CprE 381 – Computer Organization and
Assembly-Level Programming

Project A: Full ALU

Lab Partners:
Ethan McGill
Ehren Fox

Section / Lab Time:
Section 6, Thursday 4:10-6:00PM

Table of Contents

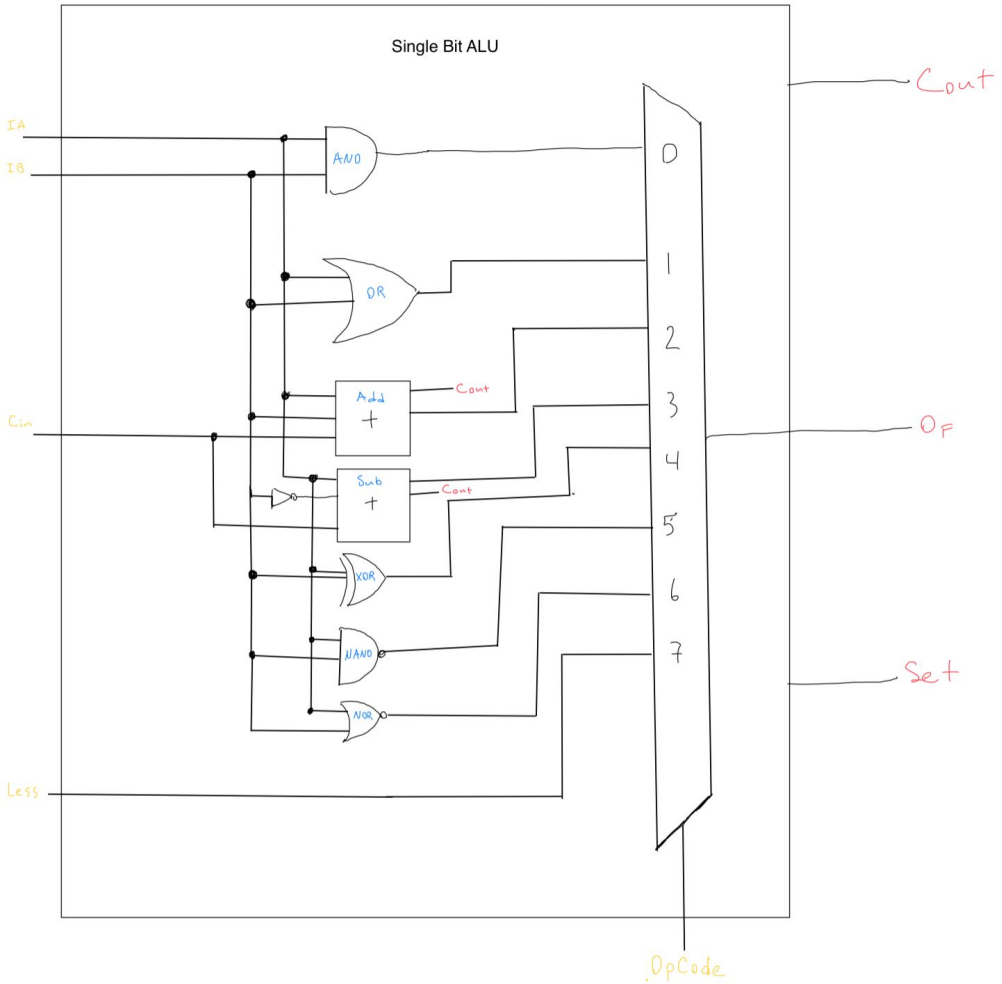
Prelab Questions

a. [Part 0] With your project group members, create a list of best practices / tips for designing, compiling, and testing VHDL modules based on your experiences so far with these labs, both working individually and as a group.

Draw what you are trying to create first, save and recompile often, create test benches to test instead of forcing values.

1-Bit ALU

b. [Part 1 (a)] Draw a schematic for a 1-bit ALU that supports the following operations: add/sub (both signed and unsigned), slt, and, or, xor, nand, and nor. What are the inputs and outputs that are needed?

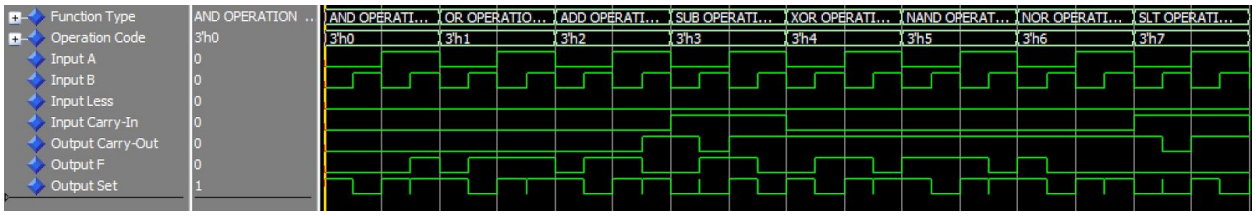


c. [Part 1 (b)] In your project writeup, describe your design in terms of the VHDL coding style you chose and the control signals that are required.

For the single bit ALU, we used dataflow VHDL coupled with behavioral case statements in order to achieve our goal. Because there were 8 different functions (and, or, add, sub, xor, nand, nor, and set less than) we need 3 bits of control.

d. [Part 1 (c)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.

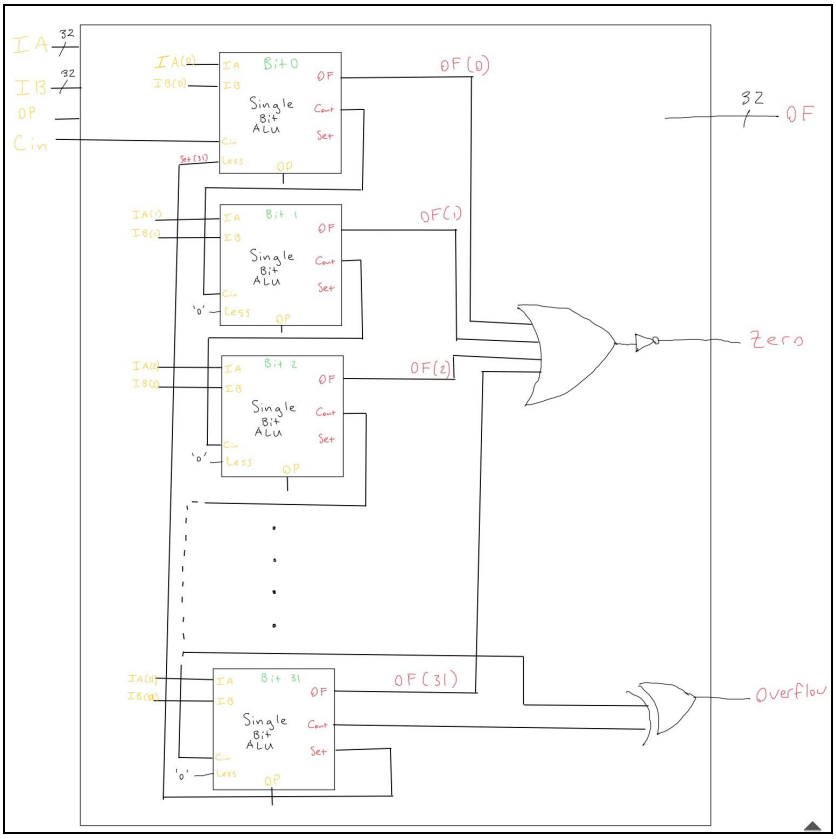
All operation types are labeled within the waveform screenshot. The set output is not important unless opcode is "111" (Set less than op). We tested all 2 bit input combinations for each operation type.



32-Bit ALU

e. [Part 2 (a)] Draw a simplified schematic for this 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is slt implemented?

Overflow is calculated by xor'ing the carry-in bit of the most significant ALU (c_in(31)) and the carry-in bit of the prior ALU (c_in(30)). Zero is calculated by hooking up each bit of the output to an or gate, and then using a not gate to invert it. SLT is implemented by subtracting input B from input A, and then xor'ing that with the carry-in bit of the ALU and the carry-out bit of the ALU.

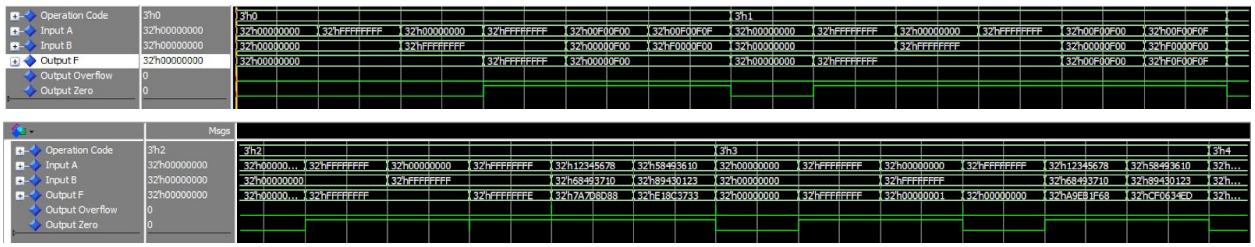


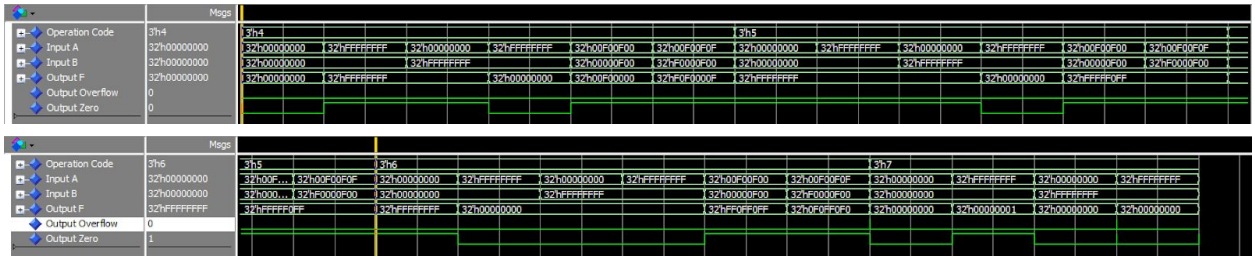
f. [Part 2 (b)] In your writeup, describe what challenges (if any) you faced in implementing this module.

