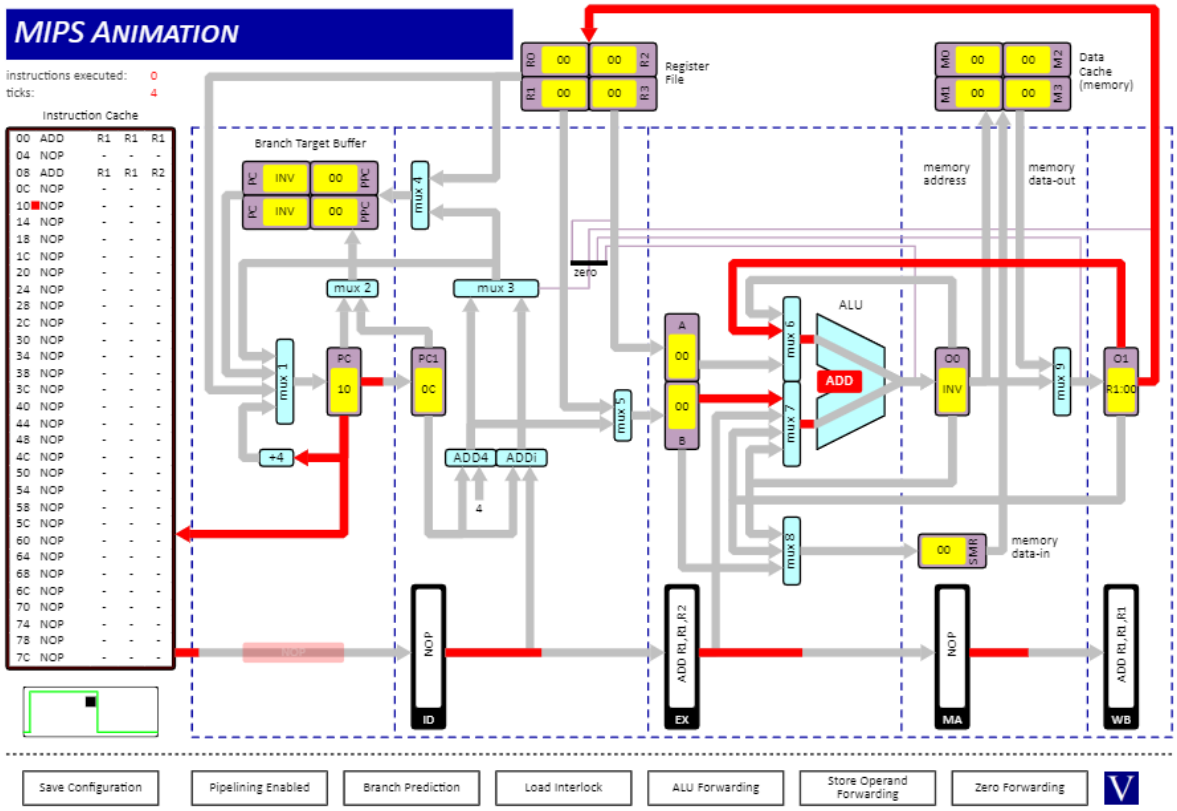
Tutorial 4

Name: Jack Joseph Gilbride
Student Number: 17340868

Q 1.1: *O1 to MUX6*
