CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: _____Alek Norris_____

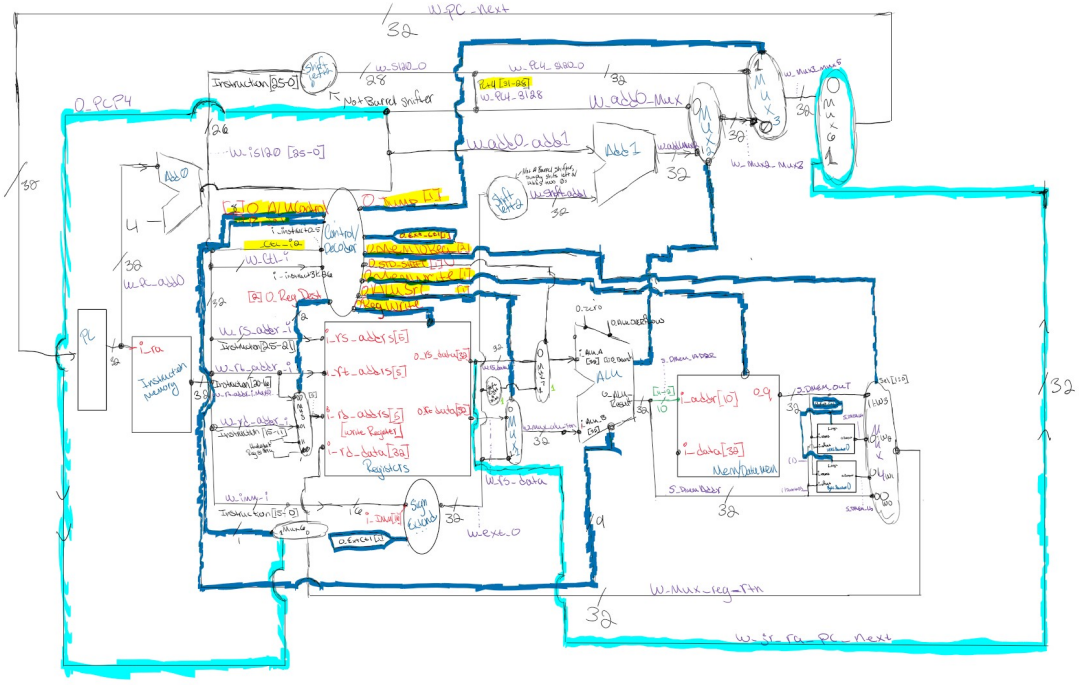
_____Drew Kearns _____

_____

Project Teams Group #:_____Term Proj1_2_07_____

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

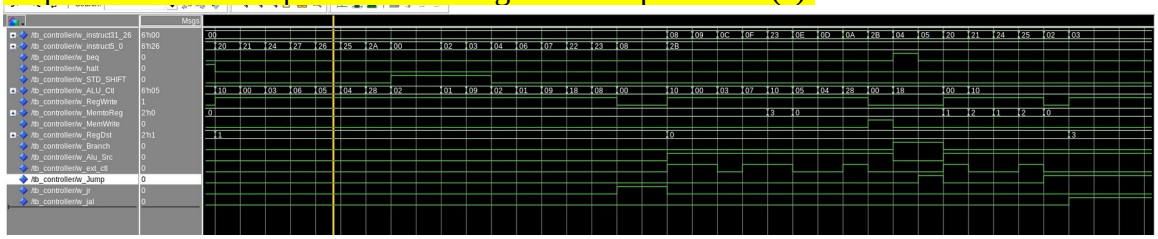
[Part 1 (d)] Include your final MIPS processor schematic in your lab report.



[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your datapath implementation. The end result should be an *N***M* table where each row corresponds to the output of the control logic module for a given instruction.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1						Control Signals												
2	Instruction	ALU Operation	Instruction Type	Opcode [Binary][31-26]	Funct [Binary][5-0]	o_halt	o_STD_Shift[1]	ALUSrc (i_ALU_src) [1]	ALUControl (i_ALU_C) [3]	o_MemToReg [2]	o_MemWrite [1]	o_RegWrite [1]	o_RegDst [2]	o_Jump[1]	o_ext_C [1]	o_jal_c [1]	o_jr [1]	Tests Pass Or Fail
3	add	add	R	"------"	"100000"	0	0	0	00010000	00	0	1	01	0	0	0	0	Pass
4	addi	add	I	"001000"	"------"	0 (not jump returning)	0 (not jump returning)	1 (using immediate value)	00010000	00 (not reading from memory)	0 (not writing to memory)	1 (writing to RegDst register)	00	0 (not jumping)	1 (addi is signed addition)	0 (not jumping)	0 (not jump returning)	Pass
5	addiu	addu	I	"001001"	"------"	0	0	1	00000000	00	0	1	00	0	1	0	0	Pass
6	addu	addu	R	"------"	"100001"	0	0	0	00000000	00	0	1	01	0	0	0	0	Pass
7	and	and	R	"------"	"100100"	0	0	0	00000011	00	0	1	01	0	0	0	0	Pass
8	andi	and	I	"001100"	"------"	0	0	1	00000011	00	0	1	00	0	1	0	0	Pass
9	lui	lui	I	"001111"	"------"	0	0	1	00010111	00	0	1	00	0	0	0	0	Pass
10	lw	add	I	"100011"	"------"	0	0	1	00010000	11	0	1	00	0	1	0	0	Pass
11	nor	nor	R	"101111"	"------"	0	0	0	00000110	00	0	1	01	0	0	0	0	Pass
12	xor	xor	R	"100110"	"------"	0	0	0	00000101	00	0	1	01	0	0	0	0	Pass
13	xori	xor	I	"001110"	"------"	0	0	1	00000101	00	0	1	00	0	1	0	0	Pass
14	or	or	R	"------"	"100101"	0	0	0	00000100	00	0	1	01	0	0	0	0	Pass
15	ori	or	I	"001101"	"------"	0	0	1	00000100	00	0	1	00	0	0	0	0	Pass
16	slt	slt	R	"------"	"101010"	0	0	0	00111000	00	0	1	01	0	1	0	0	Pass
17	slti	slt	I	"001010"	"------"	0	0	1	00111000	00	0	1	00	0	1	0	0	Pass
18	sll	sll	R	"------"	"000000"	0	1	0	00000010	00	0	1	01	0	1	0	0	Pass
19	srl	srl	R	"------"	"000010"	0	1	0	00000001	00	0	1	01	0	1	0	0	Pass
20	sra	sra	R	"------"	"000011"	0	1	0	00001001	00	0	1	01	0	1	0	0	Pass
21	sw	add	I	"101011"	"------"	0	0	1	00010000	00	1	0	00	0	1	0	0	Pass
22	sub	sub	R	"------"	"100010"	0	0	0	00011000	00	0	1	01	0	0	0	0	Pass
23	subu	subu	R	"------"	"100011"	0	0	0	00001000	00	0	1	01	0	0	0	0	Pass
24	beq	beq	I	"000100"	"------"	0	0	0	11001000	00	0	0	00	0	0	0	0	Pass
25	bne	bne	I	"000101"	"------"	0	0	0	01001000	00	0	0	00	0	1	0	0	Pass
26	j	add	J	"000010"	"------"	0	0	1	00010000	00	0	0	01	1	0	0	0	Pass
27	jal	add	J	"000011"	"------"	0	0	1	00010000	00	0	1	11	1	0	1	0	Pass
28	jr	add	R	"------"	"001000"	0	1	0	00010000	00	0	0	01	0	0	0	1	Pass
29	lb	add	I	"100000"	"------"	0	0	1	00010000	01	0	1	00	0	1	0	0	Pass
30	lh	add	I	"100001"	"------"	0	0	1	00010000	10	0	1	00	0	0	0	0	Pass
31	lbu	add	I	"100100"	"------"	0	0	1	00010000	01	0	1	00	0	0	0	0	Pass
32	lhu	add	I	"100101"	"------"	0	0	1	00010000	10	0	1	00	0	0	0	0	Pass
33	slv	sll	R	"------"	"000100"	0	0	0	00000010	00	0	1	01	0	0	0	0	Pass
34	srlv	srl	R	"------"	"000110"	0	0	0	00000001	00	0	1	01	0	0	0	0	Pass
35	srav	sra	R	"------"	"000111"	0	0	0	00001001	00	0	1	01	0	0	0	0	Pass
36	halt	sra	R	"------"	"010100"	1	0	0	00001001	00	0	0	01	0	0	0	0	Pass
37																		
38						ALU Control Signals	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]				
39							BEQ	Branch	SLT Bit	Signed/Ovflw, or not.	Shift Logical Or Arith	MUX Selector	MUX Selector	MUX Selector				
40																		
41						RegDst	[00]	[01]	[10]	[11]								
42						Mux for Rt, rd, or $31	rt	rd	Hardcoded reg 31	Hardcoded reg 31								
43																		
44						MemToReg	[00]	[01]	[10]	[11]								
45						Mux for Lw, lhu,lb,lbu	ALU Return	Lb,Lbu	Lh, Lhu	Memory Return								

