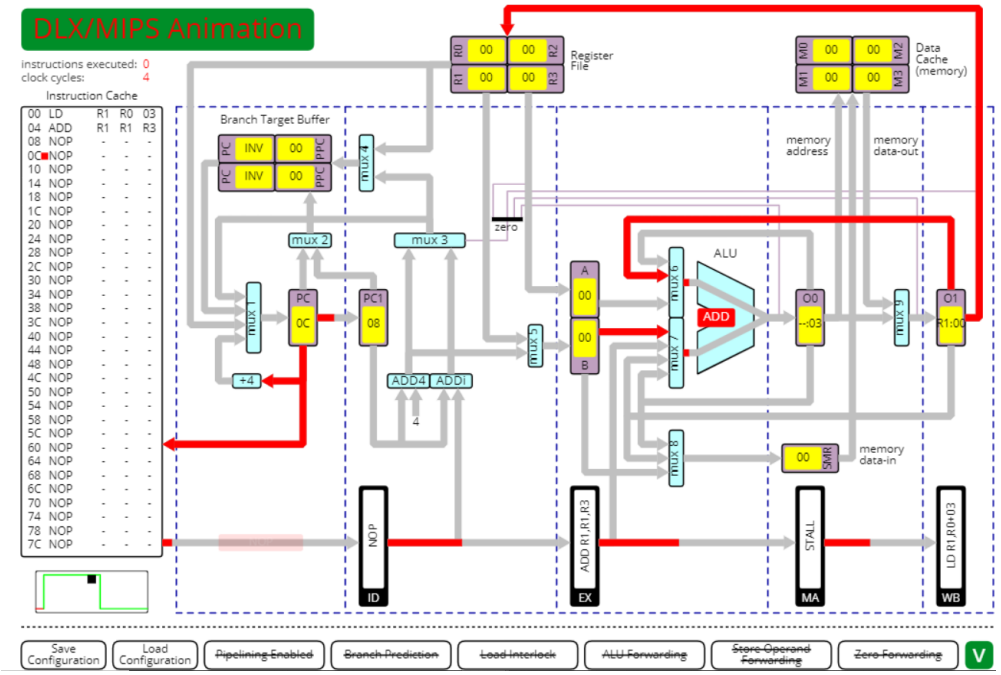
Tutorial 4

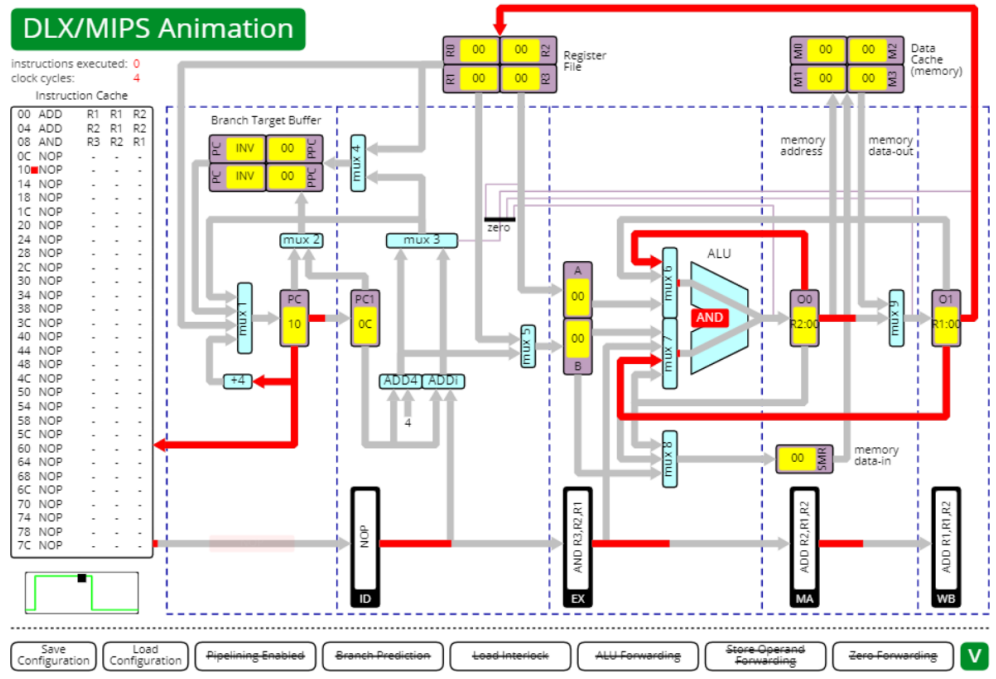
Question 1

1.



