# CprE 381: Computer Organization and Assembly-Level Programming

# Project Part 2 Report

Team Members: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Alek Norris\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Drew Kearns\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

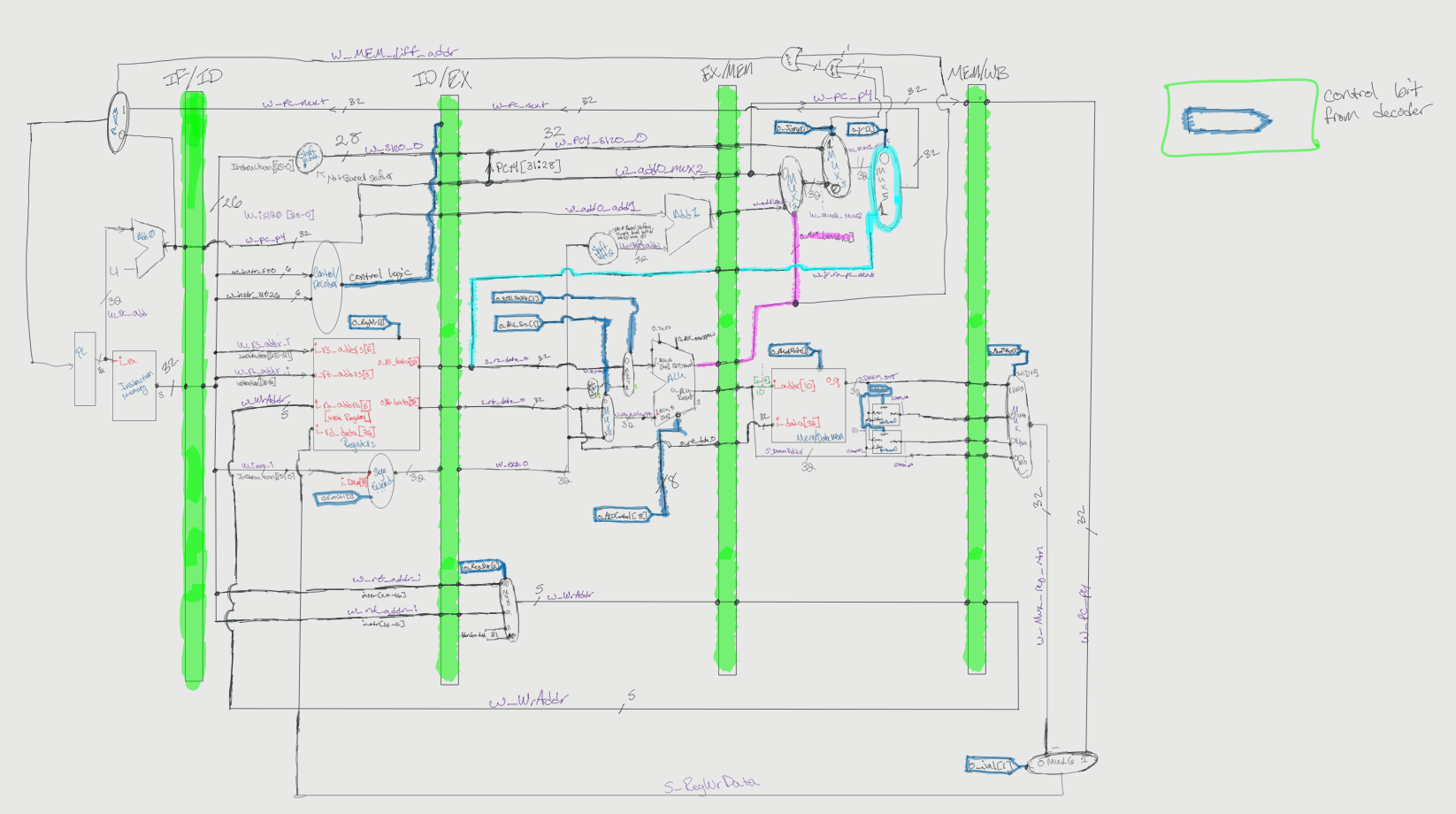
## Project Teams Group #:\_\_\_\_\_\_\_\_\_Term Proj1\_2s\_07\_\_\_\_\_\_\_

***Refer to the highlighted language in the project 1 instruction for the context of the following questions****.*

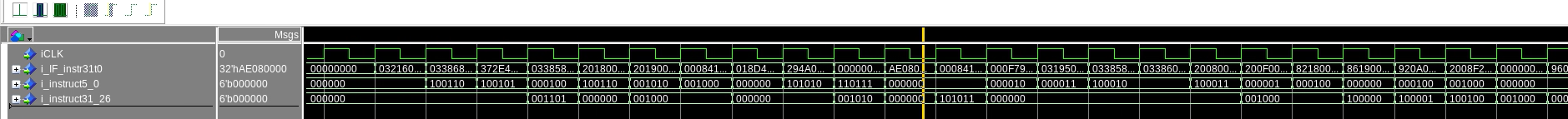
[1.a] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.

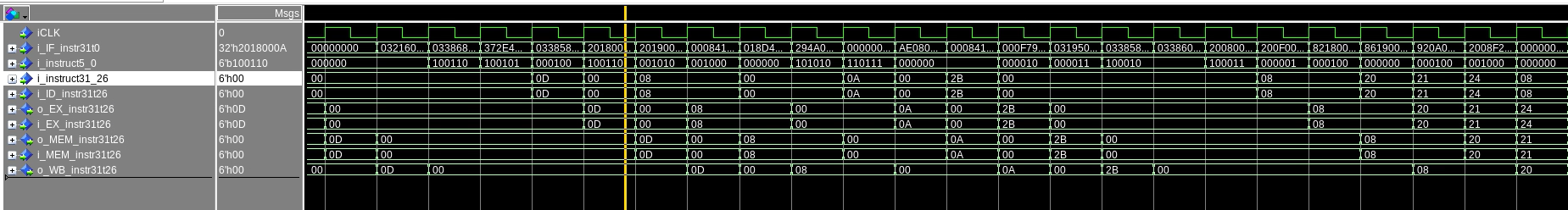


[1.b.ii] high-level schematic drawing of the interconnection between components.



