

Design and simulation of a pipeline processor based on the MIPS architecture

Raúl Méndez Álvarez

7/19/19

Arquitectura Computacional

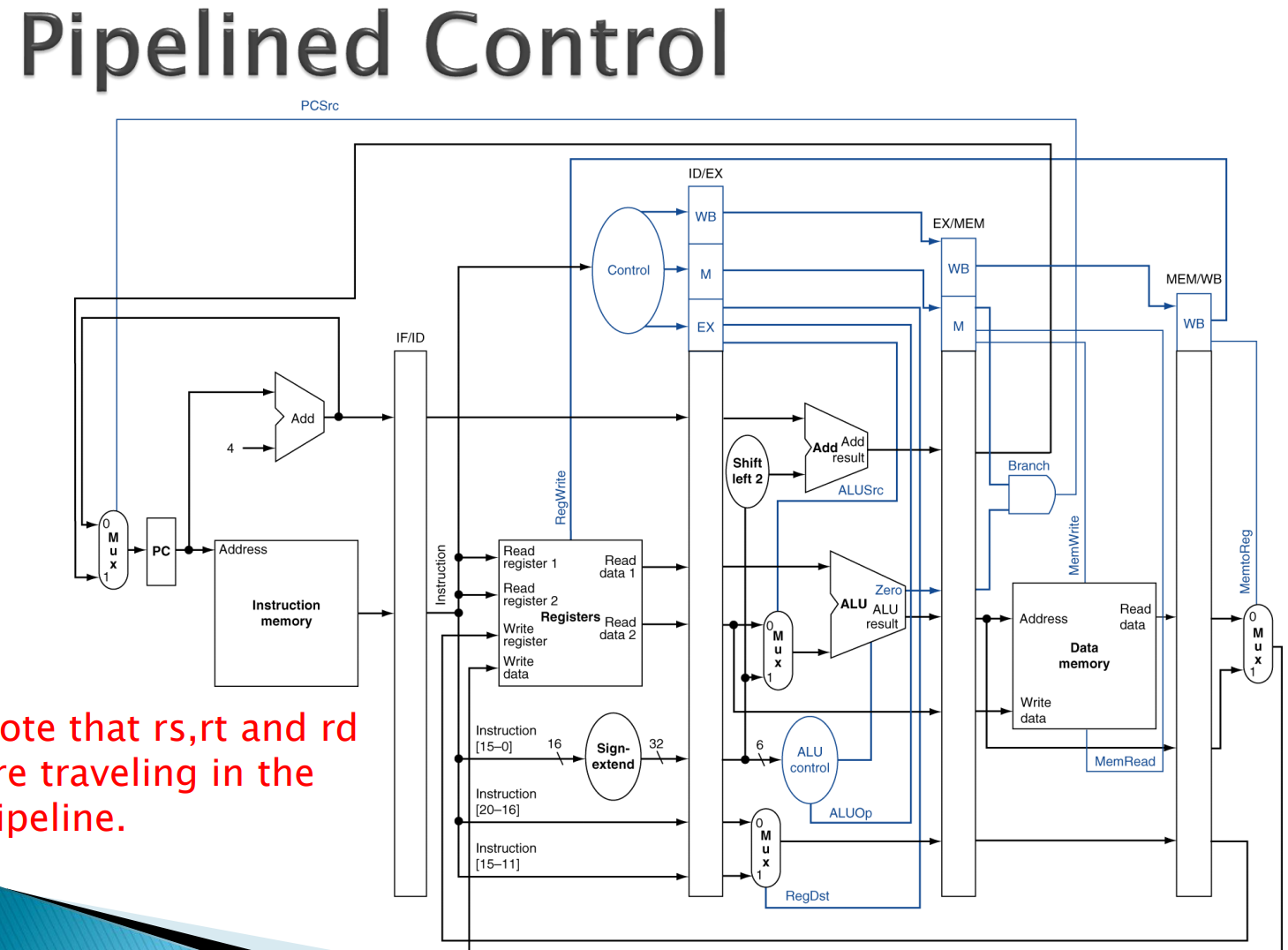
Rodrigo Aldana López

Repository: <https://github.com/RaulMendezA/Practica-3---AC>

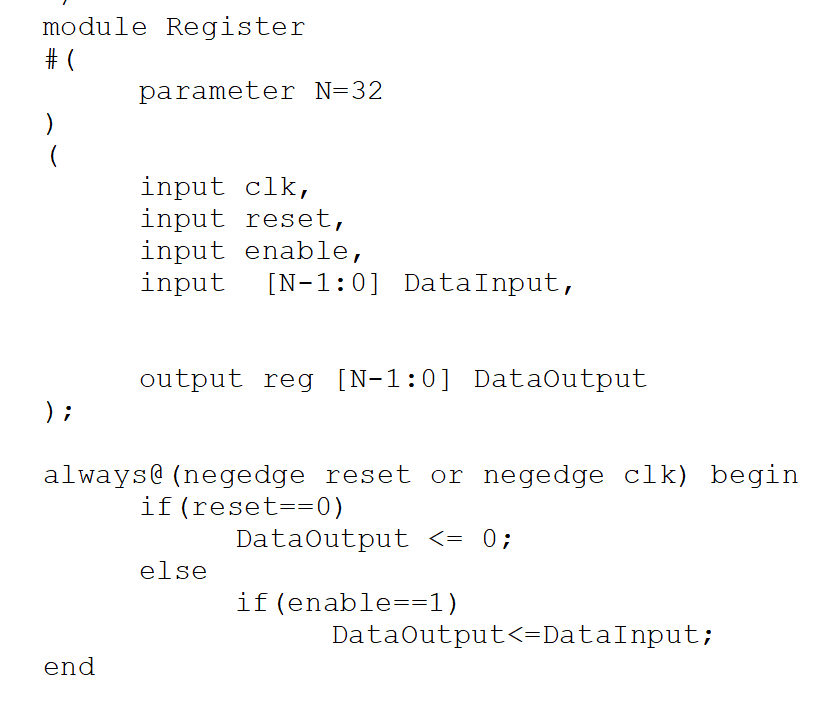
**1 – Design changes and process:**

For the pipeline, I first had to create some sort of internal language or system in order to name each of the new components for each pipeline. For example: ID\_Instruction\_wire\_EX, both for entries and exits, in this way it is easier to realize where exactly is a certain cable; basically, all of the new component had the cable or module type in the middle and its connecting components on both sides.

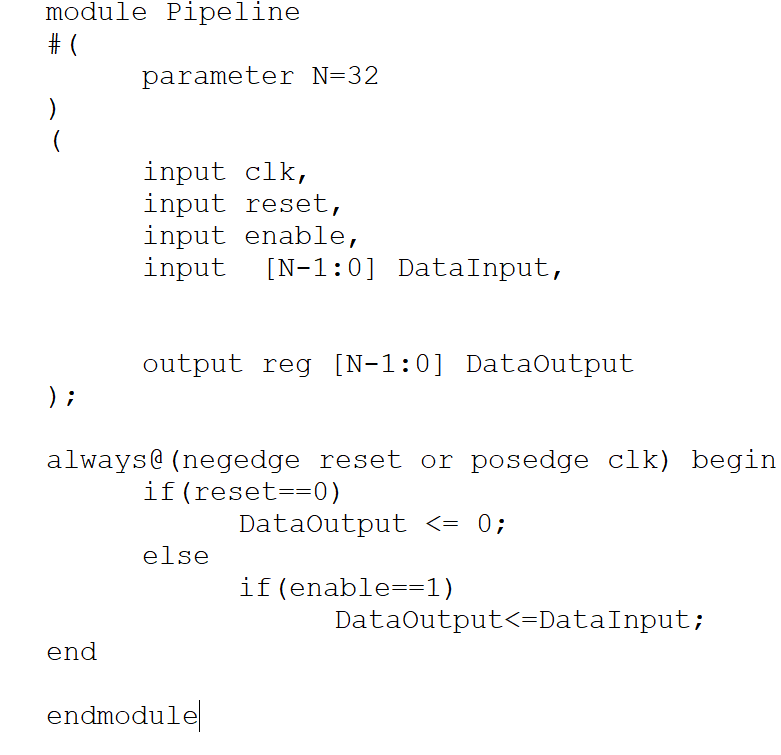
I also made the decision to only pass the fundamental cables through the pipeline so that the forward unit could work. For this, the diagram provided in the presentation "Digital design with verilog HDL" was followed since using a different approach would be a bit confusing.