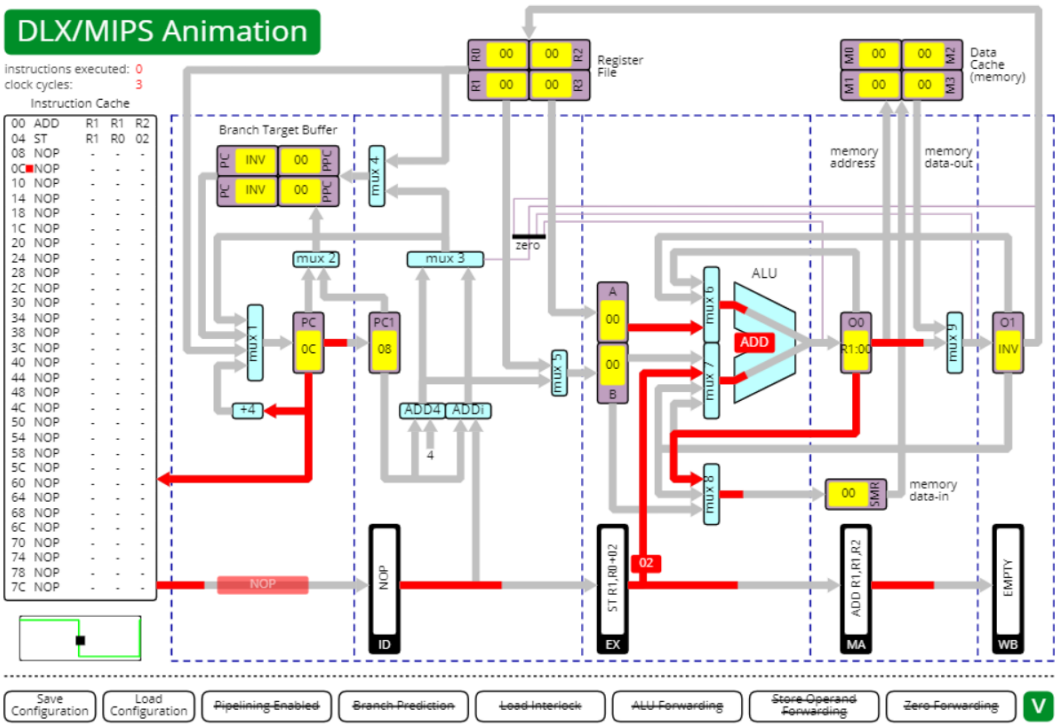
LD R1, R0, 03
ADD R1, R1, R3

2.



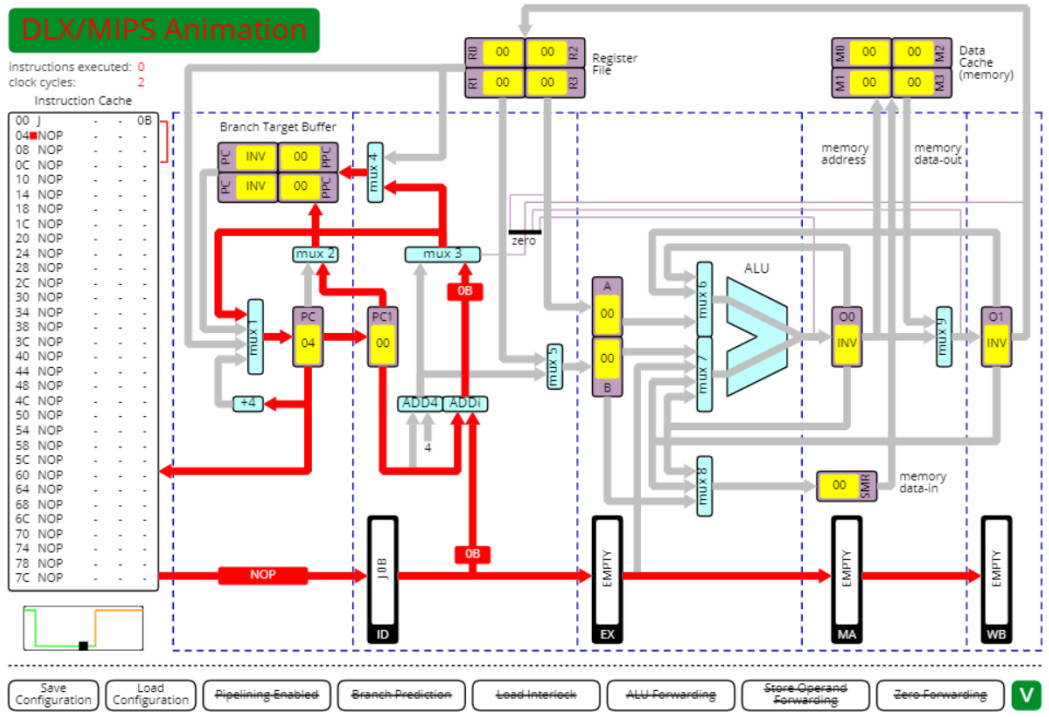
ADD R1, R1, R2
ADD R2, R1, R2
ADD R3, R2, R1

3.