ADD R1, R1, R1,
NOP
ADD R1, R1, R2

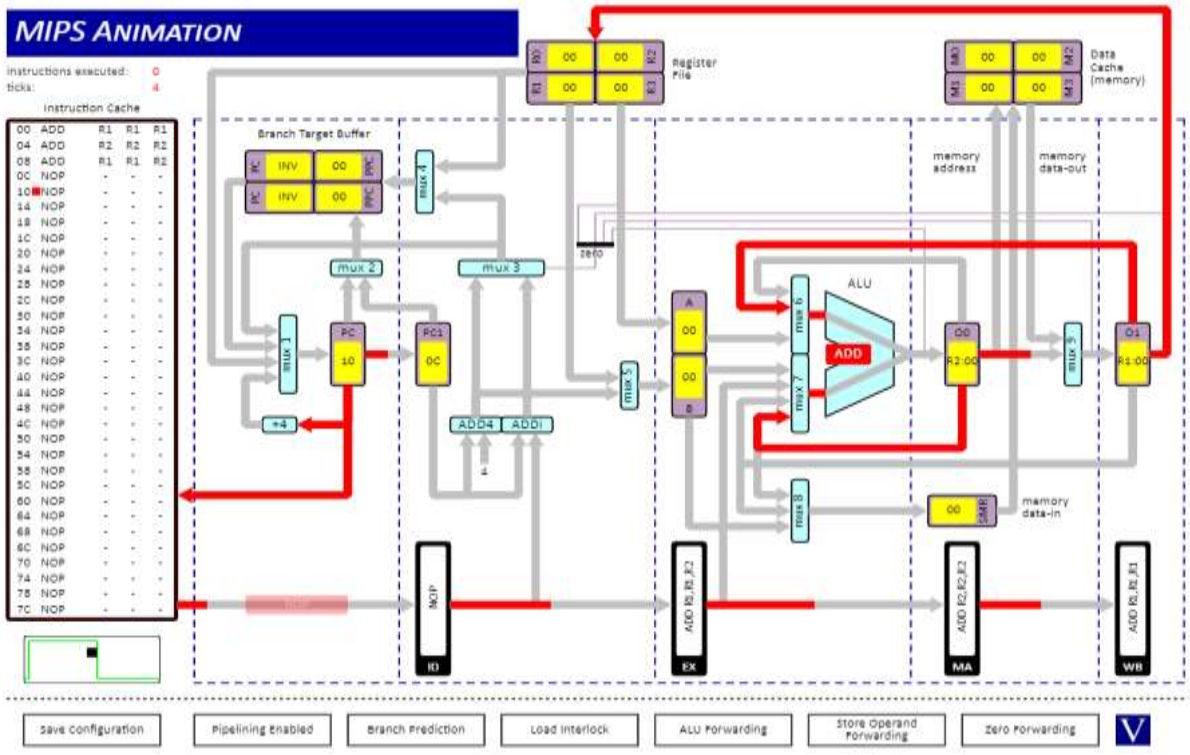


Q 1.2: O0 to MUX7 and O1 to MUX6 (simultaneously)

