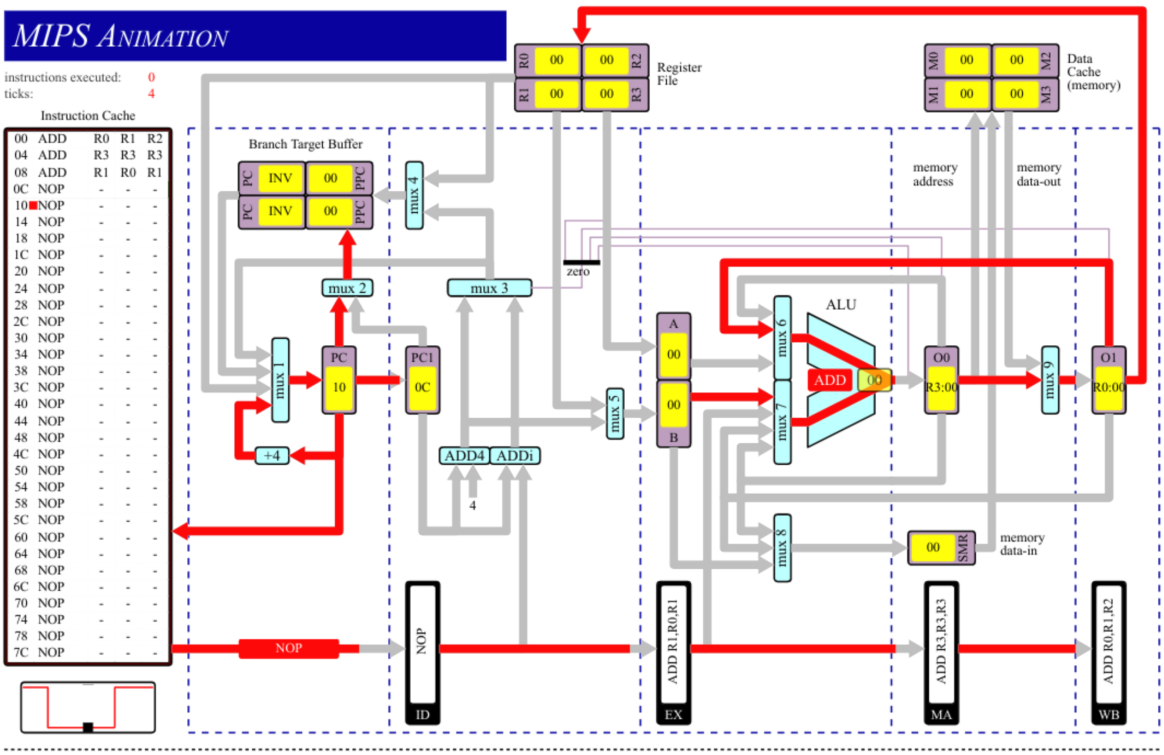
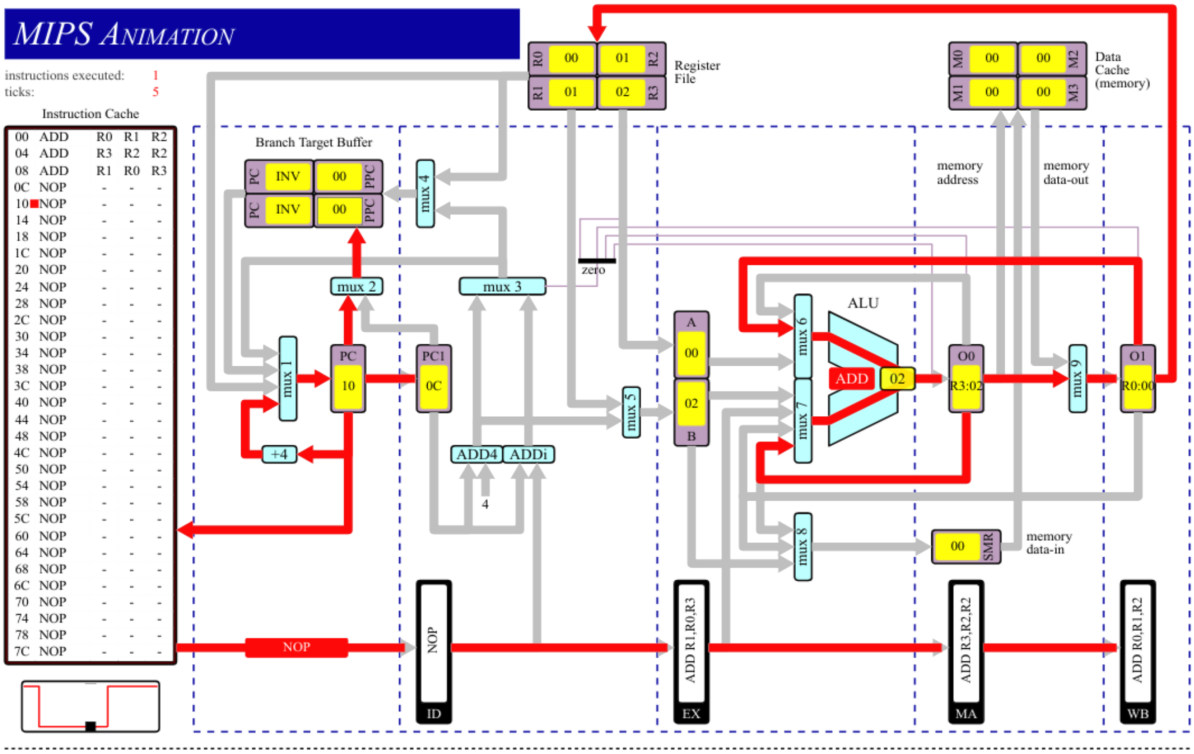
Tutorial 4 - Jack Cassidy 14320816

