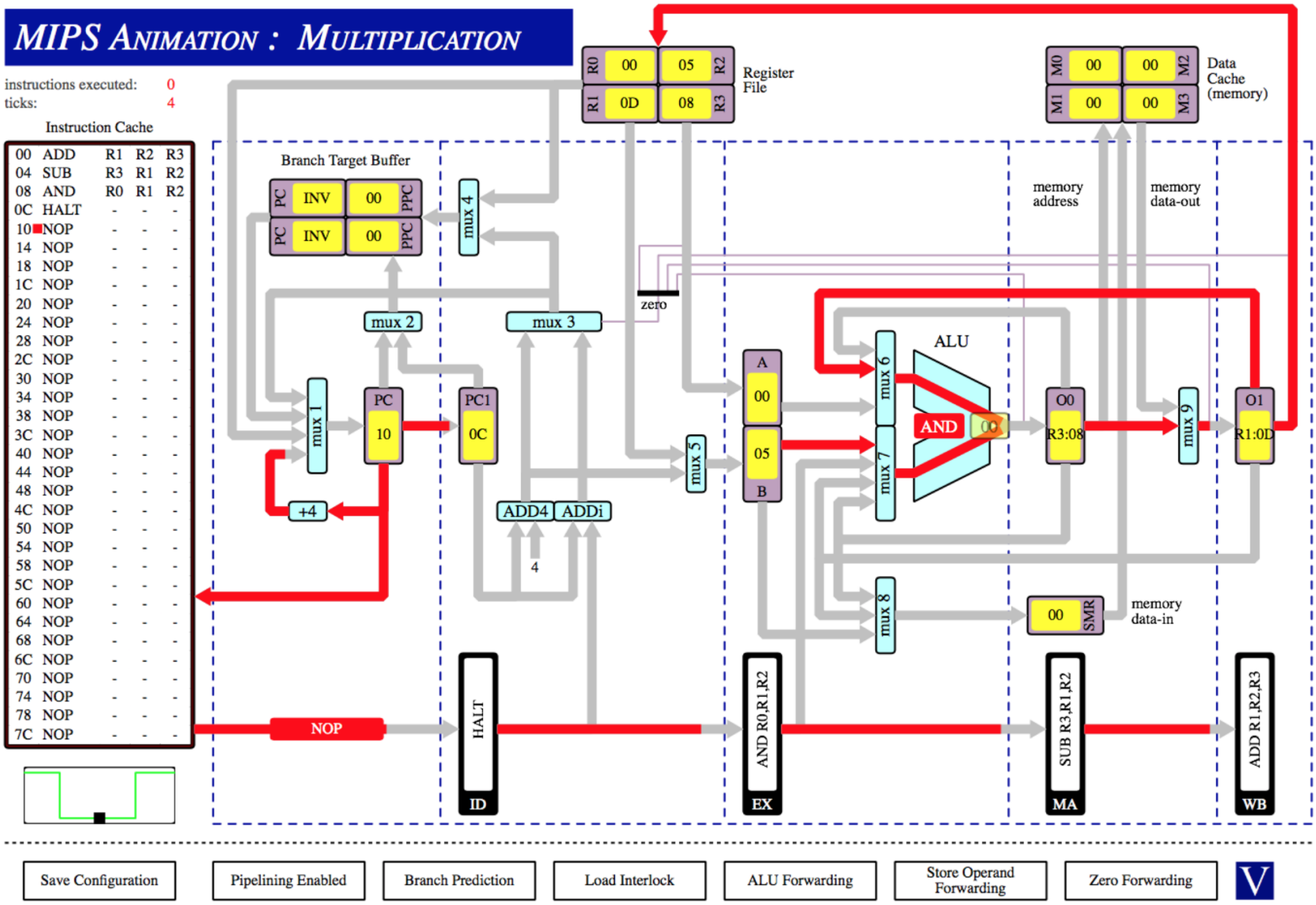
Q1

1. O1 to MUX6

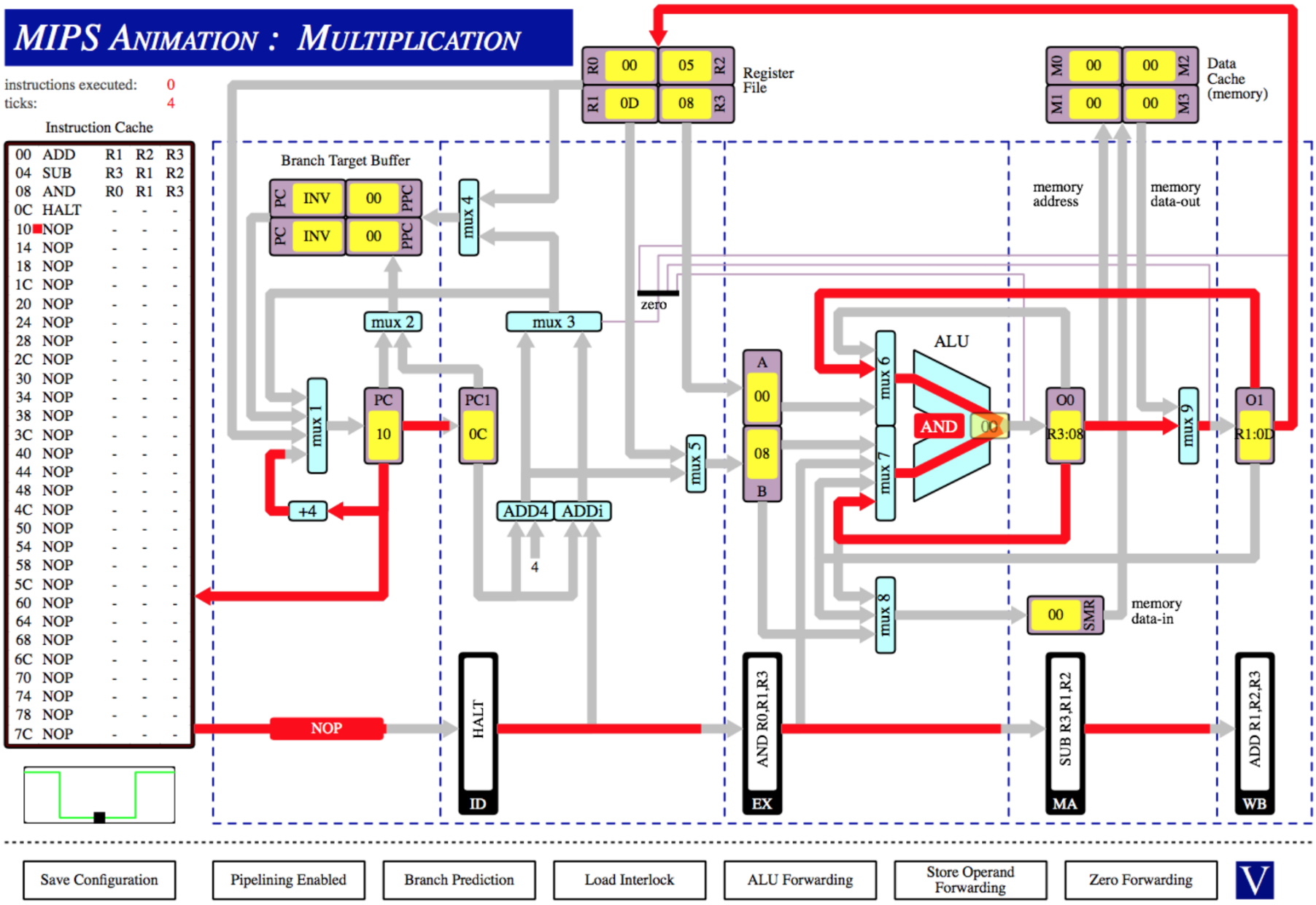
ADD R1,R2,R3
SUB R3,R1,R2
AND R0,R1,R2



Q1

2. O0 to MUX7 and O1 to MUX6 (simultaneously)

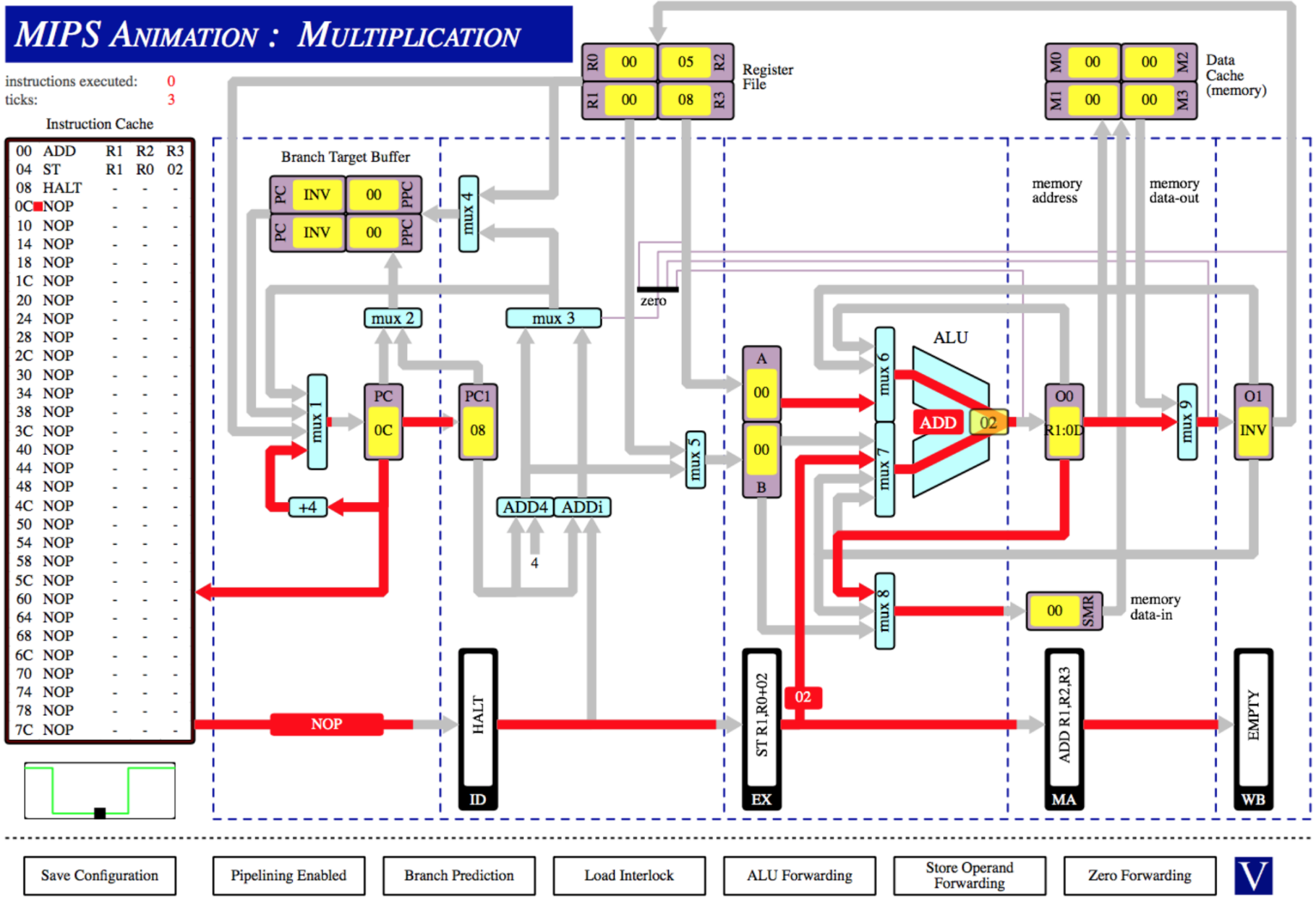
ADD R1,R2,R3
SUB R3,R1,R2
AND R0,R1,R3



Q1

3. O0 to MUX8

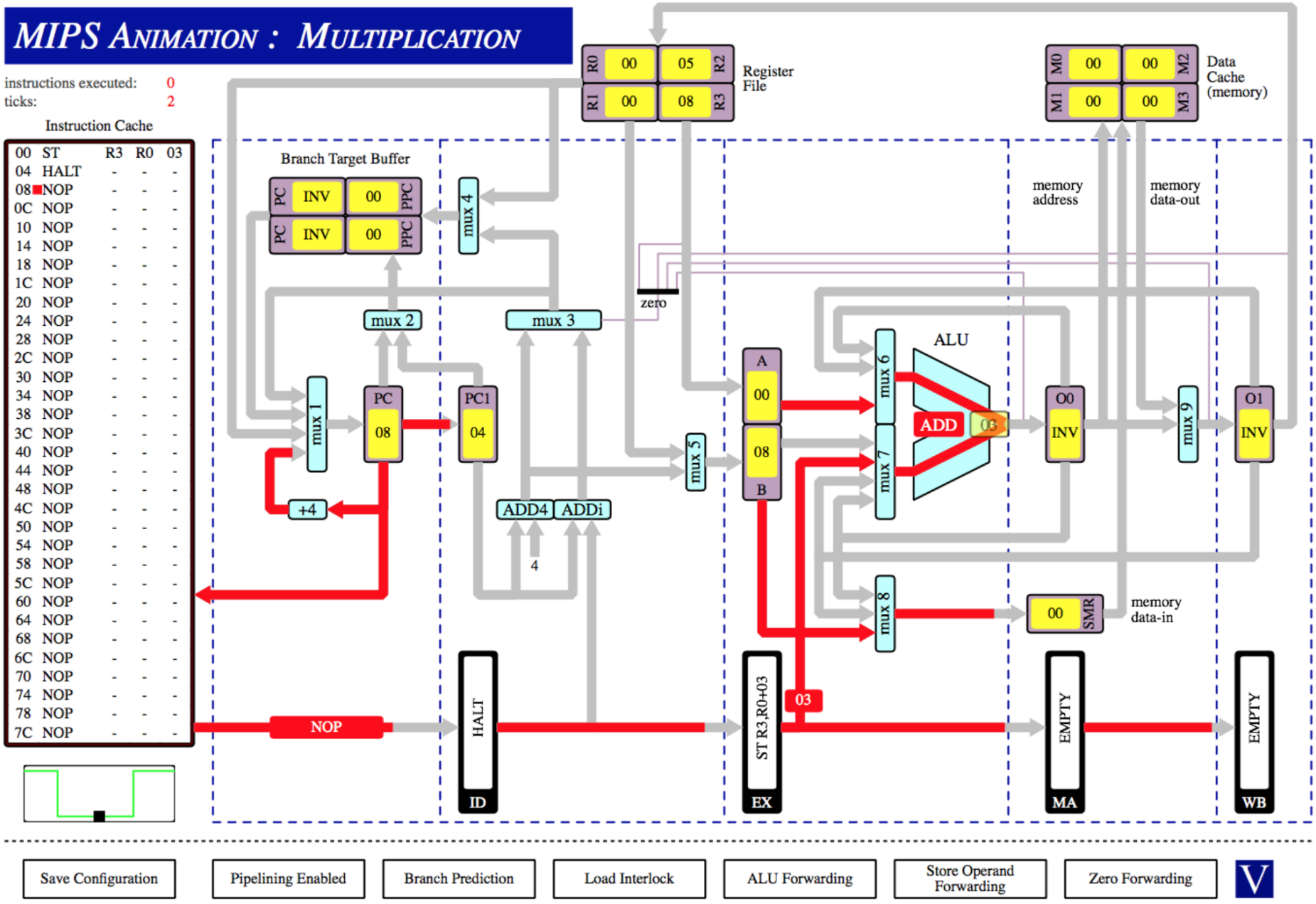
ADD R1,R2,R3
ST R1,R0,02



Q1

4. EX to MUX7

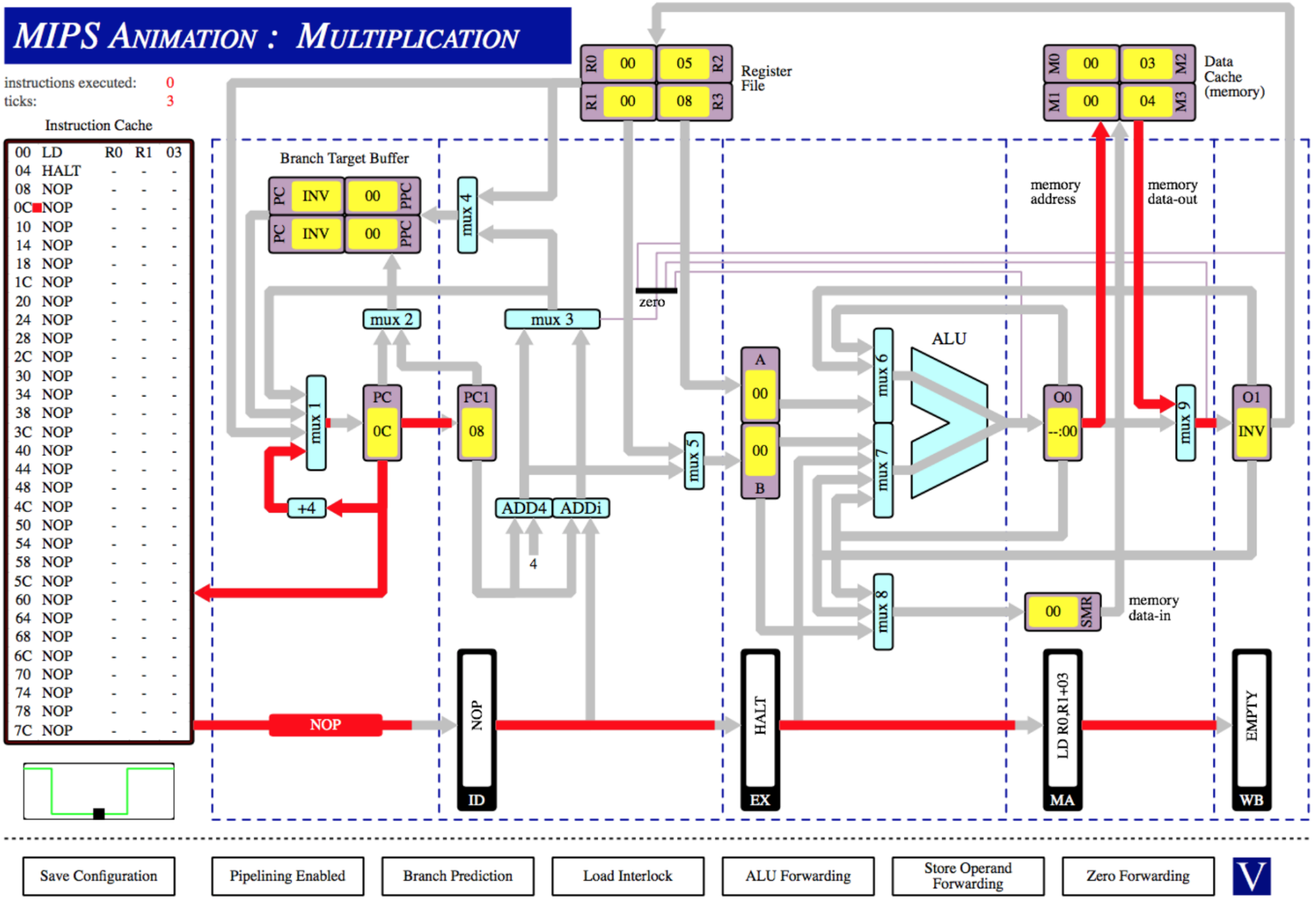
ST R3,R0,03



Q1

5. Data cache to MUX9 (memory data-out)

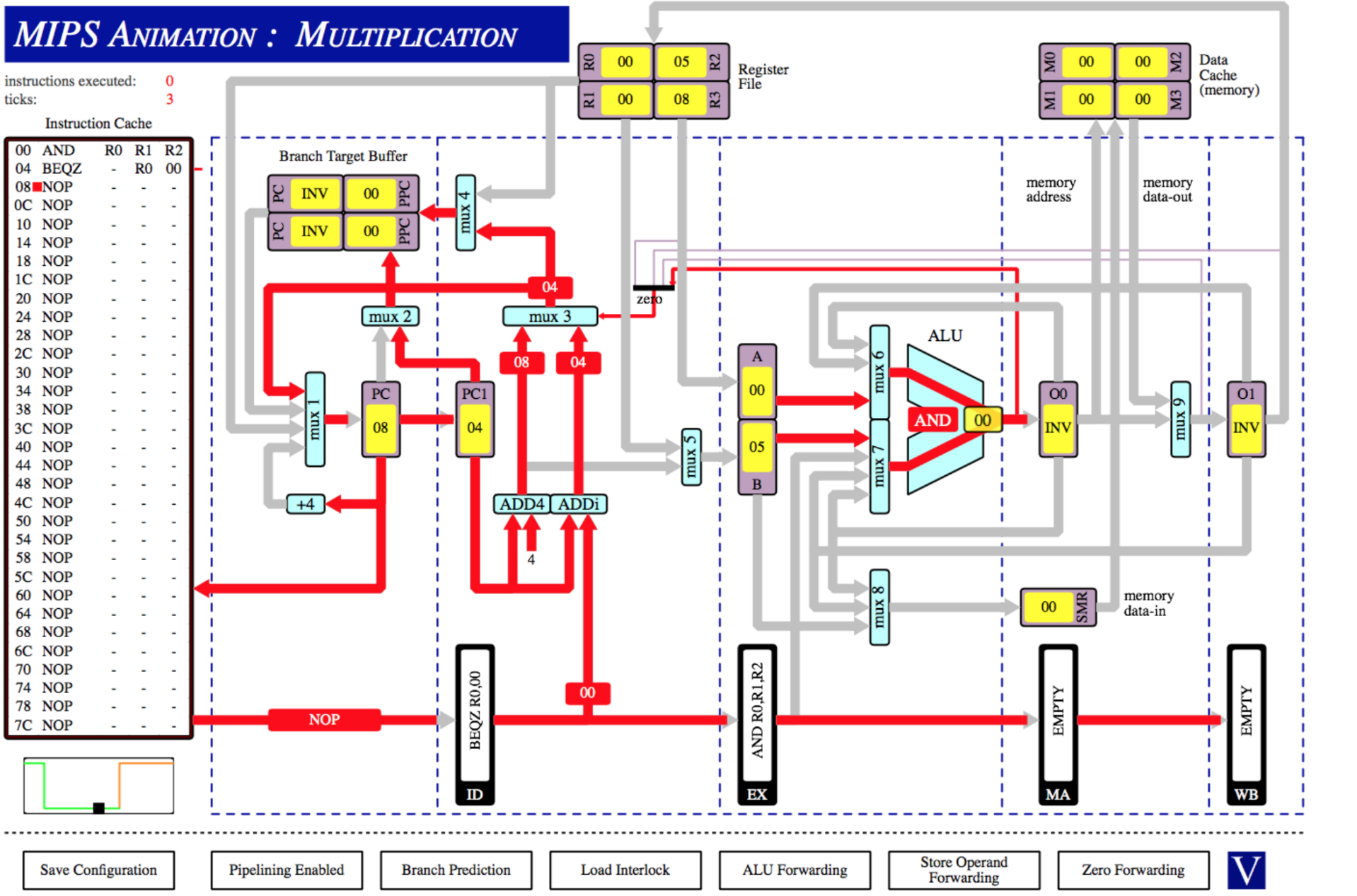
LD R0, R1, 03



Q1

6. O0 to Zero detector

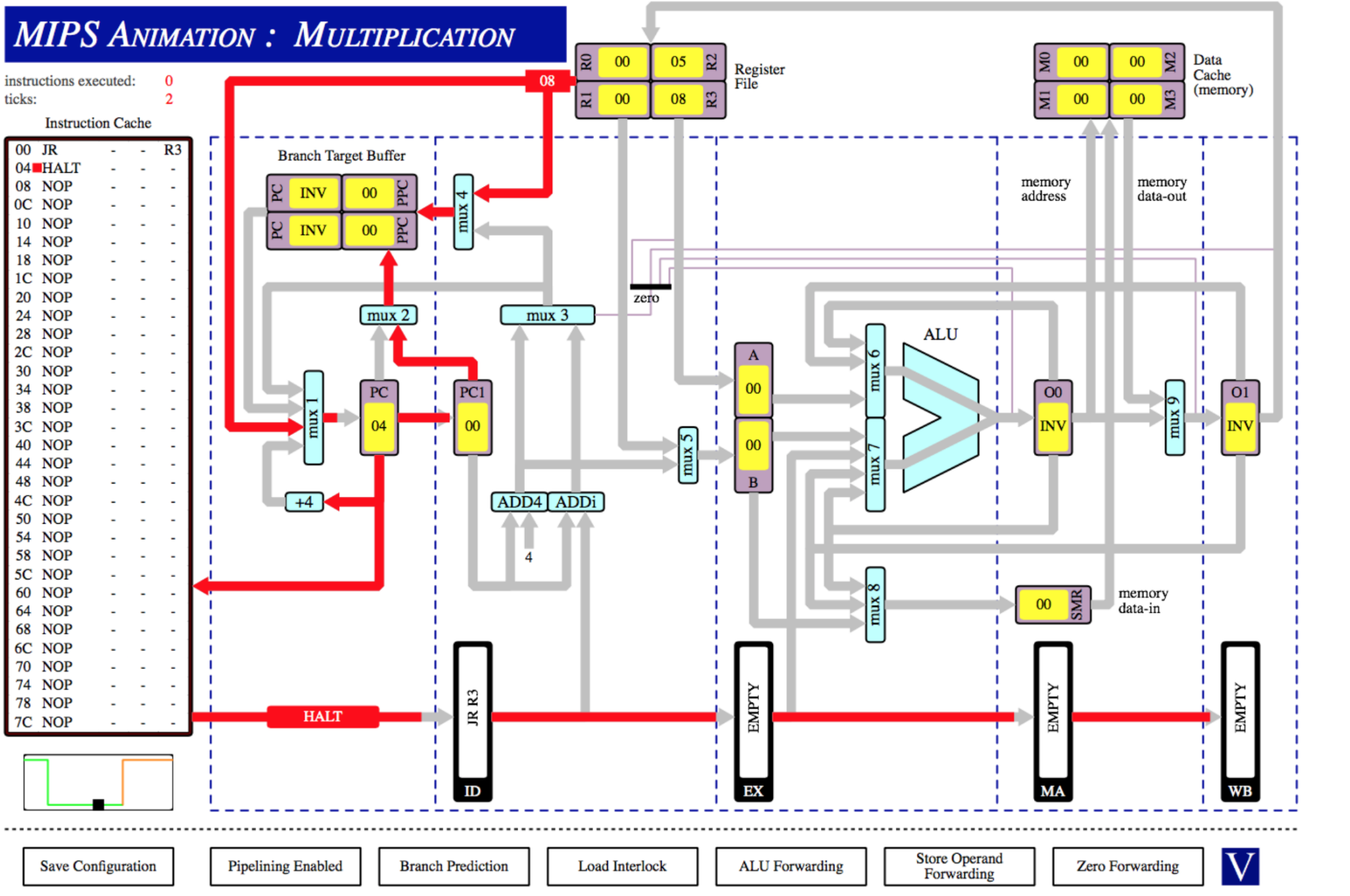