There was some confusion about having two different 1 bit ALU's and how the SLT actually worked. When using the process statement we did not have a strict enough process list so we ran into a time consuming bug that was easy to fix once found.

g. [Part 2 (c)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.

Added helper function to output the function name as the test bench runs. Also added an "expected output" to verify the results are what we are looking for. This test bench tests random inputs for the first 8 operations of the ALU (and, or, add, sub, xor, nand, nor, SLT) to verify functionality, and tests the shift left logical multiple times. All sub components (ALU, shifter) have comprehensive test cases already.





32-Bit Barrel Shifter

h. [Part 3 (a)] Describe the difference between logical (srl) and arithmetic (sra) shifts. Why does MIPS not have a sla instruction?

The difference is that an arithmetic shift preserves the most significant bit of the input being shifted, and fills from the left with whatever the value of the MSB was.

i. [Part 3 (b)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

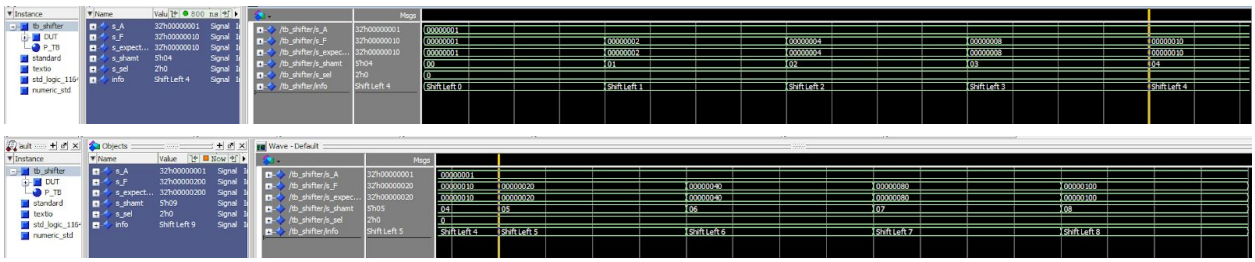
We created a chain of 32bit 2-1 multiplexers with process statements. We used two separate multiplexer chains to handle logical vs arithmetic. The logical chain only appends 0's to the given value. The arithmetic appends the most significant bit to the given value.

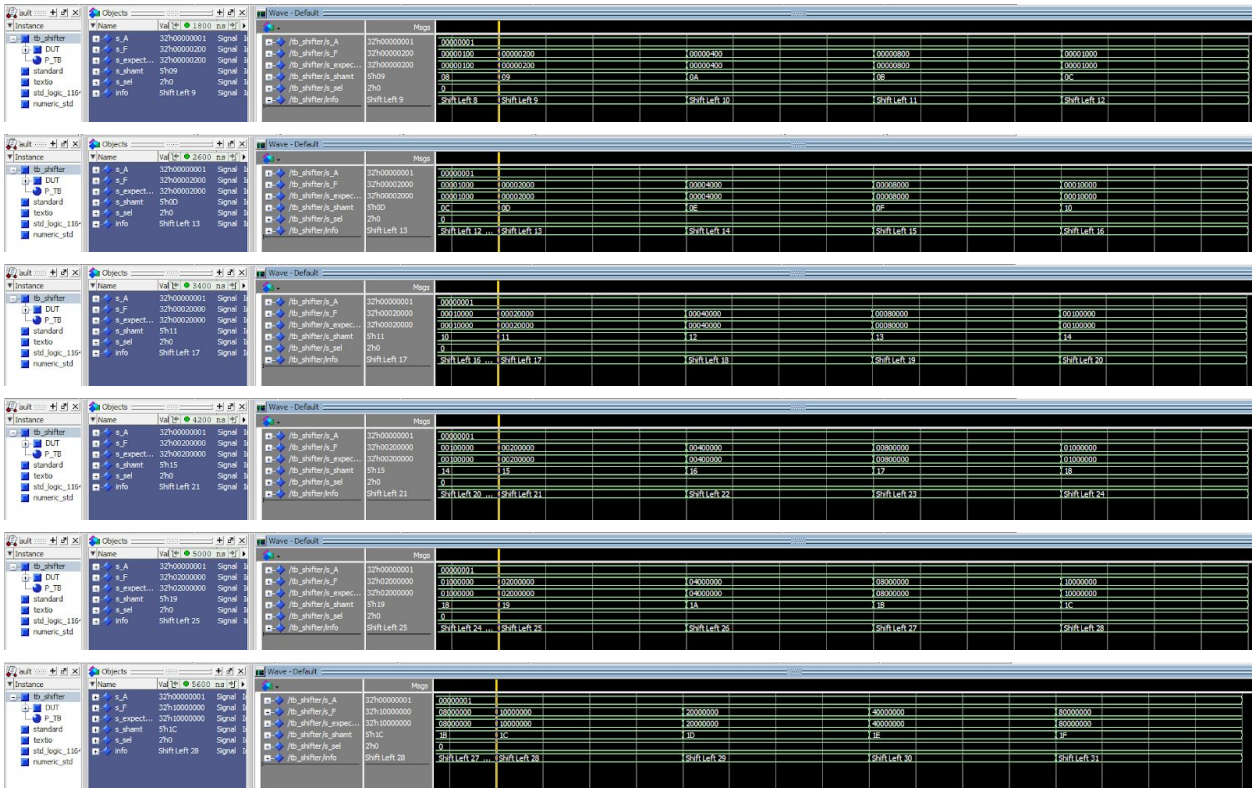
j. [Part 3 (c)] In your writeup, explain how the right barrel shifter from part b) can be enhanced to also support left shifting operations.

We added a third chain of multiplexers that appended 0's to least significant side rather than to the most significant side of the given value. We combined all three chains into one shifter object that had shared inputs, outputs and select bits.

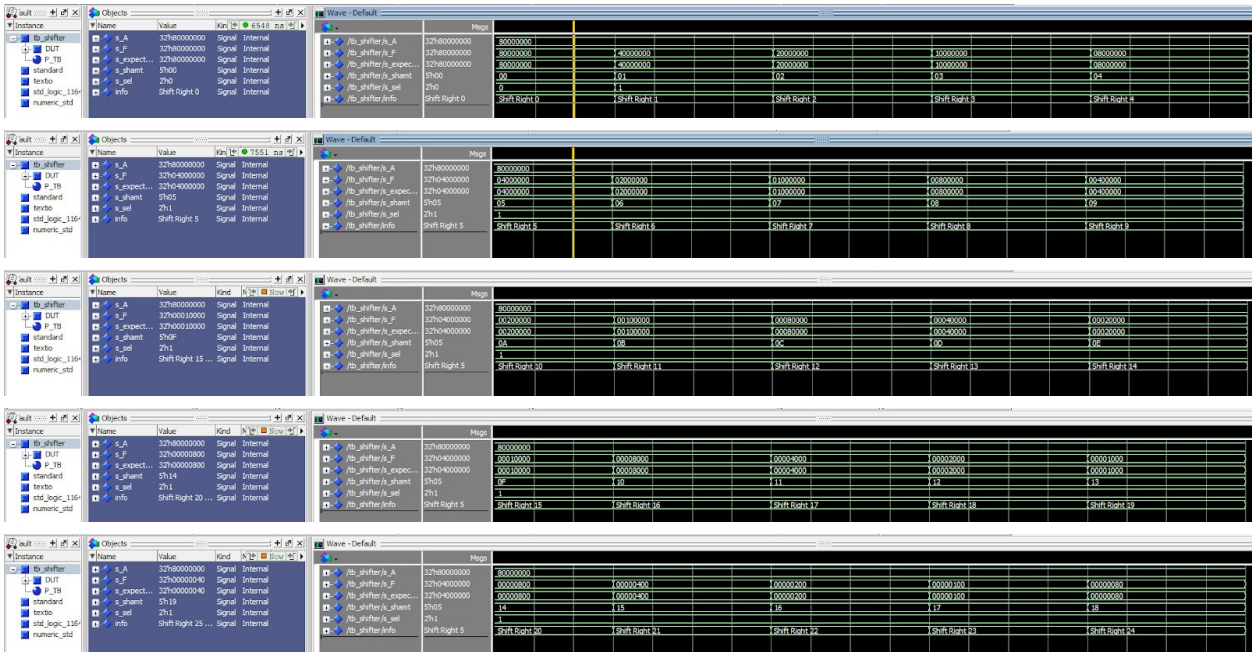
k. [Part 3 (d)] Describe how the execution of the different shifting operations corresponds to the Modelsim waveforms in your writeup.

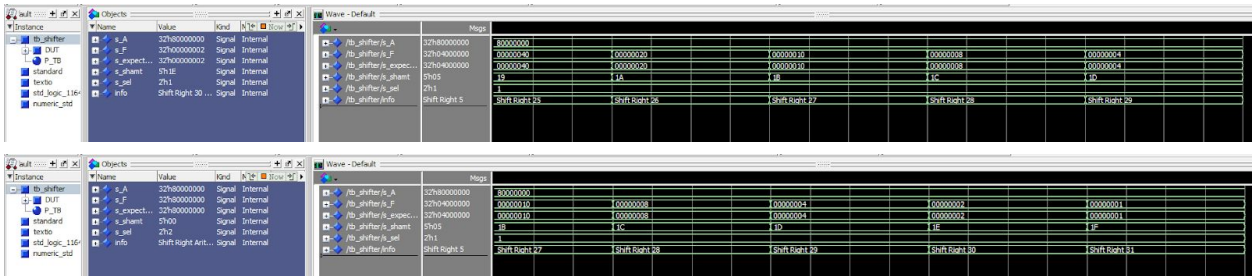
In the below waveform we are shifting left 1 bit, 2 bit … to 31 bits. As seen in the below waveforms the previous value is increasing by a multiple of two until you shift it all 31 bits and it reaches the highest possible value from shifting a single bit.



