Introduction

The goal of this lab was to implement a 3-stage pipelined datapath, consisting of a register file and an ALU, capable of executing three instructions simultaneously. Two instruction formats are supported, R-type instructions and I-type instructions, for instructions with only registers and instructions with immediates, respectively. The instruction format fields are shown below:

Type	31	format (bits)			0
R	opcode(6)	r1(5)	r2(5)	r3(5)	
I	opcode(6)	r1(5)	r2(5)	imm(16)	

The opcode field is interpreted as follows:

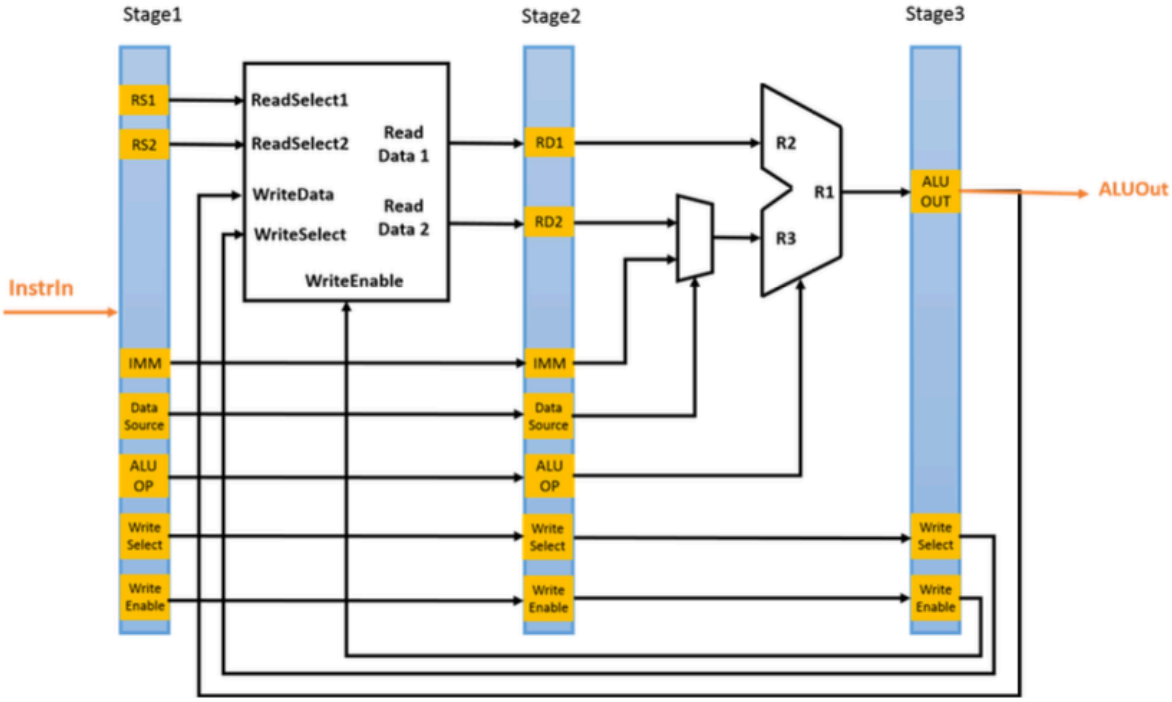
- The most significant bit (31) is unused, i.e., a don't care, set to zero during testing.
- The next most significant bit (30) determines whether the instruction is logical/arithmetic (1) or other (0). According to lab instructions, this is always set (1); thus it is always 1 during test instructions.
- The third bit (29) indicates an I-type instruction if it is set (1) or a R-type instruction if it is not set (0).
- Finally, the least significant 3 bits (28-26) of the opcode indicate the actual ALU operation, which is shown below:

AOp2	AOp1	AOp0	Output	Function Name
0	0	0	R1 = R2	MOV
0	0	1	R1 = ~R2	NOT
0	1	0	R1 = R2 + R3	ADD
0	1	1	R1 = R2 - R3	SUB
1	0	0	R1 = R2 | R3	OR
1	0	1	R1 = R2 & R3	AND
1	1	0	R1 = 1 if R2 < R3, else 0	SLT (signed)