AND R0,R1,R2
BEQZ R0,00



Q1

7. Register File to MUX1

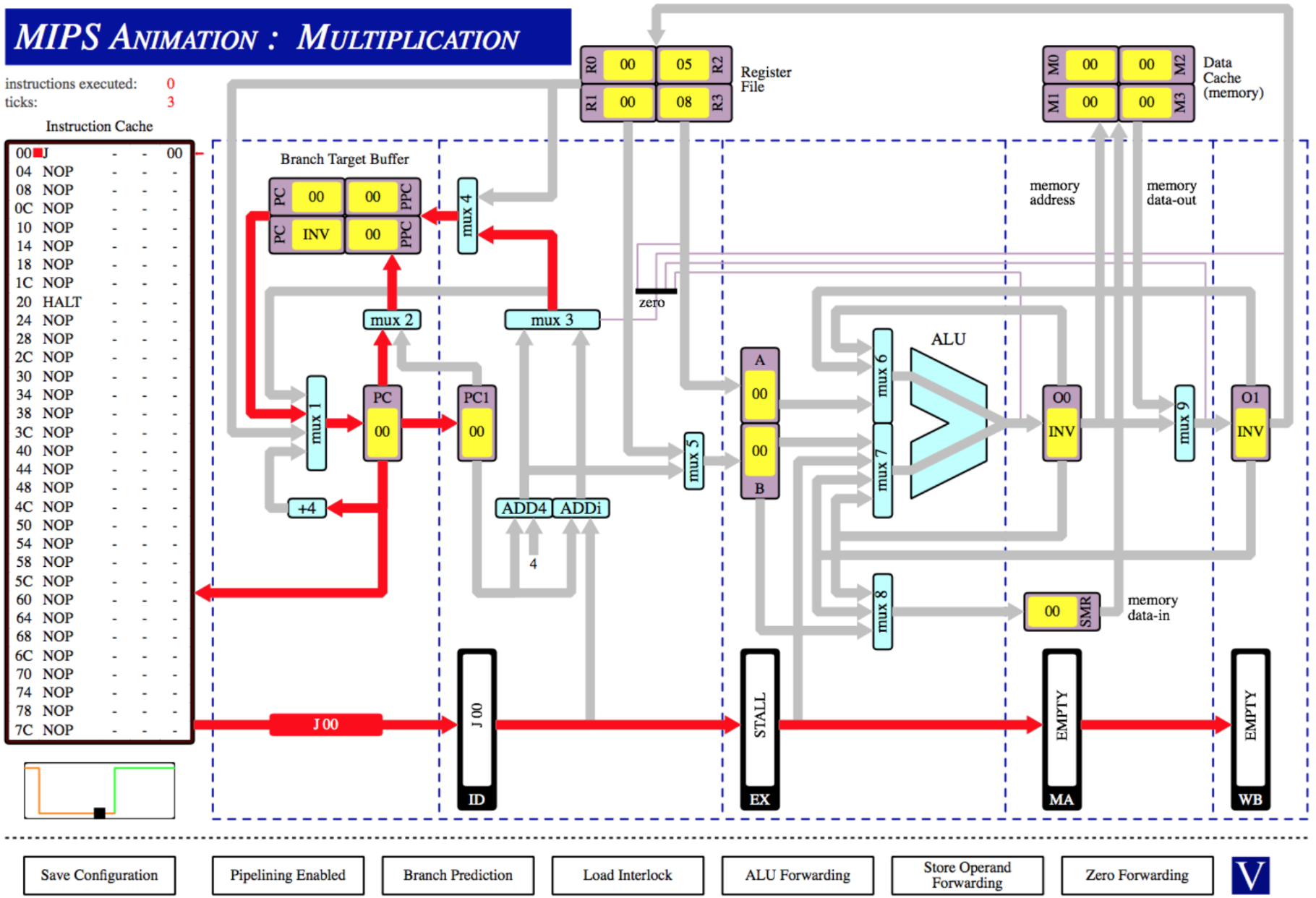
JR R3



Q1

8. Branch Target Buffer to MUX1

J 00



Q2

i) ticks = 10, result = 0x15
ii) ticks = 18, result = 0x15
iii) ticks = 10, result = 0x06

Explanation:

When ALU forwarding is enabled, results from previous instructions can be stored in the immediate registers O1 and O2 depending on which pipeline stage they are in, either Memory Access or Write Back. Unlike when ALU forwarding is disabled with CPU data dependency interlocks enabled, the processor doesn't have to implement 2 stalls between two