Computer Architecture — CSCE 3301

Project 2: femTomas

Tomasulo’s Algorithm

Names & ID’s :

Noor Emam 900222081

Omar Saqr 900223343

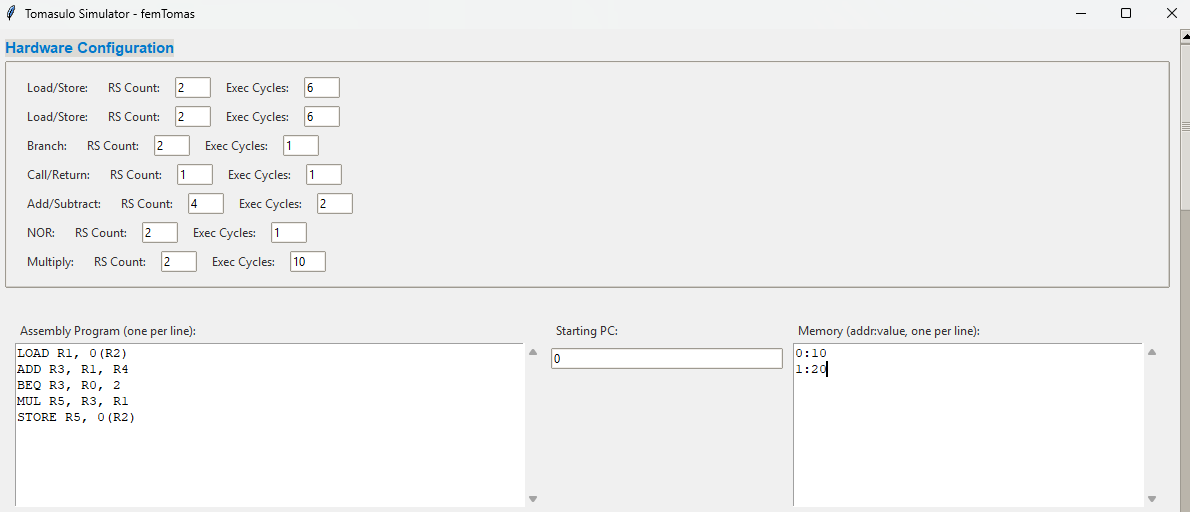
Aabed Elghadban 900223106

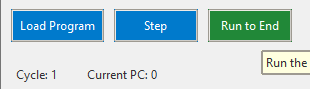
Introduction:

Our code implements/simulates Tomasulo’s algorithm, which is a hardware algorithm mainly used by common ancestor processors to allow for superscalar dynamic scheduling out-of-order execution processing. The code has implemented various features, and these features have been checked rigorously. All features are implemented correctly; however, except a few, which will be discussed in the limitations section related to our code simulator.

Bonus Implemented:

We have done two bonuses. The first bonus was related to an educational GUI that would allow support for “step by step” execution in order to facilitate the understanding of tomasulo’s algorithm for aspiring students and even computer architects eager to learn more about the algorithm. Secondly, we also implemented a variable–configurable hardware reservation status simulator that users can use to choose their own configuration of the algorithm. Here’s a screenshot proving that we did both. Please note that the screenshots down below will also be considered as testcases/small demo of our project.