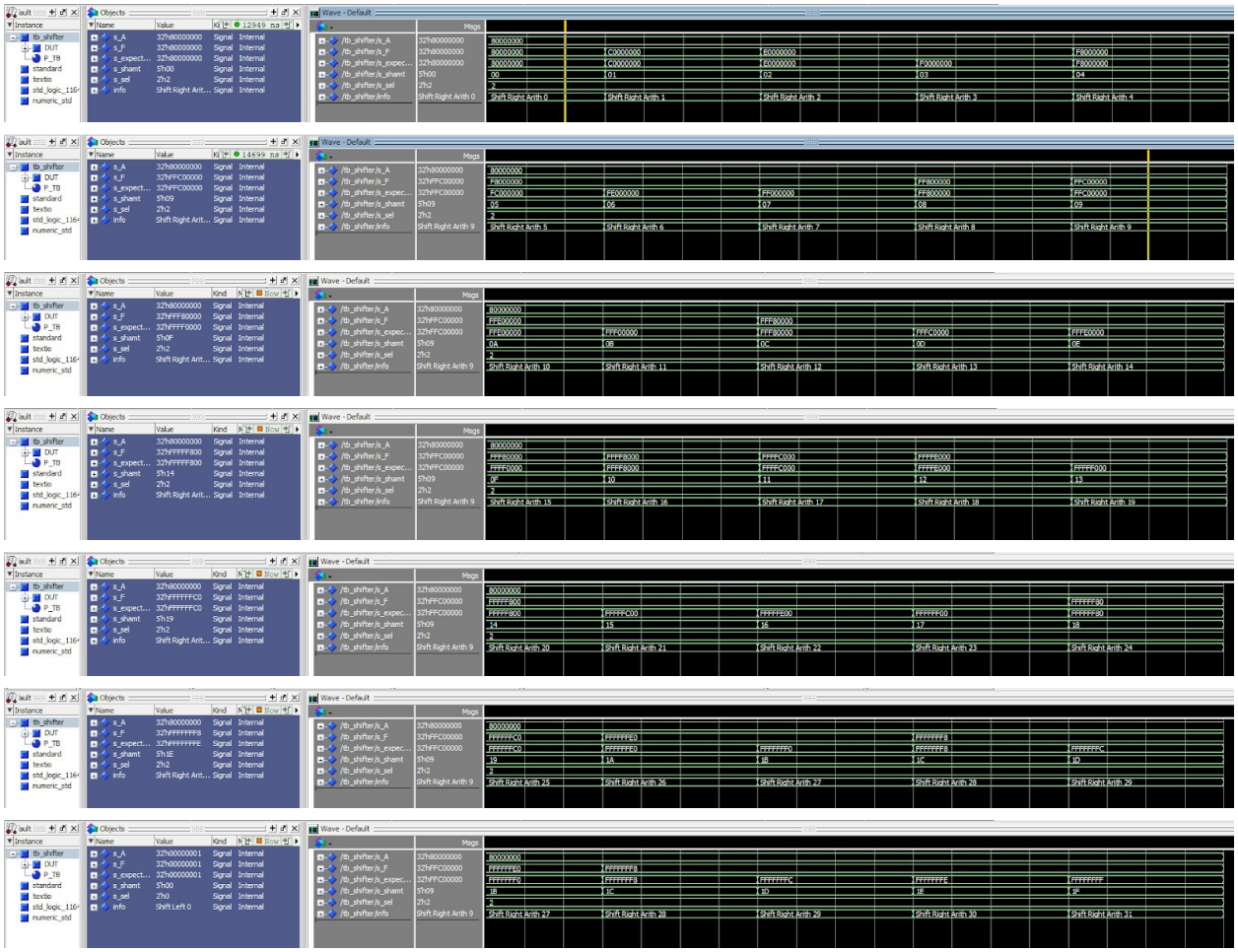


In the below waveform we are shifting right 1 bit, 2 bits … to 31 bits sequentially. As seen in the below waveforms the previous value is decreasing in by a multiple of two or the previous value is being divided by two, until we shift it all 31 bits and it's the lowest possible number for a single bit shifted right.





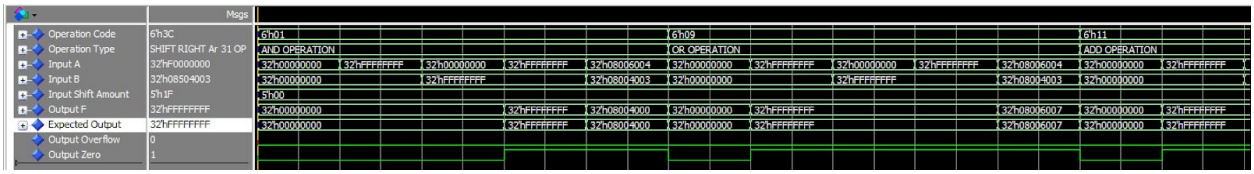
In the below waveform we are shifting right 1 bit, 2 bits … to 31 bits sequentially while preserving the most significant bit. As seen in the below waveforms the previous value is decreasing in by a multiple of two or the previous value is being divided by two, until we shift it all 31 bits and its the lowest possible number for a single bit shifted right.

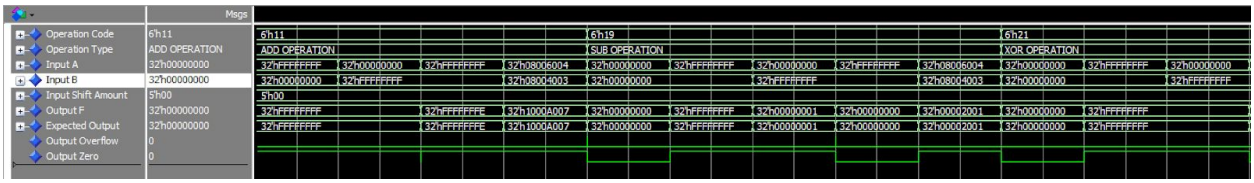


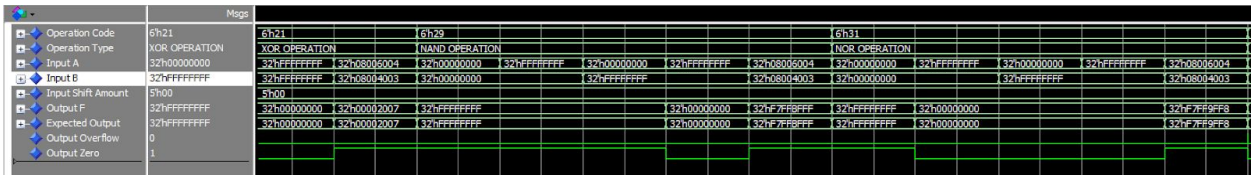
Full-ALU

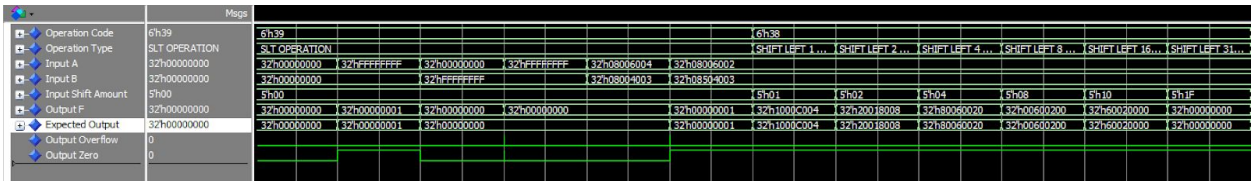
l. [Part 4(b)] Justify why your test plan is comprehensive. Include waveforms that demonstrate your test program is functioning.

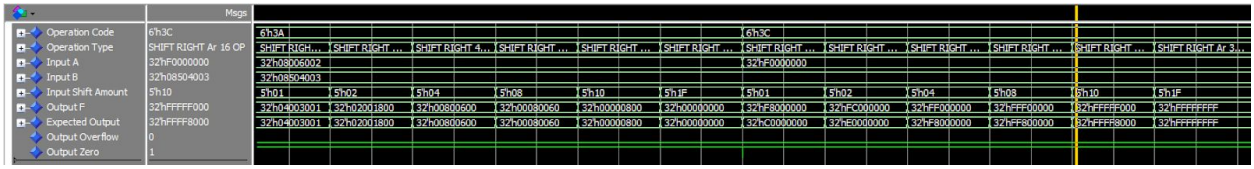
The following three waveform captures are of the Full ALU + Barrel Shifter component.











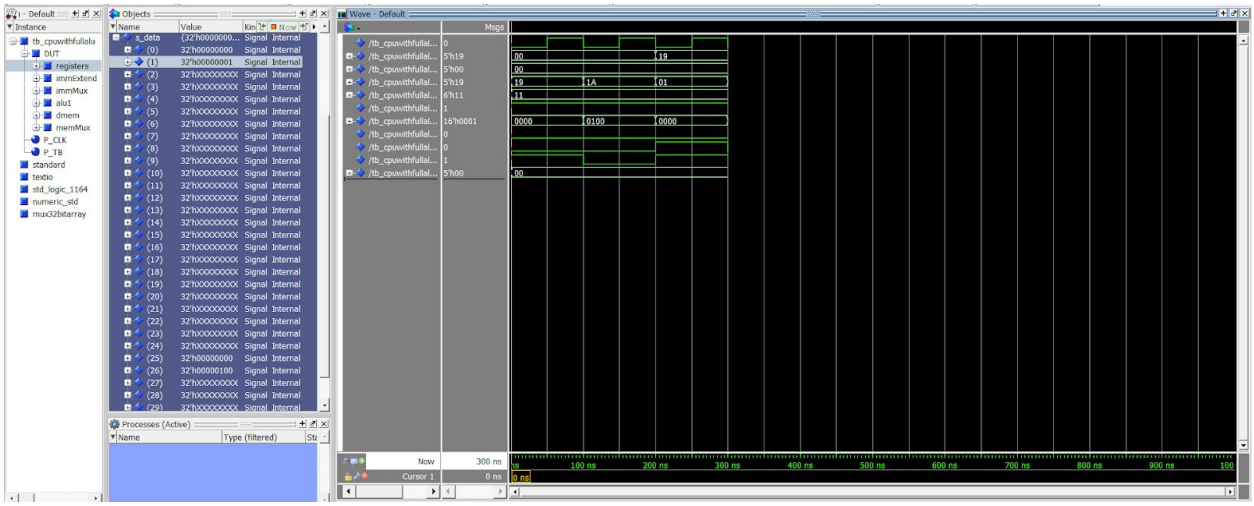
We tested each function of the full ALU against every edge case, as well as random test cases, after this we connected the completed full ALU into our data path and created a MIPS program with our new ALU to test whether it worked when being used as it would in the "real world"