[Part 2 (a.ii)] Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually and show that your output matches the expected control signals from problem 1(a).



For this part to verify everything was working, I just simply set up a test bench to push every single instruction code, in the image you can see the first half using changing in instructions [5:0] or the funct field for R types. For he second half using changing in instructions corresponding to [31:26] or the opcode for I and J type instructions. I then went through and verified I was seeing the correct output on each line for our excel sheet.

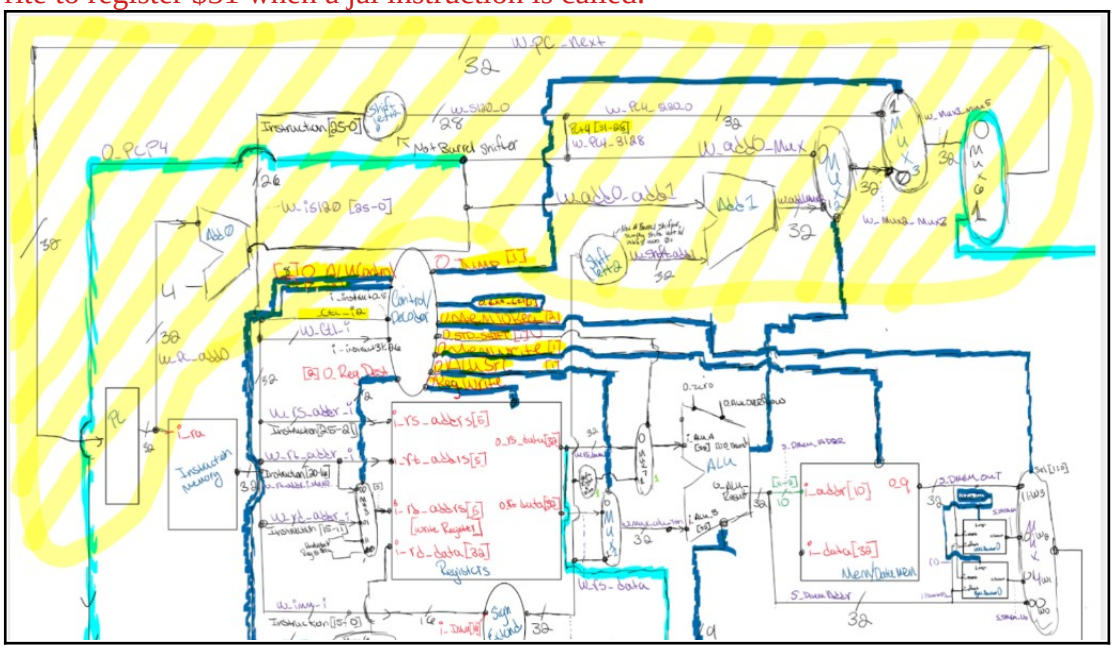
[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

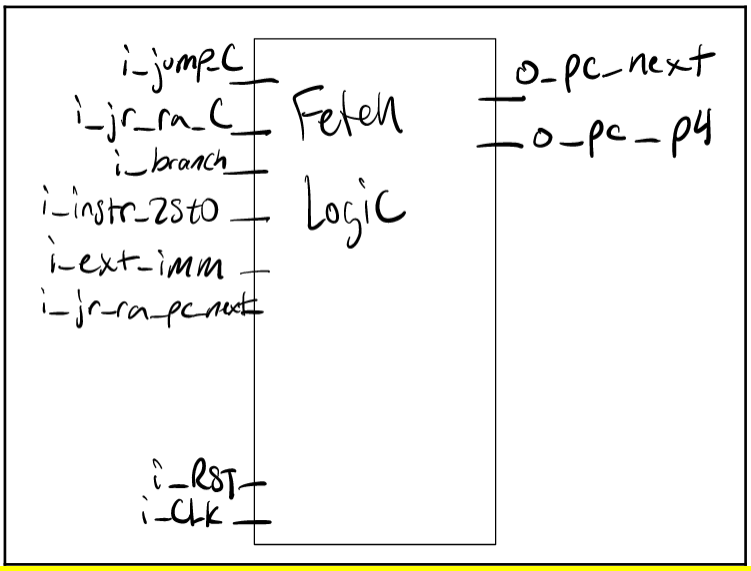
For references relating to control bits and other inputs, see either the fetch logic schematic in b.ii or the overall drawing in Part 1 d.