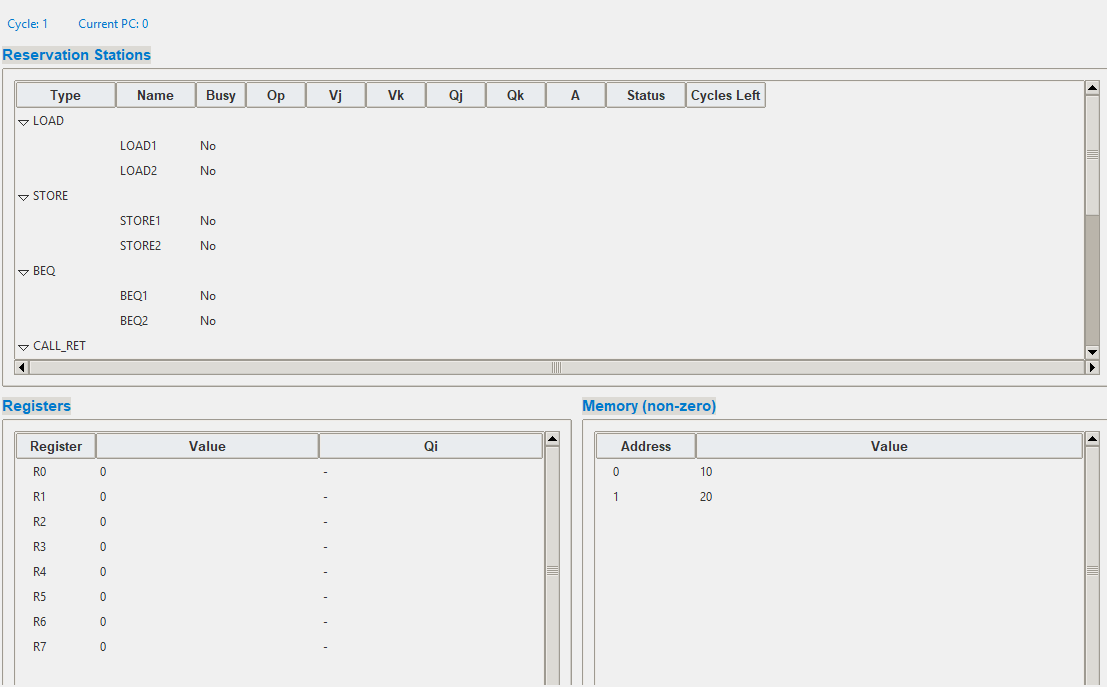


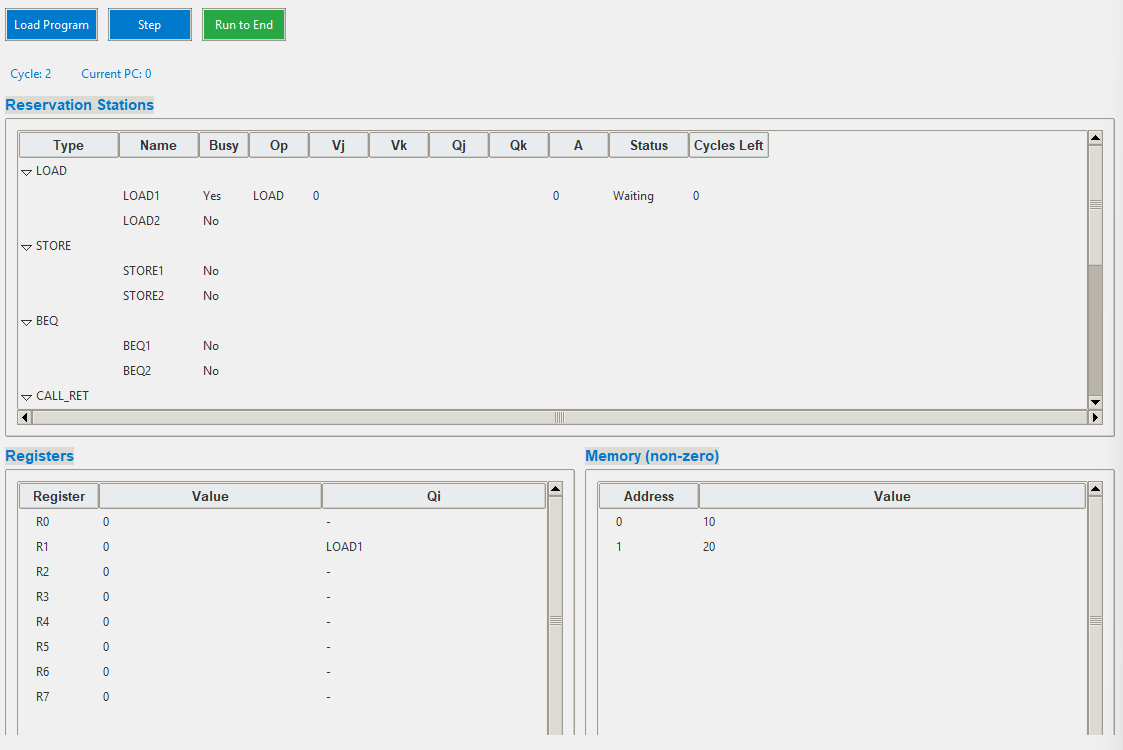


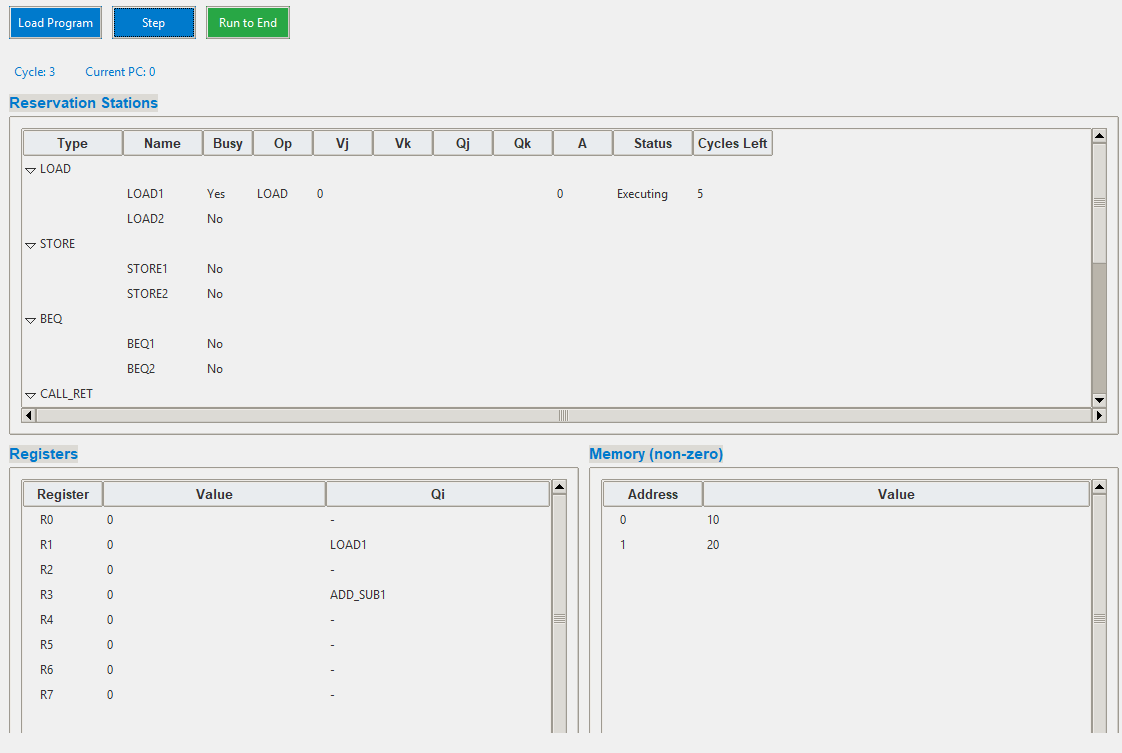
To execute the program, you should type in the assembly program in the format provided in the snapshot, along with the memory (address : value) to indicate if there’s any needed value that you assume to be stored in case of a load instruction.

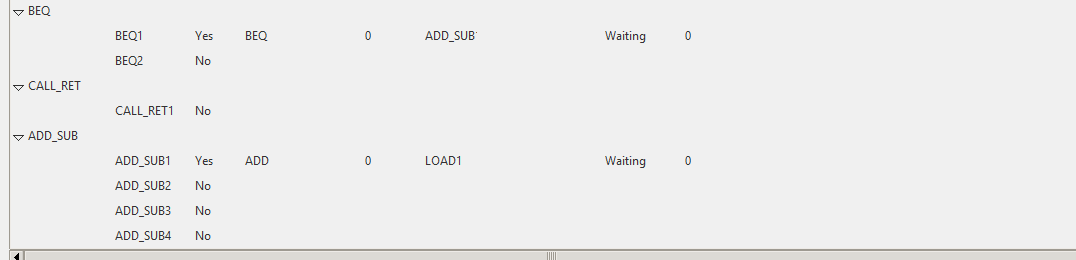
You have two options after executing the program, you either do a step, or run to end. Step option means just executing the instructions clock cycle by clock cycle. Run To End means just executing them all at once and giving the final result.

To demonstrate, we will do the step option.