MIPS Program for New CPU

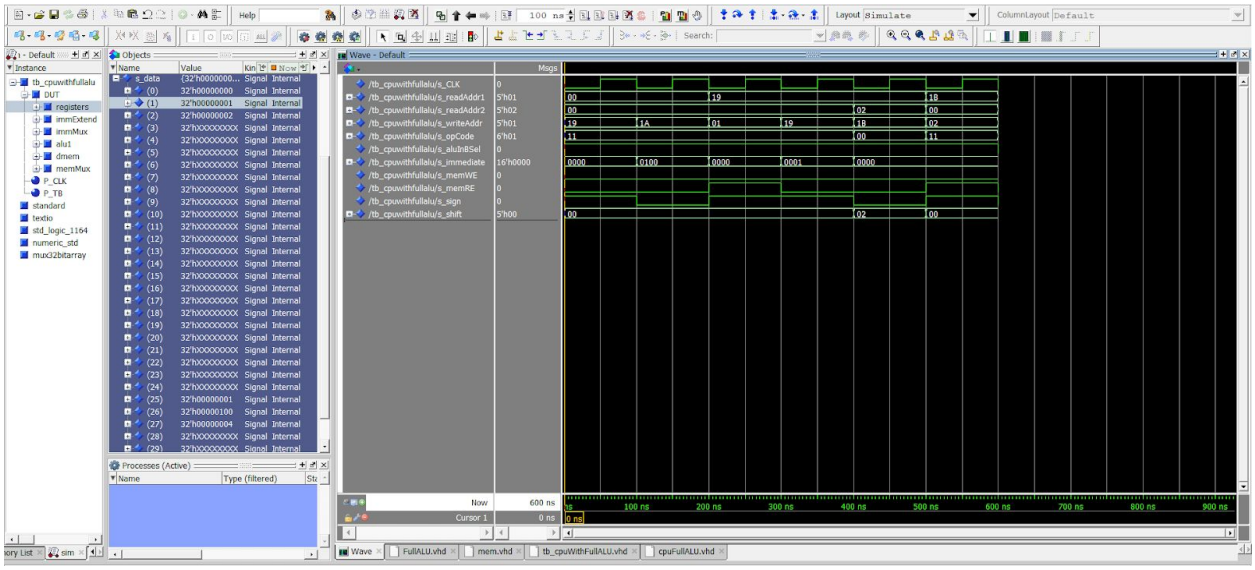
ALU Test's independent of the datapath:

MIPS "Program" with the new Full ALU Screenshots of datapath waveform and register/mem values after given instructions:

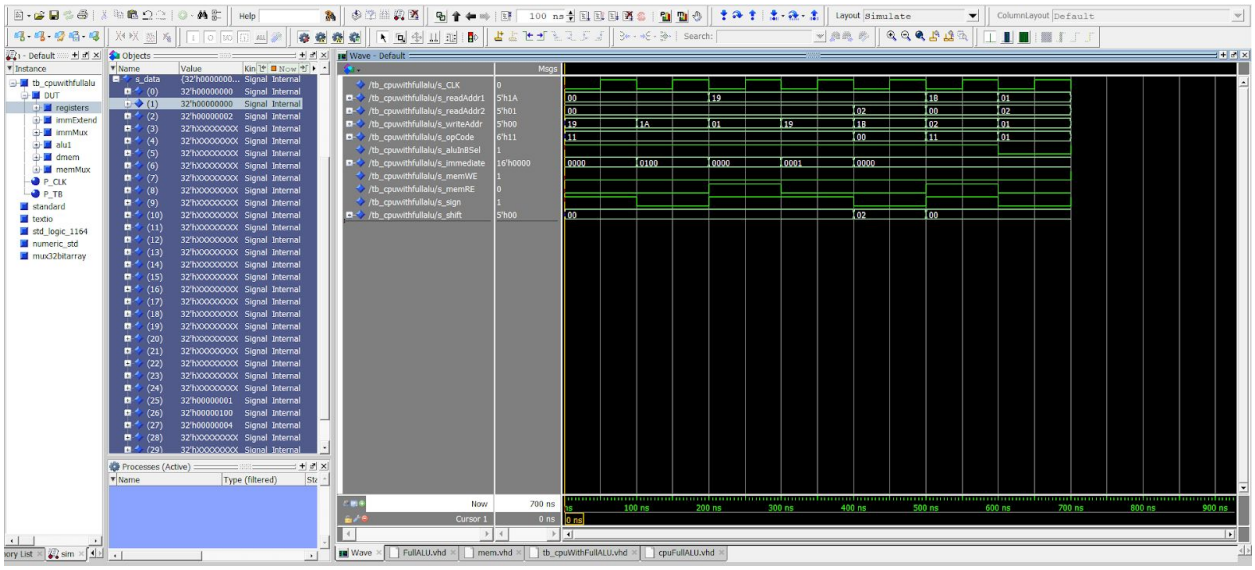
addi $25, $0, 0
addi $26, $0, 256
lw $1, 0($25)



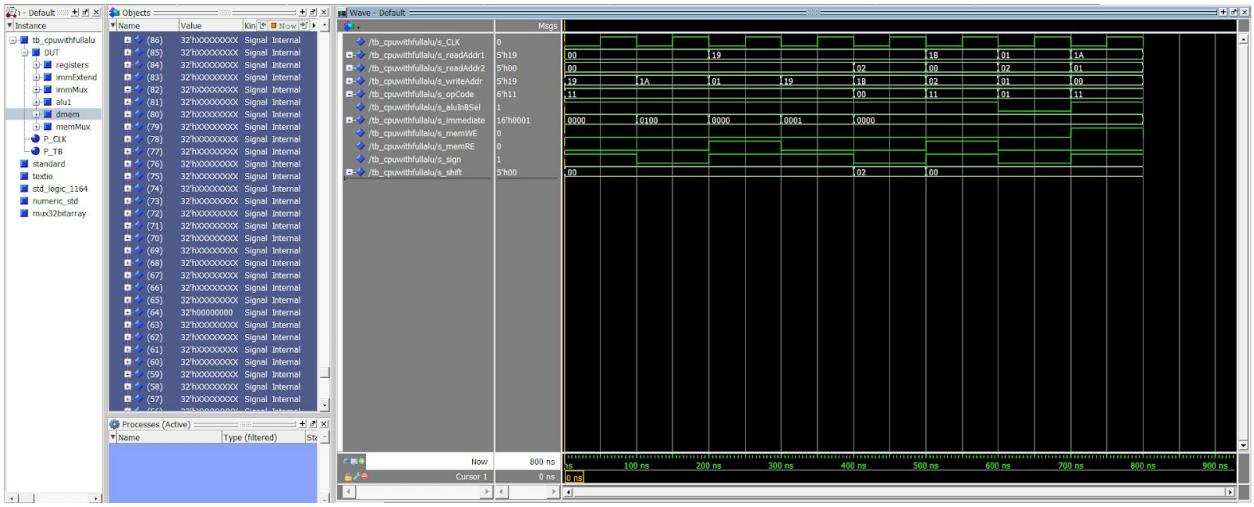
Addi $25, $25, 1
Sll $27, $25, 2
Lw $2, 0($27)