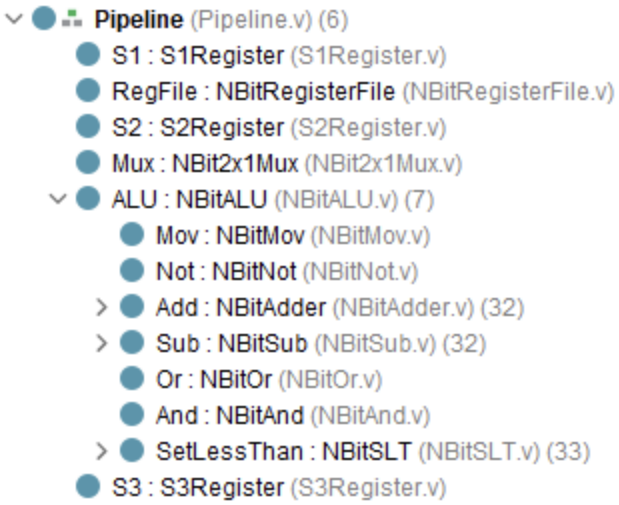
The register file holds up to 32, 32-bit registers, and the ALU takes in two 32-bit inputs and outputs one 32-bit output.

Design

My design is analogous to the schematic presented in the lab 5 instructions:



My module hierarchy is as follows:



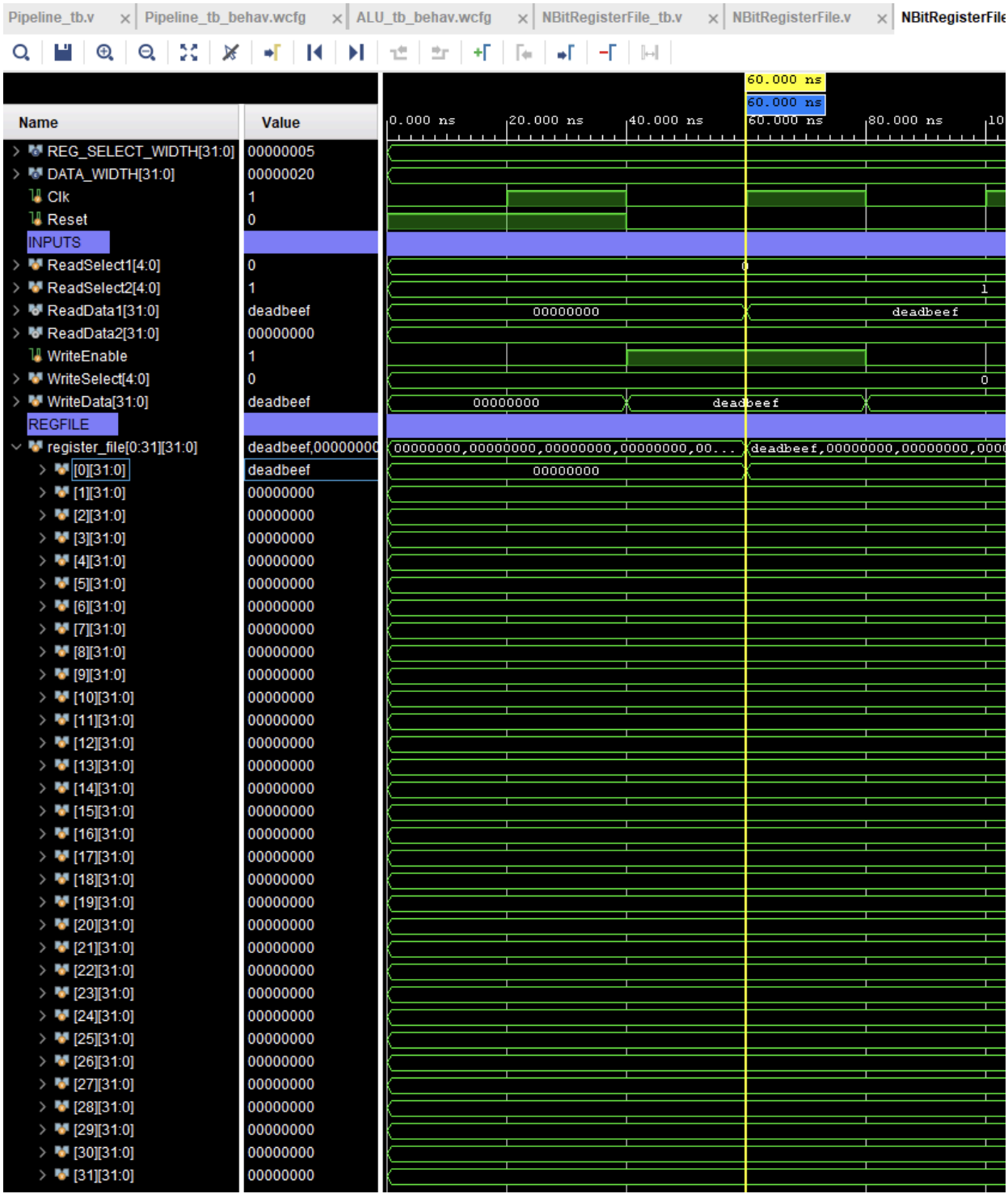
I have a top Pipeline module, which connects the S1Register, NBitRegisterFile, S2Register, NBit2x1Mux, NBitALU, and S3Register modules, exactly according to the schematic above. As the lab instructions allowed for behavioral Verilog, I used behavioral Verilog across all modules with exception of the NBitALU module, which I simply pulled from my previous lab 4 assignment. All source code of the modules can be found in the modules directory.