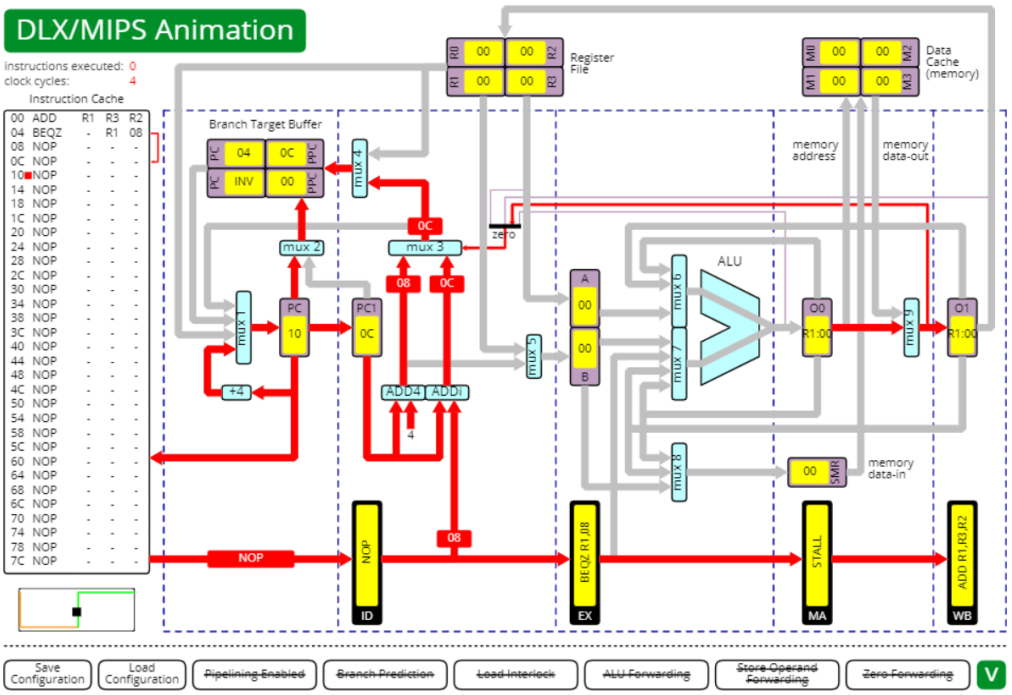
ADD R1, R1, R2
ST R1, R0, 02

4.