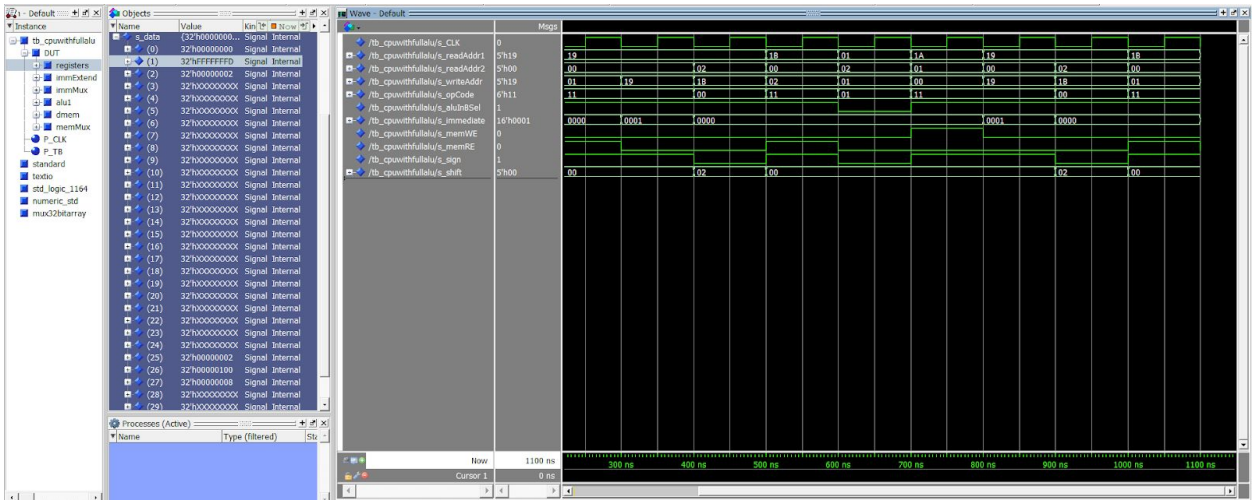
And $1, $1, $2



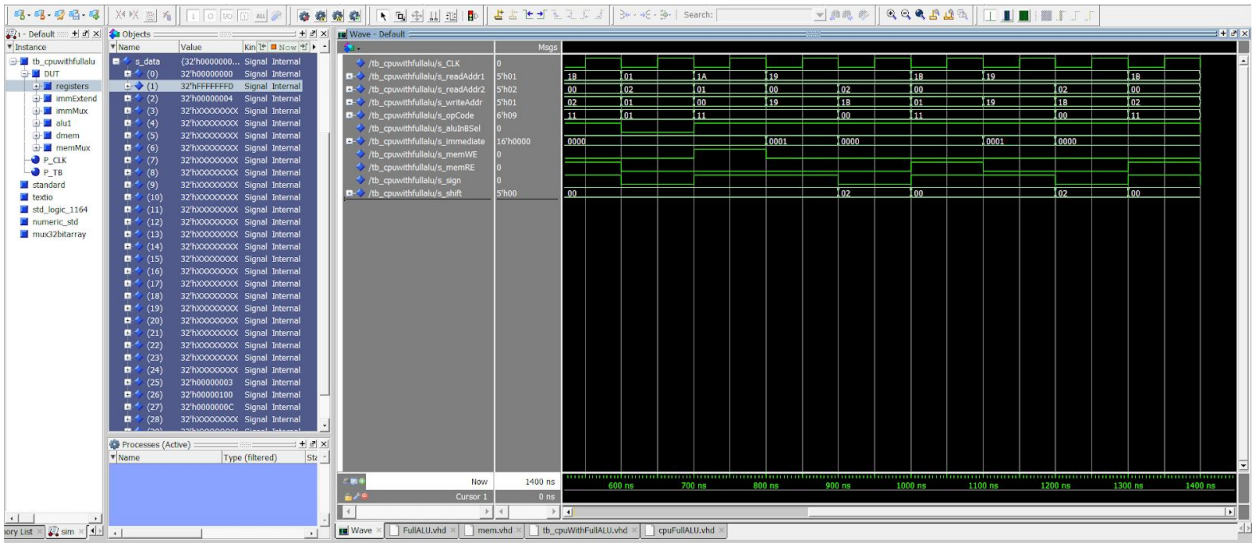
Sw $1, 0($26)



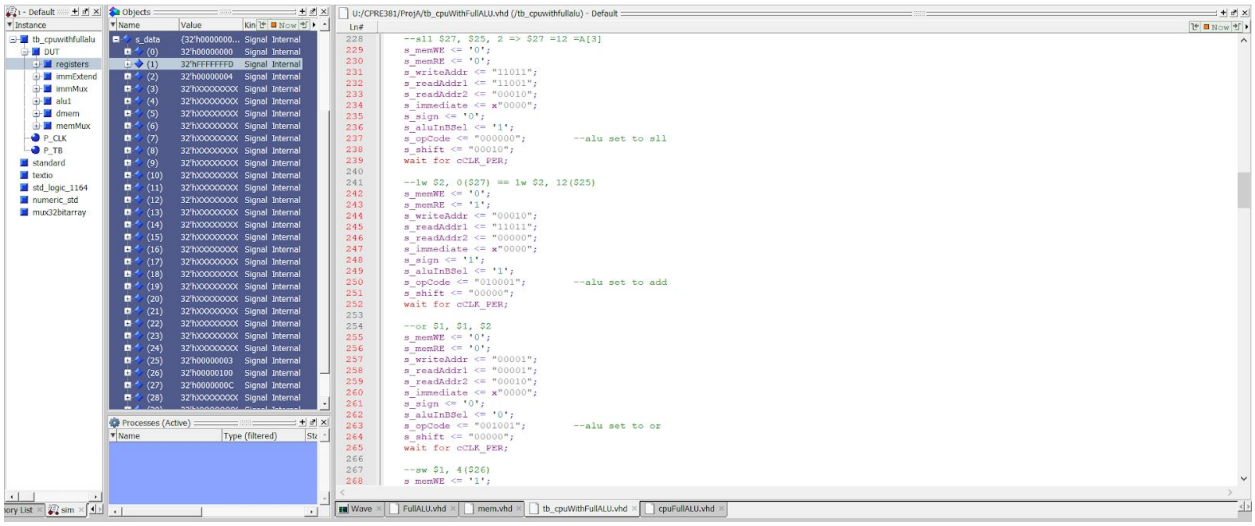
Addi $25, $25, 1
Sll $27, $25, 2
Lw $1, 0($27)



Addi $25, $25, 1
Sll $27, $25, 2
Lw, 2, 0($27)

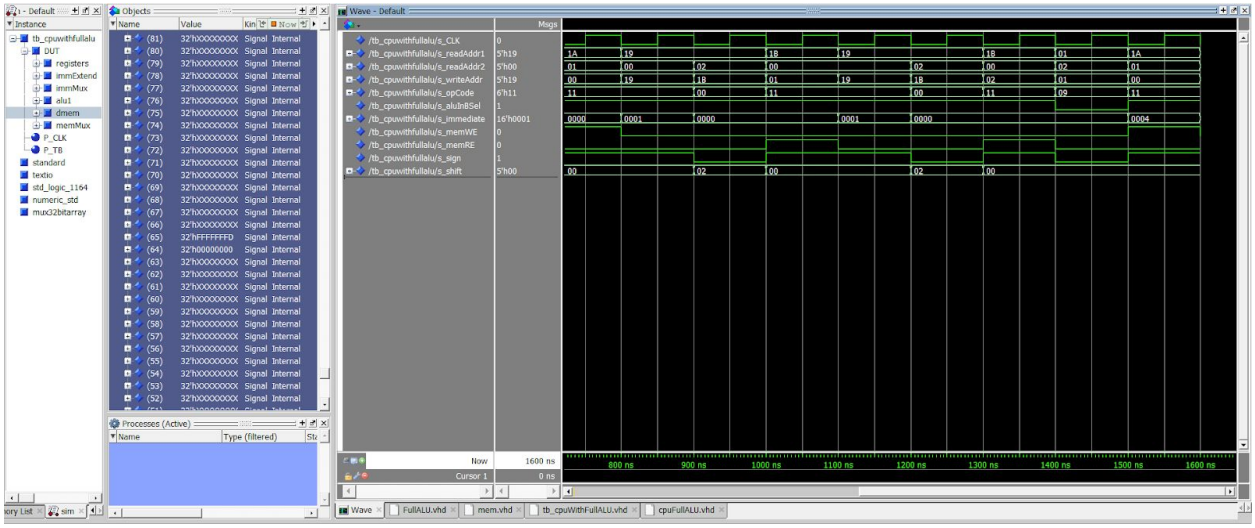


Or $1, $1, $2

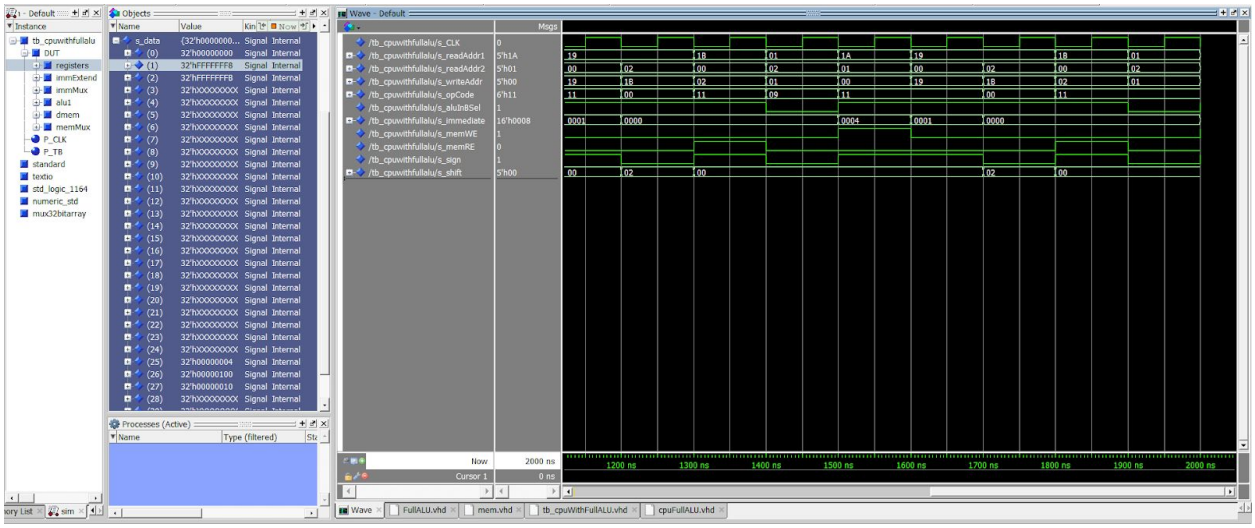


```
--sll $27, $25, 2 => $27 =12 =A[3]
s_memWE <= '0';
s_memRE <= '0';
s_writeAddr <= "11011";
s_readAddr1 <= "11001";
s_readAddr2 <= "00010";
s_immediate <= x"0000";
s_sign <= '0';
s_aluInBSel <= '1';
s_opCode <= "000000";        --alu set to sll
s_shift <= "00010";
wait for cCLK_PER;

--lw $2, 0($27) == lw $2, 12($25)
s_memWE <= '0';
s_memRE <= '1';
s_writeAddr <= "00010";
s_readAddr1 <= "11011";
s_readAddr2 <= "00000";
s_immediate <= x"0000";
s_sign <= '1';
s_aluInBSel <= '1';
s_opCode <= "010001";        --alu set to add
s_shift <= "00000";
wait for cCLK_PER;

--or $1, $1, $2
s_memWE <= '0';
s_memRE <= '0';
s_writeAddr <= "00001";
s_readAddr1 <= "00001";
s_readAddr2 <= "00010";
s_immediate <= x"0000";
s_sign <= '0';
s_aluInBSel <= '0';
s_opCode <= "001001";        --alu set to or
s_shift <= "00000";
wait for cCLK_PER;

--sw $1, 4($26)
s_memWE <= '1';
```