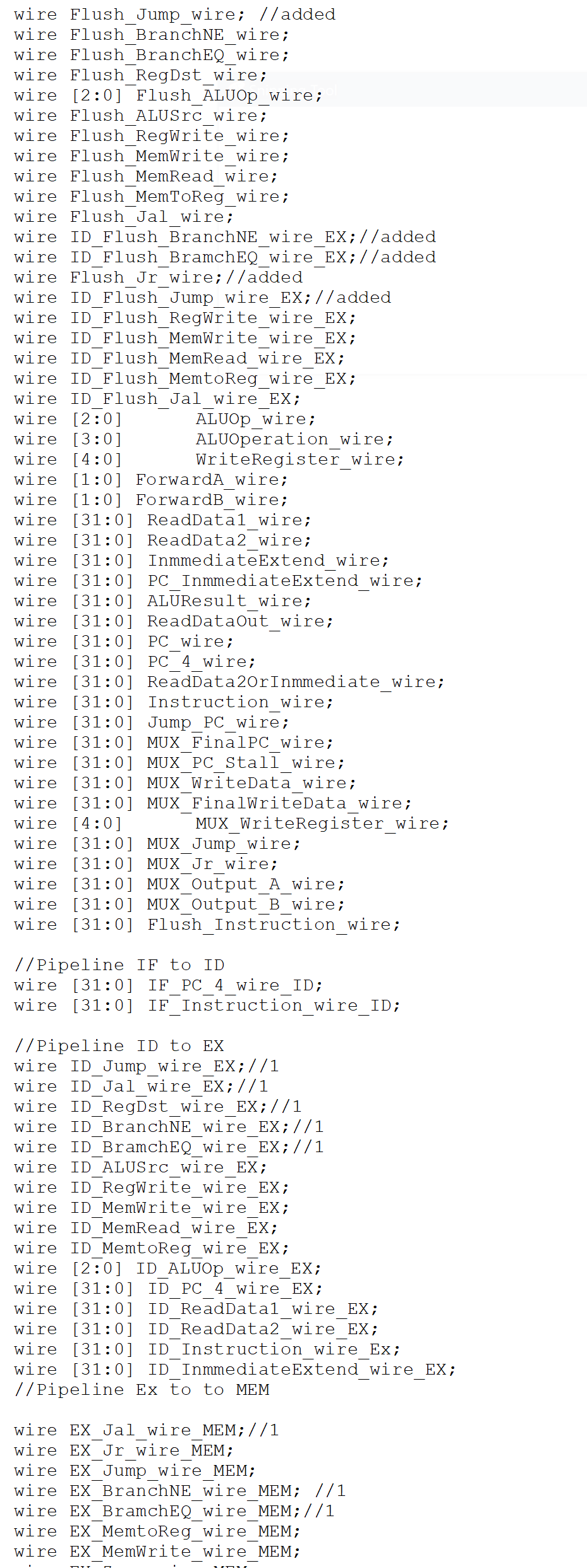
In order to begin migrating my original sincle-cycle model to a pipelined/multi-cycle approach, I had to change the register from posedge to negedge:

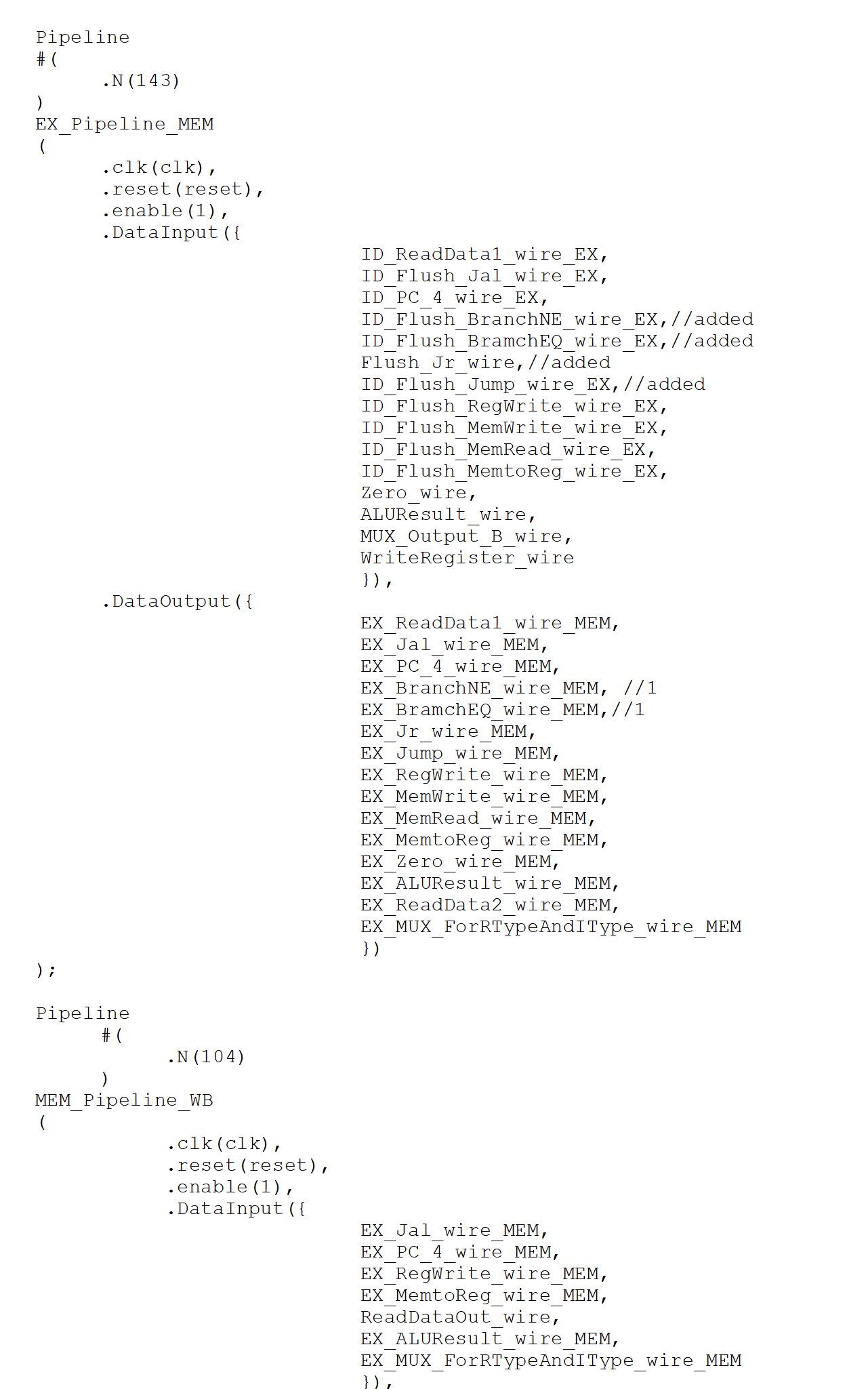


Whereas the new pipeline module created would then be updating on posedge:

