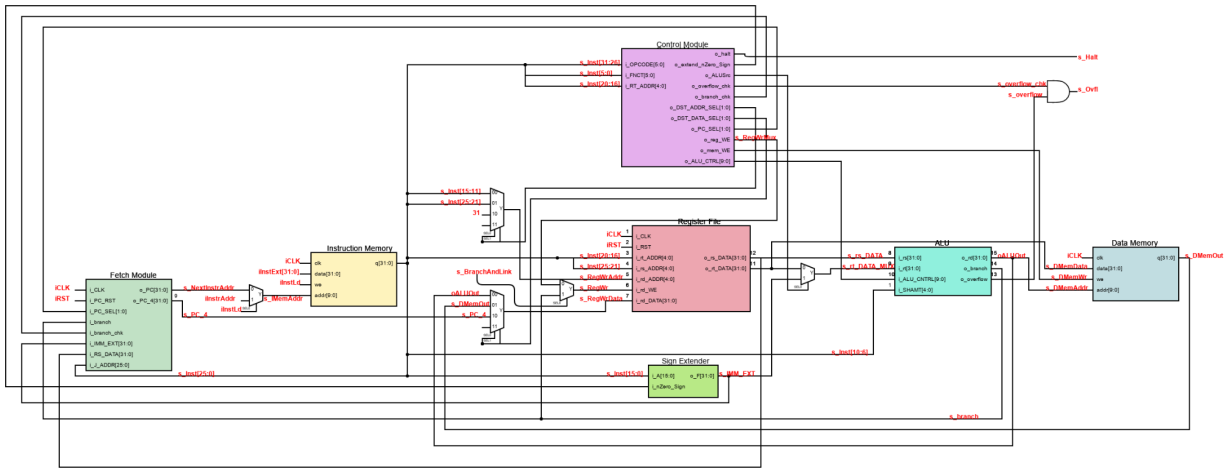
CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: Jake Hafele, Thomas Gaul