The fetch logic needs to be able to jump to a new instruction address, branch to a new instruction address, and return to an instruction address after a jr instruction is called, as well as doing the normal PC+4 operation. Jumping to a new instruction address will happen when a jump instruction (j) is called or a jump and link instruction (jal). These both perform the same jump operation in the fetch logic and the only difference is that the fetch logic output o_pc_p4 is stored in $31 after a jal instruction is called. The fetch logic will need to branch after a branch if not equal (bne) or a branch if equal (beq) instruction is called. The bne and beq operations are handled in the ALU and output a 1-bit control signal called o_branch that is fed into the fetch logic at MUX2. For a jump return (jr) instruction, the return address is fed into the 1 option in MUX6 as well as the control signal for a jr instruction.

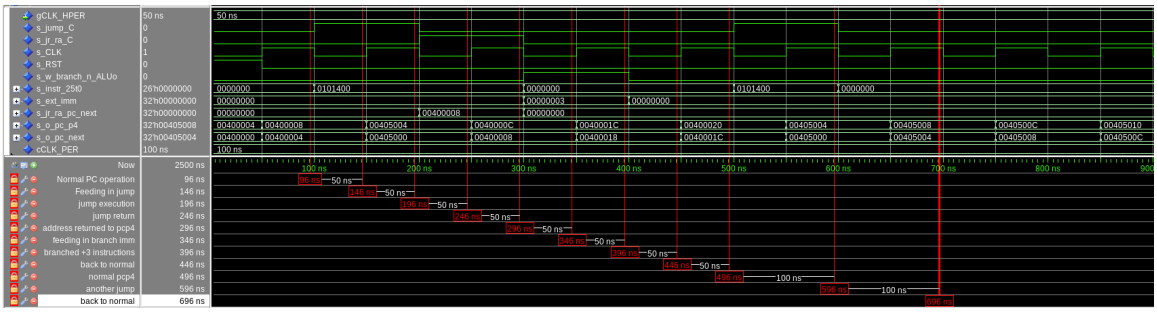
[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?

The fetch logic is what is highlighted in yellow below. It has 8 inputs; 5 control bits which are: i_jump_C, i_jr_ra_C, i_CLK, i_RST, i_branch; and 3 other inputs which are: i_instr_25t0, i_ext_imm, and i_jr_ra_pc_next. The two outputs are o_pc_next which is fed directly into the instruction memory and o_pc_p4 which is fed to a MUX that will write to register $31 when a jal instruction is called.





[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use QuestaSim to test your design thoroughly to make sure it is working as expected. Describe how the execution of the control flow possibilities corresponds to the QuestaSim waveforms in your writeup.



Note that in the screenshot above the branch control bit has a different name. This does not matter as that is all that was changed in the fetch logic since testing. As you can see in the waveform screenshot I have multiple tests for jumps, a test for a jump return, and a test for branching. All tests work as intended and while the control bits are 0 the fetch logic outputs a pc next of PC(previous)+4 as expected and a PC+4 for jal instructions.

[Part 2 (c.i.1)] Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does MIPS not have a `sla` instruction?

The difference between logical (SRL) and arithmetic (SRA) right shifts lies in how the vacated positions are filled. In a logical shift right (SRL), the vacated positions are padded with zeros. Conversely, in an arithmetic shift right (SRA), the vacated positions are filled with the value of the most significant bit (MSB) to preserve the sign of the original number; this means if the MSB is 1 (indicating a negative number in two's complement), the shift operation pads with 1s. MIPS does not include a Shift Left Arithmetic (SLA) instruction primarily because arithmetic left shifts do not require sign bit management—left shifts by their nature multiply the number by two for each shift position, preserving the sign. Furthermore, adding an SLA instruction would complicate the instruction set without offering a significant benefit, as the logical shift left operation (SLL) already achieves the desired outcome without altering the number's sign.

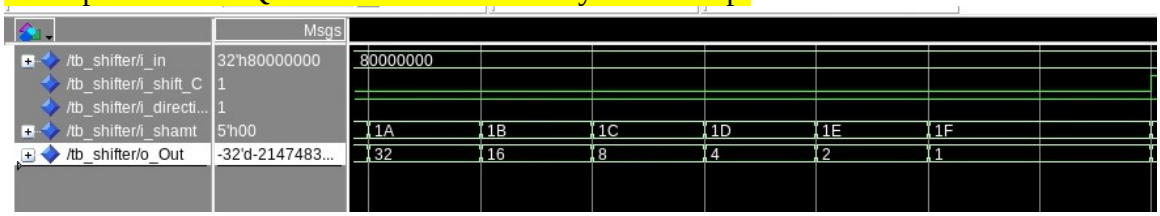
[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

It utilizes a hierarchical structure of 2-to-1 multiplexers (muxes) organized into five stages, each stage doubling the shifting ability from 1-bit shifts up to 16-bit shifts, allowing for shift amounts ranging from 0 to 31 bits. The type of shift (logical or arithmetic) is determined by a shift type signal, and the amount of shift is controlled by a 5-bit shift amount input. This design efficiently achieves variable bit shifting by selecting the appropriate path through the muxes based on the shift amount, with the fill bit for arithmetic shifts dynamically determined by the input's most significant bit (MSB).

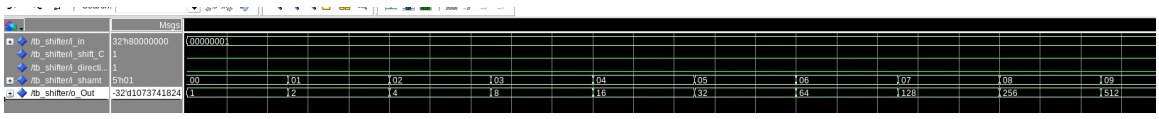
[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

To make a right barrel shifter work both ways, we add a simple trick: flipping the bits around before and after shifting. There's a new input called "shift direction" that lets us choose which way to shift. If we want to shift right, it just does its normal thing. But if we want to shift left, it first flips all the bits around, does a normal right shift, and then flips the bits back. This way, we end up with a shifter that can go both left and right.