instructions, so that the result of the preceding instructions can first update their results before the incoming ones can be executed. Since there are 4 instructions after the first one and between every two instructions, there should be 2 stalls, so 4x2 = 8 stalls, exactly the extra amount of stalls (ii) has.

When ALU forwarding with CPU data dependency interlocks disabled, there are no immediate registers to store results from previous instructions and no stalls can be implemented like in ii), the instructions will just keep reading values from registers that are not updated yet. This situation is called data hazards. As a result, all the instructions are executed directly and causing an error in results of the calculation.

Q3

I) 39 instructions, 51 ticks
Total stalls that implemented are 51 - 39 = 12.
First 4 stalls happened before the first instructions finishes its WB stage in the pipeline, so when the first instruction is fetched, 4 stalls has to be put in in front of it.

Then, every time before SRLi instruction can be executed after LD, a stall has to be put in between them, so that there is enough time for the loaded value in R2 to be stored in the immediate register, O1. This prevents the SRLi from being performed on the wrong values in R2. Since the instructions is performed 4 times, there would be 4 ticks until the iterations are completed.

Lastly, when the jump instruction, J is being decoded, the succeeding store, ST instruction will be fetched. Since the processor did not know that an unconditional jump is supposed to happened until it is decoded, a stall has to be put before the jump and executed first, so that ST can be removed and the correct instruction, BEQZ can be fetched instead. Since the iteration goes on for 4 times, then 4x1= 4 ticks.