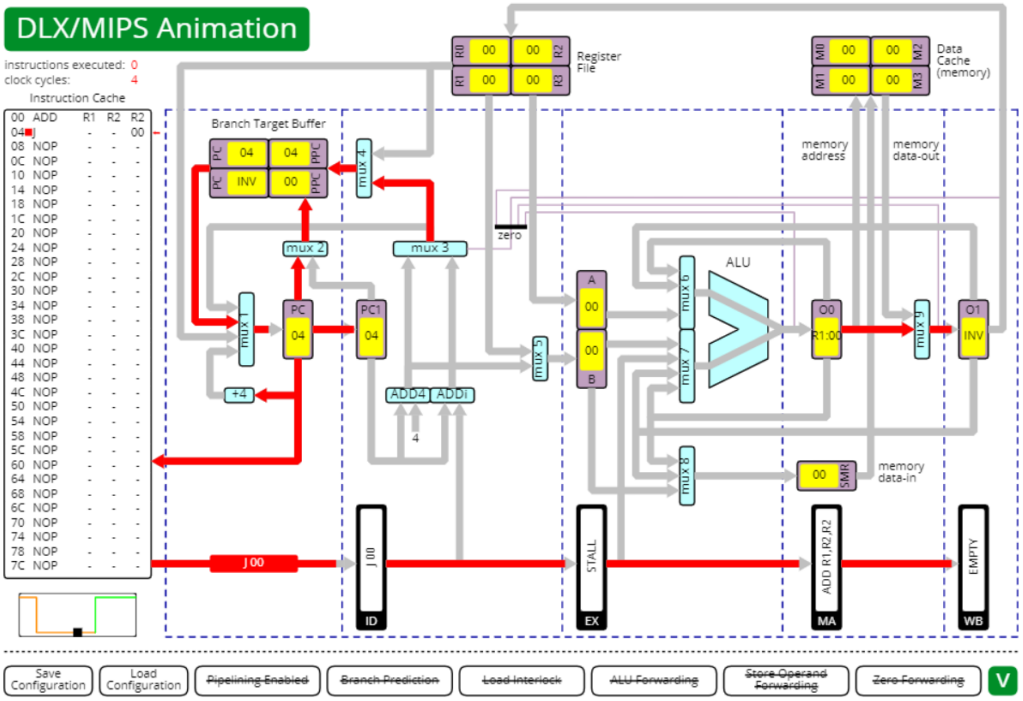
J 0B

5.



ADD R1, R3. R2
BEQZ R1, 08

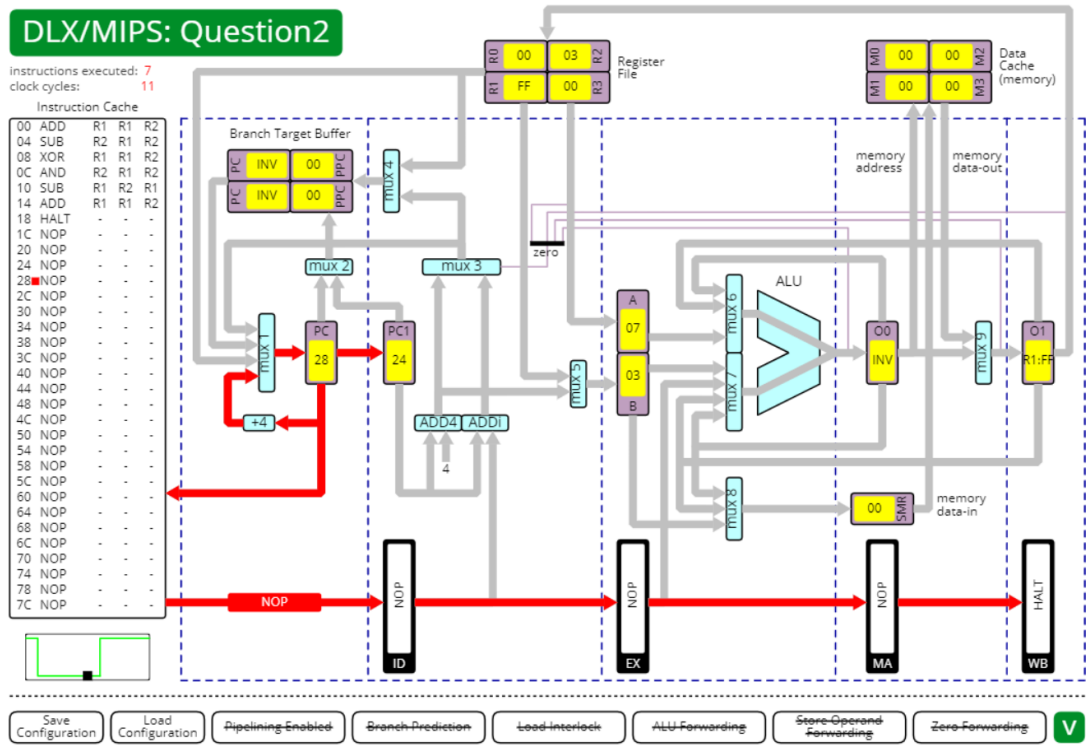
6.