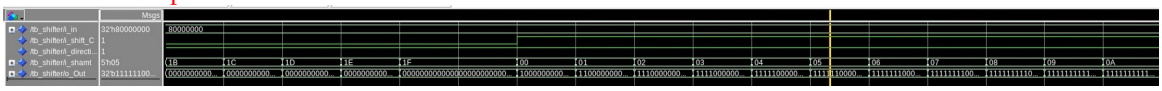
[Part 2 (c.i.4)] Describe how the execution of the different shifting operations corresponds to the QuestaSim waveforms in your writeup.



You can see here that when direction is 1(right) and the Control bit (logical), is set to 0 the the value decreases by dividing by 2 each time, until we get to 0. This is expected.



Here you can see that when the direction is set to 0, and the shift type is set to 1(logical), a normal bit shift occures, shifting the bit to the left by 1 place, effectivly doubling the value at every shift, the opposite of the above example.



In this example you can the right shift at work with the control bit set to 1 for arithmetic, in this setting the shifter adds a bit to the msb slot and removes one from the lsb slot each time. When adding bits, because a 1 is set in the msb at start, the shifter adds a 1 bit every time.

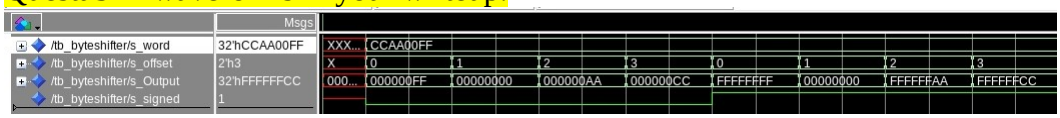
[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

The Design approach we took was to first figure out what all the major things we needed to finish the design were. We then broke them down into smaller parts such as the ALU, control, Fetch logic, and register. Anything outside these were considered extra things needing to be implemented on the overall processor design.
We then go to work making and testing the individual parts separately with testing, and building up to throwing it all together. Once thrown together we were able to start running some real test benches with the provided tests and verify what we had working and what we didn't have working. The following is a list of additional things we needed to implement.

Mux4t1_N, Mux8t1_N,Nor_N,Or_N,Xor_N,And_N, mux32t1, ByteShifter, and WordShifter … as well as I'm sure there's some missing.

One of the design choices we decided to make was implementing a ByteShifter, and a WordShifter, these both work similarly, but do different things. Because of once everything was assembled, and all the basic ALU components passed their test, as well as LW, we needed a way to be able to implement a Lb, lbu, lh, and lhu. After looking at the outputs it became clear the we had the right chunk of memory, but not the right piece we were looking for, and because we knew that out CPU was word addressable there must be another way, outside of the ALU and DMEM to get the desired output. So that's why we came up with these decoders. These decodes sit just between the mux used for Mux2reg(mux4 in our design) and the DMEM. These decoders then, in parallel hook up to the dmem and the alu output, and then into the mux4t1_N we implemented. Then depending on a ctl signal from the controller, pics between the decoders or the regular outputs etc. if Byte decoder is selected, it pulls off the desired byte, with offset. Then pads depending on the MSB and an ext type control signal. The word decoder works the same, but instead of having 4 offset possibilities, it only has 2. It also relies on bit (1) of the alu out to determine the shift amount, while the byte decoder relies on bits [1:0]

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

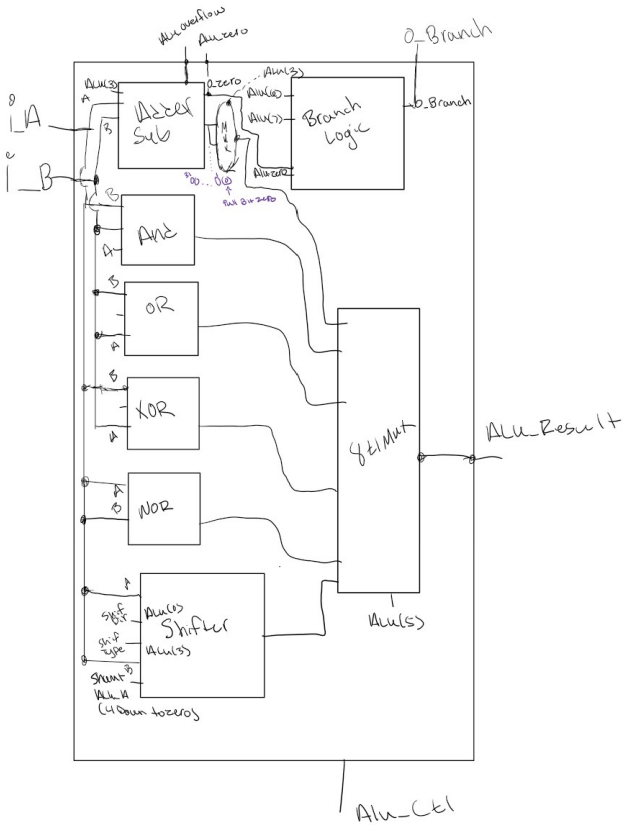


Showing the ByteShifter Above, and the test bench that was designed for it. The shifter works by having a decoder as the base, that is hooked up to an offset input. The offset input comes in, is decoded into an offset of 5 bits, then feeds into a left shifter depending on the decoder input. The shifter then outputs the fully left shifted data, and then a hardcoded shifter shifts the bits 24 times to the right. To determine sign, the 32nd bit is stripped in between the first output and the second shifter input, and is read at an and gate. If the second input to the and gate is a 1(meaning we want a signed extension) it will propagate the 1 bit to the second shifter if, and only if that 31st bit from the first output was a 1. The Word shifter, not show is the same, except with only two possible offsets, and 16 bit shifts only.

You can see in the testbench when the sign bit is 1, and the 32nd bit is 1, it sign extends, and you can also see, depending on the offset, 0,1,2,3 it will grab different bytes of the

input data, 00 = byte 1, 01 = byte 2 etc. the shift amount is determined on the byte shifter form the alu out bits [1:0] and for the word shifter, it uses bit (1) only

[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is `slt` implemented?