Q1

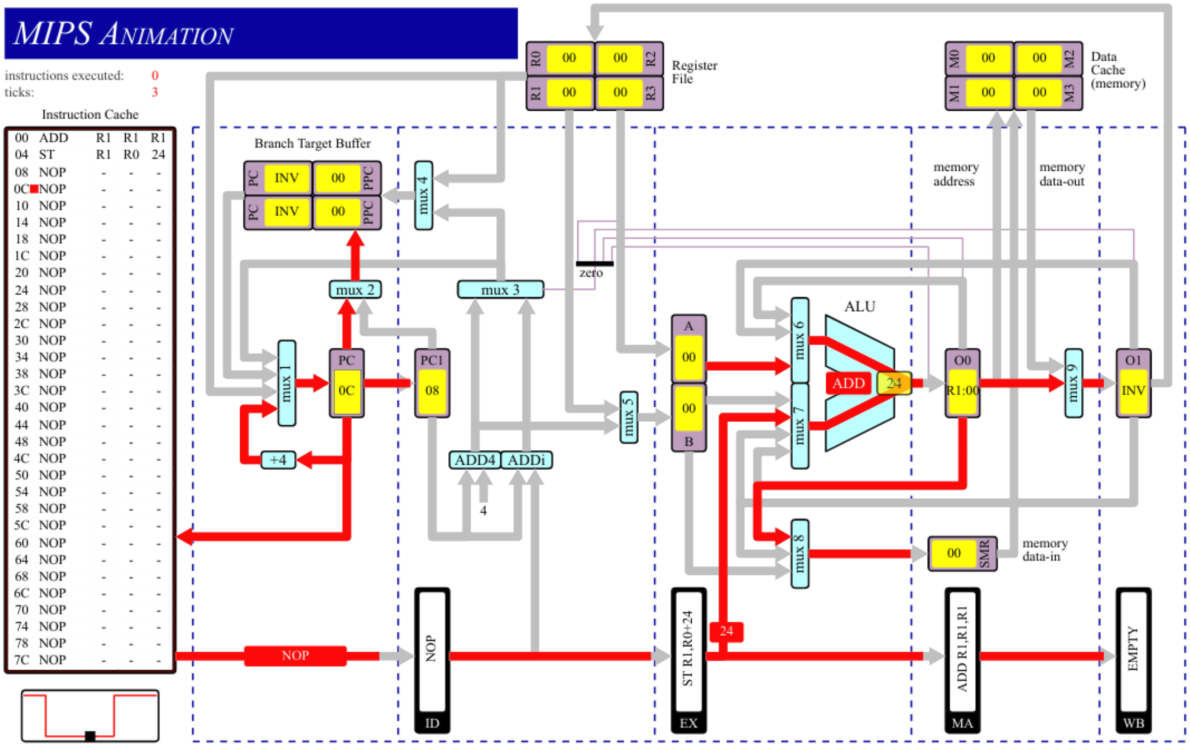
i) O1 to MUX6