ADD R1, R2, R2
J 00

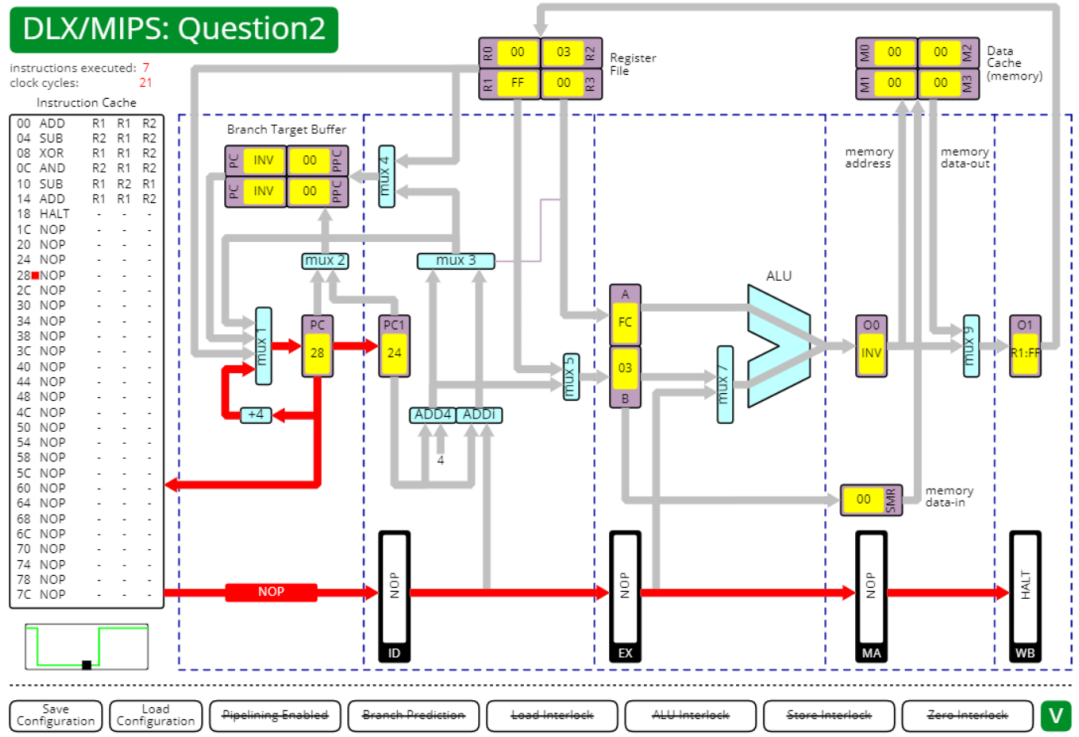
Question 2

(i) ALU forwarding enabled



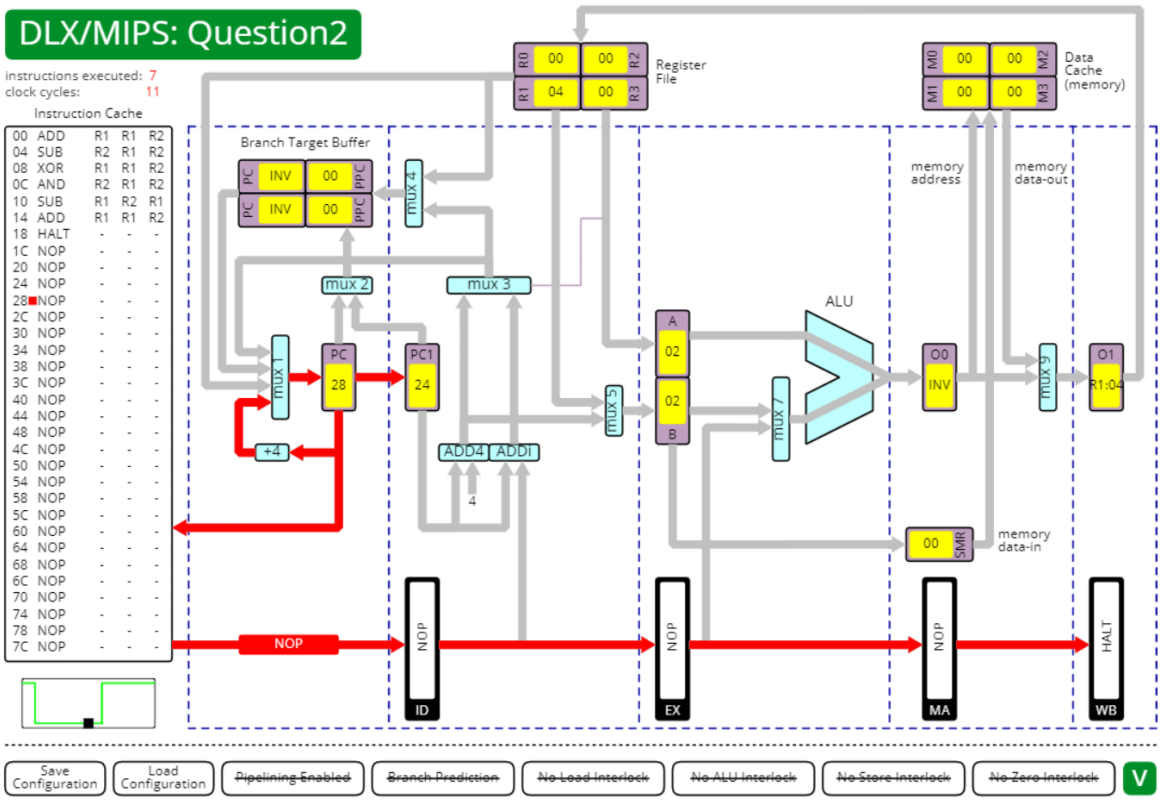
There were 11 clock cycles with a result of 0xFF

(ii) ALU forwarding disabled + CPU data dependency interlocks enabled



There were 21 clock cycles and a result of 0xFF

(iii) ALU forwarding + CPU data dependency interlocks disabled



There were 11 clock cycles with a result of 0x04

The results of the program and the clock cycles were different in these different situations due to the data dependencies and pipeline stalls.