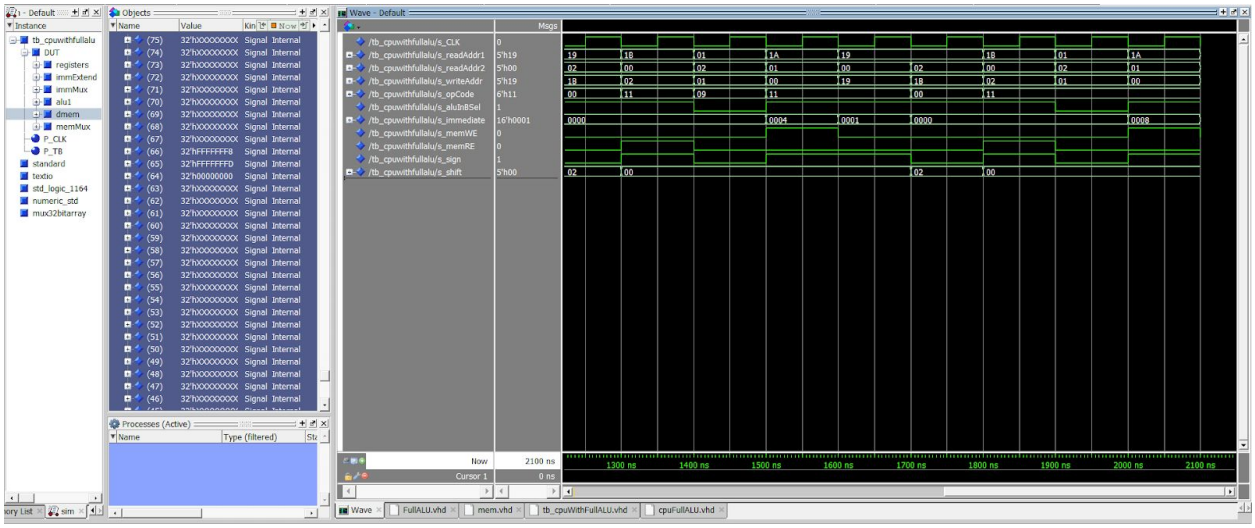
Sw $1, 4($26)



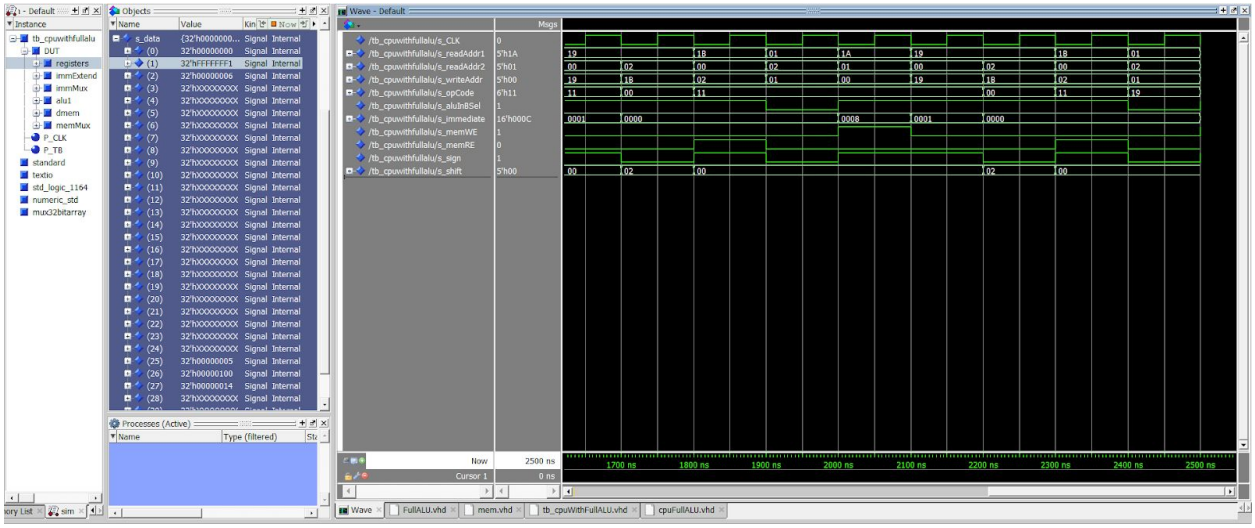
Addi $25, $25, 1
Sll $27, $25, 2
Lw $2, 0($27)
Add $1, $1, $2



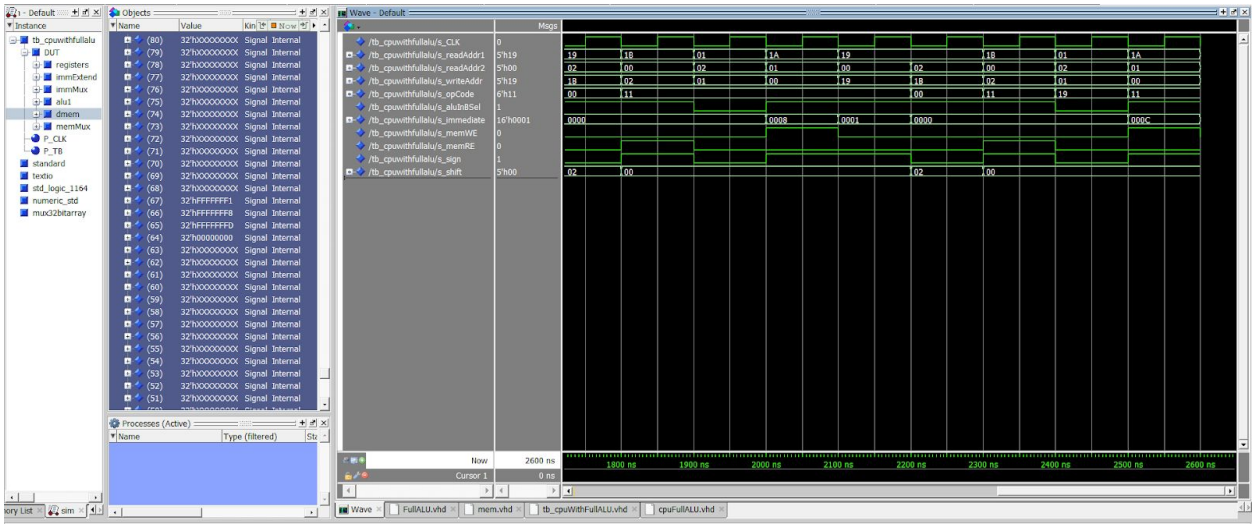
Sw $1, 8($26)



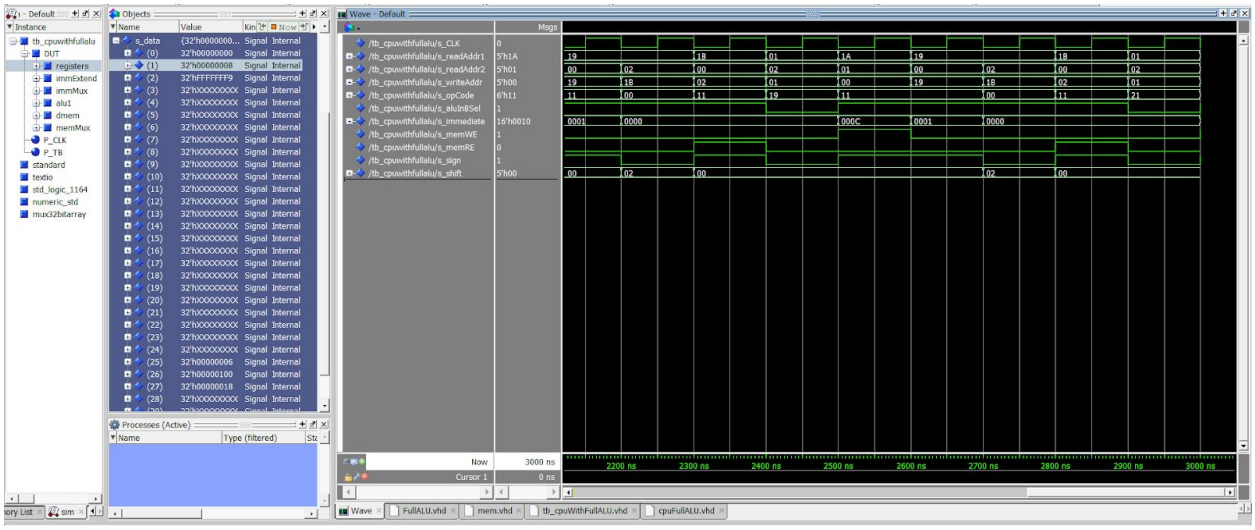
Addi $25, $25, 1
Sll $27, $25, 2
Lw $2, 0($27)
Sub $1, $1, $2



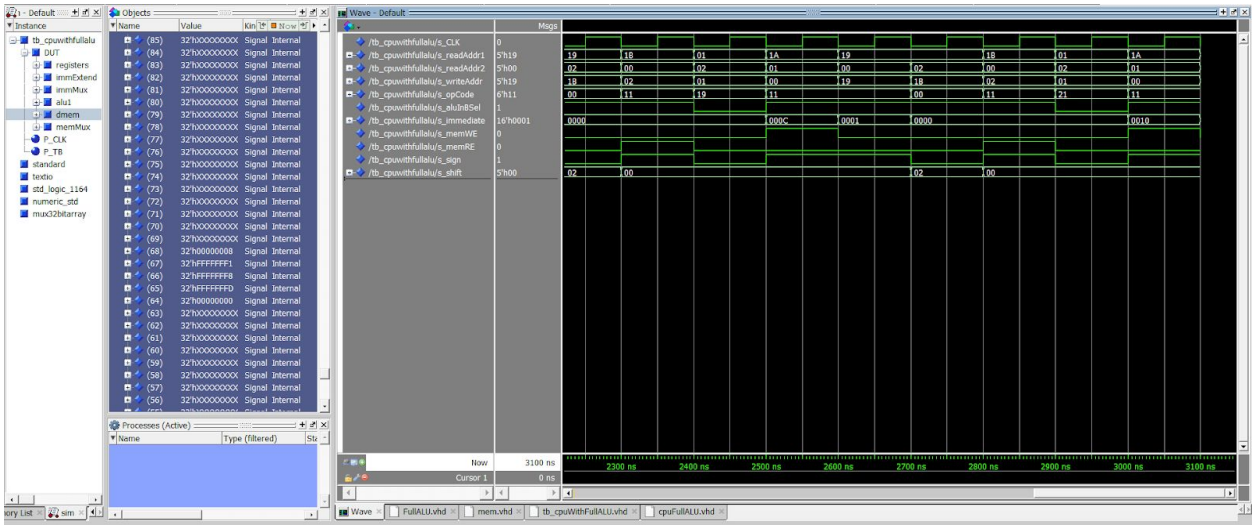
Sw $1, 12($26)