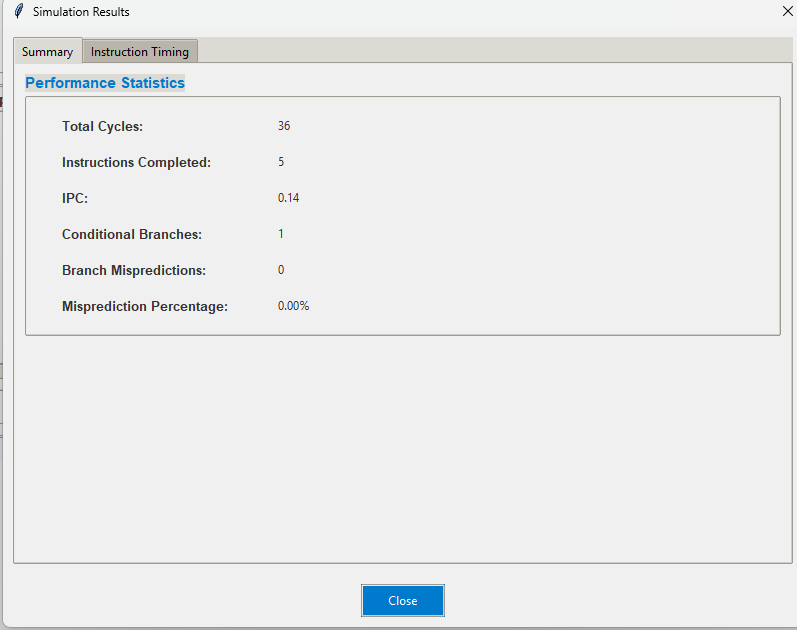


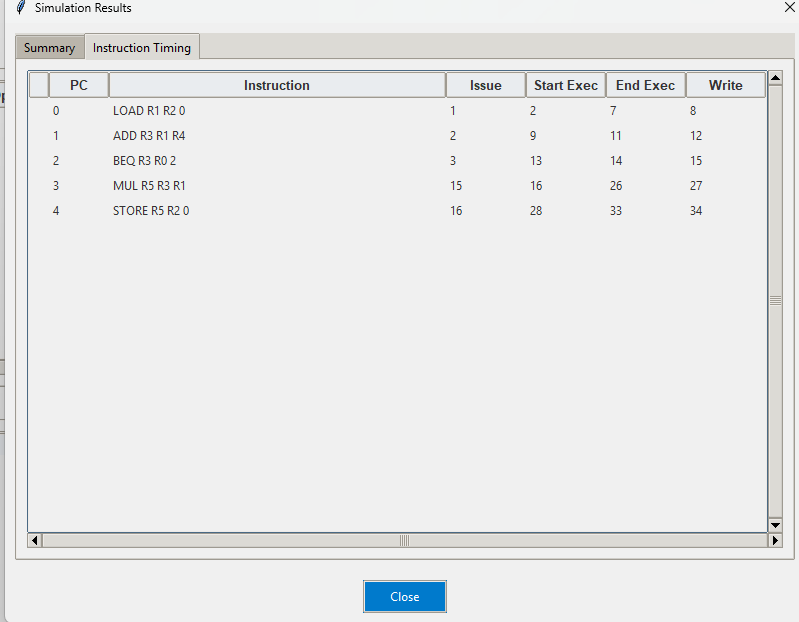




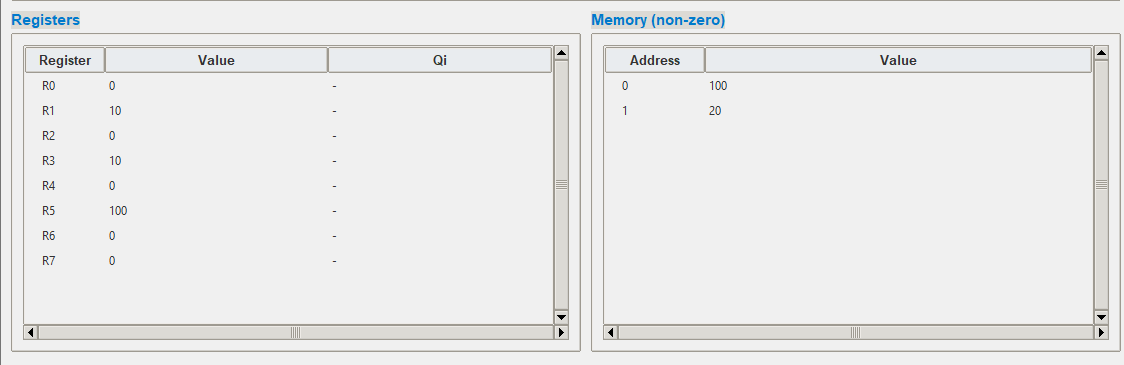
And so on. Each time you press on step, it will show you the reservation table and the register status table at every clock cycle.

There’s also the run to end option, which you can press to give the results finally.





Final register status table and memory.



As done by the program, the final results will be in the fact that the

LOAD R1, R2, 0

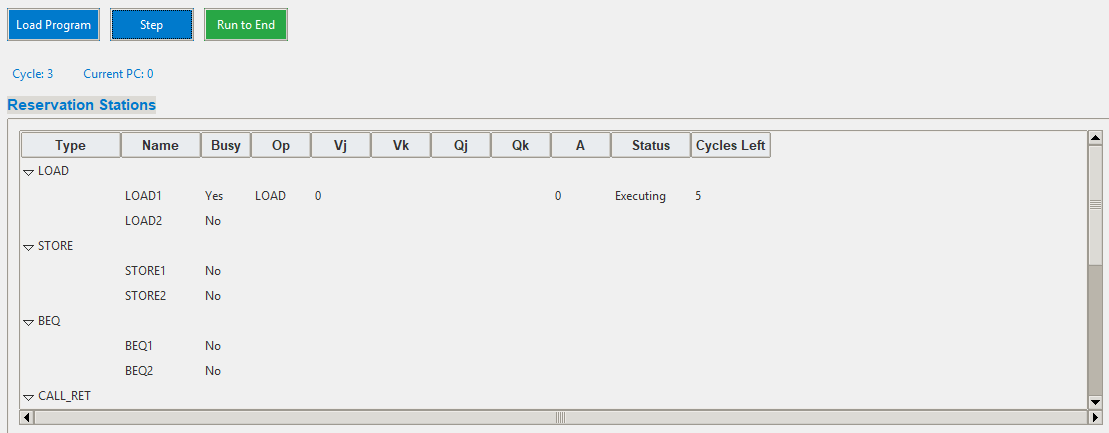
ADD R3, R1, R4

BEQ R3, R0, 2

MUL R5, R3, R1

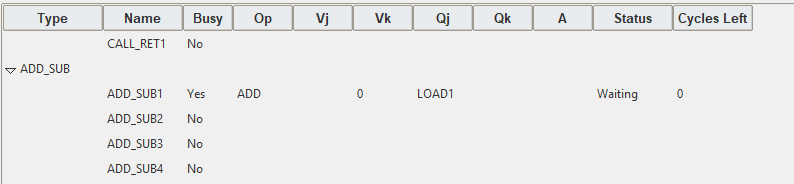
STORE R5, 0(R2)

so , load R1,R2, 0 will be issued in the first clock cycle. Then it will start executing in the second clock cycle