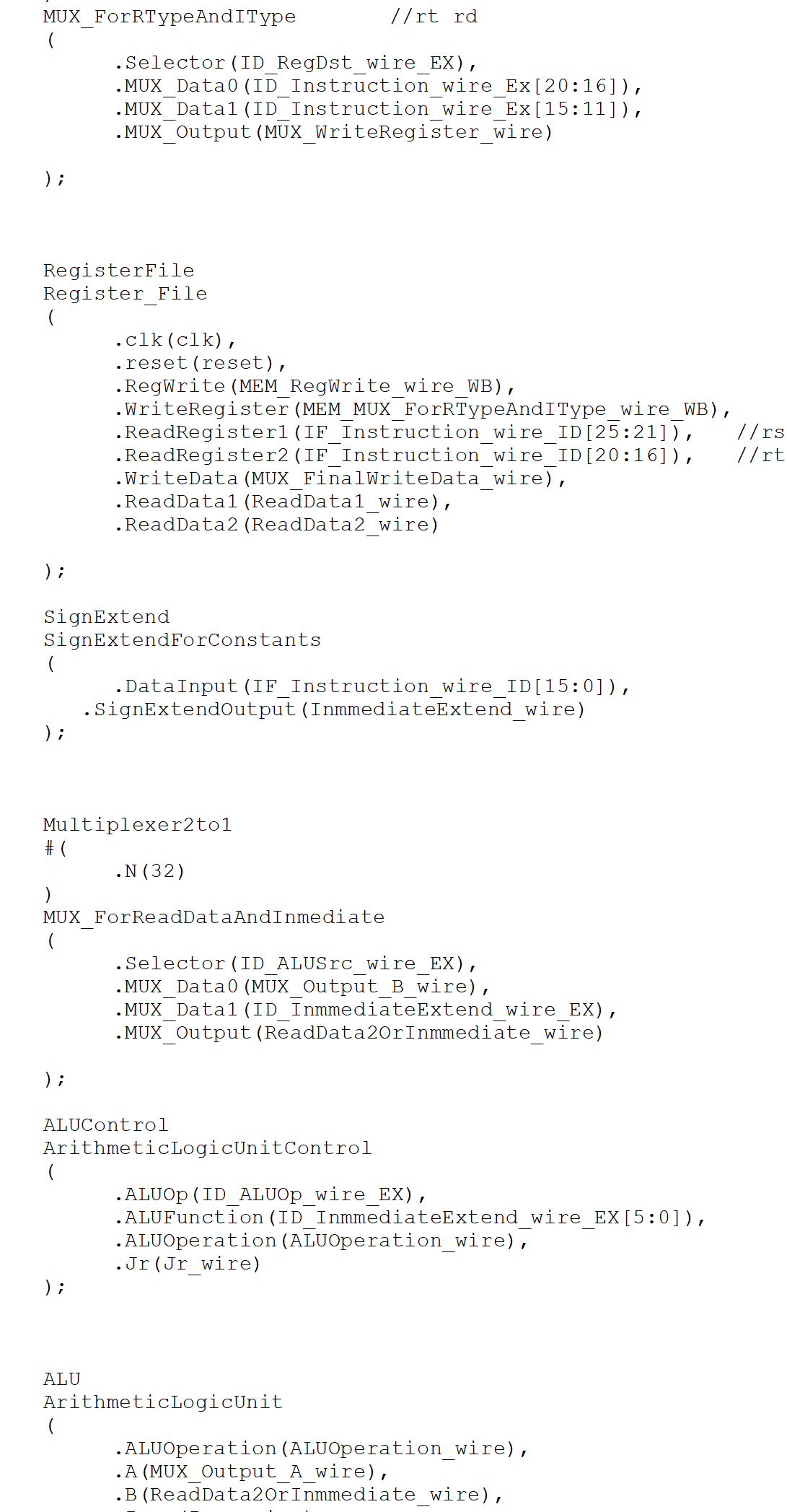


I created the new pipeline registers as well as instantiated their respective wires:

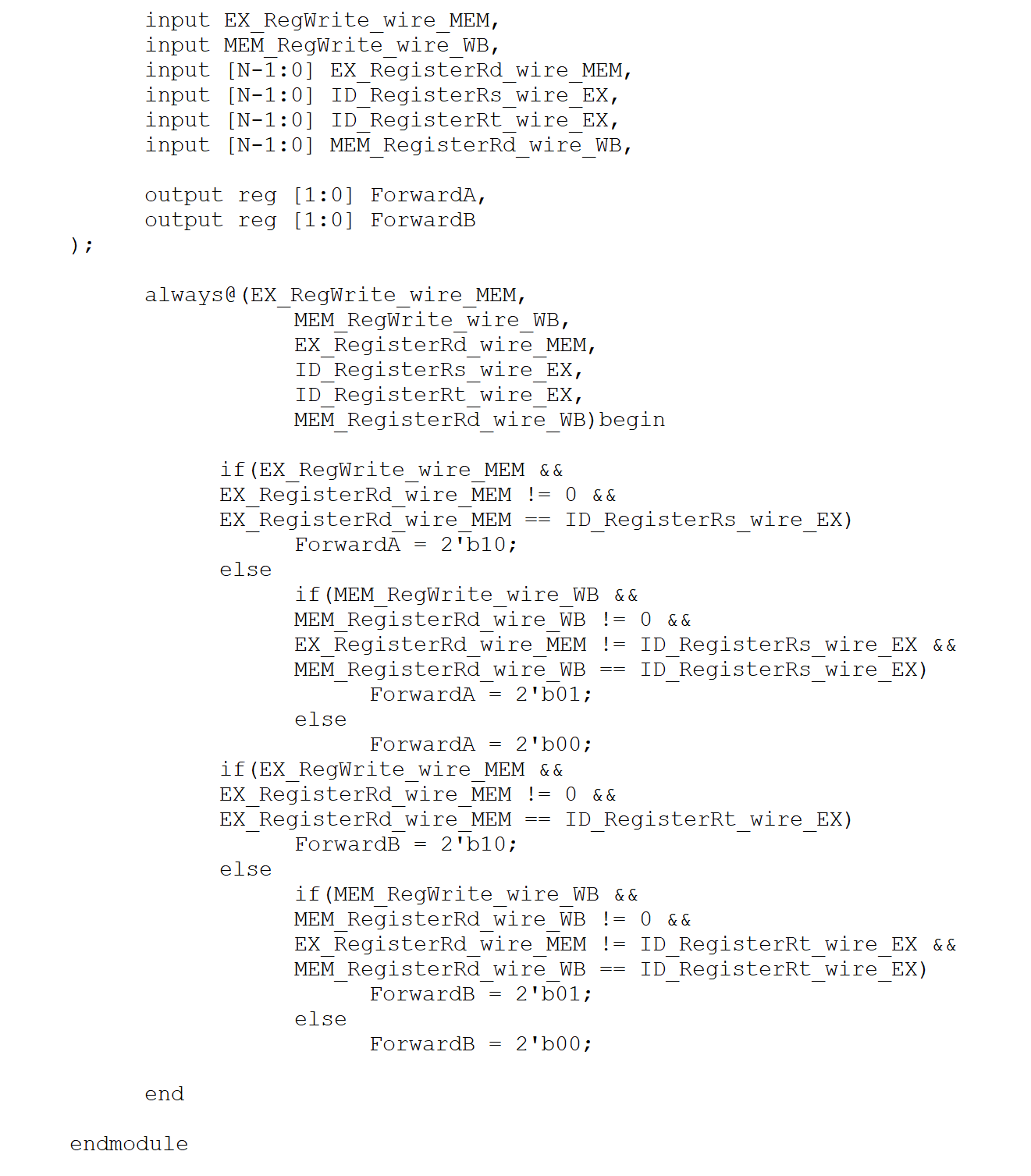


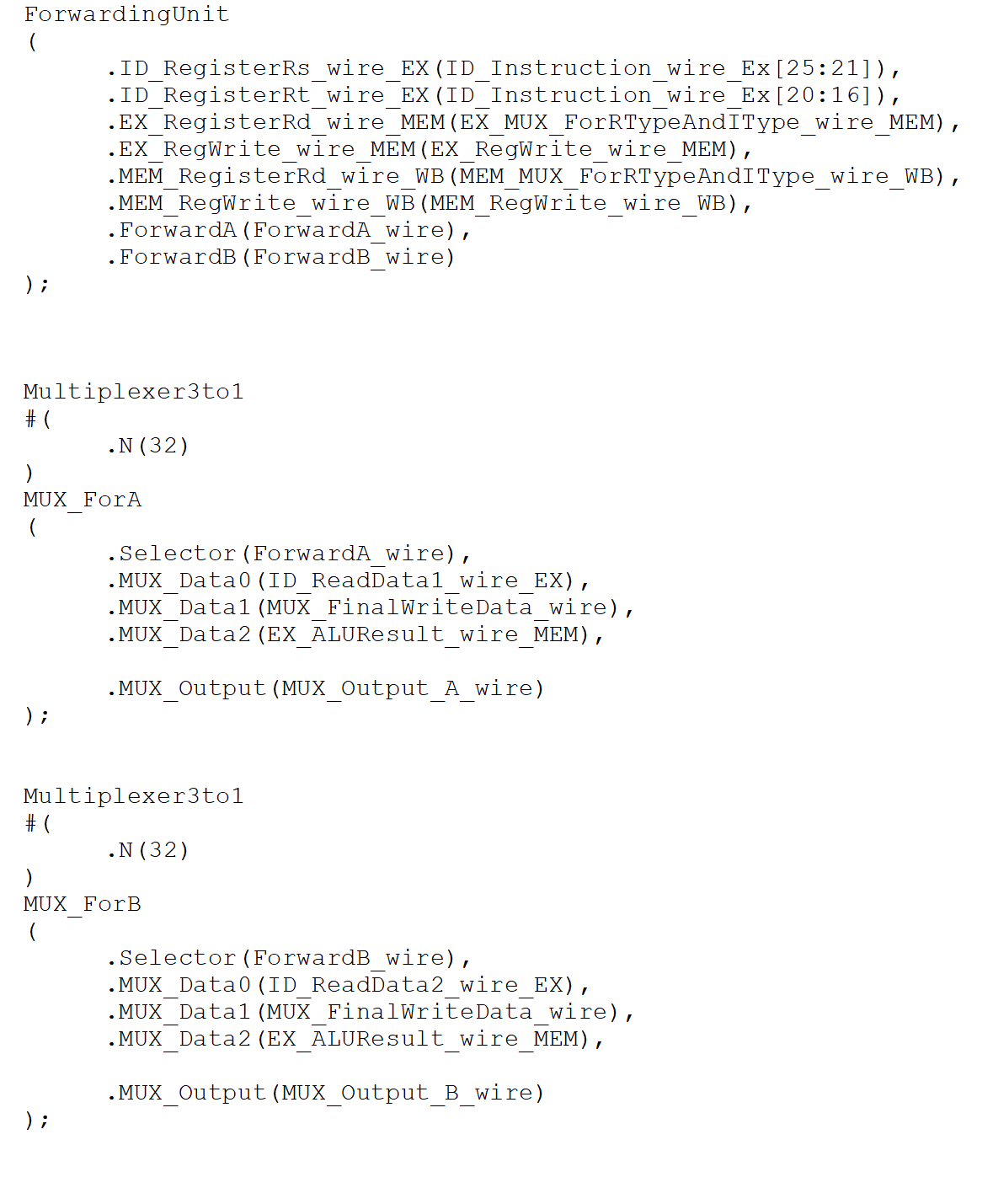


I also had to change all of the previous modules´ inputs and outputs in order to work with the new naming schema/system:

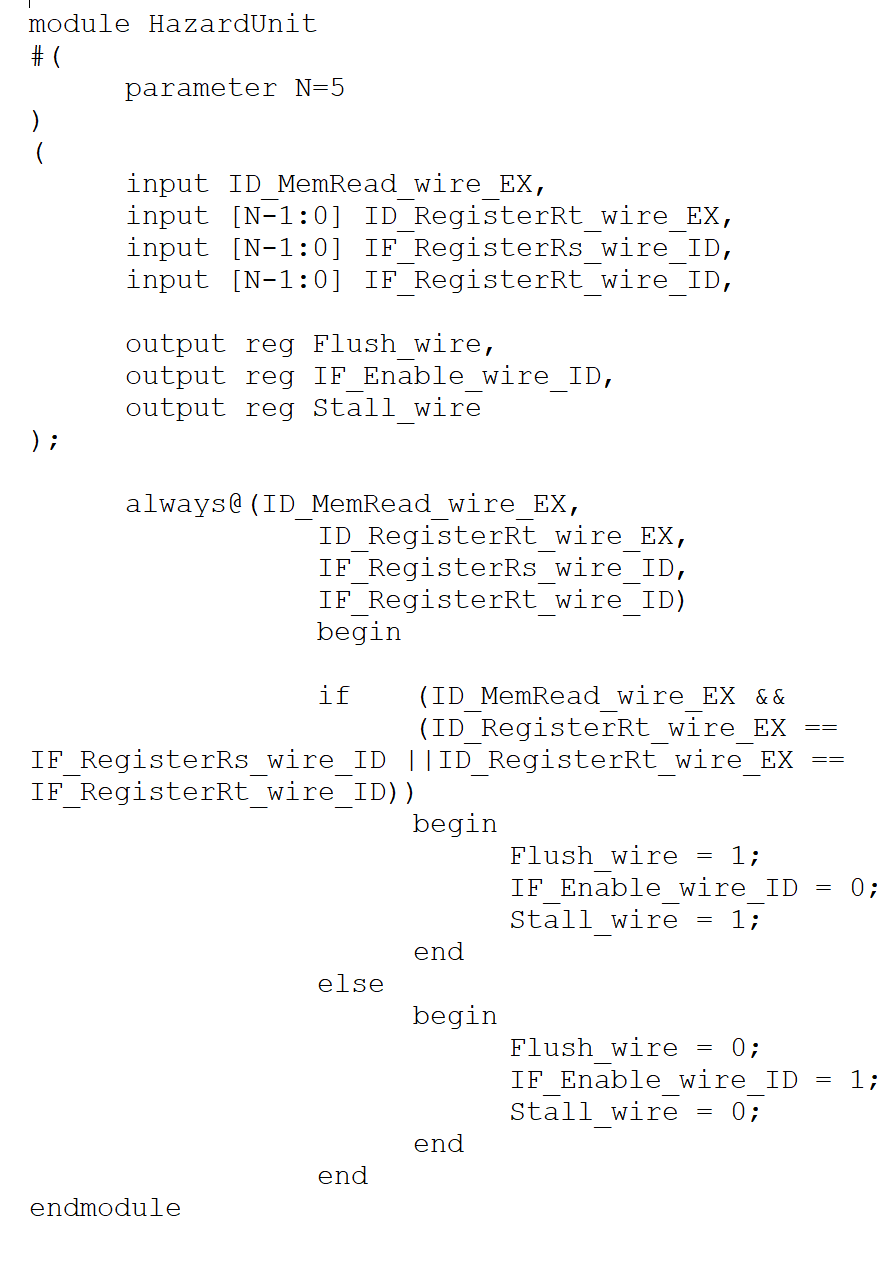


For the forwarding unit, I created a new register and added the combinational algorithm proposed in the course´s presentation:

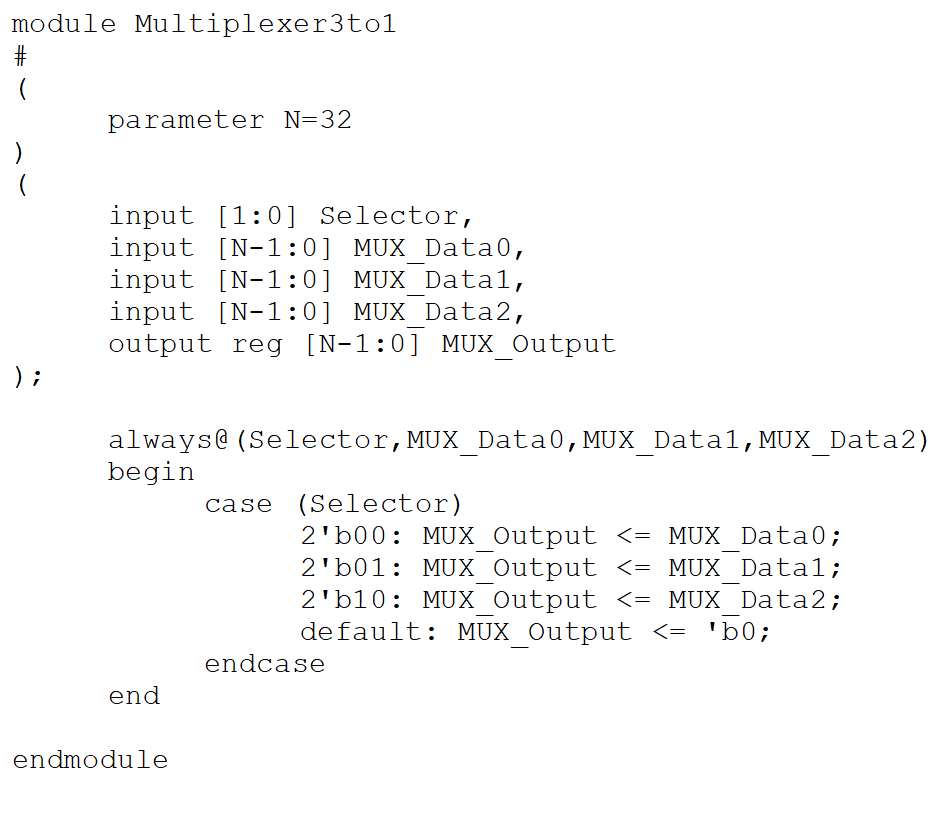


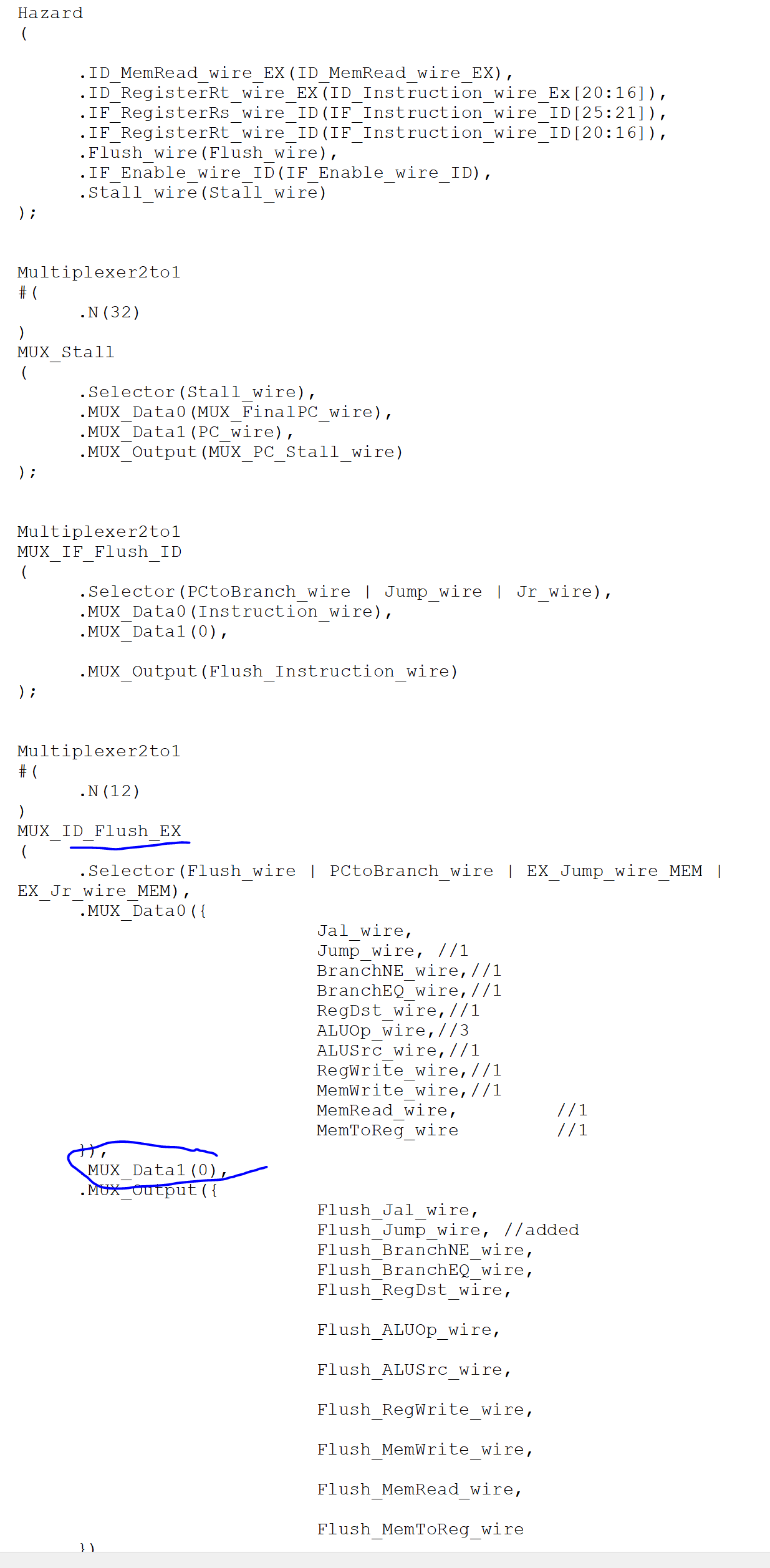


For hazard detection, I created a new register type where the ins and outs would depend on the