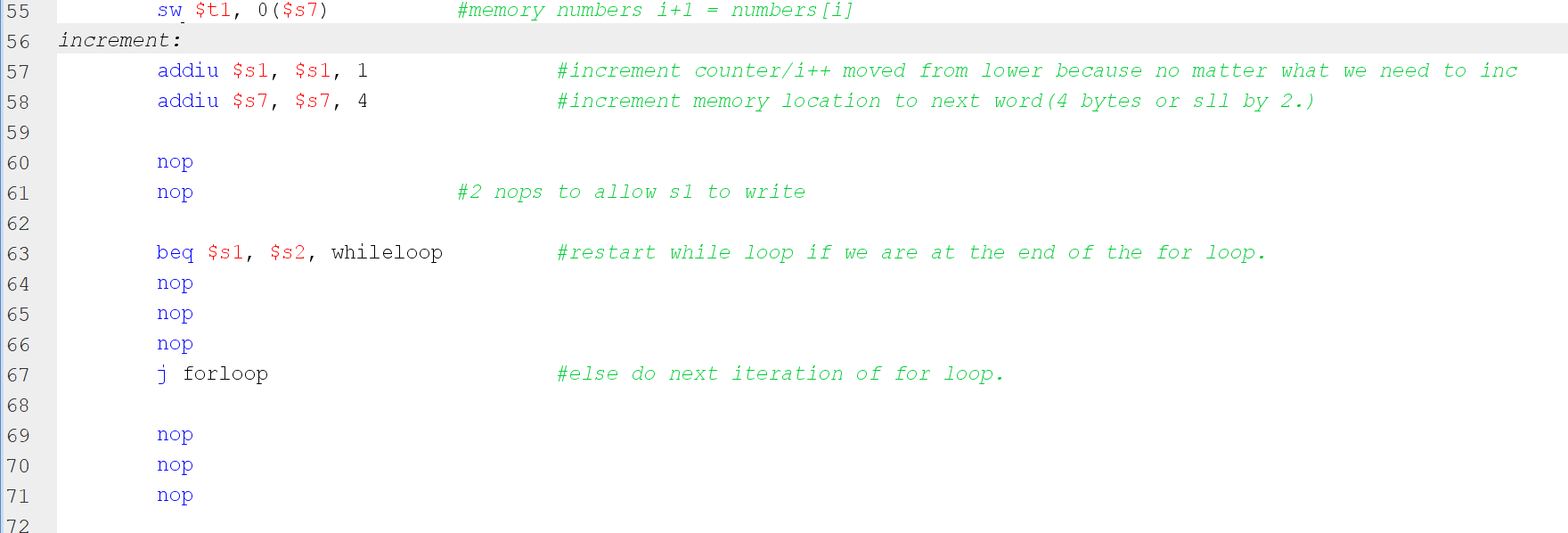
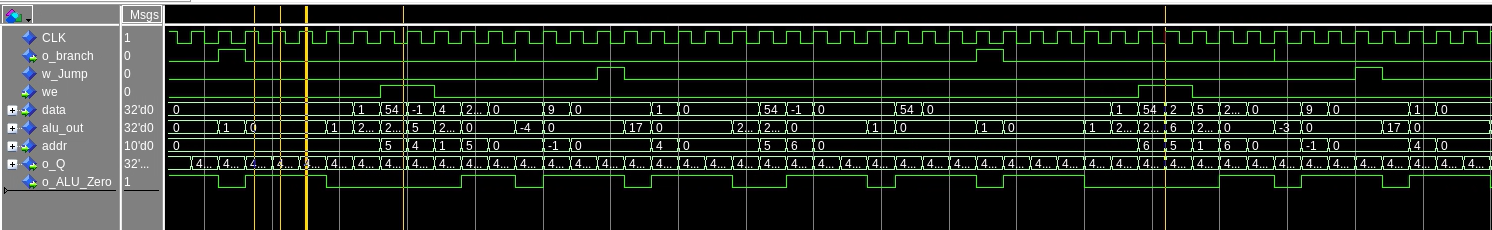
[1.c.i] include an annotated waveform in your writeup and provide a short discussion of result correctness.

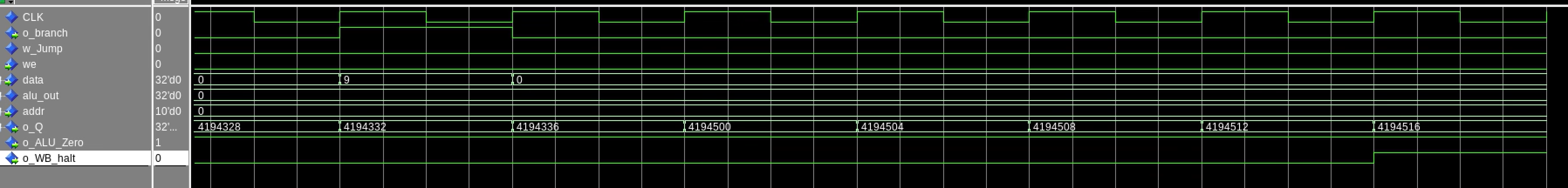
For this part I simply modified the code from out original tests we did when making the single cycle processor, then modified it, restructuring things to not have hazards, but still allowing the code to run. I verified it worked by running it through the test program which compares the mars output to the Questasim output and can be verified If you check the sw pipeline folder. But what you see above is the Decoder input for instructions 5-0 and 26-31, which is the opcode and function fields of the instruction. Any opcode that has 0x00 uses the funct, if not uses the opcode portion. Starting at 01101 of 31-26 you can see LW coming through the pipe, 001000 addi, slti etc.