By summing up the ticks that appear in the mentioned 3 conditions, we get the total, 4+4+4=12 ticks.

II) 39 instructions, 53 ticks
Total stalls that implemented are 53 - 39 = 14.
12 out of the 14 stalls are originated from the same causes mentioned in Q3.I). The other 2 occurred because of the absence of branch prediction feature in the processor to deal with conditional branches.

Branch instructions are those that tell the processor to decide which instructions to execute based on the results of previous instructions. A conditional branch in a pipeline is troublesome especially when it depends on results of an instruction that has not gone through the pipeline. In normal circumstances, 2 stalls have to be executed before the conditional branch, to allow enough cycles for the results from the previous instruction to be updated in WB stage. But we can reduce the number of stalls required to 1 by storing results of the preceding instructions in immediate registers to be used for the next conditional branch.

In this case, the stalls happened twice when ANDi instruction is executed in the 4 iterations, so 2x1 = 2 stalls to be executed to allow enough cycles for the BEQZ to be executed using the correct values in R2.

III) 39 instructions, 47 ticks
Total stalls that implemented are 47 - 39 = 8.

The difference between cases in I) and III) is that, by swapping SRLi and SLLi execution order, we avoid a stall being executed between the LD and SRLi instructions like in I). This is because SLLi instruction does not require any operands used in LD, so it doesn't have to wait for one cycle to allow the updated results to be stored in immediate register O1. Moreover, since SLLi requires exactly one cycle to be executed, this complements the one cycle needed for results in LD to be stored in O1 immediate register as in I) and used by SRLi instruction.