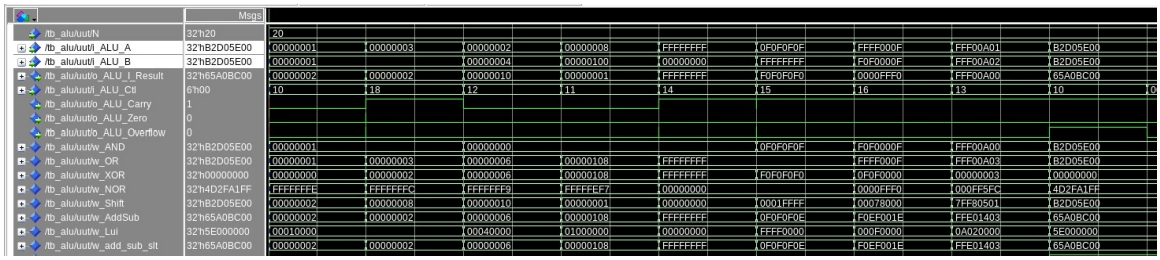


Overflow – Overflow is calculated in the adder/sub component in the adder specifically. It it calculated by xoring the N bit and the N-1 bit, or the msb bits and the msb bit-1. The carry out of the msb bit must be equal to the carry in or its considered an overflow, and

the msb has changed when it should not have, otherwise, we have surpassed the most representable number.

Zero – overflow is calculated in the adder subtractor part of the ALU, and a flag is thrown when the output is equal to 0 from here. Otherwise the output is 0 on the flag**.**

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



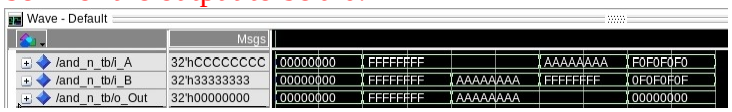
To keep things simple, I just did a simple test for everything put together. More extensive tests were done on individual parts, So all this was is a matter of throwing everything together with a mux selector and then watching the magic happen. You can see from left to Right, using ALU A, B, then result to see what came out. From left to right, each change of number, signals a change of operation. Add, subtract, bit shift B by A left, B by A right, OR, XOR, NOR,AND, AND, Add with overflow, and ADDU without overflow with the same numbers. You can see they are controlled by the ALU_CTL bits, which for the most part just handle the switching of the mux. So everyhting is used every cycle, however, only what we want leaves the ALU. You can also see here that the Zero is never triggered, and the ALU overflow is only triggered when the regular add is done and not the addu.

[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

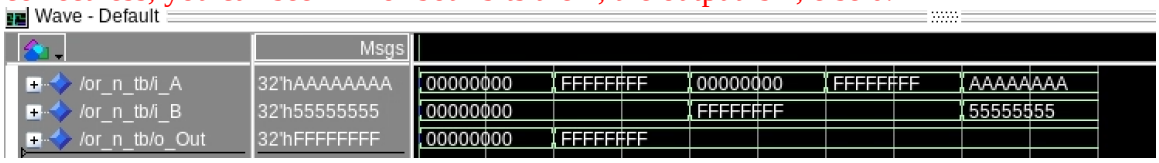
Please let me know if we need to make another test bench for the ALU, but for this, we are just showing you again, what we had done earlier. The reason no additional Testbench for the ALU were done was because additional Testing on all the individual components added were done. The Adders had been tested for overflow, and the adder subtractor has been tested and confirmed to work with the zero flag. Because the adder subtractor uses addition to subtract, and the internal components to get to the adder overflow had been tested quite well, with edge cases prior to this lab. So, we had come to the conclusion that the adder and subtractor were working good, and no unexpected behavior has or had arisen during this lab. The following are test cases from the AND gate, or gate, nor gate, and xor gate, mux8t1, and the shifter has already been mentioned

above so I Will leave that out. All testbenches have been included in the report in a file called ALU, there are additional sub sections for the test benches to be tried and ran if needed. Let me know if you'd like us to resubmit with a more comprehensive test bench, however, because of the time of writing this, we have already finished all the provided tests, I don't see the need to.

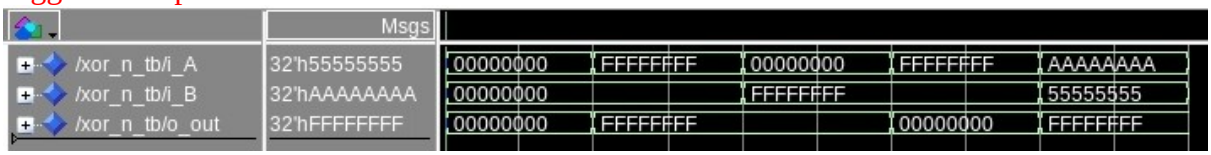
AND here is an and gate at working being test on every bit, both bits from A and B must be 1 for the output to be a 0.



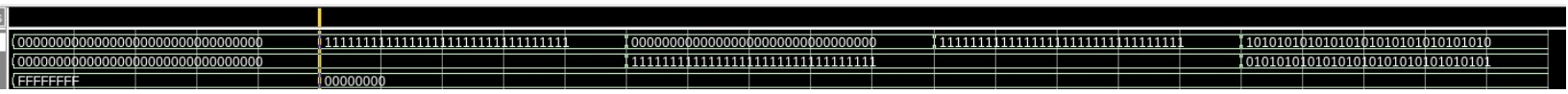
OR here you can see the test benches for the OR gate being tested on every bit, for correctness, you can see if 1 or both bits are 1, the output is 1, else 0.



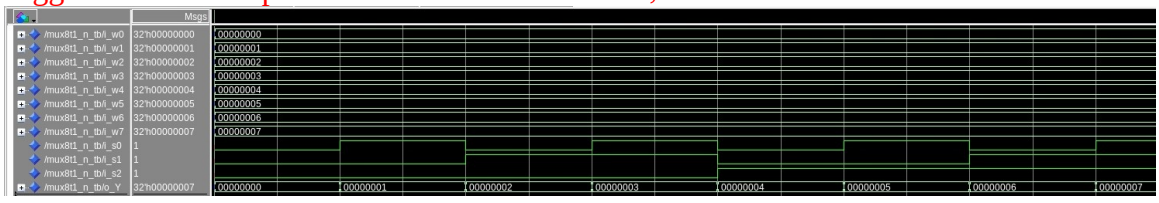
XOR – here you can see a basic xor gate at work, one or the other but not both will trigger an output of 1.