ADD R1, R1, R1

ADD R2, R2, R2

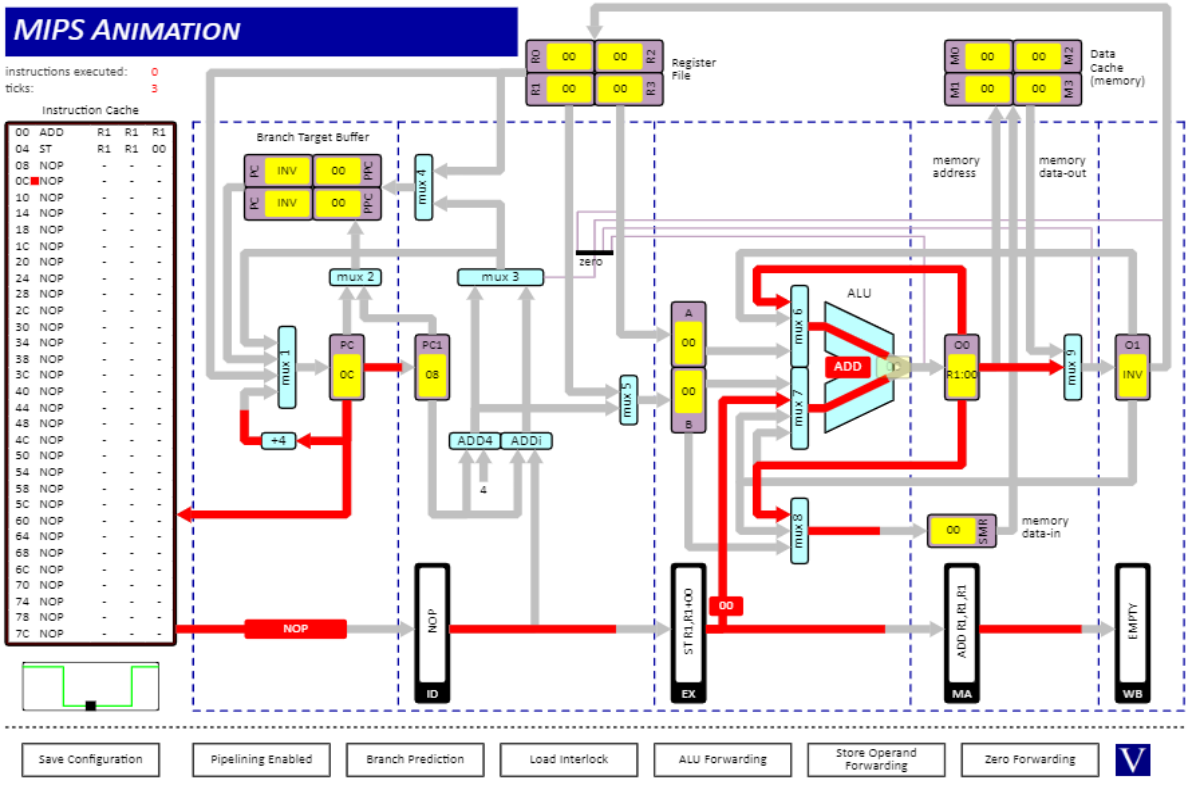
ADD R1, R1, R2



Q 1.3: O0 to MUX8

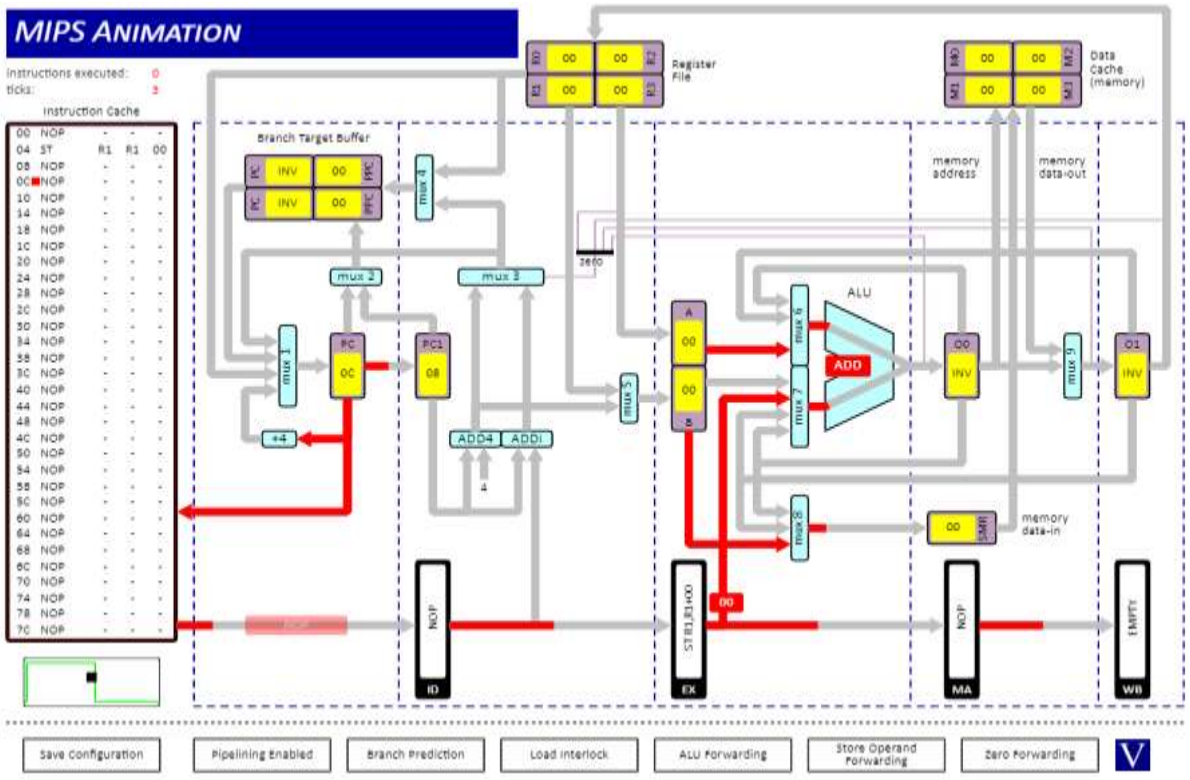
