**QUESTION 2**

Result: R1 = 0A, R2 = 0A

Clock cycles

(i) 10

(ii) 14

(iii) 10

Quick Note: “R1” means initial value for R1, “R1\*” means the value of R1 after the first operation that changes R1, “R1\*\*” means the value of R1 after the second operation that changes R1, etc.

**ALU Forwarding**

The first ADD instruction will produce a result for R1, R1\*. This result will be stored in O0 just before the start of the next clock cycle and won’t have made its way to being stored in the Register file. The next SUB instruction has a dependency on the previous instruction, as it needs R1\* in order to calculate R3\*. As a result, the value in O0 (R1\*) is forwarded to MUX6, i.e. the first operand for the operation *SUB R3 R1 R2* in the ALU, so we can now accurately calculate R3\*. In the meantime, R1\* has made its way from O0 to O1, and the R3\* will now be stored in O0. Next up is an AND instruction that will calculate R2 but needs R1\* and R3\* to do this. As neither of these results have made it to the register file yet, O0 (R3\*) and O1(R1\*) will be sent to MUX7 and MUX6 respectively to act as the operands for the AND instruction. At this point, R1\* has finally made it to the register file, R2\* is in O0 and R3\* is in O1. The next instruction, XOR, requires both R1\* and R3\* to calculate R2\*\*. We can take R1\* from the register file as operand A, and will forward R3\* from O1 to MUX7 as operand B. The final instruction, ADD, has no dependencies, so it can just read both registers from the register file as normal. The advantage of ALU Forwarding here is that there is no NOPs or Stalls required, as dependencies are recognized and dealt with before the values are even stored in the register file. As a result, if *n* is the number of instructions in the program, and *p* is the number of pipelines stages, then this takes a total of n + (p-1) clock cycles to complete, n = 6 (including HALT) and p = 5, 5 + (6-1) = 10.

**ALU Interlock**

Whenever we meet an instruction that has a dependency on either of the 2 instructions ahead of it, ALU Interlock forces a stall for 2 clock cycles while R1\* and R3\* (or whichever results are causing these dependencies) are forwarded from O0 to O1 to the register file. The program counter does not increment during these stalls, adding 2 clock cycles. So for SUB we need to add 2 stalling clock cycles so that R1\* can go to the register file, for AND we need to add 2 stalling clock cycles so that R3\* can go to the register file. With ALU Forwarding, the XOR instruction required that we forward R3\* from O1 as it had not made it to the register file yet, but here we don’t need introduce any stalls, as R3\* can just be taken from the register file since it already made it to there from the AND stall. The last ADD instruction has no dependencies on this at all, as R1\* and R0 are in register file. So we had to stall twice, meaning 4 extra clock cycles compared to ALU Forwarding, so we get 14 clock cycles for ALU Interlock.

**No ALU Interlock**

Without either ALU interlock or ALU forwarding, the wrong result will be produced UNLESS the programmer accounts for this by adding NOPs or Useful Instructions after instructions with dependencies (although, coincidentally, this time the program produces the same results for R1 and R2 as ALU Forwarding and ALU Interlock above, but R3 was wrong in the end). Without ALU Interlocking, if the operation being inputted into the ALU has a dependency on either of the previous two instructions, this will not be checked for, and the ALU will take the values of the operand registers as they are at that point in time. So, our program will not produce any stalls, and will have the same n + (p – 1) formula stated above, i.e. 10 clock cycles.

**QUESTION 3**

*Part 1*

38 Instructions executed, 50 clock cycles

A 1-cycle stall is introduced whenever the LOAD instruction at address 0x18 is executed, as this instruction requires 2 clock cycles, one for calculating the address of the offset (R0 00), and then a cycle for loading the value from memory. The STORE instruction on the other hand can be achieved in one clock cycle.

The instruction J E0 at address 0x24 also introduces a stall the first time it is executed, as this branch needs to be added to the BTB. This also gives a cycle for the PC to be changed to the new PC from the branch, and for ID to read the instruction at PC’s value.

We go through this loop 4 times, so this is 4 stalls for load, plus 1 for J E0. We still need to find why the difference of 8 remains.

BEQZ at instruction 0x10 introduces a stall whenever R2 is equal to 0. This value is taken directly from the result of the ALU and not from the register file R2, as R2 is at that moment being calculated in the ALU. This stall is (i) to allow the element for 0x10 to be added to Brach Target Buffer, with 0x18 being the address that it branches to. This is a compulsory miss, as this branch hadn’t been taken before and was not stored in the BTB before. This happens two times, so now we’re left with a difference of 6 between instructions/clock cycles.

The last stall is introduced by BEQZ at 0x04, after the final iteration of the loop. As this branch was not taken, then we need to store this branch into the BTB, fetch the next instruction, and decode it. This adds just one stall.

The remaining different of 5 can be found due to how long the machine takes to carry out any program. With 5 pipeline stages, the number of clock cycles should equal the 5 (pipeline stages) + N – 1 (with N being instructions executed. As we also have a HALT instruction, this this completes the explanation for why there is a difference between the number of instructions executed and the number of clock cycles.

*Part 2*

Instructions executed 45, clock cycles 53

The stalls introduced by BEQZ in Part 1 are not present in when we have switched the Delayed Branches, as we have gotten rid of the BTB. This will lead to problems as the instructions directly after BEQZs that should not be executed are executed. So, we need to introduced NOPS after the BEQZ. Unfortunately, there are no Useful Instructions that we can put in the place of the NOPs here for either BEQZ, so this is two extra instructions, executed once for every time the loop is iterated, +1 for the if(=0) break; BEQZ at address 0x04. Since there are 4 loops executed, then this means we have 9 more instructions to execute. We also have to add a NOP after J 0E, as the instruction ST R1 R0 00 will be executed in every iteration of the loop otherwise. But in this case, we can replace the NOP with the instruction currently stored in 0x20, i.e. SLLi R3 R3 01, as this needs to be executed every time anyway. So in total, we can get a valid program by increasing the Instructions executed by 9 and the clock cycle by 9, so 44 and 62 respectively, for Delayed Branches.

*Part 3*

The data dependency is on the first BNEZ instruction, as it requires reading from the result of R2 from the ALU in order to judge when to break from the loop. We can use this, as the JUMP instruction at the end of the loop always jumps to this instruction (and as a result, causes a stall in the pipeline for every loop). So, if we remove BEQZ R2 E4 at the start of the program, and replace the J 0E with **BNEZ** R2 E4, then we can achieve the same result, shaving off 4 1-cycle stalls from the execution. As a result, this reduces both the instructions executed and number of clock cycles down by 4 each, resulting in 34 and 46 respectively.