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

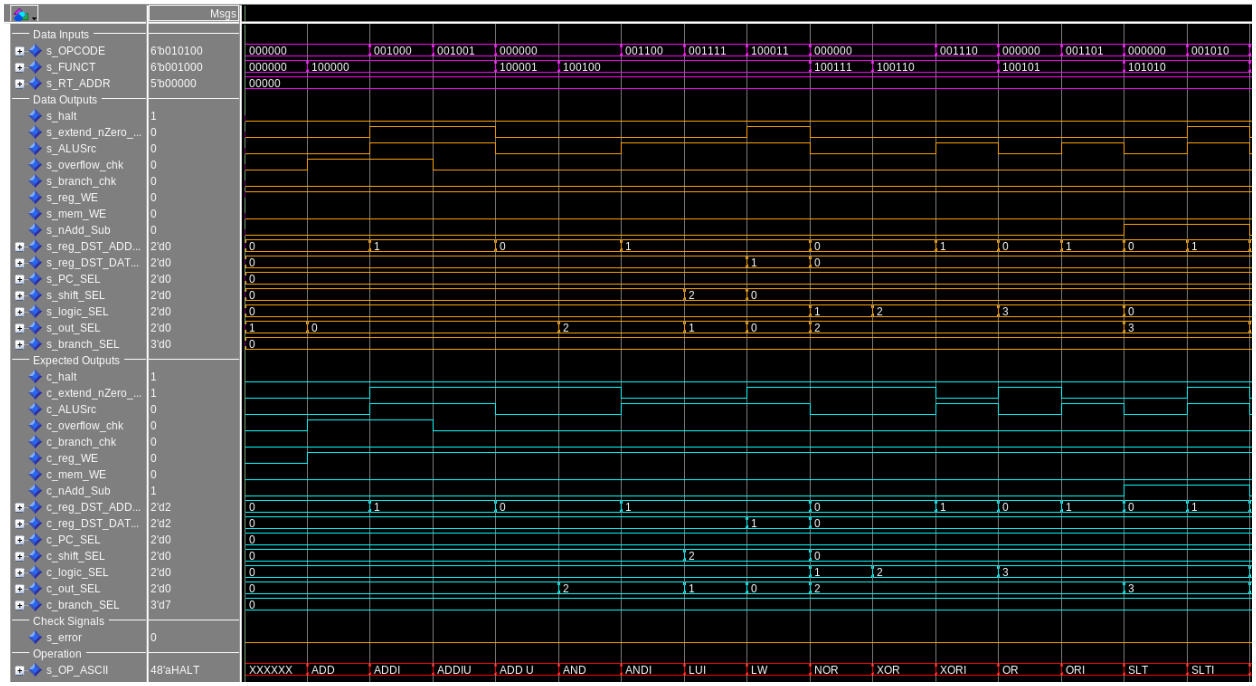


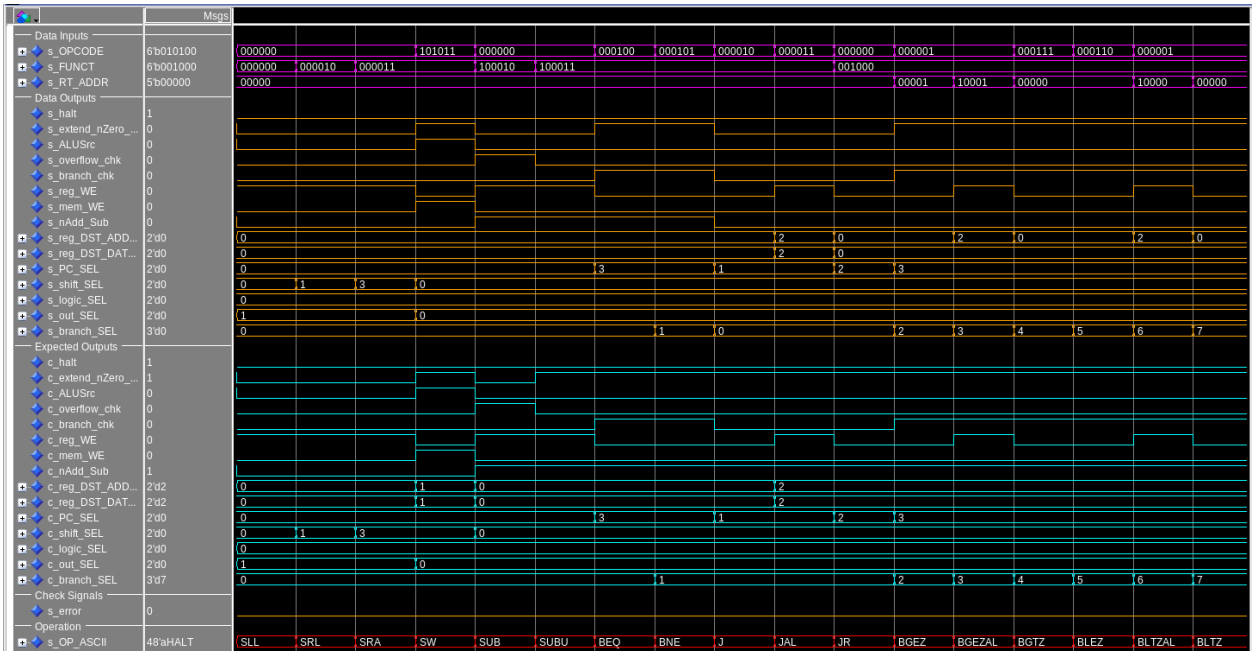
Top Level Schematic

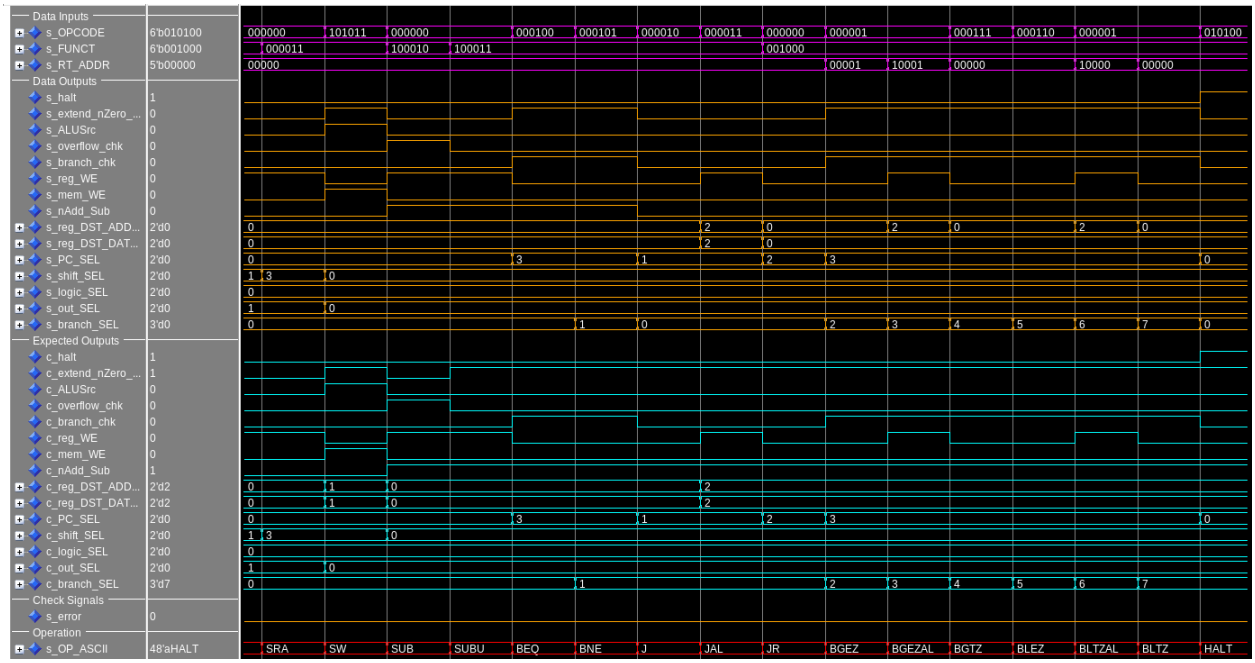
[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your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

We have attached the spreadsheet as a separate document in our zip submission for better readability.