ADD R1, R1, R1

ST R1, R1, 00



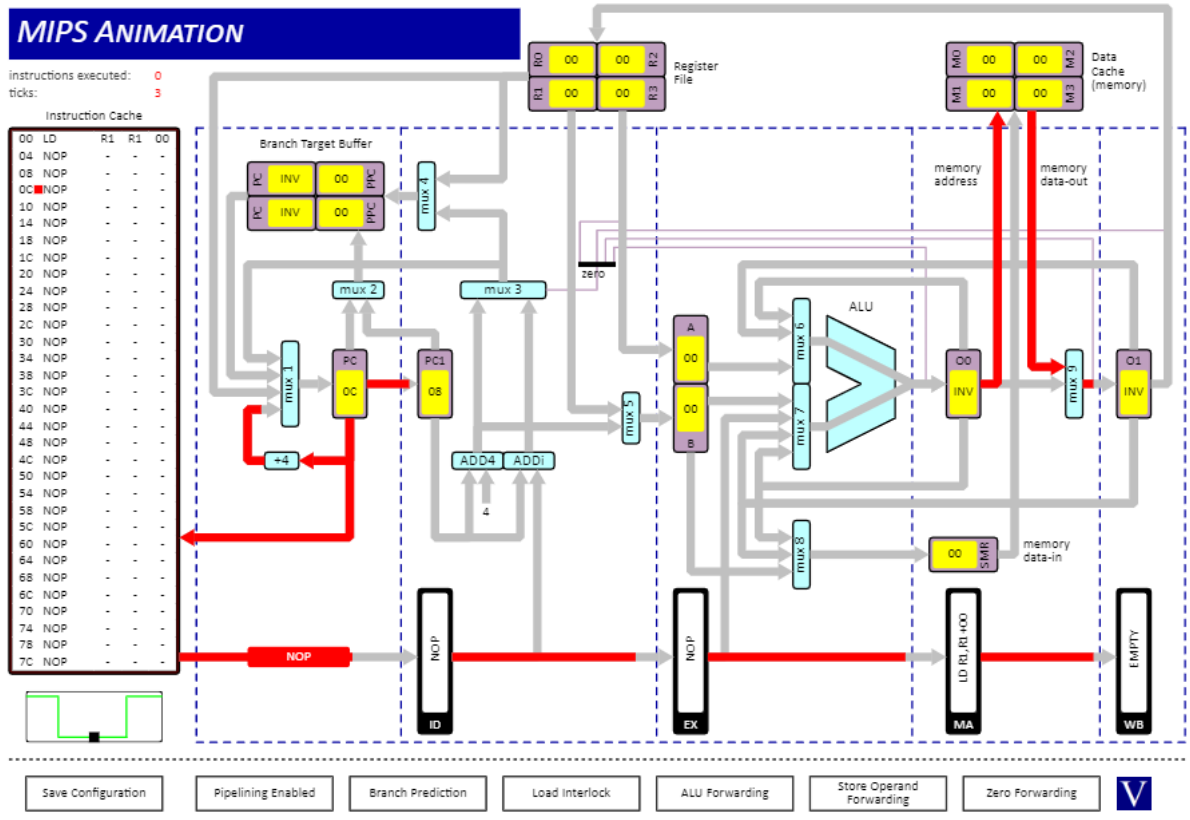
Q 1.4: EX to MUX7

ST R1, R1, 00



Q 1.5: Data cache to MUX9 (memory data-out)