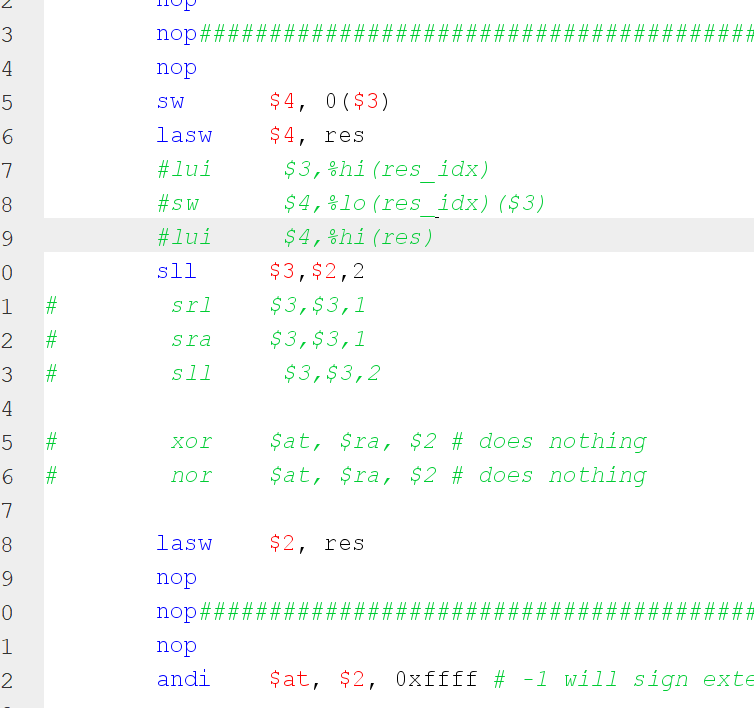


Then in this photo you can follow the instructions specifically 31-26 as they go in and out of every register changing on the clock cycles.

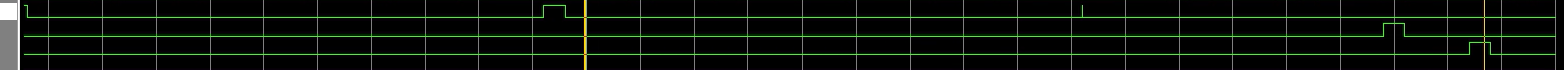
[1.c.ii] Include an annotated waveform in your writeup of two iterations or recursions of these programs executing correctly and provide a short discussion of result correctness. In your waveform and annotation, provide 3 different examples (at least one data-flow and one control-flow) of where you did not have to use the maximum number of NOPs.

Seen above is the bubble sort example of a data hazard and a control hazard avoidance. You can see its bubble sort being performed by seeing the memory locations of the first (Addr) 5,4 being compared, then 6,5 being compared, you can also see if write enable is up, then we were doing a swap of the numbers in memory. You can confirm the swap by seeing that 54,-1 swapped, then the next comparison is 54,0. This is because it always moves the smaller number to the left more position. To see the data hazards being delt with you can see the code, I was able to swap the addiu around to allow only 2 nops before the BEQ comparison. That’s why the larger address is on the left and not the right, and originally it made more sense to have the lower address on the left and the higher on the right as visually it made more sense. You can then see that after every Beq or jump there is 3 nops. Confirmed with the yellow lines, you can see 3 alu zero signals being asserted, meaning that only 3 nops had to be used. We were able to reduce our control hazard potentials from 4 to 3 by moving our jump logic into the Memory stage. Later we’d go on to reduce it even further by updating the main registers to update on a negative clock edge, but that was explicitly stated not to do for the software pipeline.

Besides being able to pass the bubblesort with the given testing software, we can tell the program executed correctly, because of the final branch, , followed by 3 nops, then 3 cycles later in the WB stage the Halt signal is high, meaning we successfully exited the program at the desired ending.



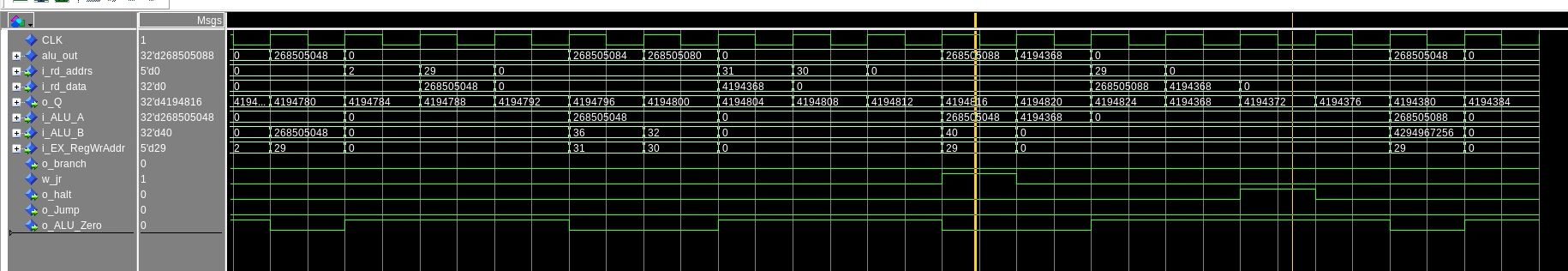
Here we were able to avoid data hazards by just removing redundent instruction. I saw while looking at them them that we were shifting and the value of $3 was not changing, reguardless of the three instrucitons running as no matter what they’d end with the same value. This was an example of data hazard avoidance.(because of it being very hard to find instruction ~insturction 1000 on the waveform I did not include a waveform corresponding for this.)