NOR - You can see here, line 1 = A, line 2 = input b, line 3 = output, only a 0, 0 will trigger a 1 output.



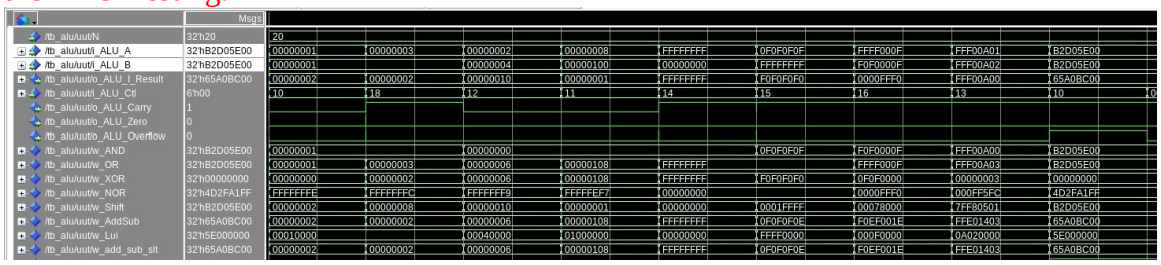
Mux8t1_N – Here you can see the 8 to 1 being tested for 32 bits, each select line is triggeredand the output can be seen below. 0 = w0, 1=w1….



SHIFTER – Testbench can be seen above, in earlier section of report.

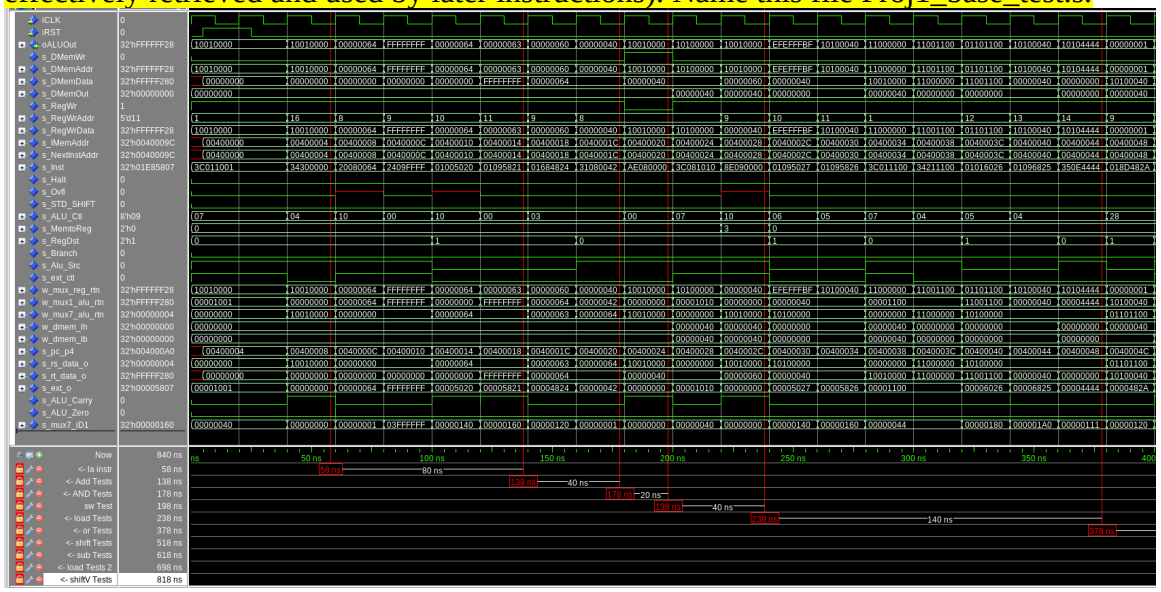
ALU- Below is the test bench for the ALU, as I mentioned above, because of our design approach to test everyhting individually, and because of the simple construction of our

ALU, and armed with the knowneledge of more gueling test benches coming up once everything is assembled we choose to wait to test it more exetesivly, as It would take a lot of time, and the basic component testing has already been done quite well, so, since evrything was able to be seen clearly switching between eachother, we stopped here with the ALU Testing.



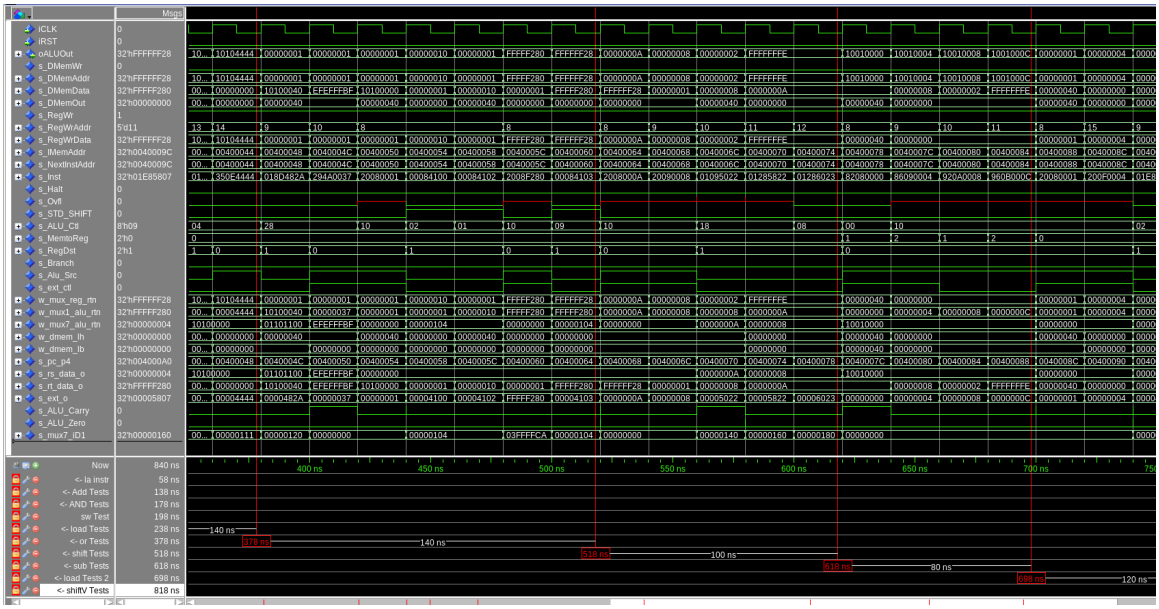
[Part 3] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

[Part 3 (a)] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 1 per cycle while data written into registers can be effectively retrieved and used by later instructions). Name this file Proj1_base_test.s.