Testing

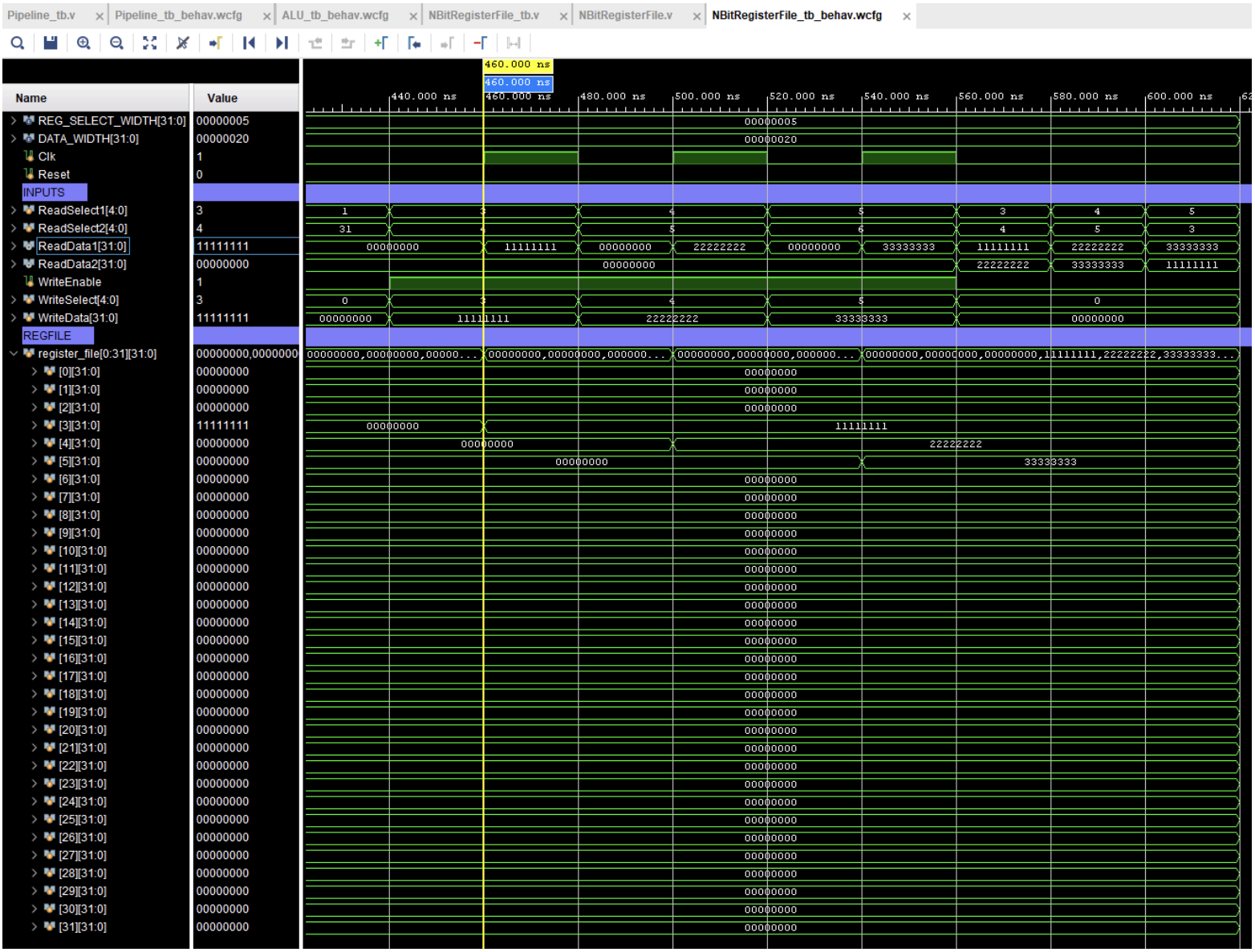
Disclaimer: I have included selected sections of my simulation waveforms here, but you can find full screenshots of the waveforms in the waveforms directory, with the corresponding waveform configurations I used to produce the following results. Of course, you will also be able to find the testbenches in the testbenches directory.