Because of the length of this phot it does look kind of weird, and I choose to crop out the signals. However, I felt it showed the jumping and recursion the best, see below for more details. But this is at the exiting of grendle, where it attempts to do a branch inside welcome, does not branch, then executedsthe last loaded $31, using jr, which takes it to the pump label. It then halts and this is seen as the halt signal becoming high.



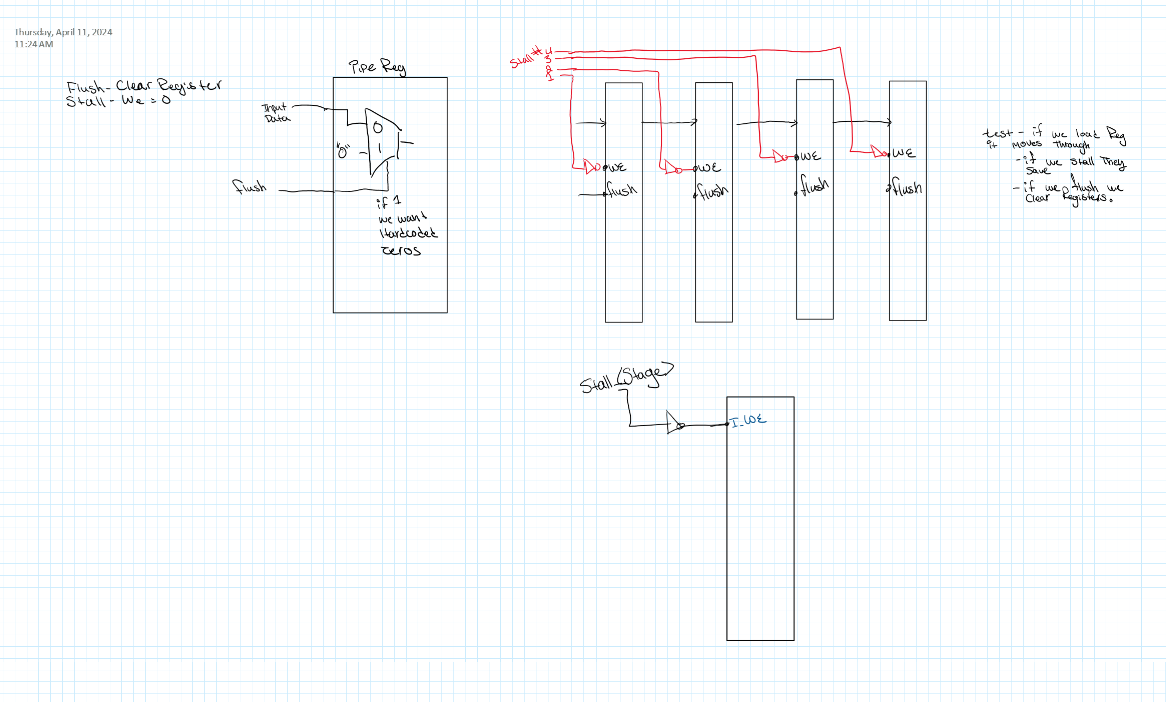


Here at the two yellow lines you can see that first the Jr Signal is asserter, then its followed by a halt. On the left red circle you can see when register 31 is written to via the load word, then in the middle you can see the alu out putting out the new jump location, and 2 cycles later the pc changing to the same location. You can also look at some additional data hazard avoidance by seeing that if there are 3 cycles where alu a and alu b are zero, a nop was performed(less than the max of 4) to avoid data dependencies, or if we were looking at a jump occuring just before the nops, a case of avoiding a control hazard.

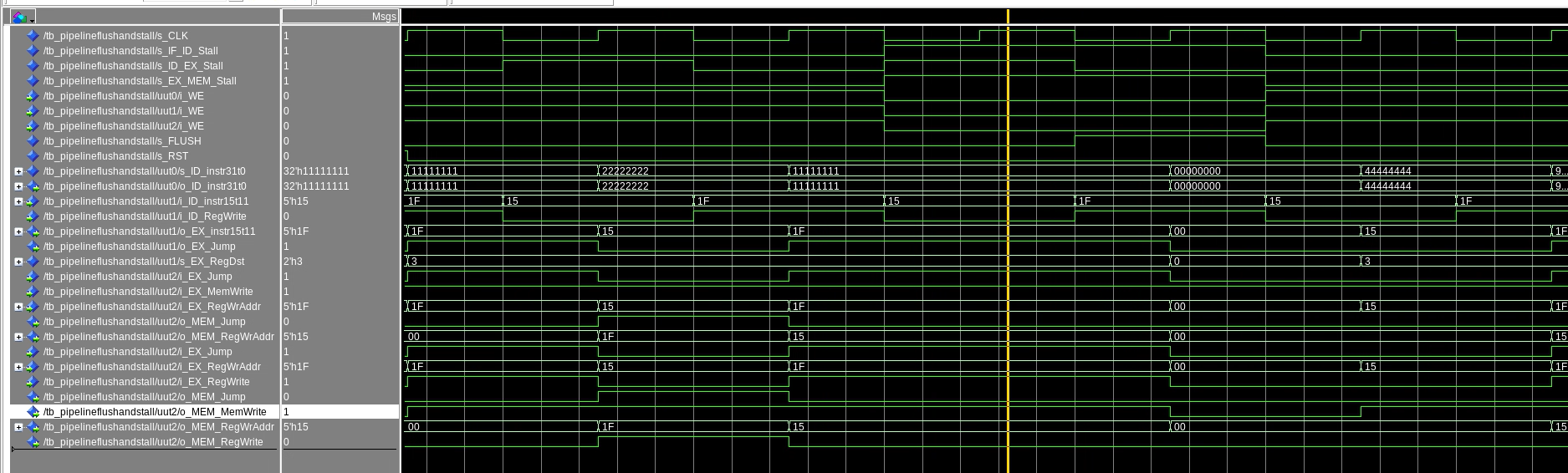
[1.d] report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

53.02mhz, the critical path for this is DMEM/WordDecoder/MEMWBPipe. The memory stage takes the longest as the DMEM is a beefy boi. So the critical path for this processor is as above.