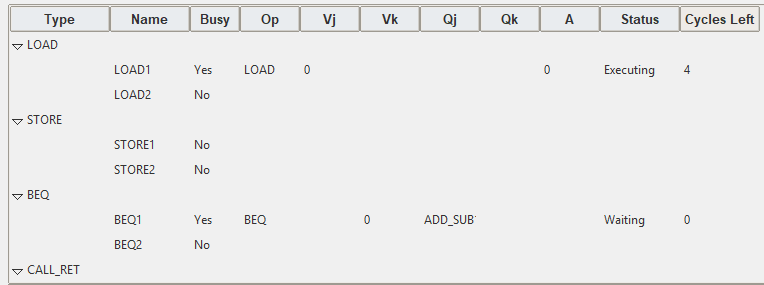
And it will keep on the executing status until after 5 clock cycles.

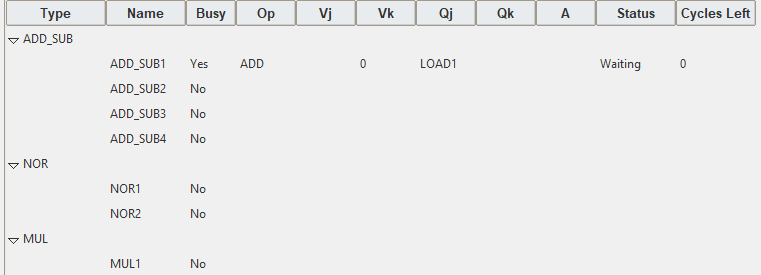
Clock cycle 2:



Add is issued and load1 is executing.

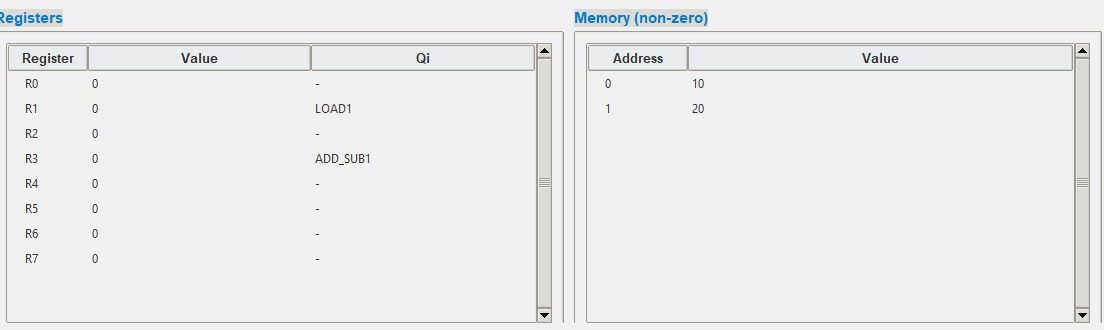
Then at clock cycle 3:





The third instruction (beq instruction) is issued.

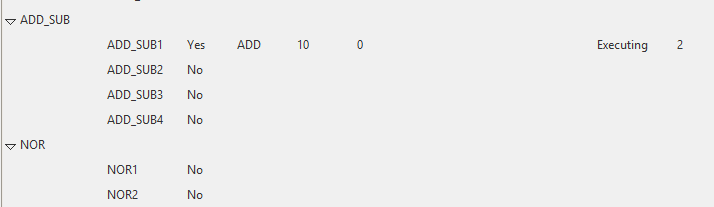
Clock cycle 5:



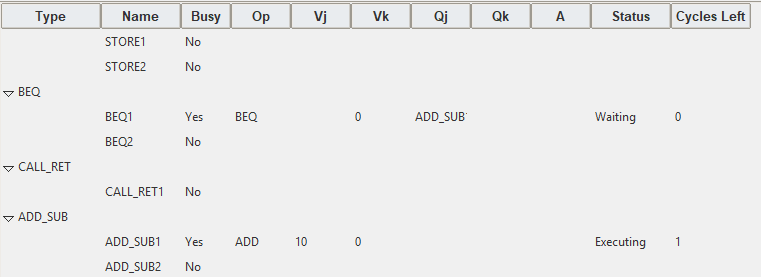
This is the register status table and the memory, as we can see, the LOAD1 is still being used as the LOAD instruction takes up 6 clock cycles to finish executing, and add\_sub1 will take 3 clock cycles to also finish executing.

Since there’s a RAW hazard between the add and the beq, then this means that beq can be issued but not executed as it would need the value coming from add\_sub functional unit. However the value coming from add\_sub would also depend on the add\_sub being executed as add\_sub is still in the waiting stage. Remember that there’s a RAW dependency between the three instructions, hence, the ADD should wait until the value coming from the LOAD1 comes out, and then the ADD will start executing.