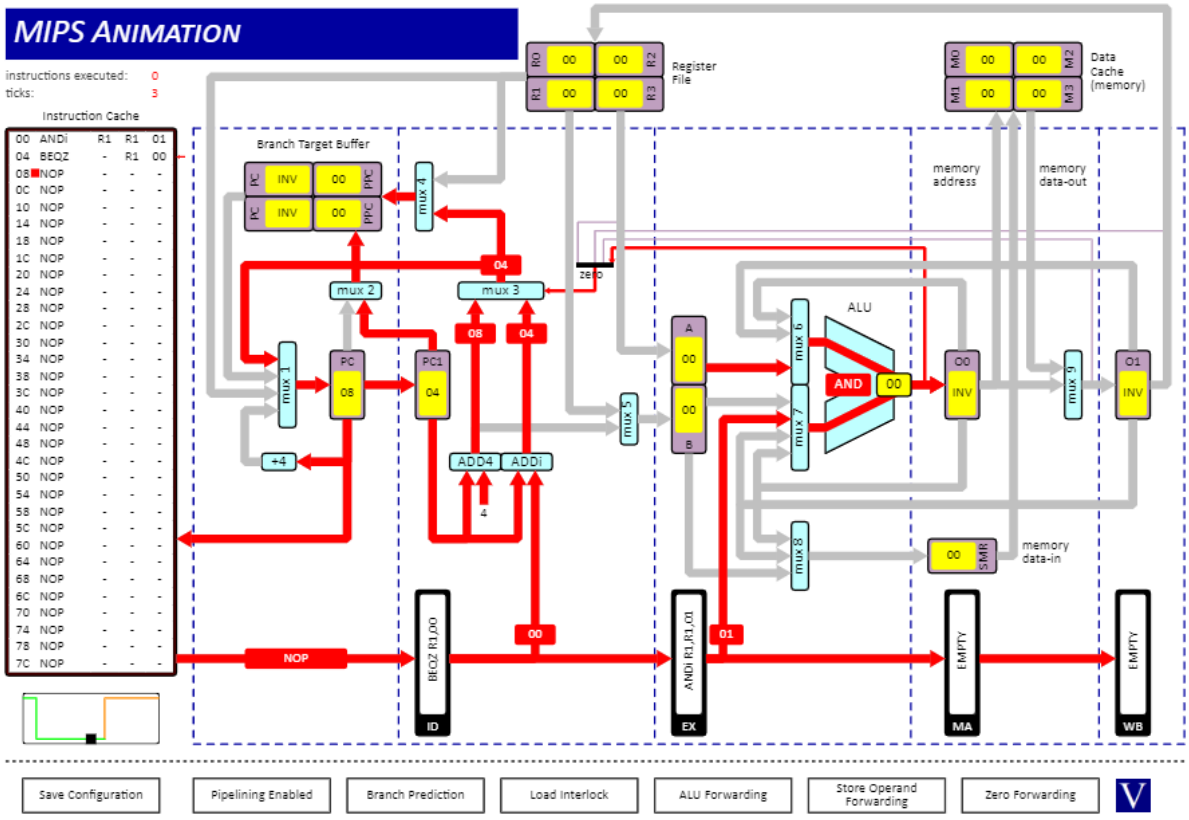
LD R1, R0, 00



Q 1.6: O0 to Zero detector

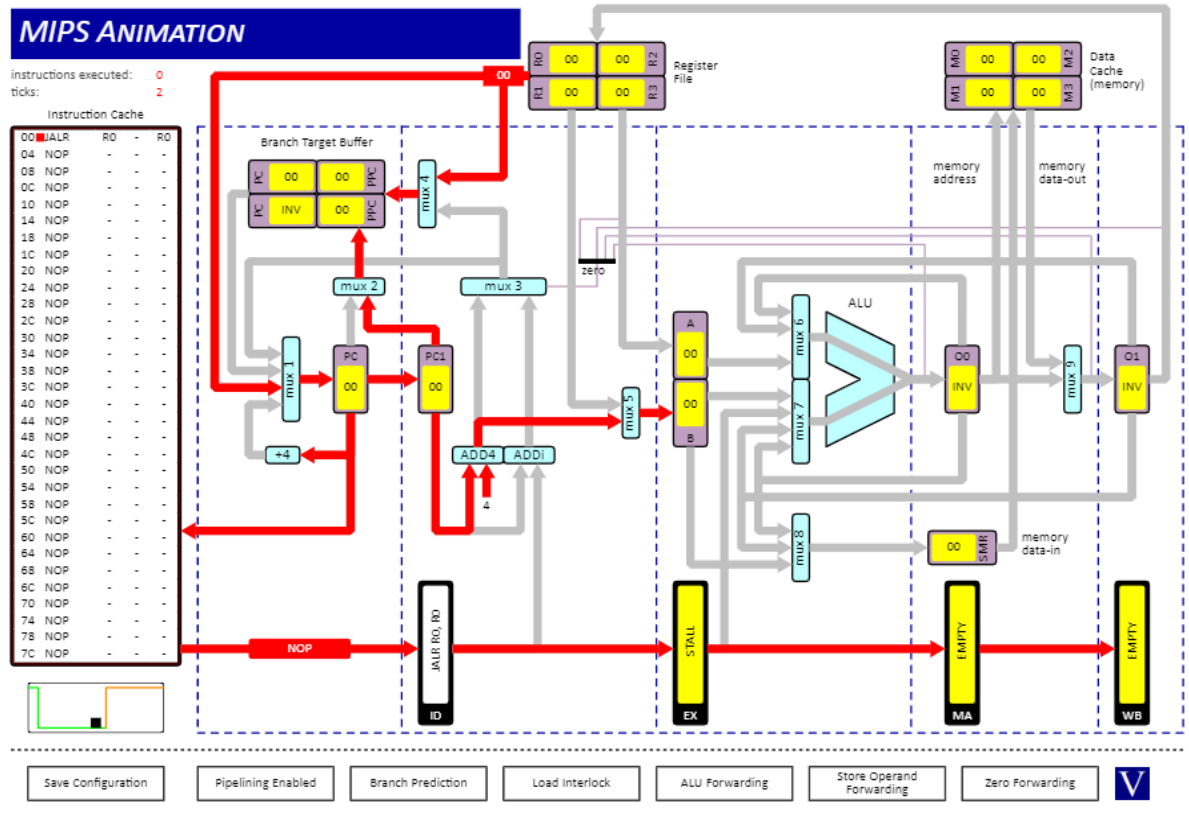
ANDi R1, R1, 01

BEQZ I, R1, 01



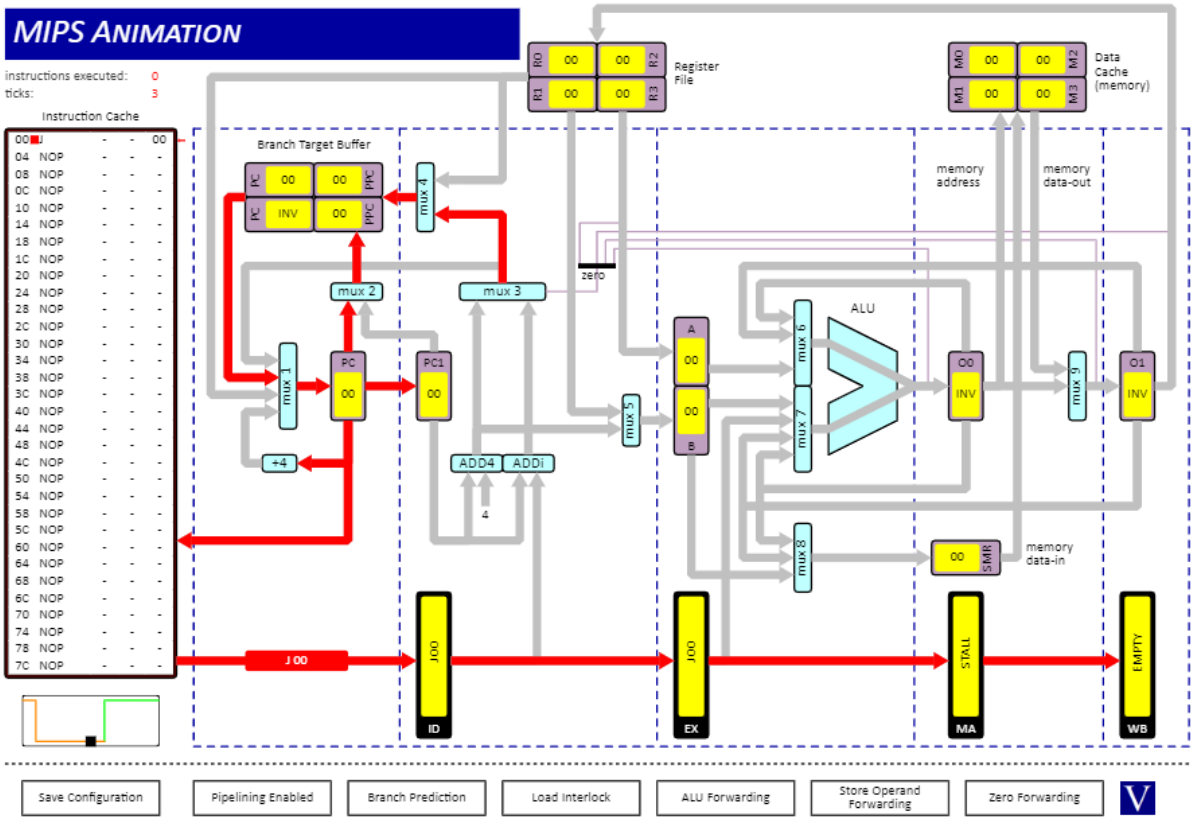
Q 1.7: Register File to MUX1

JALR R0, -, R0



Q 1.8: Branch Target Buffer to MUX1

J -, -, 00



Q 2.1

R1 = 0x15 = 21

Ticks = 10

Reason: ALU forwarding is enabled. ALU results from previous instructions can be forwarded to the ALU inputs to the next instruction within a clock cycle. This is done by putting extra logic and registers in the pipeline. In other words, if r1 or r2 are going to be written to, their new values are immediately cached in the pipeline during the execution stage, write back to the register file occurs two stages later. The next execution stage checks these registers before checking the register file, ensuring the correct value is always read even when the previous instruction has not written back yet. There is no gap in the pipeline as we never have to wait for writeback. Thus the answer is correct and takes 6 instructions + 4 ticks to complete last instruction = 10 ticks.

Q 2.2

R1 = 0x15 = 21

Ticks = 18

Reason: ALU forwarding is not enabled but data dependency interlocks are enabled. As there is no ALU forwarding, an instruction cannot execute if it uses a register written to by the previous instruction, until that instruction has ended write-back phase. The data dependency interlocks mean that the CPU is aware