Register File Testing

I tested my register file to see that writes would only happen sequentially, i.e., on rising clock edges, and that writes would only occur when the WriteEnable bit is set (1). I also ensured that after a register was written to, the corresponding ReadData would show the correct value of the register if the register was selected via ReadSelect. A portion of my testbench's simulation waveform is shown below, and the full testbench module can be found in my testbenches directory. For instance, at 60 ns (the first marker), 0xdeadbeef is written into register 0, despite being present in the WriteData register input before that and WriteEnable being set. This is due to sequential writes—the register in the register file will only be written on a rising clock edge, which does occur at 60 ns.

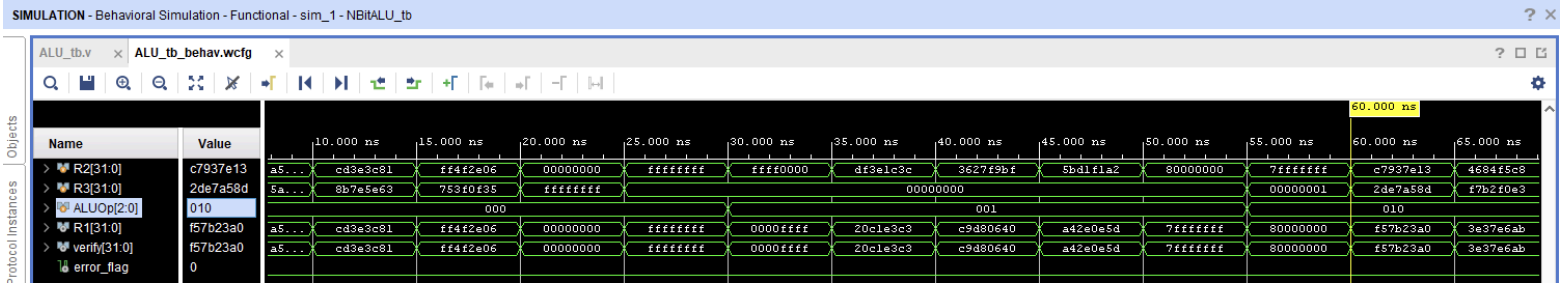


Another example of testing combinational reads is at the second marker at 460 ns: once the sequential write is performed to register 3 (WriteSelect = 3), ReadData1 also shows the new written value, 11111111, as ReadSelect1 = 3.