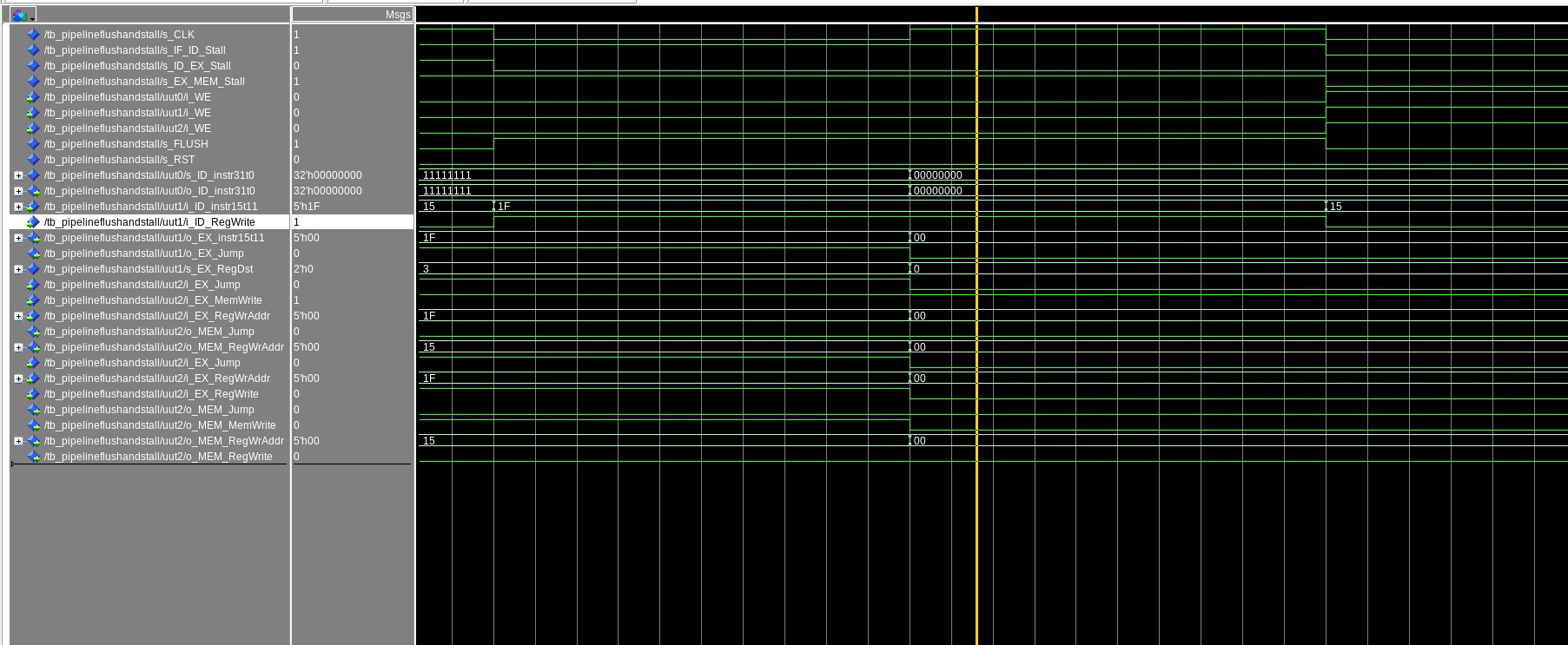
[2.a.ii] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.



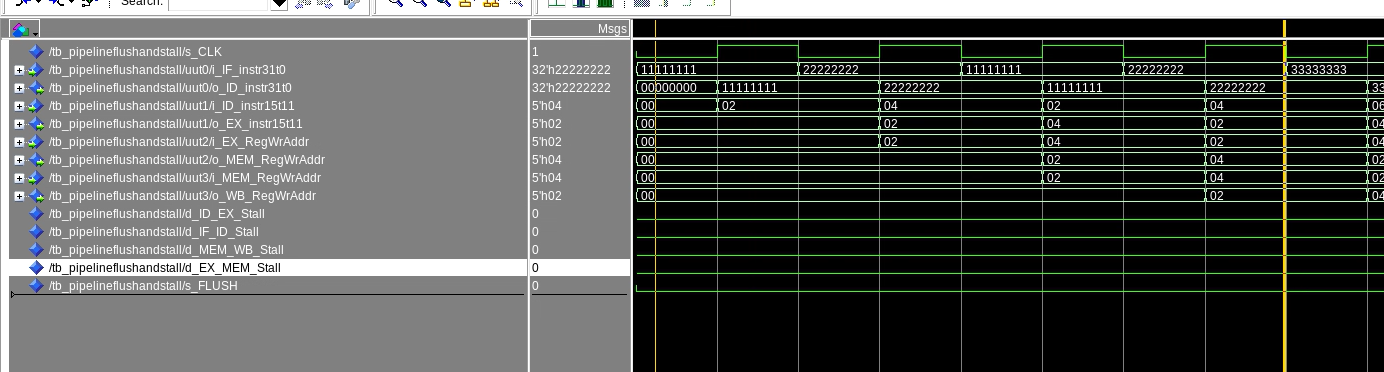
[2.a.iii] Create a testbench that instantiates all four of the registers in a single design. Show that values that are stored in the initial IF/ID register are available as expected four cycles later, and that new values can be inserted into the pipeline every single cycle. Most importantly, this testbench should also test that each pipeline register can be individually stalled or flushed.