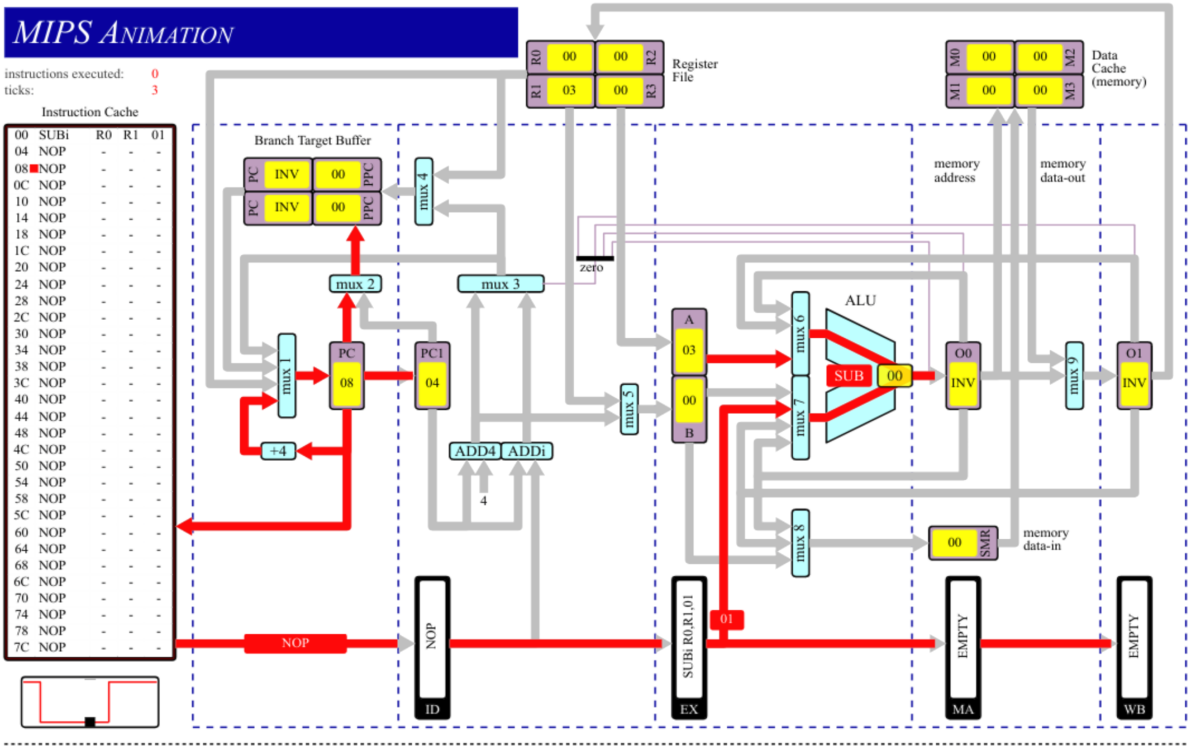
ii) O0 to MUX7 and O1 to MUX6 simultaneously