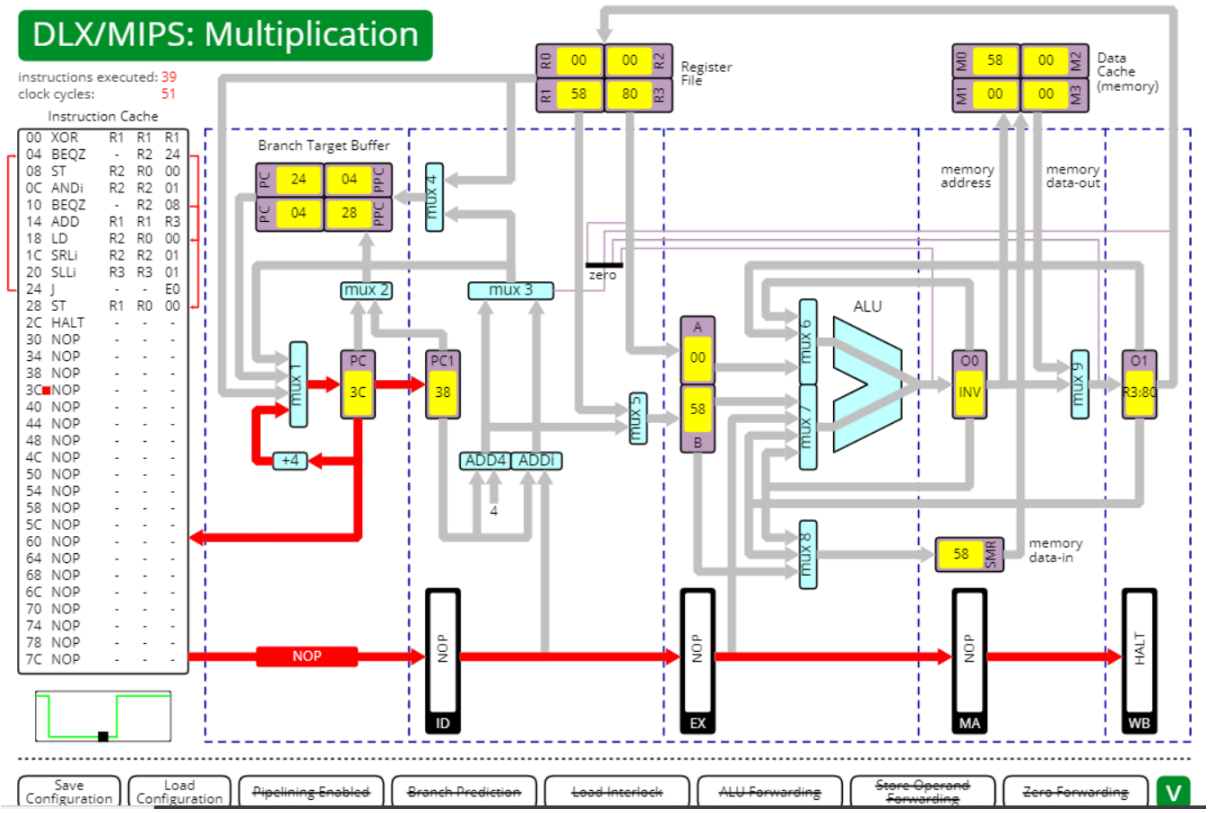
When ALU forwarding is enabled we can avoid some of the data dependency stalls that are required when accessing data from a register. Values that are used in a previous ALU operation can be reused in the next operation. This will keep the number of clock cycles low.

When the ALU forwarding is disabled we will have data dependencies. However, the answer will be correct due to the CPU data dependency interlocks being enabled. These interlocks will stall in between instructions that require data and so they will take more clock cycles to complete the program which is why there are more clock cycles required.

When both ALU forwarding and CPU data dependency interlocks are disabled the result will be incorrect. This is because there is no stalling in between instructions that require data so the processor will use whatever data is in the current register rather than “waiting” for the correct data to be stored in the register. The lack of stalling also means that there will be the same number of clock cycles as when the ALU forwarding is enabled.

Question 3

1.