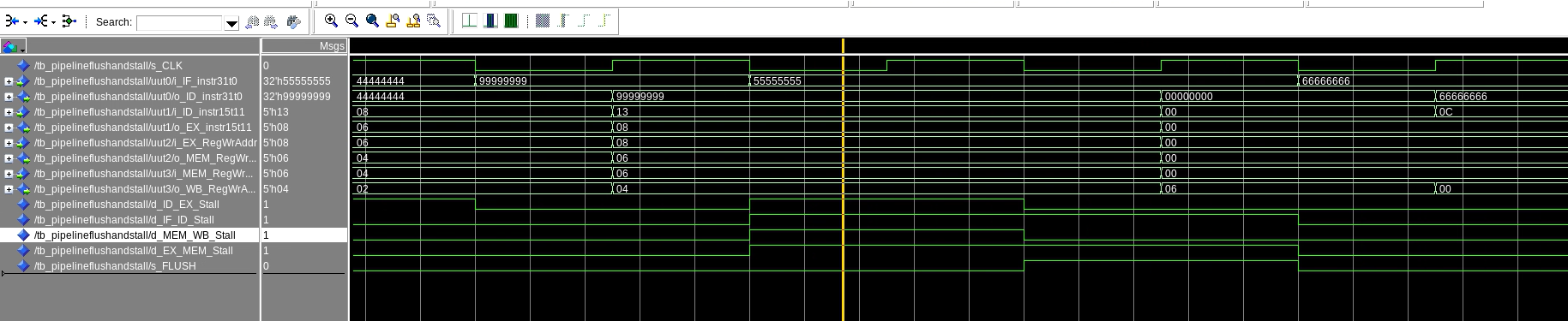
You can see here that when stall is active, none of the outputs change, even though the physical input i\_idInstr15t11 is still driving them.



You can then see here, that when flush is enabled, everything is cleared to 0 on the positive edge of the clock. Flush for this tb was hooked up to one signal to test how it works, but in the final design is hooked up with individual signals, you can also see here that stall is enabled on a couple of the registers, but they still default to 0.

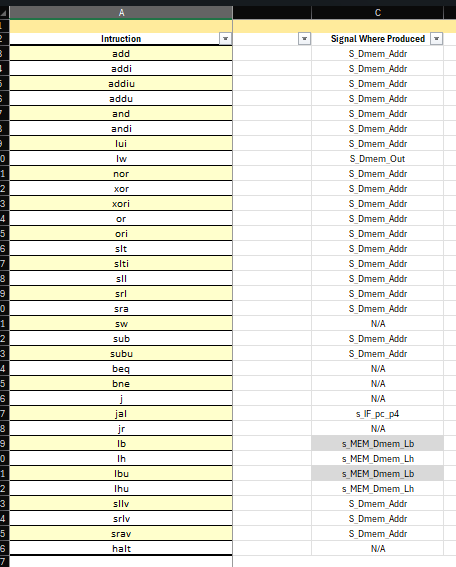


You can see here that the written information in the IF stage is available in the WB output 4 stages later



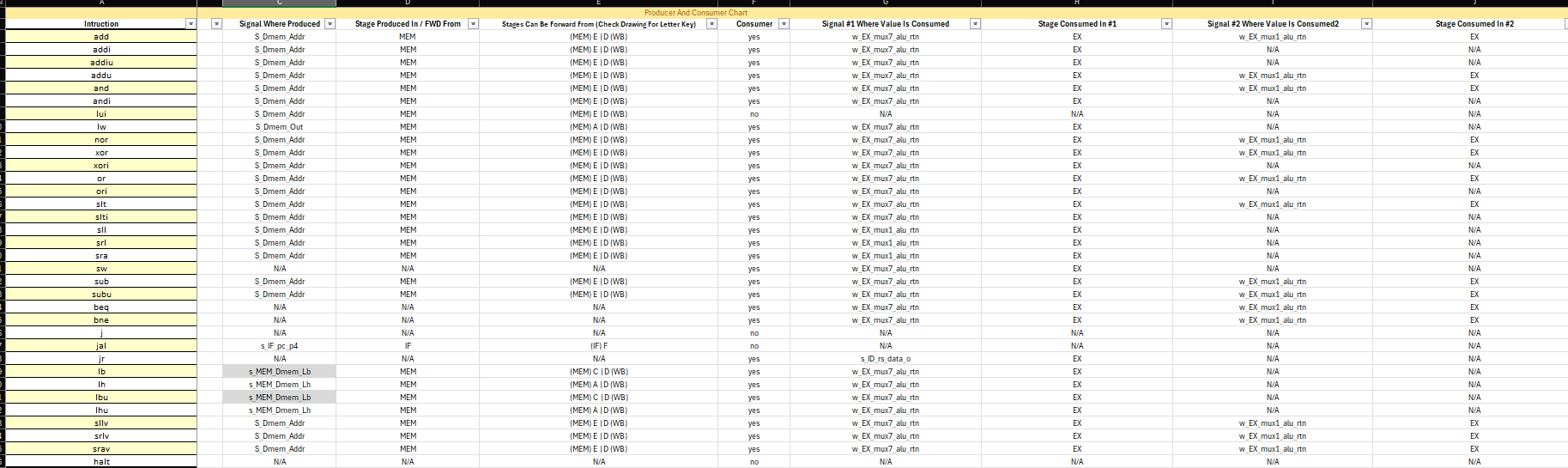
If a stall occurs across the board or just in a local value, you can see the written output values do not change in regards to that register stack.