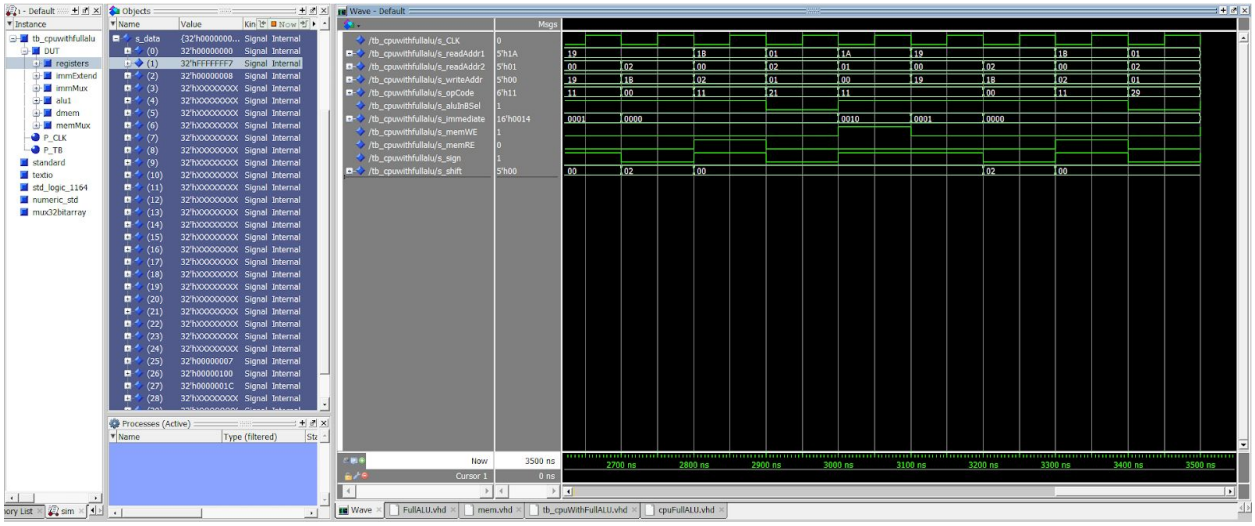
Addi $25, $25, 1
Sll $27, $25, 2
Lw $2, 0($27)
Xor $1, $1, $2



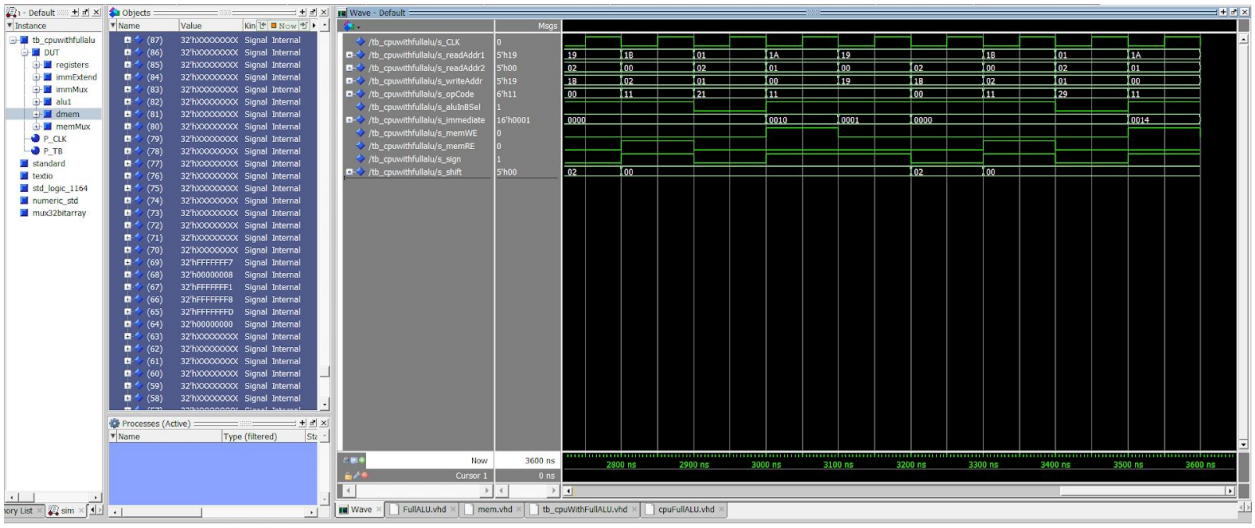
Sw $1, 16($26)



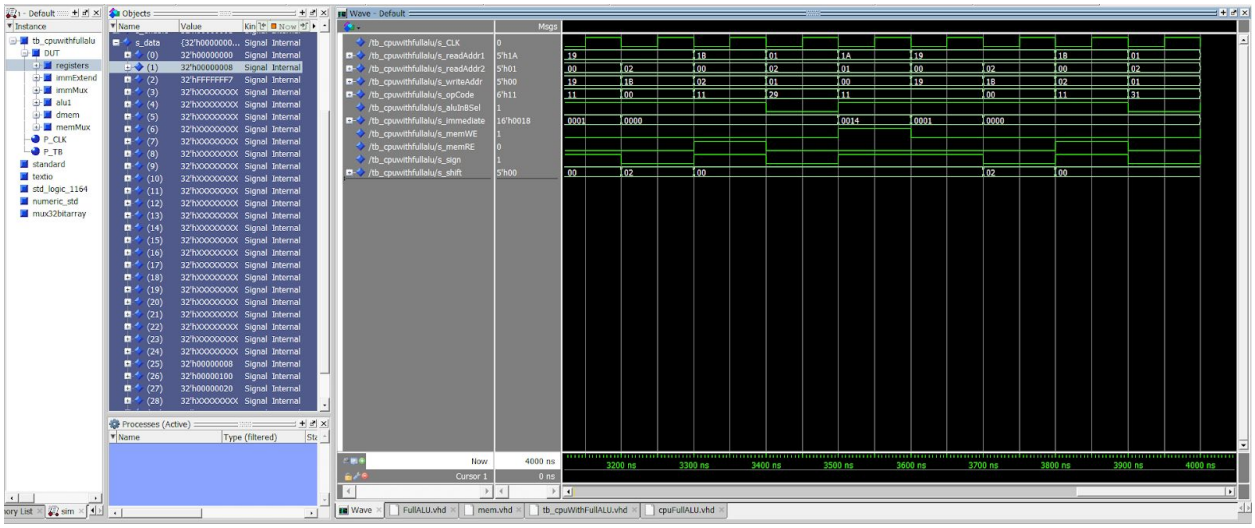
Addi $25, $25, 1
Sll $27, $25, 2
Lw $2, 0($27)
Nand $1, $1, $2



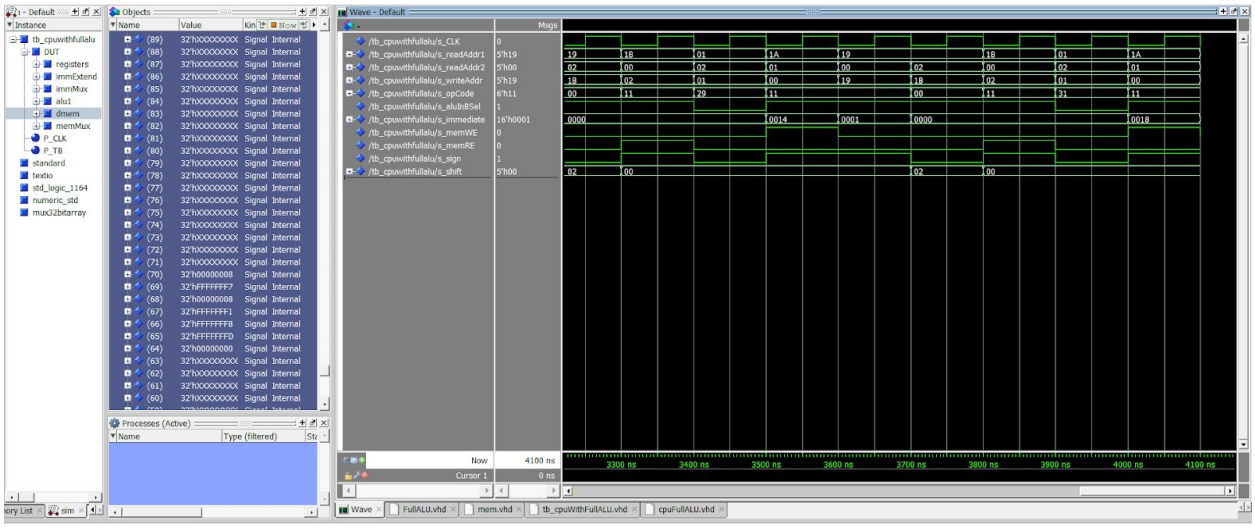
Sw $1, 20($26)