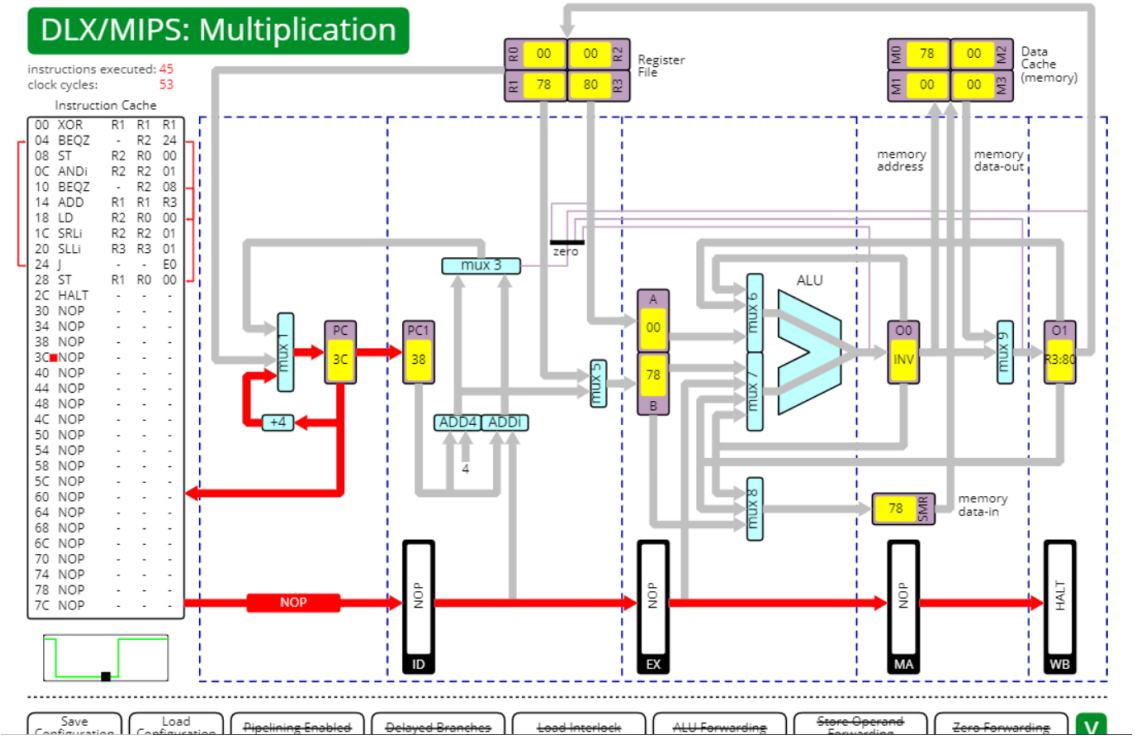
This is a snapshot of the program from question 1 after being run. The number of instructions executed was 39 as can be seen in the snapshot and the clock cycles needed was 51. For four instructions more than four clock cycles are required due to the need for them to filter through the pipeline. There is also a delay after certain instructions such as the load. This is due to the need for a delay after fetching any data. So the program stalls on the SRLi instruction due to this data dependency.

There is a stall after the first unconditional branch when it is executed the first time but after this there is stall as the branch prediction is set and the offset of the address does not need to be calculated.

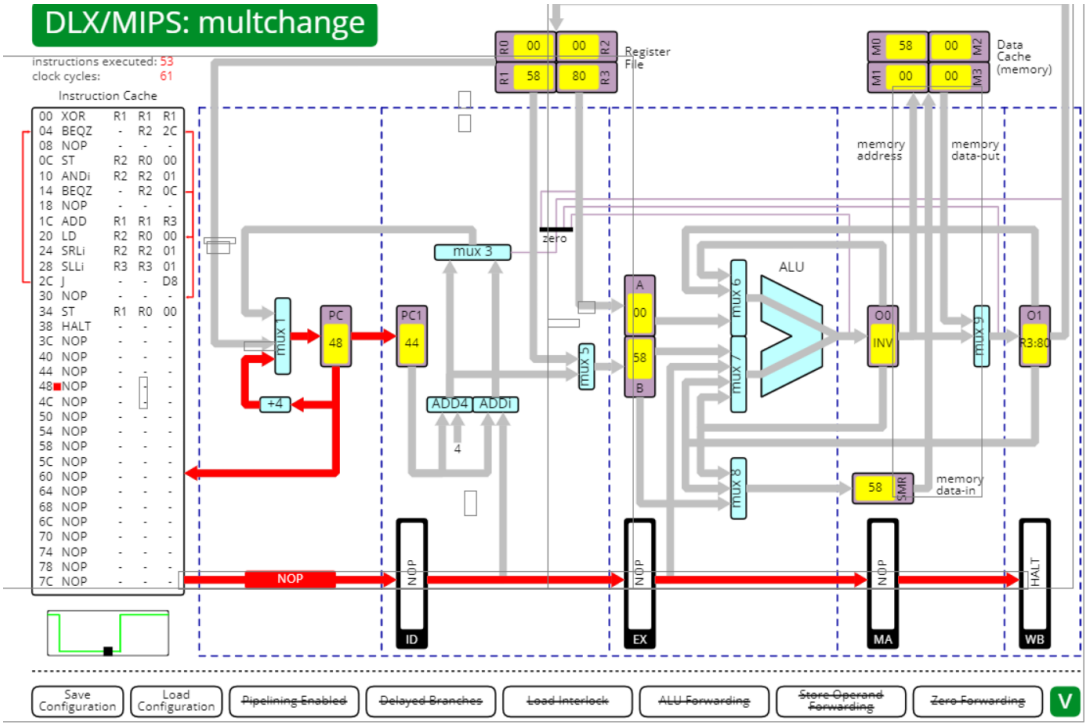
After each branch instruction, when the condition is met there is a stall to calculate the offset as well and this means there will be another delay. Such as when the loop exits at the first condition.