[2.b.i] list which instructions produce values, and what signals (i.e., bus names) in the pipeline these correspond to.

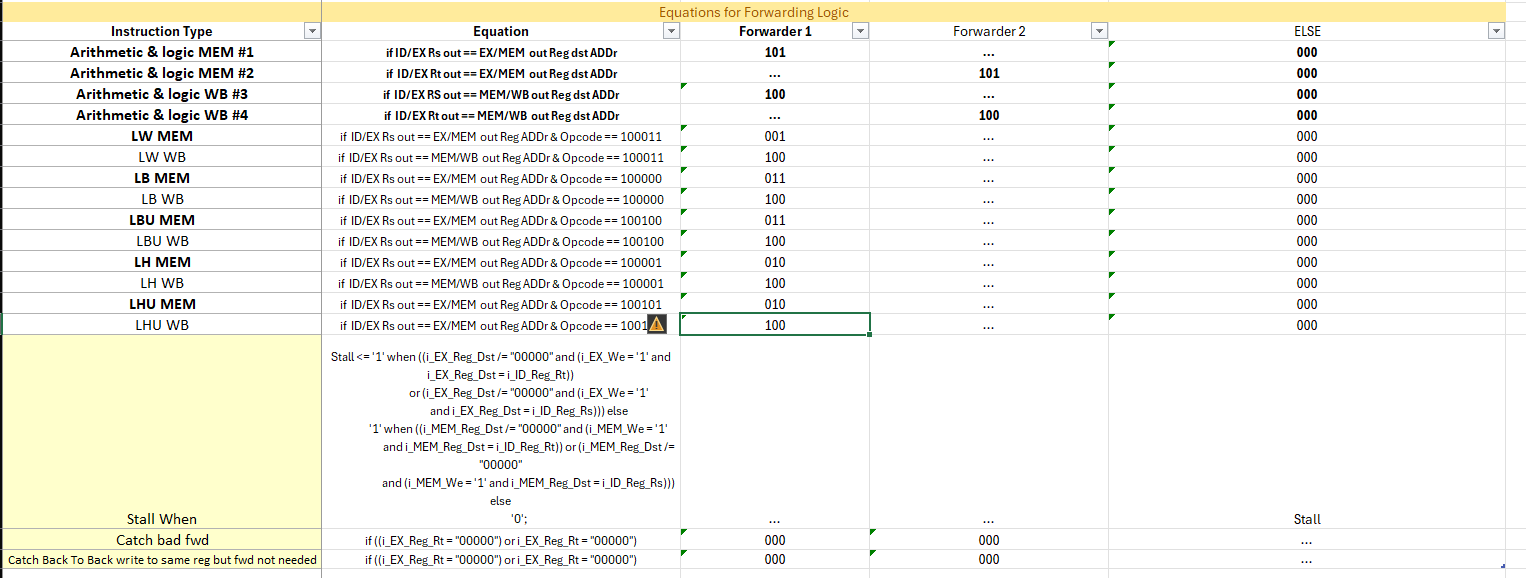


[2.b.ii] List which of these same instructions consume values, and what signals in the pipeline these correspond to.

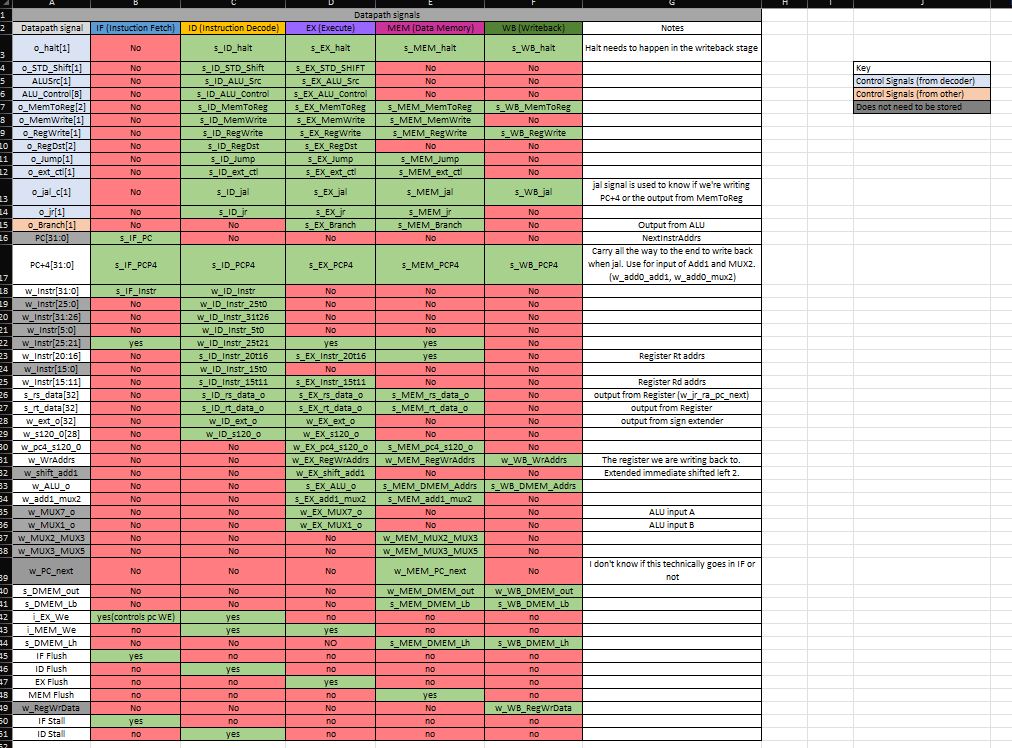




[2.b.iii] generalized list of potential data dependencies. From this generalized list, select those dependencies that can be forwarded (write down the corresponding pipeline stages that will be forwarding and receiving the data), and those dependencies that will require hazard stalls.