[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).







To test the control module I ran every single possible instruction with the corresponding opcode function and rt fields. I then compared the signals with the control spreadsheet we made and checked to make sure those made sense. I found issues with a handful of bits on the control spreadsheet that I thought were mislabeled. I then made the expected waveforms as an output and compared them by considering if they were dont cares and output if they had an error or not on the error line.

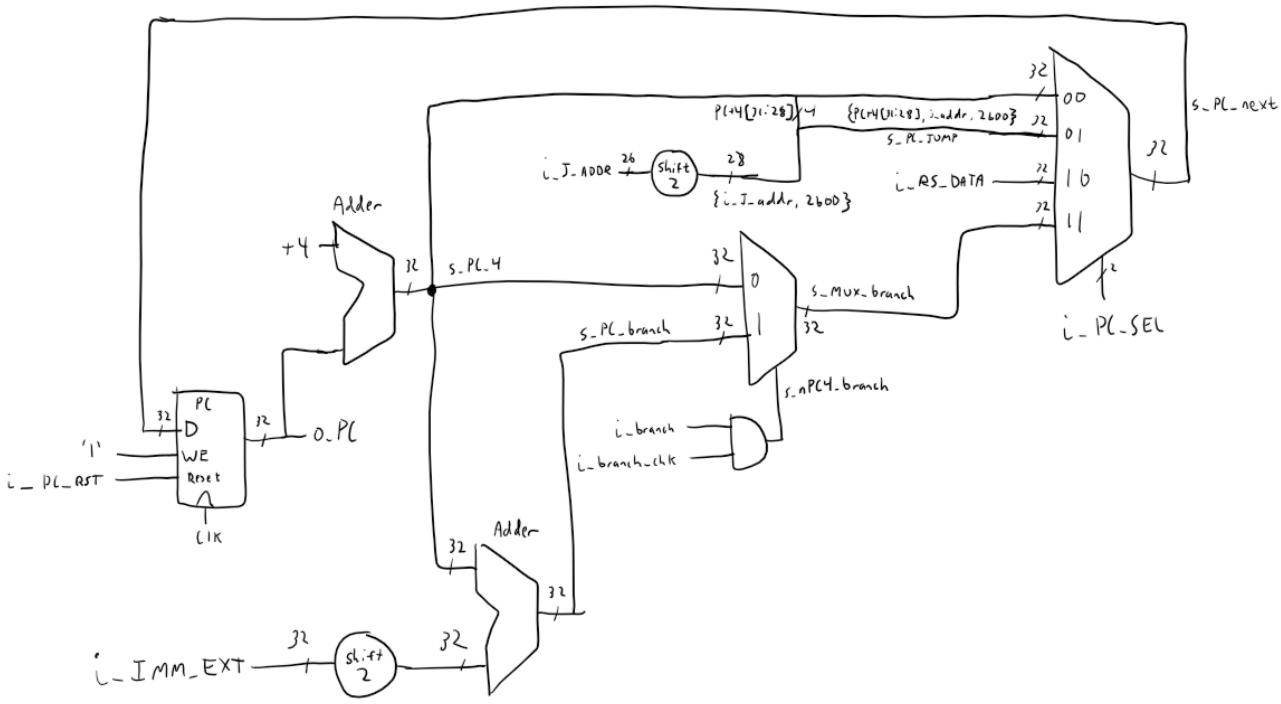
[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

The program counter can be updated as follows from the given instruction set:

- **PC + 4:**
  - The “common case” update that will increment PC by 4 to read the following instruction from the instruction memory
- **Branch Instructions, PC = Branch Address:**
  - Branch address = Sign extended immediate, shifted left 2 bits
  - Impacted by beq, bne, bgez, bgezal, bgtz, blez, bltzal, bltz
- **j instruction, PC = Jump Address:**
  - Jump address = {PC+4[31:28], address, 2’b00}
- **jr instruction, PC = R[Rs]:**
  - PC set to contents of register at RS address
  - Used to return from procedures with jr $ra, since jal updates $ra to PC + 4

To choose between these options, we will use a 4-1 MUX with 2 control bits for a select line, based on the opcode of the 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?