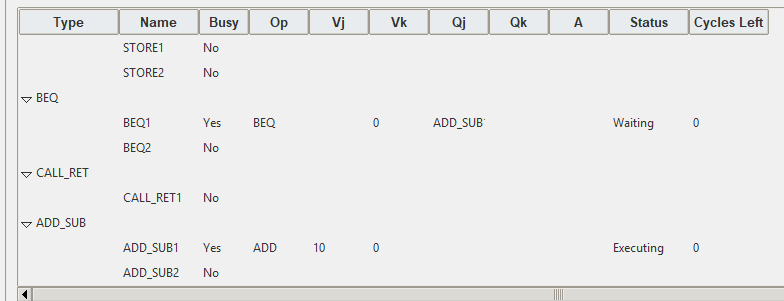
Fast forward to clock cycle 9:



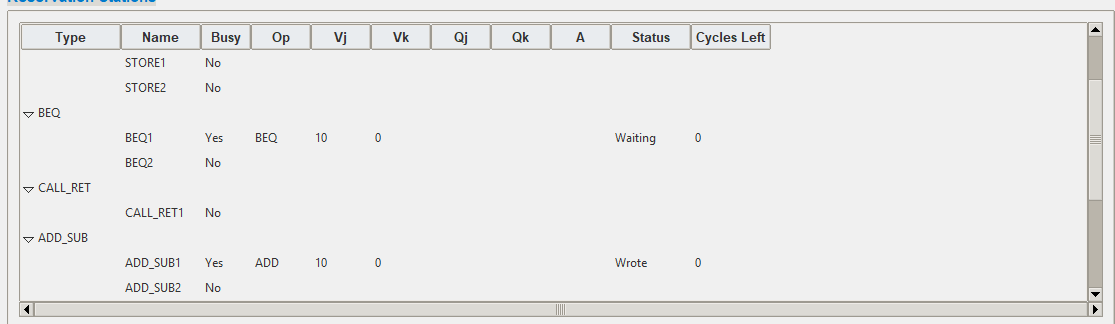
Add\_sub unit is now in the execution status, this is because it already received the value coming from the first instruction.

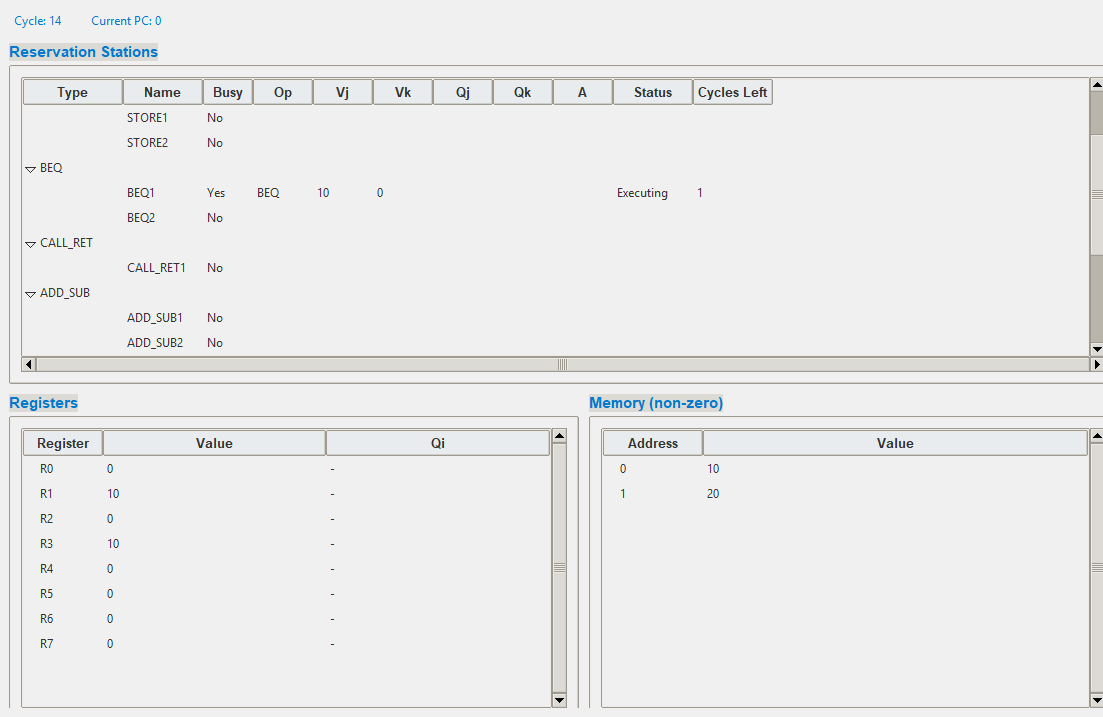


Fast forward to clock cycle 11:

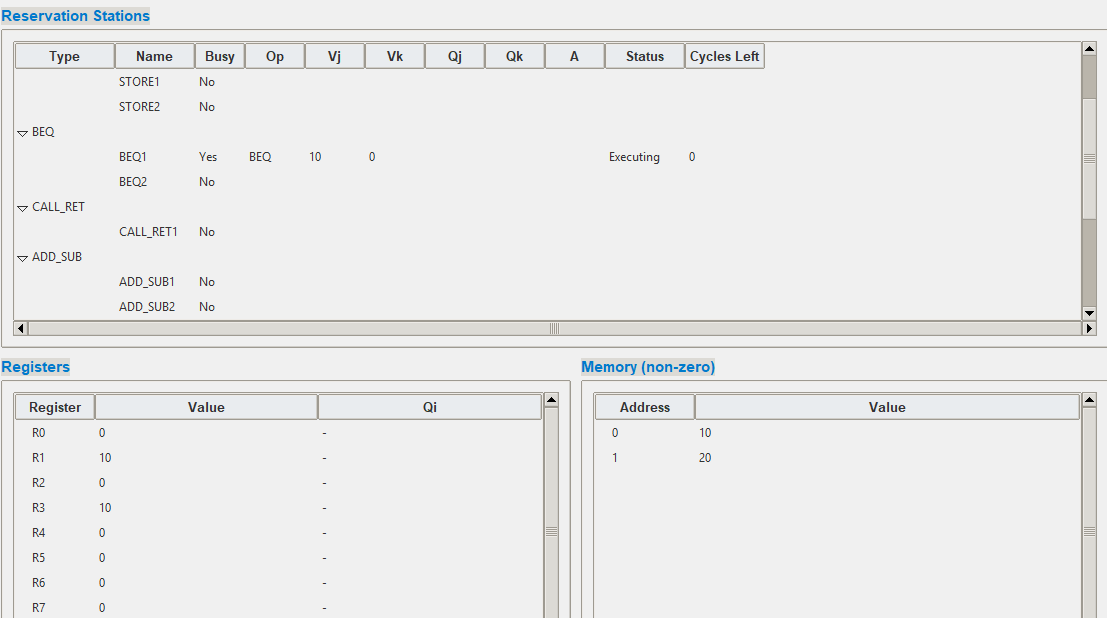


The ADD\_SUB unit finished executing, now it should write back in the next clock cycle.