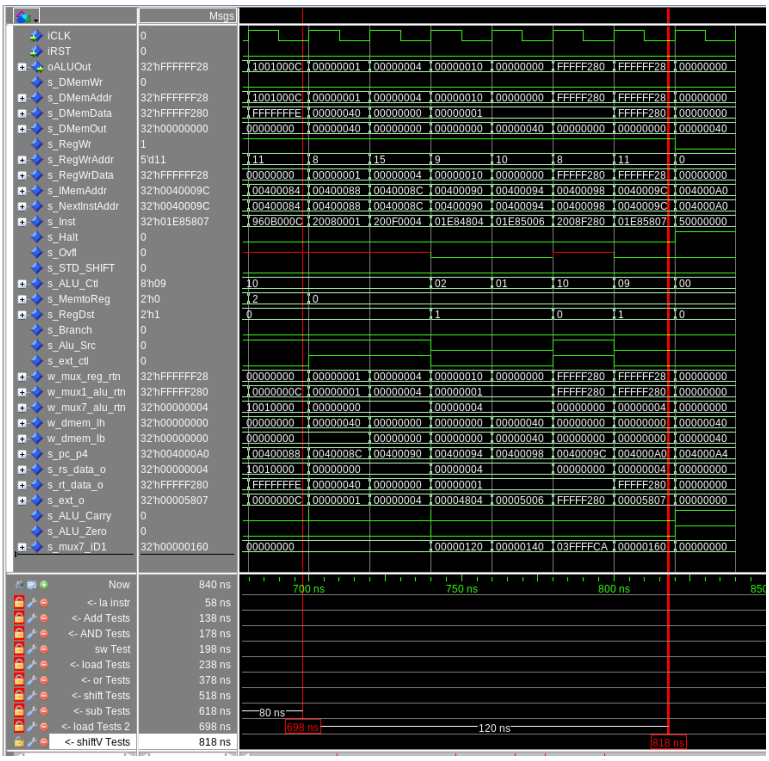


Seen above are the addTests (58-138), andTests (138-178), swTest (178-198), loadTests1 (198-238), and orTests (238-378). As you can see the two instructions to the right of the red line, the ext_ctl is 1 due to those instructions being addi instructions and then 0 after because add does not require sign extension of immediates. The ext_o signal is also outputting the correct data (0x64=100d for the first instruction and 0xFFFFFFFF=-1d for the second instruction). For the RegDst signal, it is 0 for the first two instructions because I-type instructions write to Rt and 0 for the second two instructions because R-

type instructions write to Rd. The RegWrData also shows the correct data for each instruction.
For the and instruction you can see that the rs/rt_data_o is showing what was just calculated in the add instructions. Both and instructions have ext_o as 0 because they don't sign extend. The values in RegWrData are what I'd expect to see from this instruction.
For the sw instruction I see that the DMemWr signal has gone from 0 to 1, which is exactly what I expect, and the RegWr signal goes from 0 to 1. The DMemData is the value from $t0 as expected.
The load tests reset the DMemWr signal to 0 and set the RegWr signal to 1. ALU_Src is 1 because an immediate is used for both load instructions.
For every or instruction we see that the RegWr is always 1, DMemWr is always 0, the ext_ctl is always 0 except for psuedo instructions. All Data outputs are what I would expect to see.

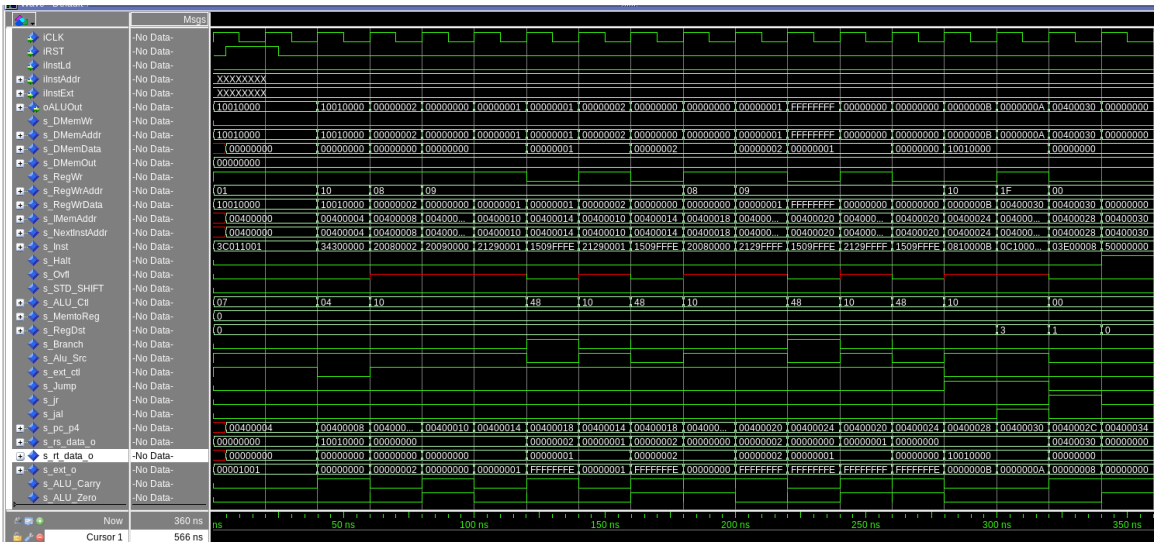


Above are the tests for shift (378-518), sub (518-618) and load 2 (618-698).
For the first shift test I see that all non ALU signals are 0 except for RegWr which is expected since shifting is done in the ALU. For the second shift test I see ext_ctl is 1 and so is Alu_Src which is expected because it's an I-type instruction. For the shift instructions after the addi instruction I the STD_SHIFT is 1 which is accurate because they are non-conditional shifts.
The first two instructions after the shift tests have an ext_ctl and Alu_Src of 1 because they are addi instructions, setting values in registers for the sub tests. We can see in the first sub test we have 10 – 8 which yields 2 in the RegWrData output, and in the second sub test we have 8-10 which yileds -2 in the RegWrData. For the subu test, the RegWrData is the same as the previous test but will show up as positive in future tests. The control bits for the sub tests are as expected.
For the load2 tests, the first two instructions are signed as we can see with the ext_ctl bit being 1 and the second two instructions are unsigned as seen with the ext_ctl bit being 2. Since lb uses an immediate, the Alu_Src is 1. All other control bits appear as expected.