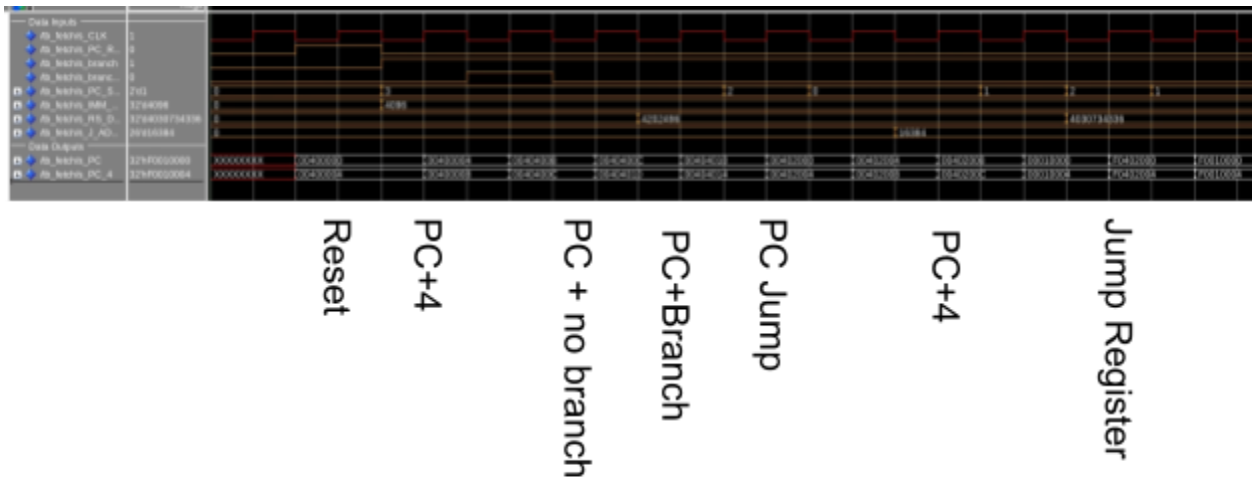


For control signals, we included:

- **i_PC_RST**: A reset signal for the PC D Flip Flop, which reset to 0x0040 0000
- **l_branch_chk**: output from control module dictating whether a branch instruction is being called. Does NOT guarantee that a branch SHOULD occur, that comes from the ALU!
- **l_branch**: output from ALU determining whether a branch should occur or not, depending on operands and the branch_SEL control bits
- **i_PC_SEL**: 2 bit select line to MUX between various PC addresses, which would be the next PC address

[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.



To test the fetch module I tested each possible use case that the fetch would have to be used for.

[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 srl and sra is srl shifts 0's to replace the shifted out binary digits whereas sra shifts in whatever the most significant bit is maintaining its sign. MIPS does not have an sla instruction because there is no clear use case for it. Sra can be used to shift while maintaining sign whereas sla has not obvious use.

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

My barrel shifter block is made up of 3 barrel shifters, one for sll, one for shifting right with 0's and one for shifting right with 1's. They the sra control line is anded with the most significant bit of the rt so if a sra instruction is selected is shifts in whatever the most significant bit is. In all cases with the shifting it is made up of a multiplexer for each bit of the shamt and either maintains the previous value or shifts the number that amount. These are all cascaded together to make the shift block.

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

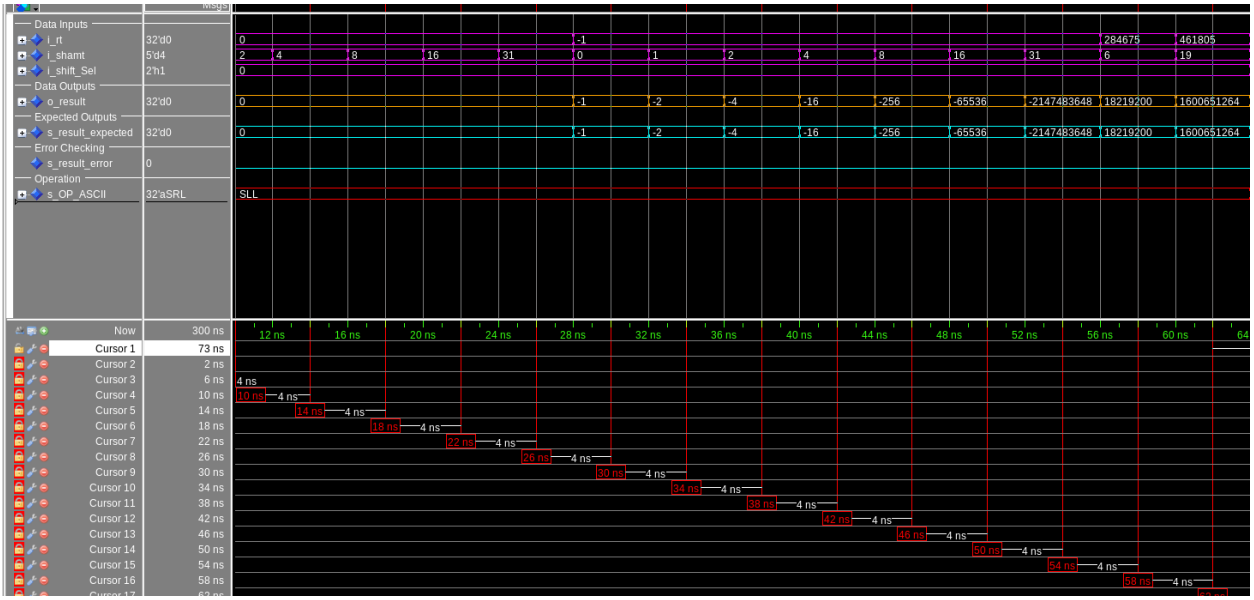
To enhance the right barrel shifter we added another set of multiplexers for left shifting I then had a multiplexer between shamt and “10000” and we then with that we can select between shifting and a lui.