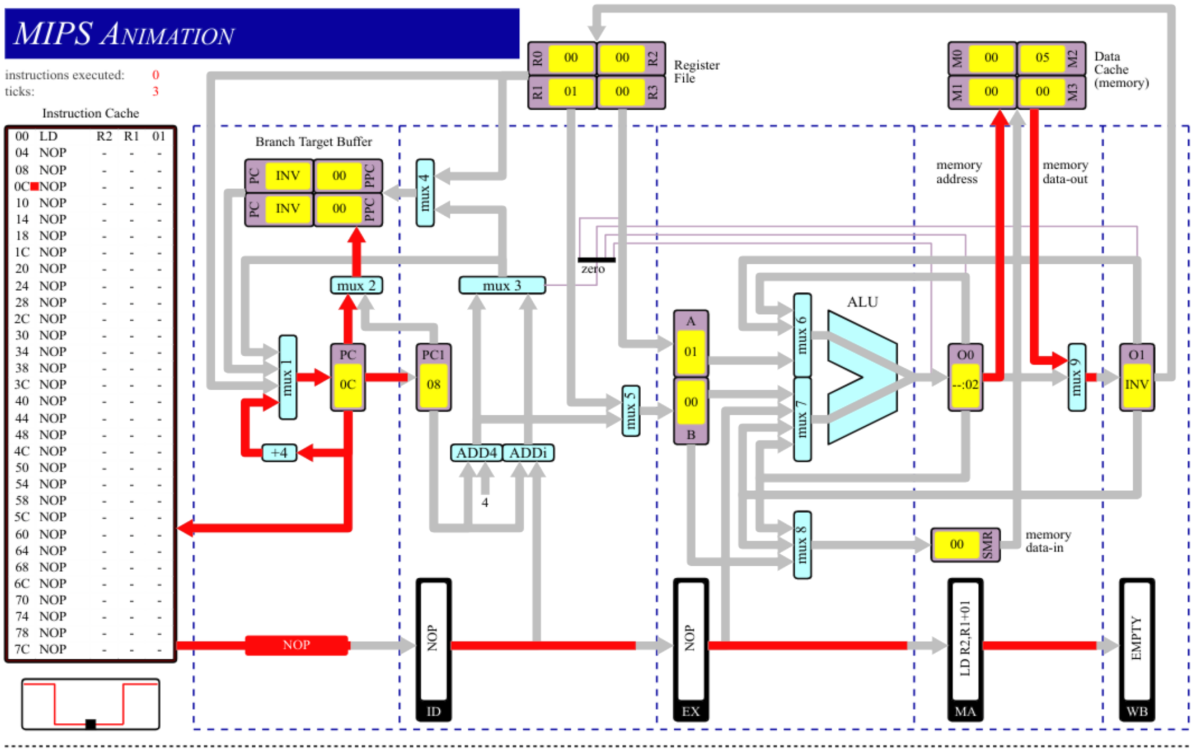
iii) O0 to MUX8



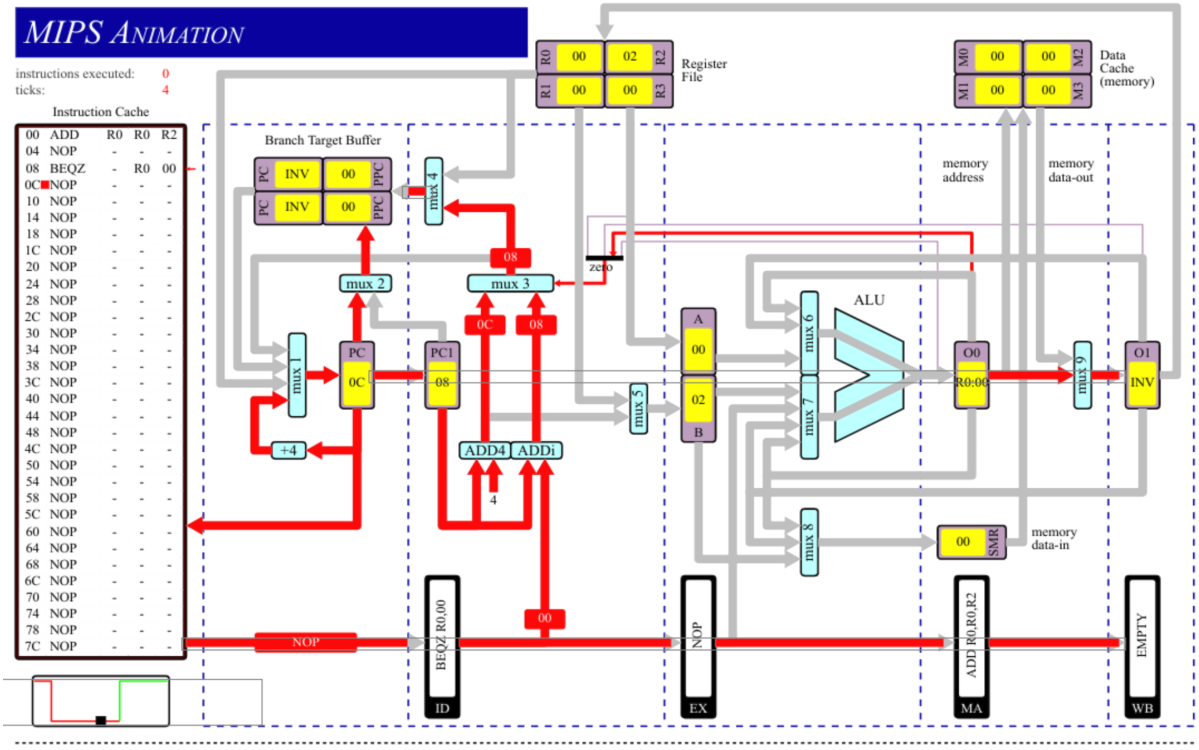
iv) EX to MUX7



v) Data Cache to MUX9



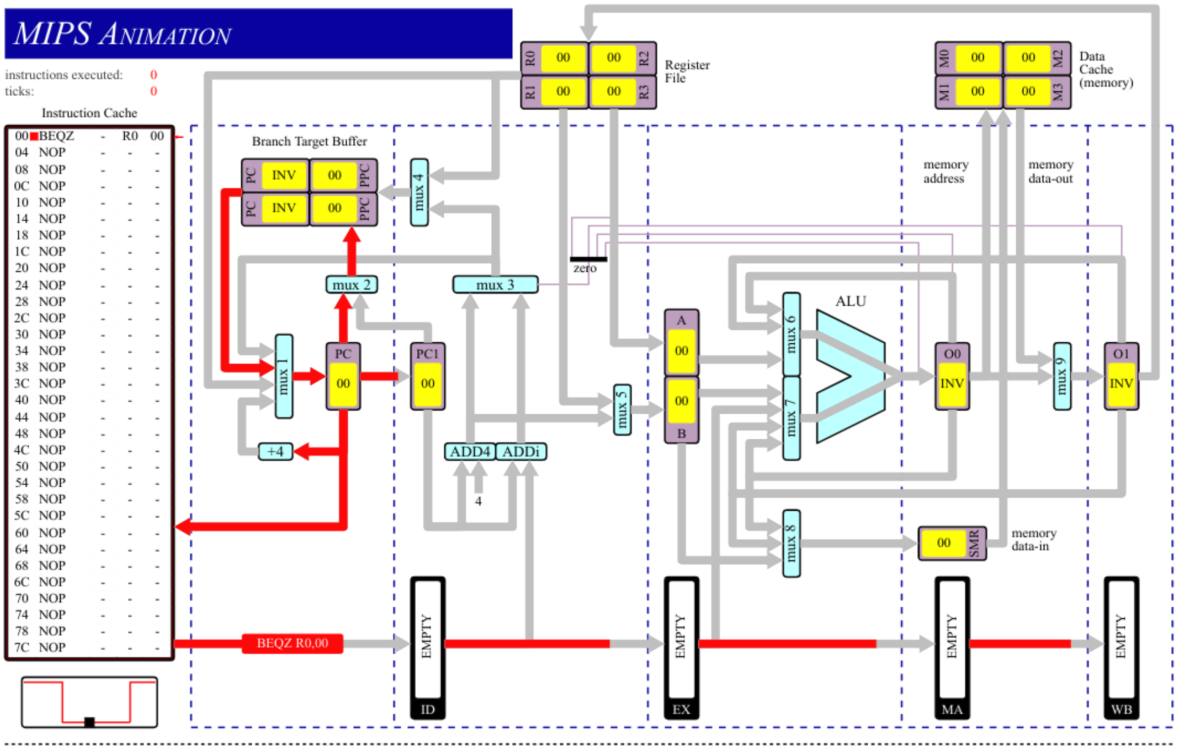
vi) O0 to Zero Detector



vii) Register File to MUX1

I was unable to create this path on the Visio animation. My thought process was to use the JR (Jump Register) instruction, but this did not give the desired path.