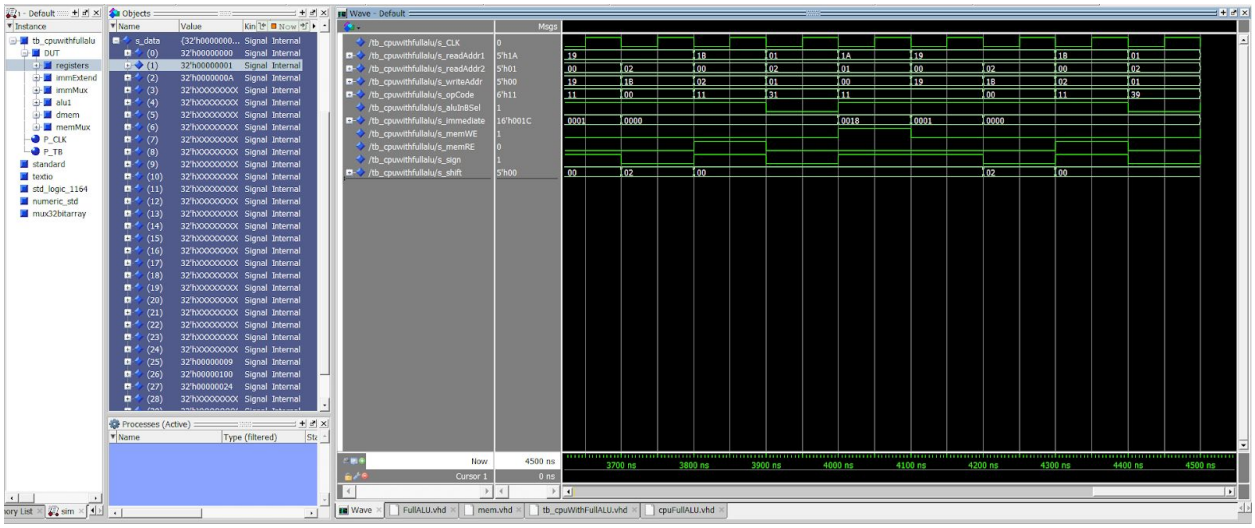
Addi $25, $25, 1
Sll $27, $25, 2
Lw $2, 0($27)
Nor $1, $1, $2



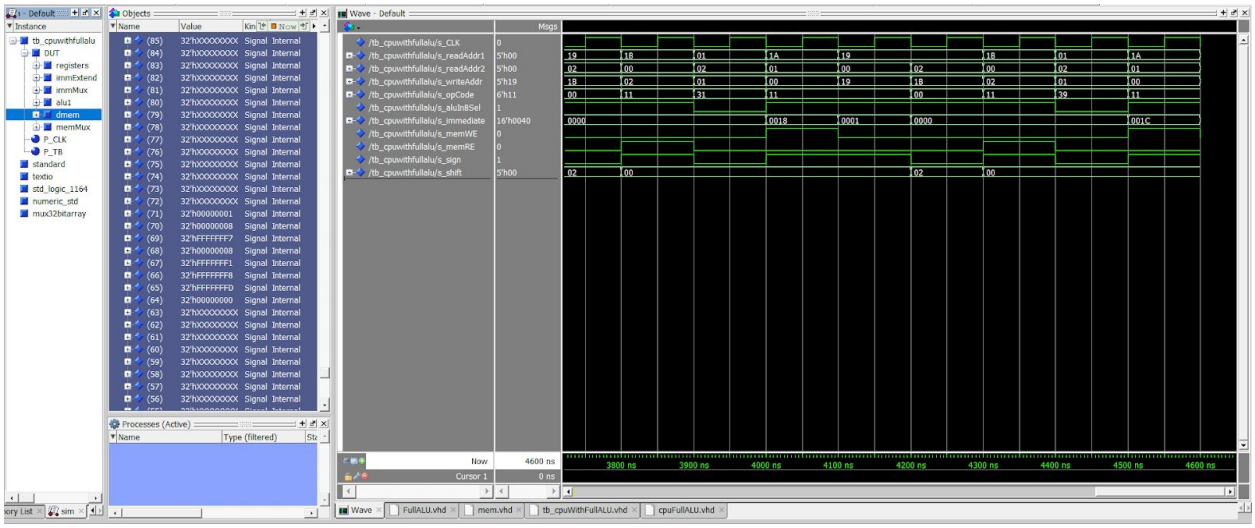
Sw $1, 24($26)



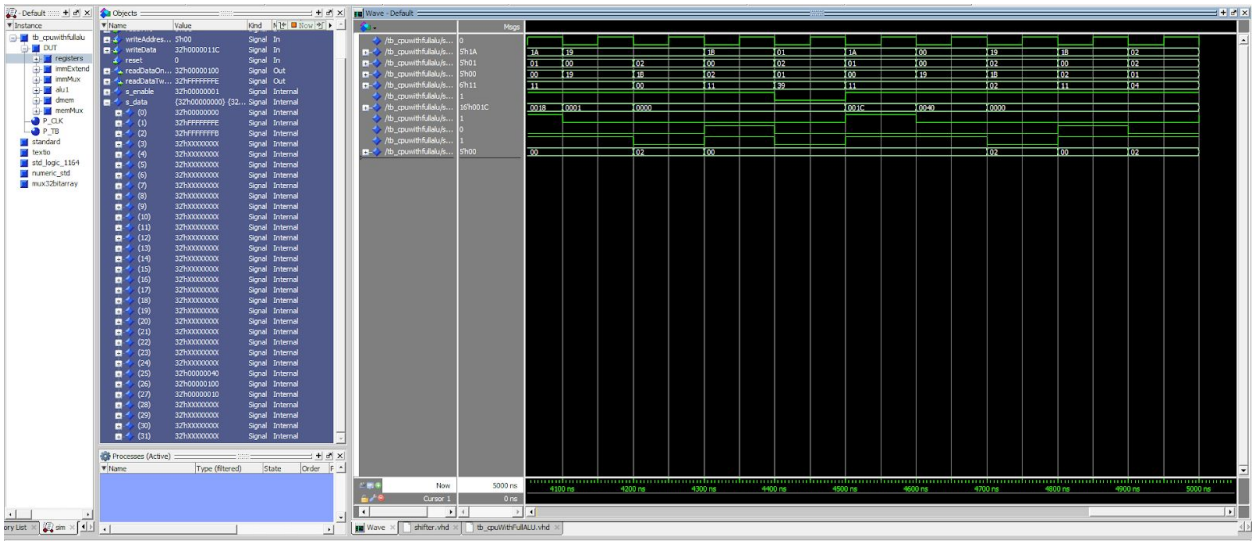
Addi $25, $25, 1
Sll $27, $25, 2
Lw $2, 0(27)
Slt $1, $1, $2



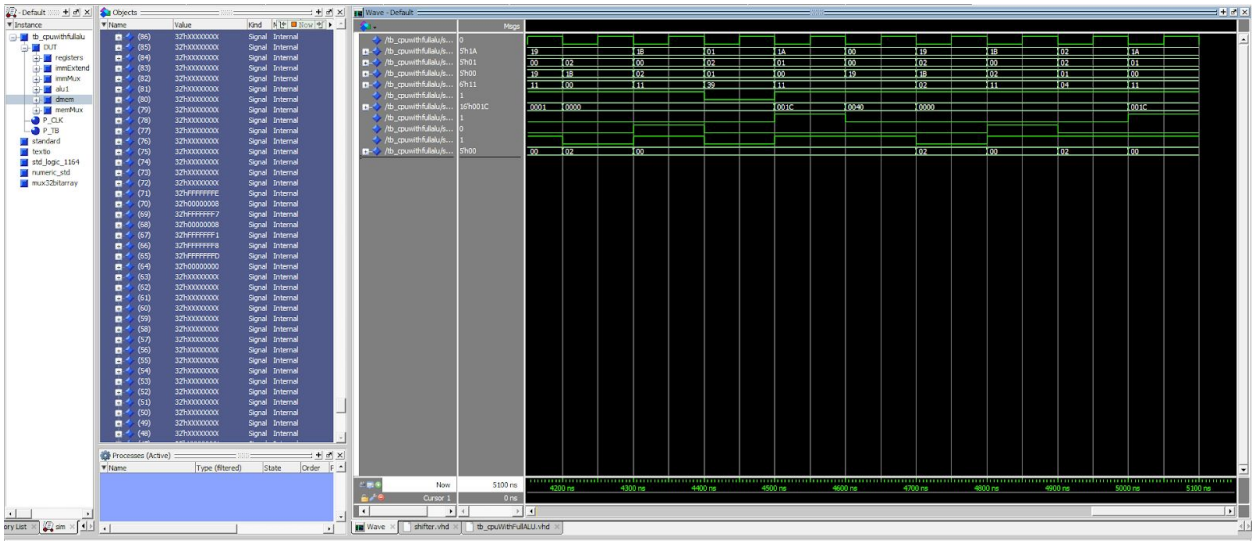
Sw $1, 28($26)



Addi $25, $0, 64
Srl $27, $25, 2
Lw $2, 0(27)
Sra $1, $2, 2



Sw $1, 28($26)



m. [Part 5(c) BONUS] Justify why your test plan is comprehensive. Include waveforms that demonstrate your test program is functioning.

Post-Lab Questions

n. [Feedback] You must complete this section for your lab to be graded. Please complete each column **separately** for each team member; I expect it to take roughly 10 minutes (do not take more than 20 minutes).

i. How many hours did you spend on this lab?

Task	**During lab time**			**Outside of lab time**		
Team Initials	EF	EM		EF	EM	
Reading lab	1 hr	1 hr		1 hr	.5 hr	
Pencil/paper design	.4 hr	.4 hr		.5 hr	1 hr	
VHDL design	1.6 hr	1.6hr		4 hr	6 hr	
Assembly coding	0	0		0	0	
Simulation	1 hr	1 hr		2 hr	2 hr	
Debugging	0	0		1 hr	3 hr	
Report writing	0	0		2 hr	2 hr	
Other:						
Total	4 hr	4hr		10.5	14.5	

ii. If you could change one thing about the lab experience, what would it be? Why?
EF: Off the top of my head I don't think I would change anything about this lab.

EM: Outside of the small bug that held me up for a while this was a pretty straight forward lab. I would suggest possibly making it clearer that we are meant to have two separate 1 bit ALU's, and how slt works.

iii. What was the most interesting part of the lab?
EF: For me, the most interesting part of the lab was eventually understanding how the Barrel shifter worked. I also enjoyed the ALU part of this lab as well, because it's kind of fun once you get your code working and start seeing desired results in the simulations.

EM: Getting to see the entire alu work with our datapath that was connected to memory like an almost fully functioning MIPS processor. I also think its pretty cool that we got to think ahead about what we might want our op codes to look like for our instructions later on.