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

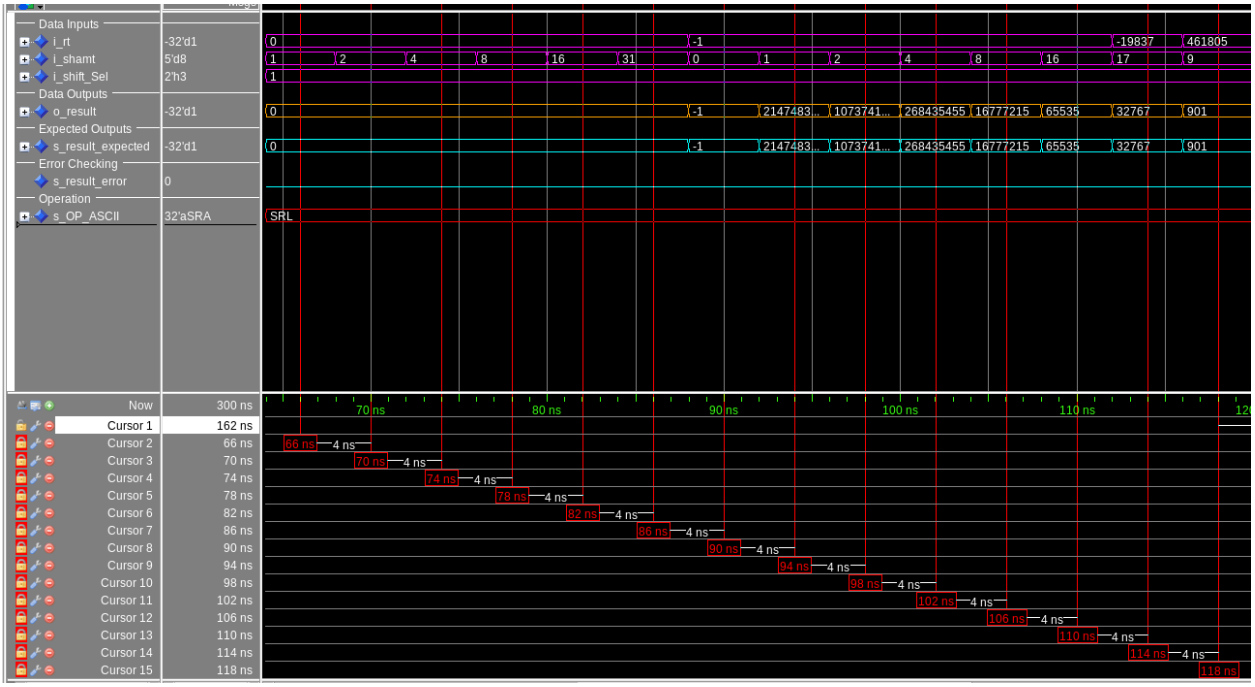
Each shifting operation corresponds to a different operation code and ASCII vector, denoted by s_OP_ASCII. Each of the following four waveforms for the shift module represent a different ALU function, supporting the sll, srl, sra, and lui instructions. For each operation, the shift module takes in a 32 bit wide input i_rt, which would act as the input operand in the ALU normally to shift. The amount of bits shifted is dictated by the input signal i_shamt, which can range from 0 to 31 bits, since i_shamt is 5 bits wide. The input i_shift_Sel denotes the control signal that is normally routed through the ALU to the shift module, to output the correct corresponding shift operation. This table is shown in the second page of our Control spreadsheet.

For each shift operation, we decided to shift the inputs for i_rt 0 and -1 by each individual bit of i_shamt first, leading to a shift of 0, 1, 2, 4, 8, and 16. The idea behind this was to catch a case of if one specific bit of i_shamt was not wired correctly through the multiplexers in the shift module. We also included a shift for both a 0 and -1 input of 31, combining each of the bit shifts together for an edge case. The expected output for each shifted 0 value was always 0, since sll and srl always shifted in a 0, and sra shifted in the MSB of the value, which in this case was 0. The shifts with -1 were beneficial since it helped us verify 0’s were also shifted in for another input and it was easier to identify how many were shifted in. We also included a common case with a few random values and shamt amounts to test that different combinations of each shift operation would output the expected value.