of this, so stalls the CPU while waiting for the previous instruction to write back. The result is correct again because of this, but it takes longer. There are four additional two-tick gaps in the pipeline, which results in 18 ticks.

Q 2.3

R1 = 0x06 = 6
Ticks = 10
Reason: ALU forwarding and data dependency interlocks are both disabled. The CPU does not immediately cache outputs from the "previous" execution stage for the next execution stage. However, there is no logic in the CPU to stall to wait for the write back stage of the previous instruction. As there are no gaps in the pipeline, the time to execute is the same as part 3.1. But the answer is incorrect as on multiple occasions, the register reads r1/r2 before their values have been updated from the previous instruction.

Q 3.1
Instructions executed = 38
Ticks = 50
Explain in detail why these two numbers are not equal and account for each stall cycle: These numbers are not equal because of pipeline stalls. It takes at least one tick for each instruction to execute, which brings it to 38 ticks. It takes four ticks at the start of the program to get the first line of code to the execute stage, which brings it up to 42. There is a stall between the ST and ANDI instruction as it takes time for the value in R2 to be stored into immediate register O1. This happens four times, so four more ticks, which brings the ticks to 46. The jump J at the end of the loop causes a stall when branch prediction is incorrect, as the correct address must be fetched. The branch prediction is wrong twice which brings the ticks to 48. Both BEQZ execute once each over the course of the program, so each stall one clock cycle as they fetch the correct address. This brings it up to 05 ticks.

Q 3.2
Instructions executed = 38
Ticks = 53
Explain in detail why this number differs from your answer to part (i): Branch prediction has been turned off, so where there is a branch there will always be a branch in the pipeline. Branch prediction was correct three times in the implementation in part 3.1, so without branch prediction there will be three additional stalls in the pipeline. All of the other stalls still apply. Therefore, there are 50 + 3 = 53 ticks.

Q 3.3
Using "Branch Prediction", what is the effect on execution time if the two shift instructions are swapped and why?: There will no longer be a stall between LD and the following shift instruction as the shift left instruction does not rely on R2 (which is written to by LD) while the shift right does. By the time we get to the shift right, we have passed a clock cycle running shift left, so we can read R2's value straight away from O1. This reduces one clock cycle per loop, so we have **four less ticks**. It is also important to note that the shift instructions being swapped does not affect the functionality of the program as there is no read-write, write-read or write-write dependency between them.