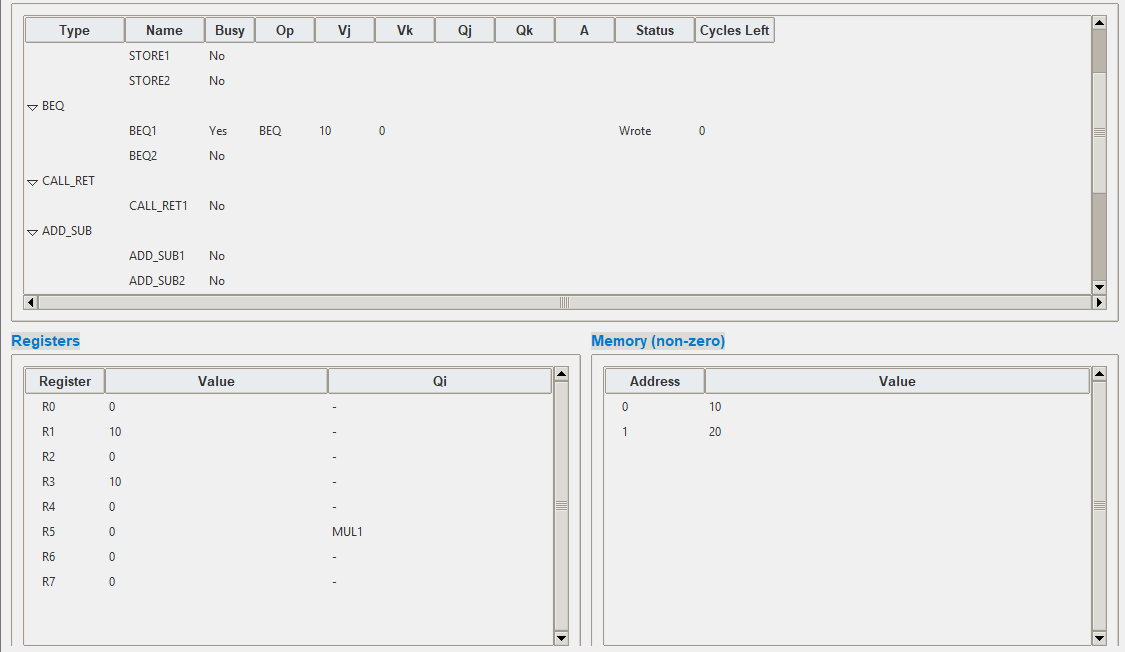




Clock cycle 14:



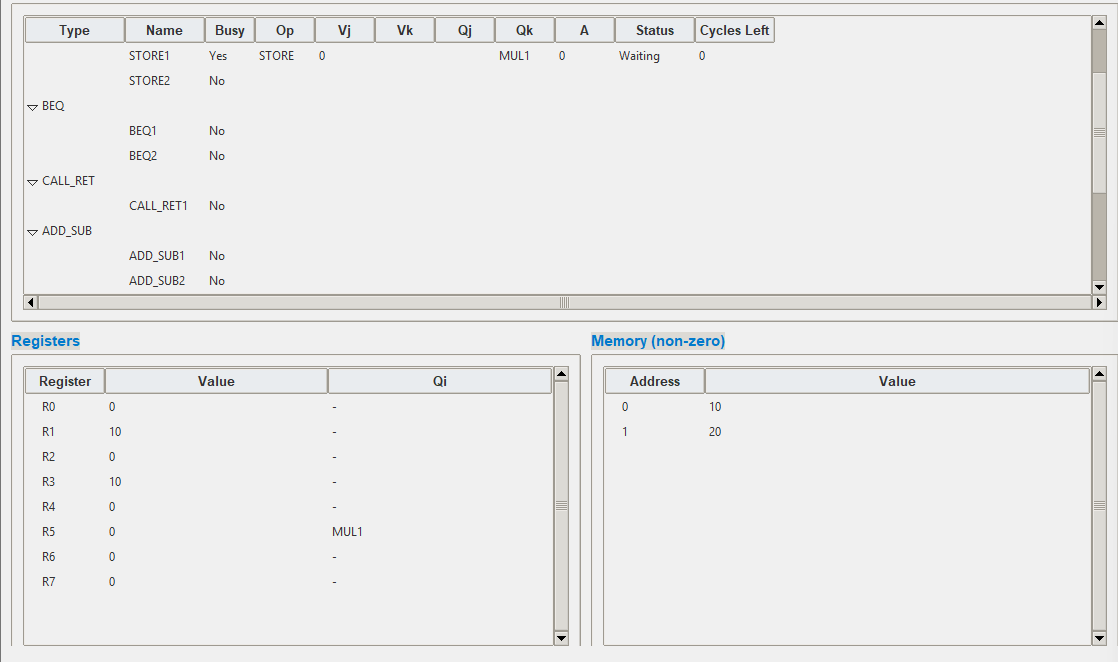
Clock cycle 15:

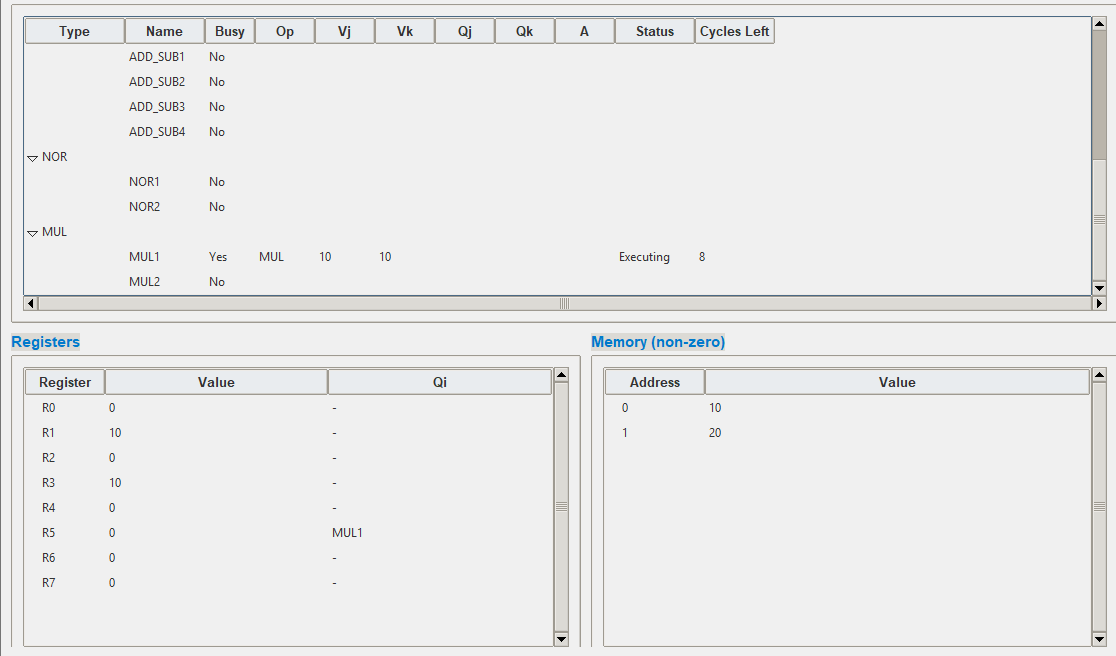


Now, the branch has computed the value; however, in the given project description, we know that we always predict not taken, and this is actually correct, as our program is designed for the branch to not go to the last instruction as the two registers are not equal to each other.

Now, the multiplication instruction starts.

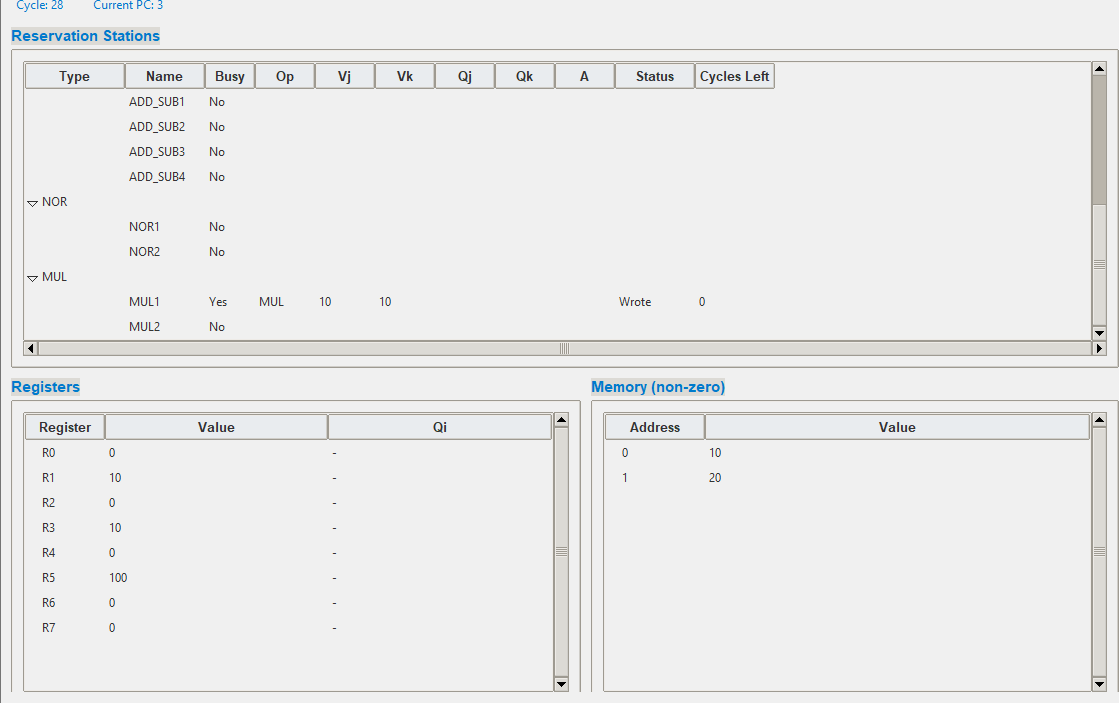
Fast forward to clock cycle 18

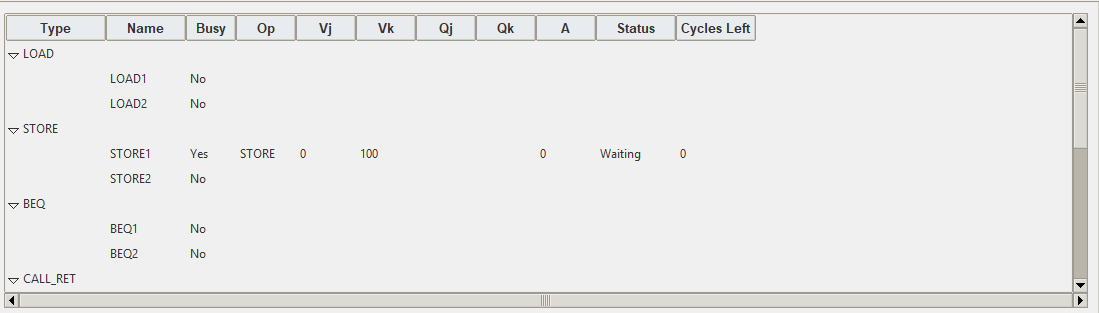




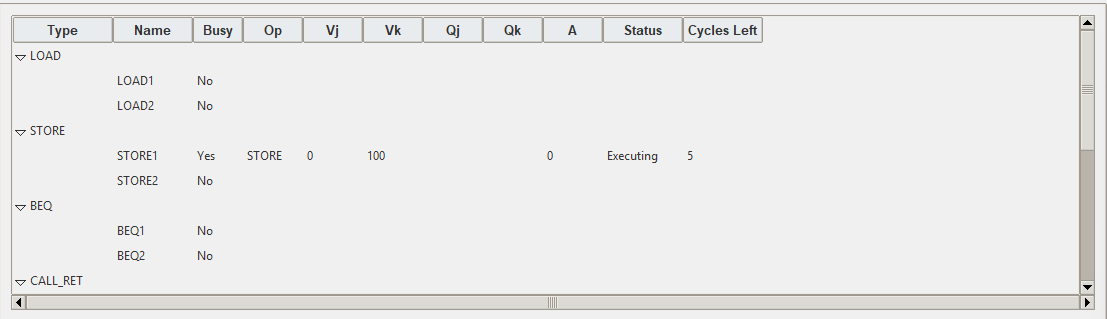
The store would not store as it would wait for the value coming from the multiplication instruction.