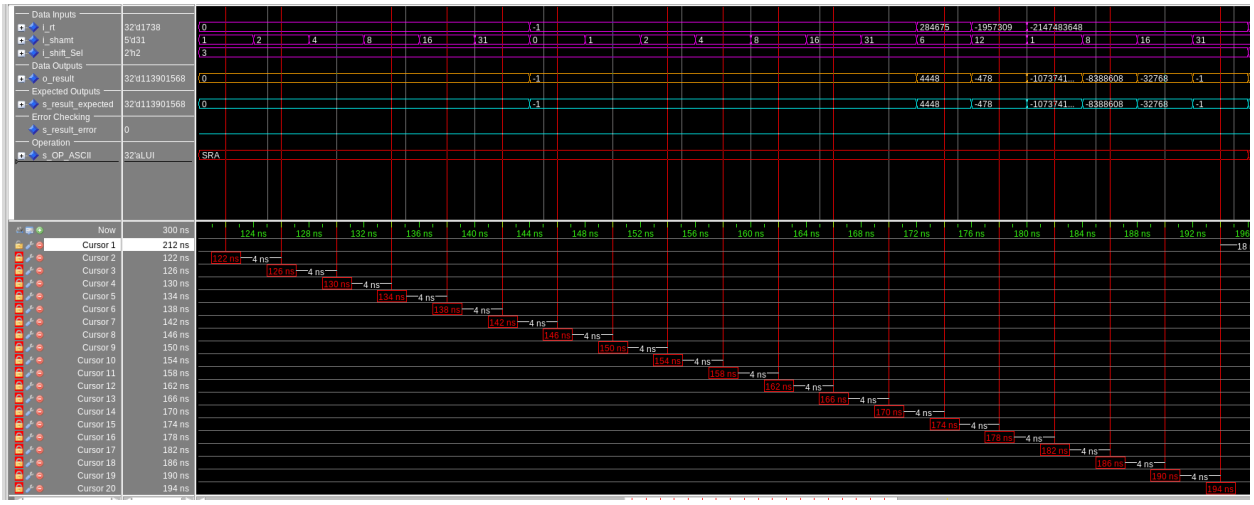
For error checking, we created an expected internal test signal, s_result_expected, that we would enter for each test case manually so that it could be compared to the shift module's final output, o_result. We also created an error flag that would be raised if the expected and actual output for s_result_expected and o_result differed. This made it much easier to sweep across the test cases and verify that s_result_error was never raised as 1 (an error).



shift Submodule SLL Operation

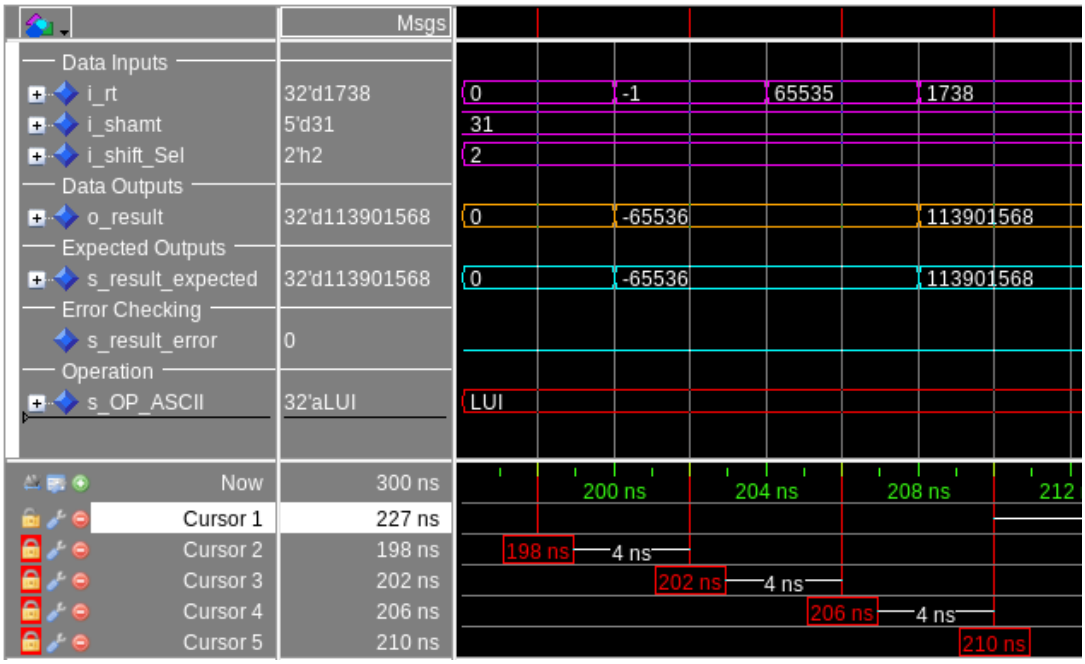


shift Submodule SRL Operation



shift Submodule SRA Operation

Since the LUI instruction always shifted 16 bits to the left, while shifting in 0's, we did not need as many test cases for it. For this, we decided to use an input for i_rt of 0 and -1 again, since it included all 0's and all 1's, making it easy to identify common errors in the shift. We also included a few common random numbers to verify bits were properly shifted left 16.



shift Submodule LUI Operation

[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 following additional modules were implemented in our ALU:

1. Adder/Subtractor
2. Branch
3. Logic
4. Shifter
5. Set Less Than

To implement the adder subtractor we had a full adder with subtractor as done in earlier labs which is used for some later instructions. I decided to calculate overflow in the adder subtractor along with a zero flag as an output of the Adder Subtractor sub-block.