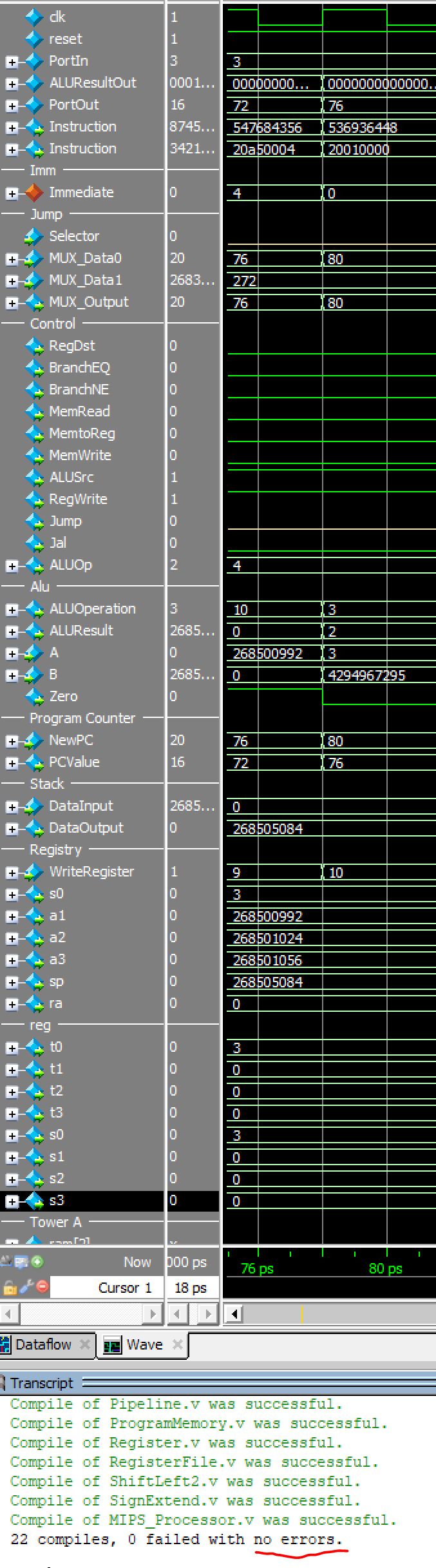
For both of the new HDU/FWU, I had to create a and update new mux types that would either reset a signal after ID or to just pass 0 as a predefined parameter (flush or stall the stage) according to the instruction scenario – for the branch prediction status, I would always assume the branch would not be taken so that a flush mux would then pass a predefined 0 signal to the IF/ID out pipe in order to “cancel” the rest of the conflicting/not needed instruction:





**Test and debugging:**

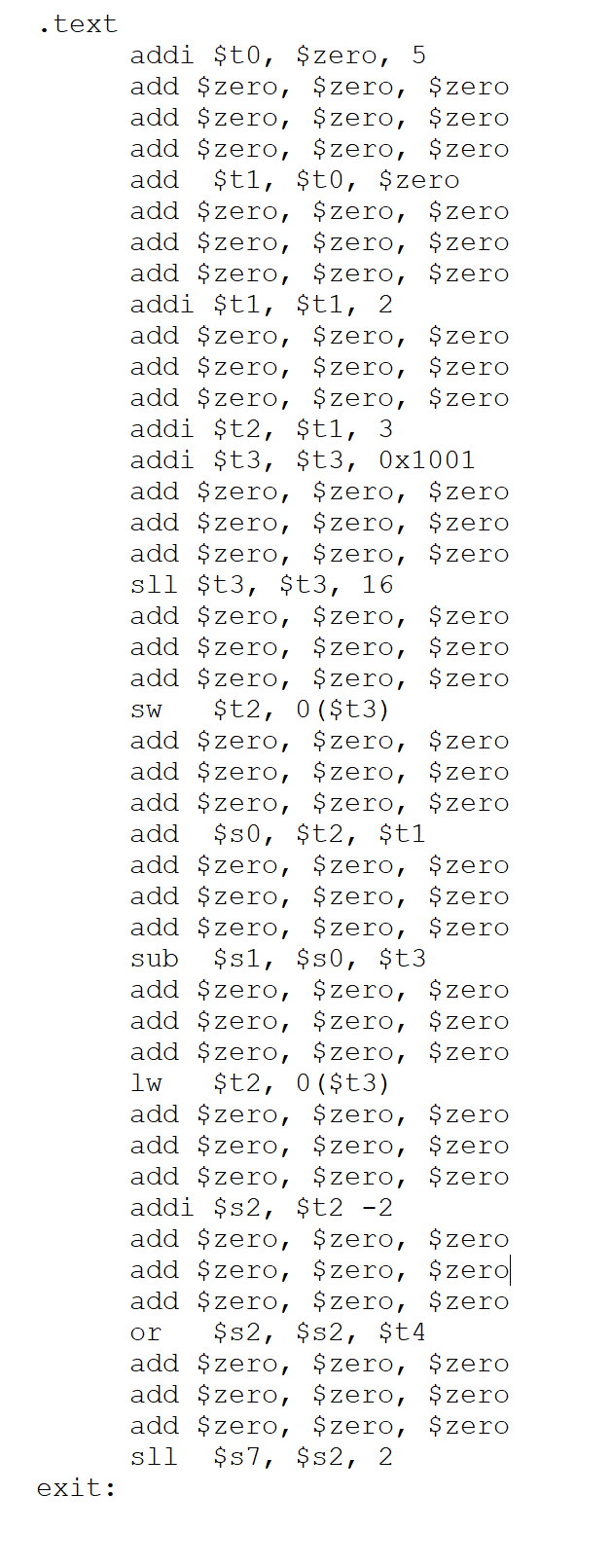
For simulation on ModelSim, I first created a wave viewer/map where I would be adding new registers ins and outs from time to time according to my needs for debugging each stage:



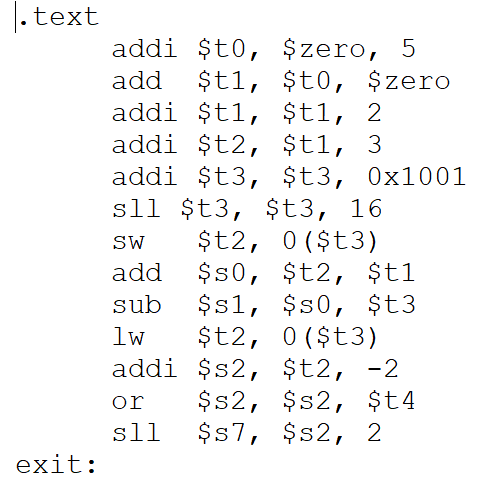
Instruction tests:

I created a set of programs that would test each of the initially implemented instruction at its most basic operation level. All of them are included in the archive.

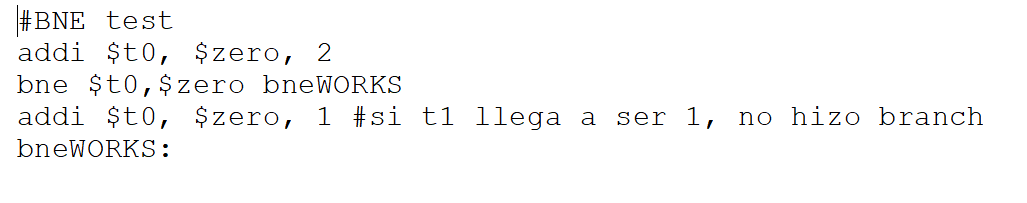
pipeTest (test with nop´s stalling)



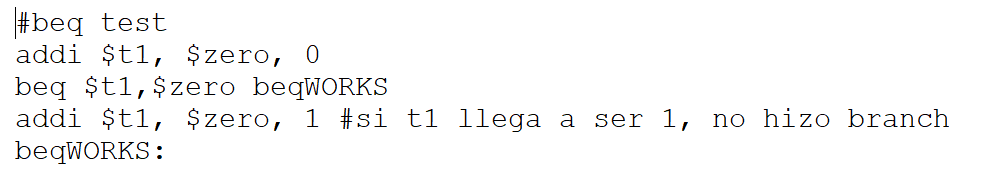
pipefwTest (test no nop)



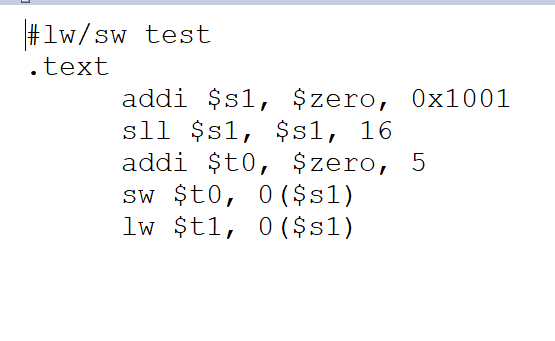
Bne (test flush after ID on conflicting instruction)



Beq (test flush)