Above are the final tests which are shift variable tests from 698-818 and the halt instruction right after. The first two instructions in this section are addi instructions, setting register $t0 to 1 and $t7 to 4 which is the value to be shifted by. The RegWrData is showing the expected values for these instructions and as expected, all non Alu control bits besides RegWr are 0 as shifting happens only in the ALU. The last instruction is testing the srav and we can see that sign extension is happening because the sign extension bit is 1 in the ALU control signals.

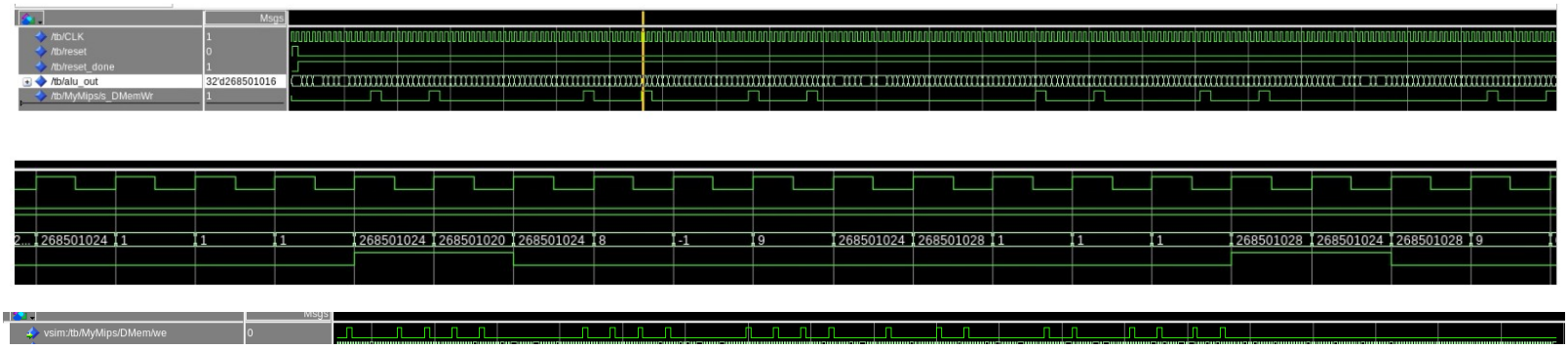
[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.



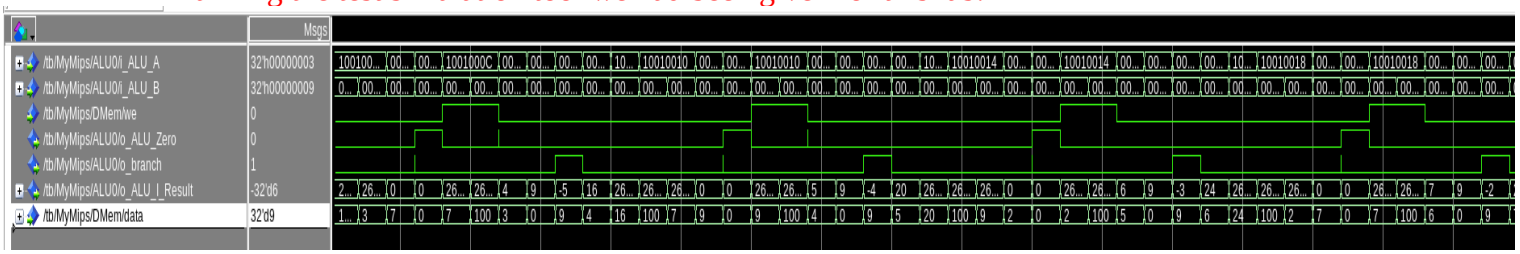
Above is the waveform for the control flow tests. The first two instructions are using add to get values in the registers, and the the following tests upuntil the s_Jump signal becomes 1 are beq and bne tests. When the s_Jump signal first becomes 1, that is an unconditional jump which jumps to a jal instruction which jumps backwards and links to a jr instruction which then returns to the halt instruction. This way we could test all the jump instructions in only 3 lines of code.

[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file Proj1_bubblesort.s.

THE TOP PHOTO IS FOR LOOKING AT THE DMemWr PATTERN ONLY ON A LARGER SCALE, PLEASE READ BELOW FOR CONTEXT.



While it was kind of hard to show this in an easy to understand way while describing it in terms of wave forms, you can see that every time the DMemWr is enabled, we give 2 memory addresses, each 4 apart(**Middle photo**). This is done during the Swap part of the code, this indicates that a swap of two numbers was needed and we wrote the swap to memory, the new address. Then, as we get further and further down the line of swaps, the need to swap happens less and less(**Bottom photo shows the dmem Write Enable being toggled until it stops near the end of the program**) we then compounded this data and confirmed it to being working correctly as it also had been given a pass by running the test simulation tool we had been given for this lab.



If you look at this photo things become more clear. You can see that when a swap is detemrined, the ALU_Zero, Branch, and MemWrite all happen in the same sequences. Branch, Zero, Write. This is then compounded by the fact the number 100 in this case is always on the right, and a new lower than 100 number is on the left, the reason is because it is swapping, then comparing the higher number to the next number, and if needed swapping again.

In this last photo, where the lines are youre able to confirm their location in the array being written to as well, and you will notice, it writes to N then N+4, both times, this is because of them being word aligned, and each being written to every 32 bits in their memory location in the array.

[Part 4] Report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematic from part 1. What components would you focus on to improve the frequency?
-Max: 24.74mhz Clk Constraint: 20.00ns Slack: -20.42ns

- To improve the Freq We would probably have to make adjustments to the ALU, as currently the ALU houses a ton of signals and logic things going on, how to improve it, currently I am unsure, however, I see the ALU as being the major thing that every single instruction has to go through, this could maybe be changing from a ripple carry adder to a look ahead, or maybe finding a faster way to shift bits inside the ALU. The next option would be to update the memory access, but given how that was a given to us file, I wouldn't know where to start on updating it, so I didn't choose that. **After looking at the Synth report, it does look like DMEM took the longest** to go through as well.