viii) Branch Target Buffer to MUX1



Q2
i) ALU Forwarding ON
Number of cycles: 9
Answer in R1: 21 (correct)

ii) ALU Forwarding OFF. CPU Data Dependencies ON
Number of cycles: 17
Answer in R1: 21 (correct)

iii) ALU Forwarding OFF. CPU Data Dependencies OFF
Number of cycles: 9
Answer in R1: 6 (incorrect)

A data hazard occurs when an operand register of an instruction is being written to as a destination register in the previous instruction, known as a data dependency. The instruction cannot be guaranteed the correct value for it's operand until the correct value has been written to the register file in the previous instruction. In pipelining terms, the current instruction cannot enter the EX phase until the previous instruction has completed the WB phase. There is a data hazard on every line of this program except the first.

ALU forwarding circumvents the data dependency problem by having intermediate registers at the end of the EX and MA phases, which store the results of these phases. This enables the 'forwarding' of ALU calculation results to be concurrently wrapped around and used as inputs to the ALU for the EX phase of the next instruction and to be passed onto the MA phase of the current instruction as normal.

Hence, the reason for the number of cycles nearly doubling when ALU forwarding is turned off is that each time there is a data hazard, the pipeline must wait for the data hazard to clear also known as 'stalling.' Due to the fact that there are many data hazards in succession in this program, the pipeline stalls accumulate as shown in the attached spreadsheet. As instruction are blocked from entering the EX phase, they are obviously stuck in the ID phase. As each phase can be occupied by one instruction at a time, this stall propagates backwards so that the IF phase becomes congested, similar to a traffic jam on a highway.

However, the correct result is arrived at eventually. When CPU data dependencies are turned off, we arrive at an incorrect answer because the CPU cannot resolve data hazards. It naively pushes through the pipeline, with incorrect values for source operands being read from the register file.

Q3
i)
Number of instructions: 22
Number of cycles: 28

There are two reasons for the difference in number of clock cycles and instructions completed.

a) At the beginning of the program the pipeline is empty. Thus only one phase of the pipeline is at work at any one time until it is filled. It is not until tick 5 when the pipeline will be fully utilized.

$$\#\ \text{cycles} = \#\ \text{instructions} + (\ \text{pipeline size} - 1\ )$$

This gives us 26 cycles, so two are unaccounted for.

b) The two missing cycles are from the branch instructions, J (unconditional branch) and BEQZ (Branch Equal to Zero). The first time both of these instructions are executed, meaning the branch is taken, the pipeline stalls in the ID phase for one cycle as the new value of the program counter is calculated. The Branch Target Buffer (BTB) caches a mapping of the branch instruction address to the new program counter value, which can prevent subsequent stalls whenever these branches are executed in the future as branch prediction comes into play.

ii)
Number of cycles: 29
When ‘Branch Interlocking’ is turned on, it removes the BTB availability. This means that there are no cached branch instructions and hence, no branch prediction can take place. As eluded to above, branch prediction helps to reduce extra cycles by always taking a branch if there is a BTB cache hit. In the scenario where the branch is not taken, the penalty is only a single cycle stall as the incorrectly fetched instruction is aborted.

The reason for the extra cycle in this scenario, is that the program counter must again be evaluated for the second J instruction, whereas before the destination address was cached and branch prediction cut out the extra cycle.

iii)
Number of instructions: 22
Number of cycles: 30
The execution time is increased by 2 cycles. This is due to the introduction of a load hazard every iteration. SRLi is dependant on the correct value of R2 being loaded from memory by the previous LD instruction, hence there is a 1 cycle stall as the SRLi instruction waits in the ID phase for the LD instruction to complete the MA phase. The correct value for R2 can be forwarded using ALU forwarding as discussed above, which cuts the cycle penalty from 2 (where the SRLi instruction would have to wait until WB phase had completed) to 1.