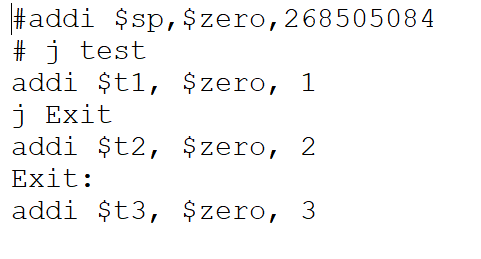
lwswTest

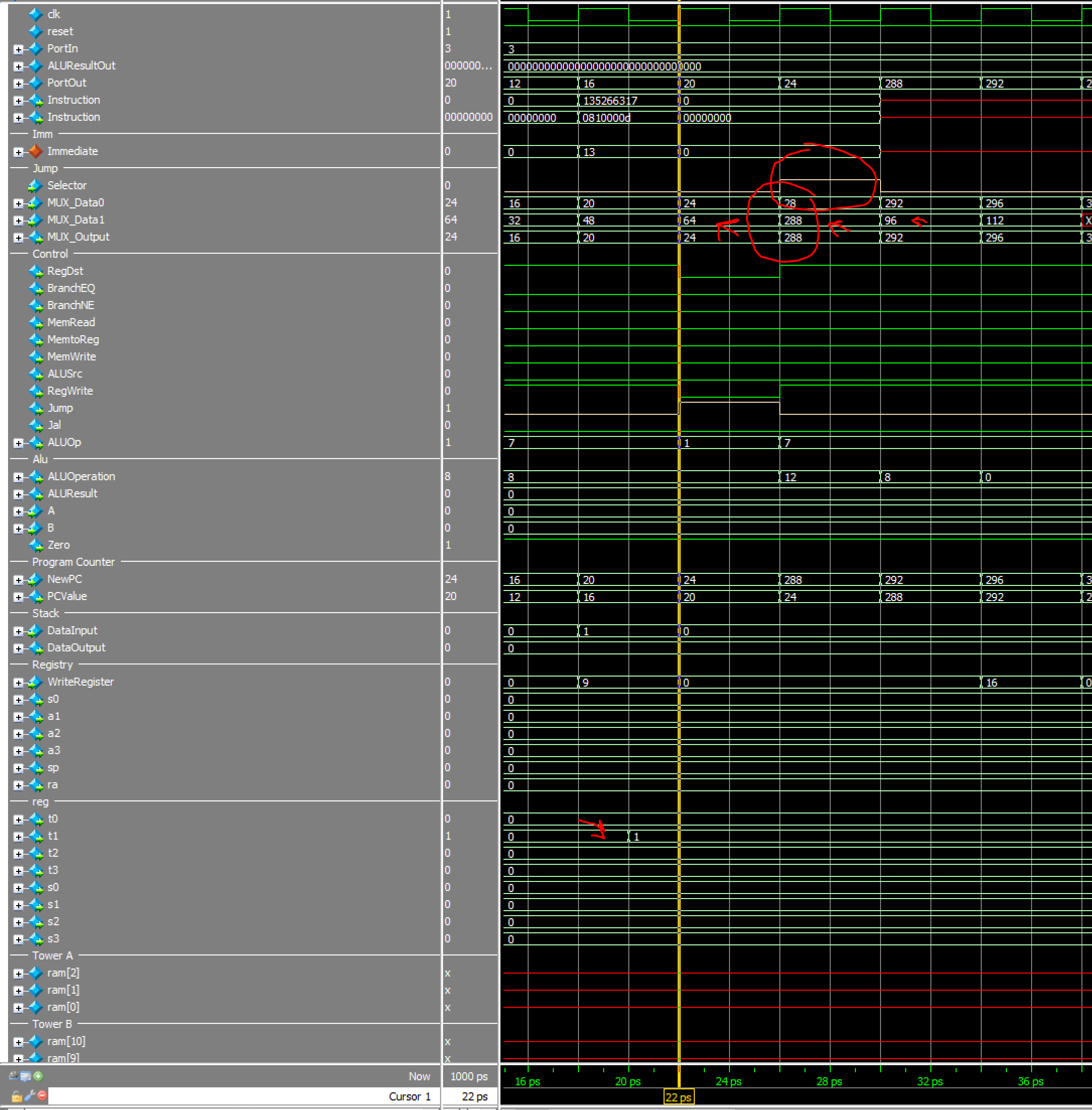


**J types and Hanoi:**

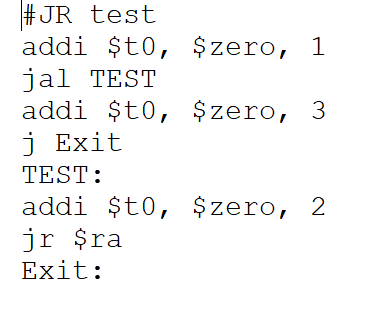
Even though the above test did work/pass, unfortunately, and here´s where I failed, I never seemed to either fix or even understand the J type instructions through a pipeline approach; I couldn’t seem to figure the Datapath that a jump address would have to go through/by in order to go to the stage it is needed at the precise cycle it is needed. And yes, I tried using nop´s…