Fast forward to clock cycle 27

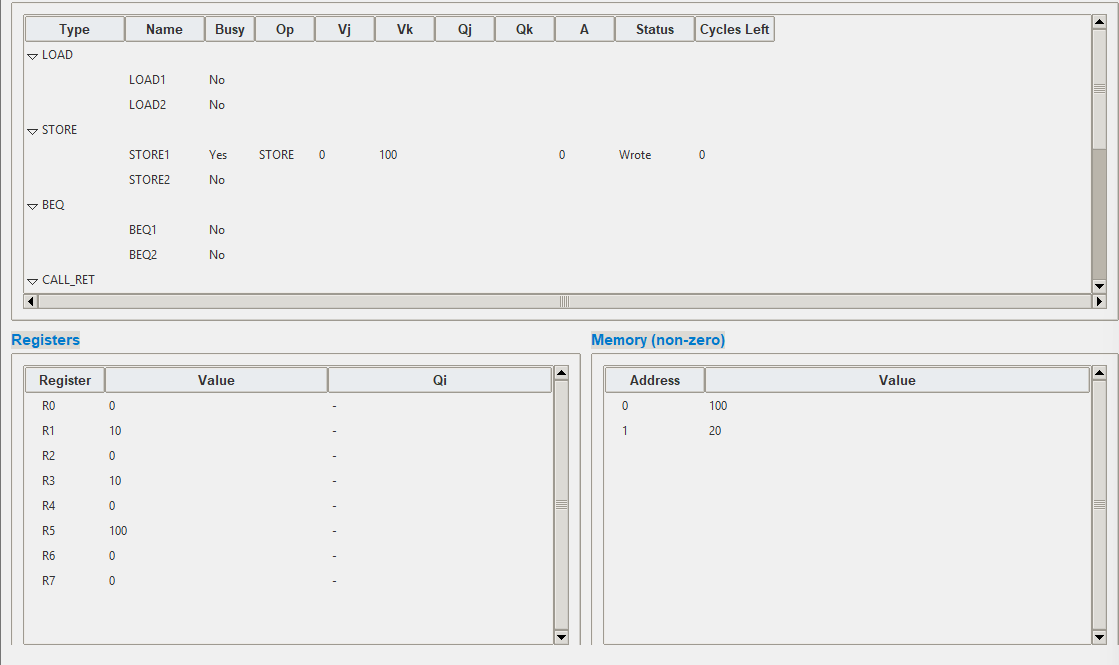




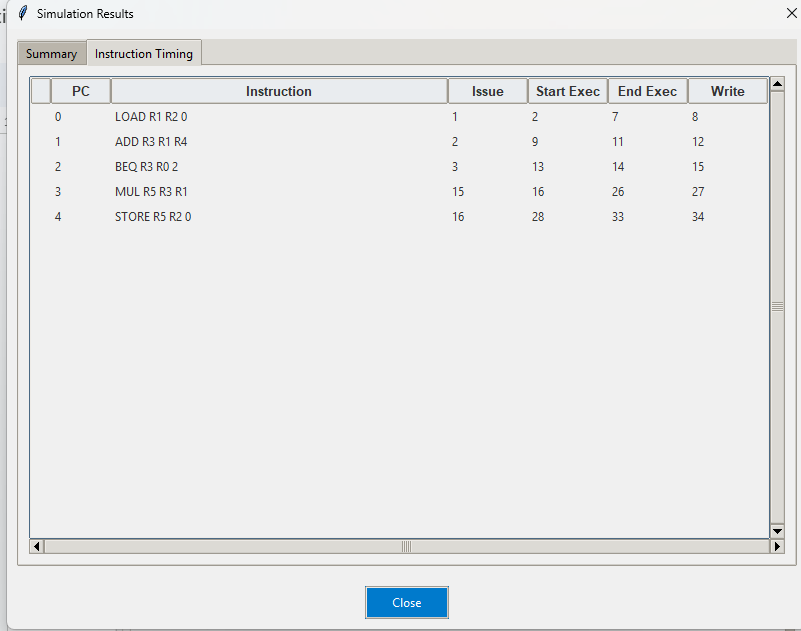
Clock cycle 28



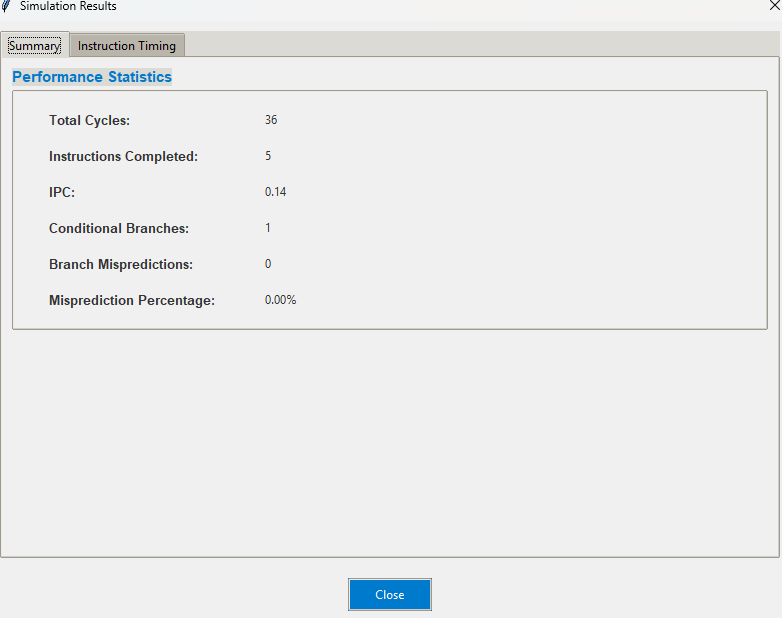
Fast forward to clock cycle 34



This is the final result:



And this is the summary:



Now this would do the same thing for the others; however note that in our program, we did not implement features for dealing with load-store hazards, and hence, this is one of the limitations of our program. Also, concurrent writing requests could cause un-expected behavior depending on instructions that are written.

AI:

1-) It was used to decide the programming language.

2-) It was used in the implementation and design of gui.

3-) It helped in the debugging process, and suggesting which parts of the code needed to be fixed.

4-) It was used to structure the code, give some comments, and give variable more meaningful names.