[2.b.iv] global list of the datapath values and control signals that are required during each pipeline stage

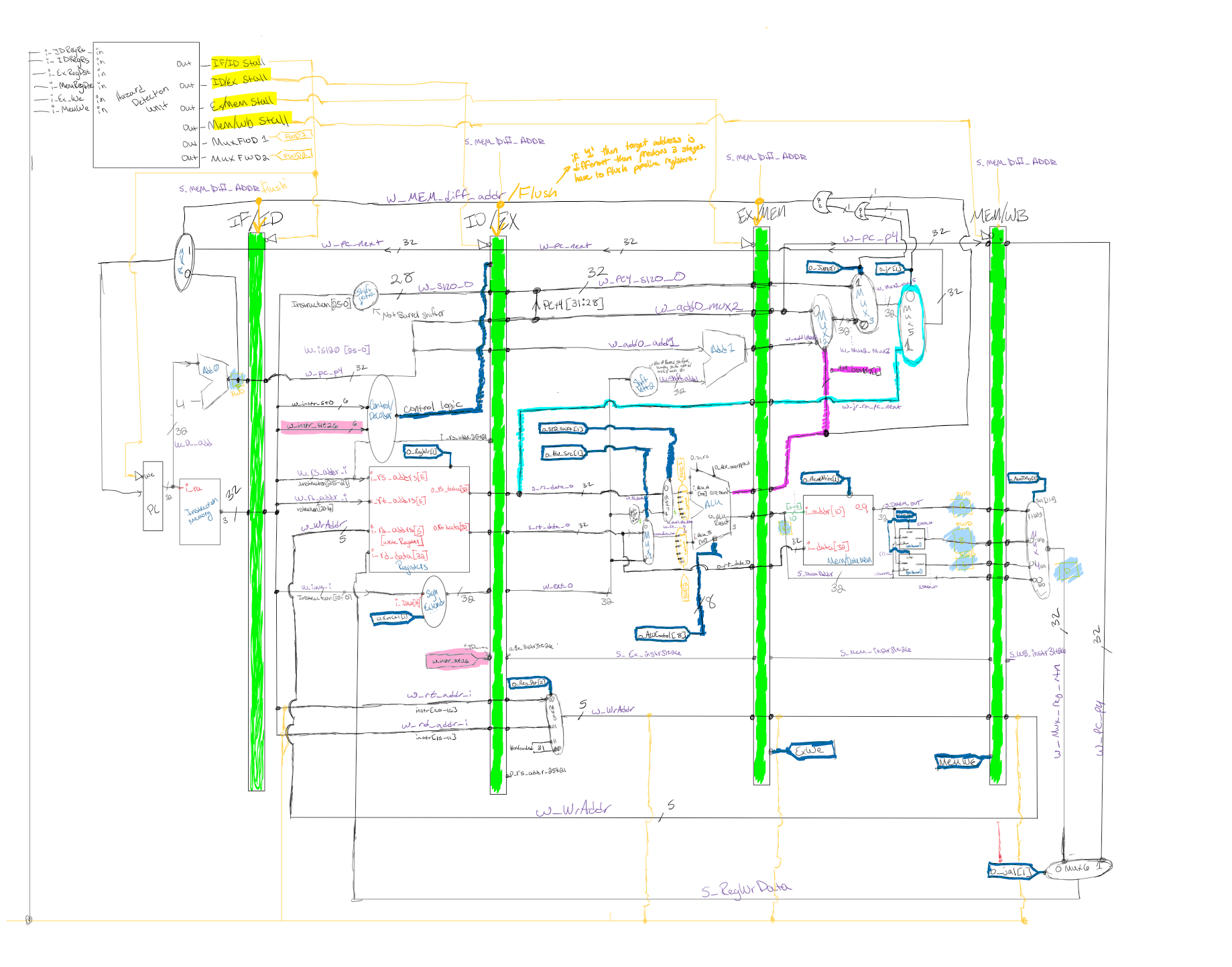
Updated from above 

[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs. Branch, jump, and jal (or anything that jumps.) this is executed for us in the Memory stage, because of this we had to add a special instruction to the PC We to not allow a stall to occur in the PC if flush or stall is occurring at that exact time (flush would occur during the execute stage for us in anything that results in a jump)

[2.c.ii] For these instructions, list which stages need to be stalled and which stages need to be squashed/flushed relative to the stage each of these instructions is in.

For Anything that meets the criteria for the stall we gave above we had it stall the IF\_ID registers, as well as the pc. Along with this, we also had the ID\_EX registers flushed as to make sure a junk instruction doesn’t make its way through.

[2.d] implement the hardware-scheduled pipeline using only structural VHDL. As with the previous processors that you have implemented, start with a high-level schematic drawing of the interconnection between components.



[2.e – i, ii, and iii] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

TODO

[2.e.i] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

TODO

[2.e.ii] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

TODO

[2.f] report the maximum frequency your hardware-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

The maximum frequency of the hardware-scheduled pipelined processor is 47.39mhz. And just like the software\_scheduled pipelined processor, the critical path goes through DMEM/WordShifter/MEM\_WB\_Pipe because the memory stage is the longest stage and that’s the longes path through that stage.