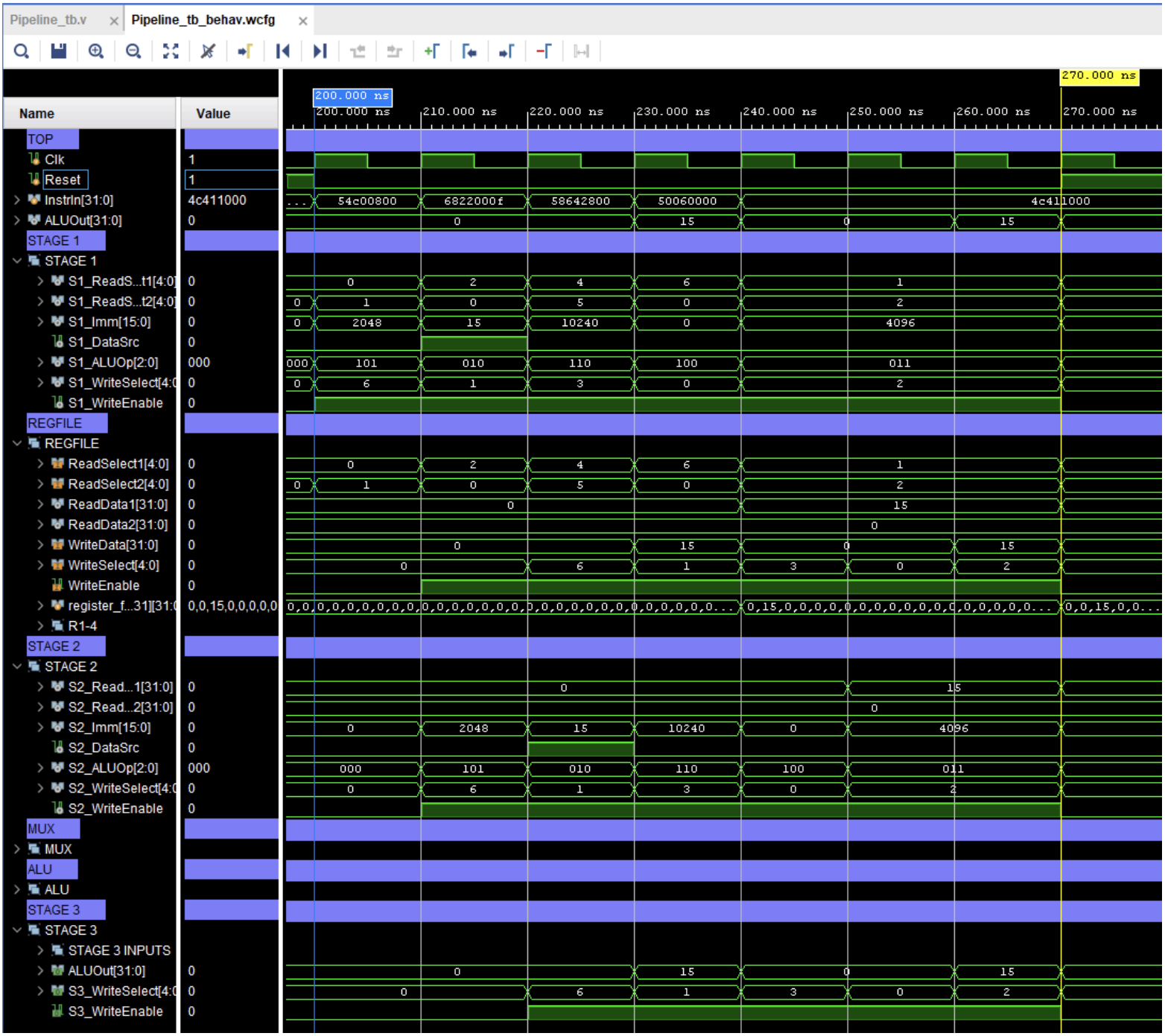
ALU Testing

I tested my structural ALU the same way as in lab 4 (and previous labs): using a behavioral ALU that receives the same operation and inputs, and checking the outputs with each other, setting an 'error_flag' bit if the outputs do not match. A section of my simulation waveform is shown below, with the full waveform (split into two images) in the waveforms directory. Note that R1 is the structural ALU output (the ALU used in my pipelined datapath) and verify is the behavioral ALU output (used only in the testbench). As can be seen, R1 always equals verify, and the error_flag is never set, ensuring my ALU is functioning correctly.