To implement the branch we took the zero flag from the adder for BEQ and BNE after the adder subtracts them. The other flags are done by generating a zero flag and negative and a positive flag and doing basic logic between them to determine the branch. We had each one separate even though some share the same functionality to make it more straightforward dividing it.

Logic is done with just the bitwise operators and multiplexers for select; we didn't do it structurally to make it simpler to look at.

The shifter was included in the ALU but is described above.

We implemented Set less by adding another full adder outside of the adder subtractor and took the most significant bit after the subtraction. It is explained later on.

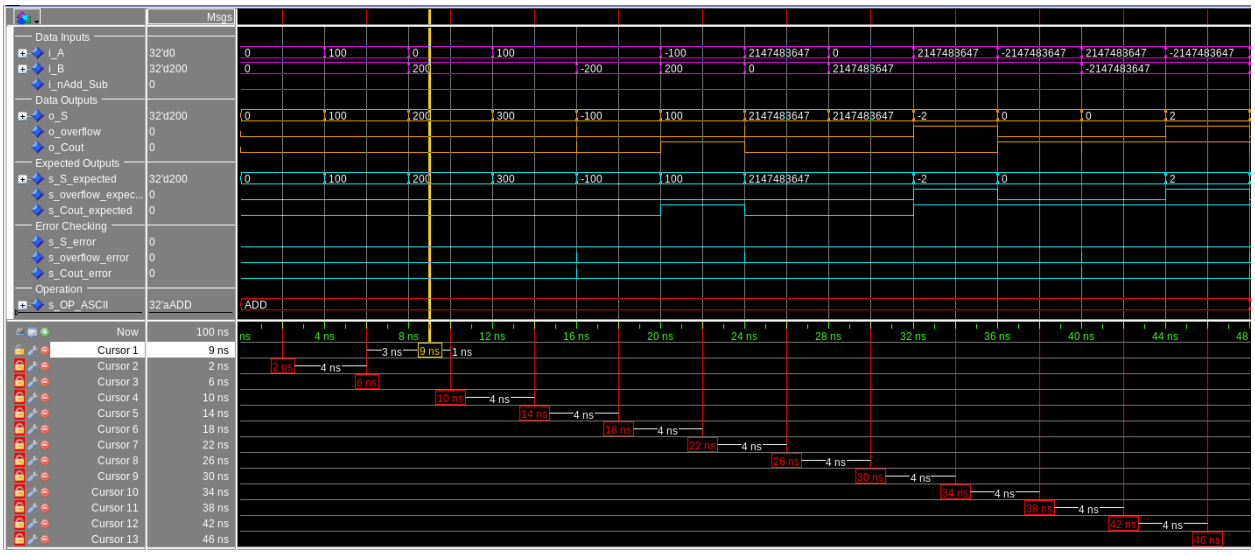
[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

Each of the following units has a similar testbench setup to the above shifter module. The expected outputs are either set manually or with a process statement based on the updated inputs, and an error flag is raised if any of the outputs disagree with the expected values. The intent of this testing was to verify both common and edge cases worked for varying inputs that would act as operands for different instructions in the ALU.