2.

Delayed Branches:

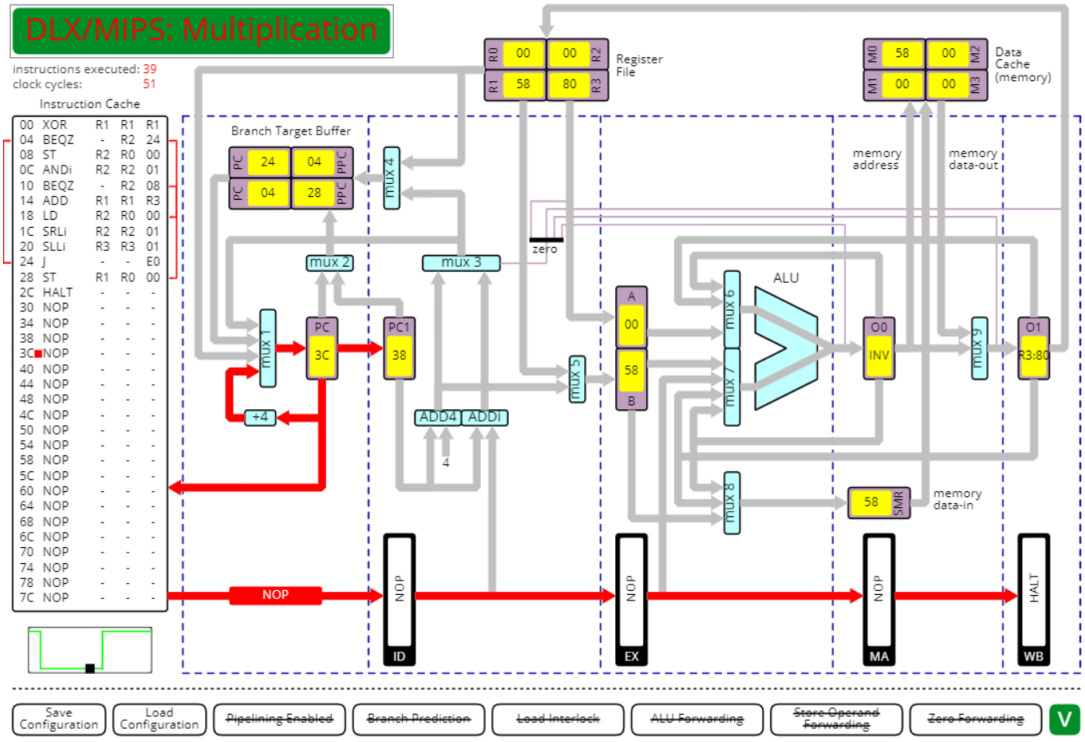


The delayed branching increased the clock cycles and the instructions executed. This meant that the data dependencies meant that the answer that was returned was incorrect. There needs to be a stall after the branches to ensure that the correct values are used in each instruction. Putting a nop instruction after each branch will ensure that the correct answer is gotten.



3.

This is the normal multiplication running.



There is a data dependency after using the load instruction where it loads from memory into R2. This can be removed by not having the next instruction rely on this value. To fix this you can switch the two shift statements and this will speed up the program and decrease the clock cycles by giving the load instruction time to load the correct data. You can see that the clock cycles executed was 47 in comparison to the 51 of the original program.