Pipelined Datapath Testing

To test my pipelined datapath, I used a combination of instructions provided in the lab assignment, the discussion slides, and instructions from the provided sample testbench. First, I computed the lab assignment instructions and discussion slide instructions by hand, i.e., a paper testcase. They can be found in the lab5_paper_testcase.xlsx Excel sheet I have included in the same directory as this PDF. I then compared my simulation waveform with the paper test case, and was able to confirm all the data is transferred from stage to stage correctly, with the final result being the same as the one I calculated in the paper test case. I do want to note that the instructions given in the lab assignment does run into a data hazard (reading stale data) and requires data forwarding, and since we only slightly covered that, I simply added in two cycles of stalling, waiting for the data to propagate correctly, and calculated my results based on this modification and compared it with the simulation waveform similarly (I made this change after having taken the screenshots below, as I had completed the lab before the class that covered it, so the screenshots in this report do not reflect my current testbench settings, but they do still explain how I tested my datapath). I have included an image of the simulation waveform of the five instructions from the discussion slides that does not run into any data hazards (200 ns to 270 ns in the image below), as well as my paper test case that matches the values depicted. Note that the 0 in ALUOut after the first 15 in ALUOut lasts two cycles, not just one.



	Time	T10	T11	T12	T13	T14	T15	T16
	32-Bit Instruction	010101_00110_00000_00001_00000000000	011010_00001_00010_00000000_00001111	010110_00011_00100_00101_00000000000	010100_00000_00110_00000_00000000000	010011_00010_00001_00010_00000000000	-	-
	Type	R	I	R	R	R	-	
	Decoded	AND R6, R0, R1	ADDI R1, R2, 15	SLT R3, R4, R5	OR R0, R6, R0	SUB R2, R1, R2	-	-
Stage 1	ReadSelect1	0	2	4	6	1	-	-
	ReadSelect2	1	dc	5	0	2	-	-
	Imm	dc	15	dc	dc	dc	-	-
	DataSrc	0	1	0	0	0	-	-
	ALUOp	101 AND	010 ADD	110 SLT	100 OR	011 SUB	-	-
	WriteSelect	6	1	3	0	2	-	-
	WriteEnable	1	1	1	1	1	-	-
Register File	ReadSelect1	0	2	4	6	1	-	-
	ReadSelect2	dc	15	dc	dc	dc	-	-
	ReadData1	0	0	0	0	15	-	-
	ReadData2	0	dc	0	0	0	-	-
	WriteData	-	-	0	15	0	0	15
	WriteSelect	-	-	6	1	3	0	2
	WriteEnable	-	-	1	1	1	1	1
Stage 2	ReadData1	-	0	0	0	0	15	-
	ReadData2	-	0	dc	0	0	0	-
	Imm	-	dc	15	dc	dc	dc	-
	DataSrc	-	0	1	0	0	0	-
	ALUOp	-	101 AND	010 ADD	110 SLT	100 OR	011 SUB	-
	WriteSelect	-	6	1	3	0	2	-
	WriteEnable	-	1	1	1	1	1	-
ALU	R2	-	0	0	0	0	15	-
	R3	-	0	15	0	0	0	-
	R1	-	0	15	0	0	15	-
Stage 3	ALUOut	-	-	0	15	0	0	15
	WriteSelect	-	-	6	1	3	0	2
	WriteEnable	-	-	1	1	1	1	1
Simulation Waveform Timestamp (ns)		200	210	220	230	240	250	260

As for the rest of my testing, I used the instructions from the provided sample testbench. Since my pipelined datapath has 3 stages, I expected (and calculated by hand above) instructions to take 3 cycles (in my specific testbench's case, 30 time units) to complete. Given this fact, I simply checked the ALUOut for the correct output 3 cycles/30 time units after each new instruction (the correct output is given in comments after each instruction in my pipeline testbench). One final note is that the paper test and testbench/waveform configuration files in my submission reflect the changes after adding in the stalling I mentioned earlier, so they will not appear like the screenshots above but are still correct.