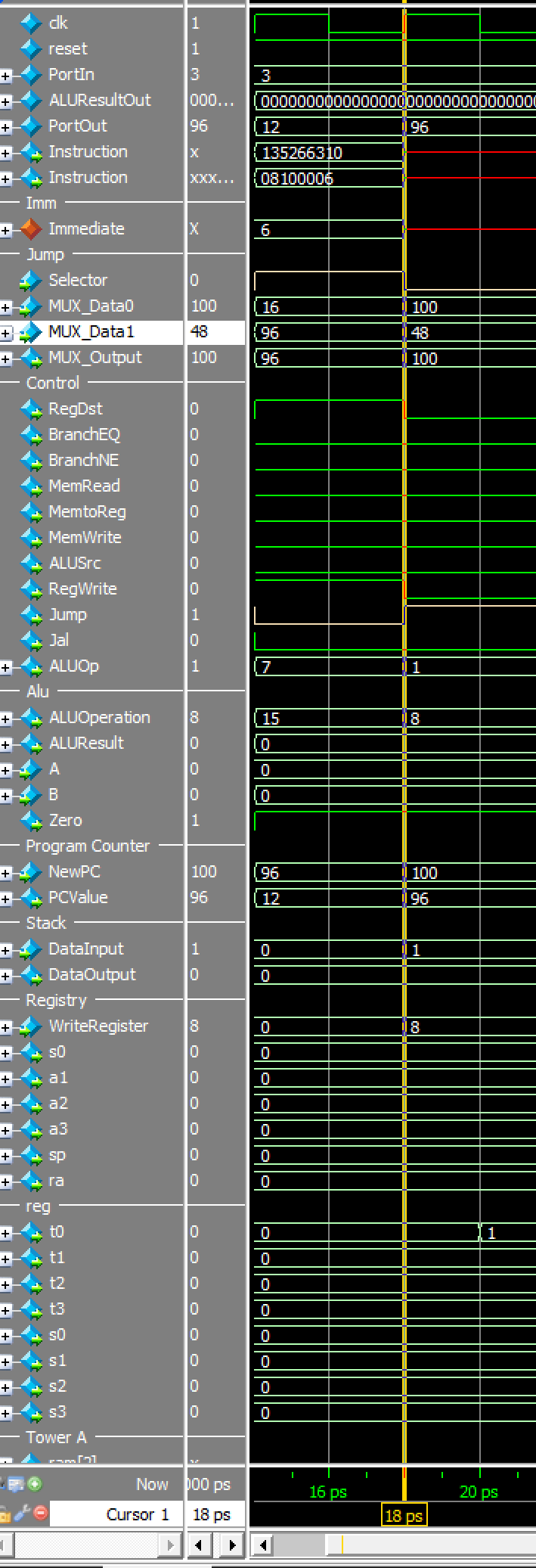
jTest





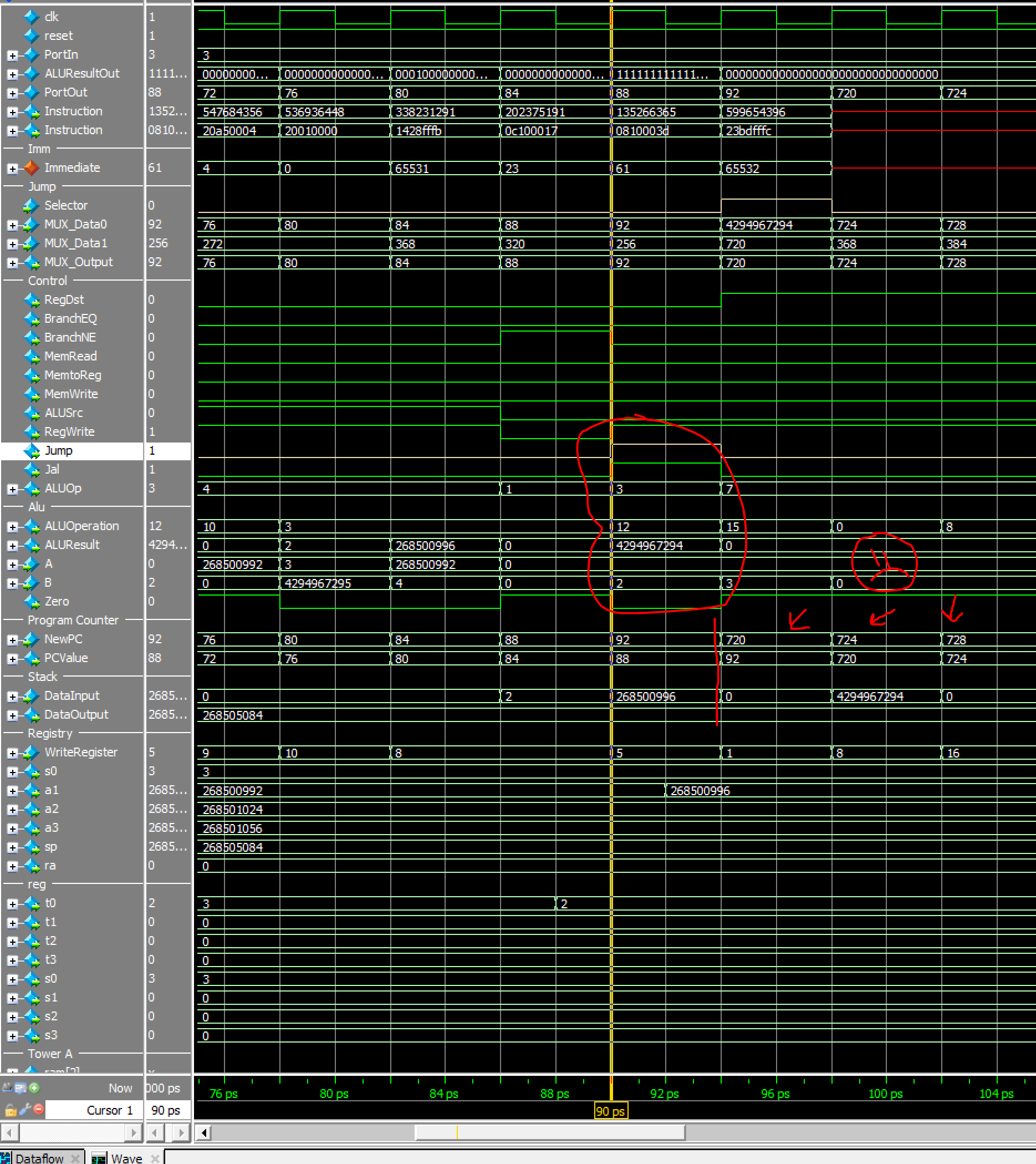
jrTest



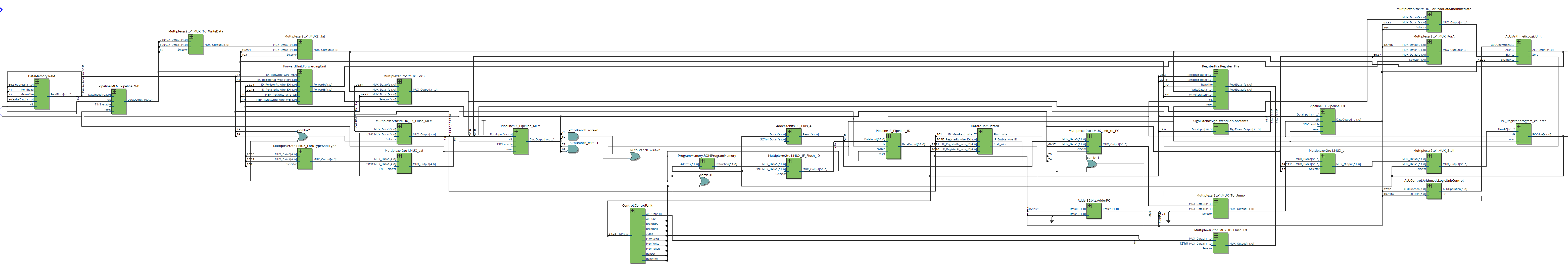


Hanoi (“hanoiTest”)

For Hanoi, as the j type instructions were broken the moment I tried to pipeline my original approach, the program did not work right at the moment where the first Jal is used: it calls the Hanoi Tower recursively after linking its current address, but there were lots of different problems – sometimes it even wrote the $ra data on a random register like $so. In this case, I also tried to nop the hell out of the original program and no, neither did work:

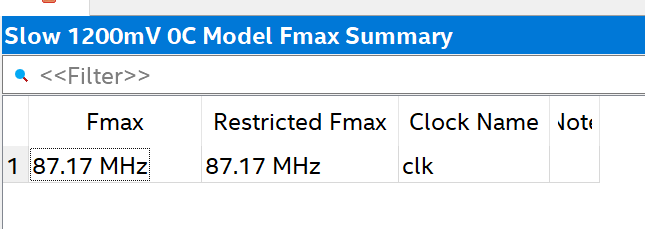


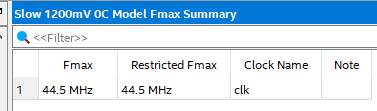
**3. RTL DIagram**



**4. Clock rate and CPU comparison:**

Single Cycled and Pipelined



****

**A screenshot of a cell phone

Description automatically generated**

**5. Flow diagram and description:**

**A screenshot of a cell phone

Description automatically generated**

Flow description:

My algorithm would begin by associating each of the addresses to each of its respective tower as well as register. We´d also always have n stored in s0, even though such variable is inputed by the user beforehand.

The loadDisks function or subroutine would then begin to load each and every disk onto its respective space or place on tower A/origin tower until the amount of disks are complete. And then it would proceed to call the HanoiTower function for the first time, which would initially allocate the memory necessary to store the ra registry onto the top stack position.

The function would first try to validate whether the amount of disks is or isn´t 1 in order to jump to the base case (which would move the respective tower to its according position) or to the internal function steps (which would swap the pointers or addresses on each of the towers accordingly in order to follow the algorithm described on the assignment).

The process would loop in and out of the steps, unwinding and then rewinding when its ready to swap the disks to their respective next tower, step by step, all by switching the auxiliary tower accordingly. Once the algorithms takes us to the last step or to the top of the rewinding process, it would put the last disk onto its final place and finally it would restore the size of the sp as well as s0 to finally be done with the process.

**9 – Files**

Check the included archive, where the top and rest of the v files, pdf architecture and flow diagrams are also included.