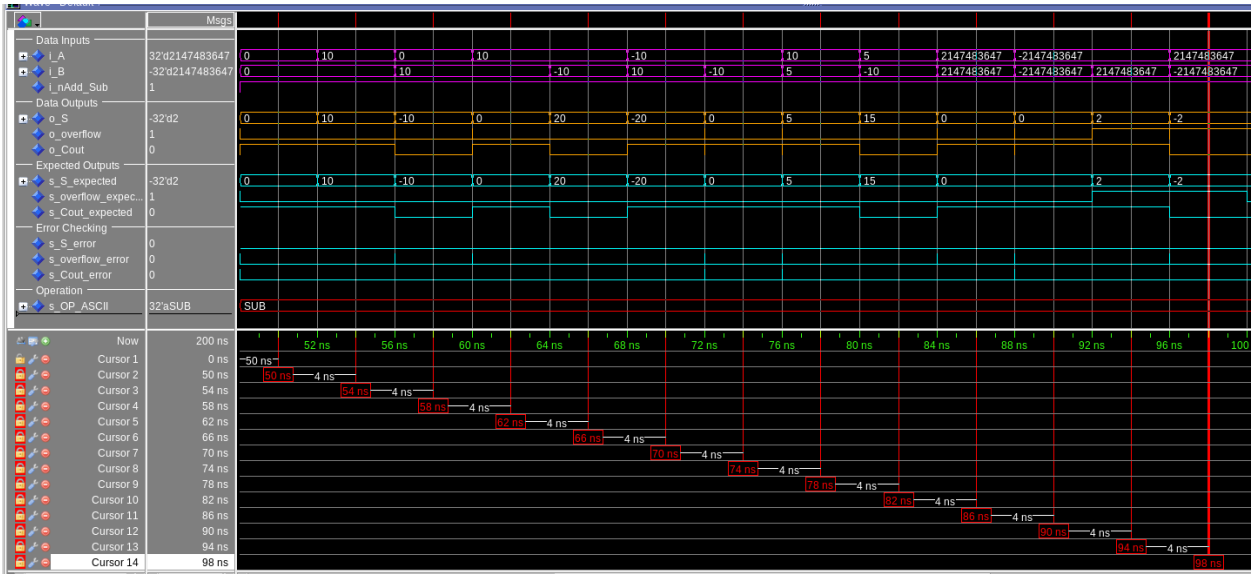
Adder/Subtractor ALU Module

For our adder/subtractor unit, nothing had changed compared to our implementation in Lab 2. The inputs i_A and i_B will act as the operand inputs for both add and subtract operations. Like before, the add and subtract operation will be dictated by a 1 bit wide control signal, i_nAdd_Sub, where we will add if 0 and subtract if 1. There are three outputs, including the output sum o_S, the overflow indicator o_overflow, and the carry out o_Cout. If the overflow indicator is 1, then overflow has occurred in the operation. But, this does not mean it should always be flagged, since outside the ALU in our design, we have a control signal to dictate if overflow should be checked, due to the difference in the add and addu instructions. But, for the case of this submodule in the ALU, we do not need to worry about this control signal since it is out of scope. The same standard error checking signals are also applied.

For the common cases, we included some add and subtract operations using numbers such as 0, 100, 200, -100, and -200 to verify 2's complement values were added and subtracted correctly. We also included some edge cases with the largest and smallest two's complement values for a 32 bit wide logic vector, to verify if overflow would occur when adding together to large values of the same sign or subtracting two large values of opposite signs. This was important to test since overflow needed to be indicated for the add and sub instructions.



Adder_Subtractor Submodule ADD Operation

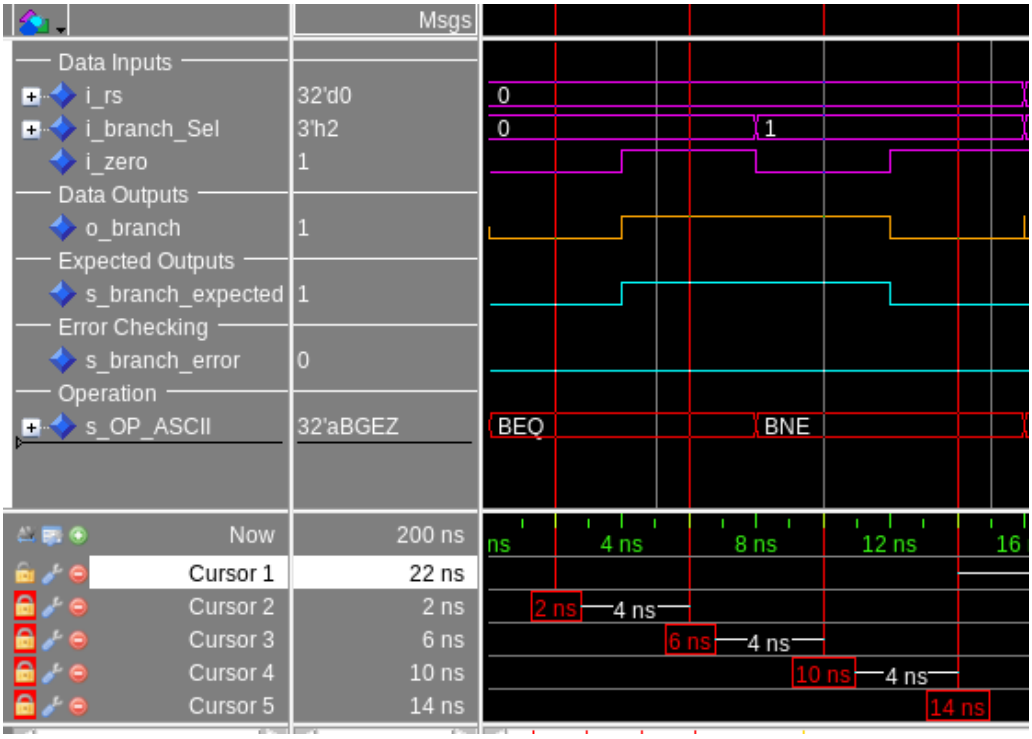


Adder_Subtractor Submodule SUB Operation

Branch

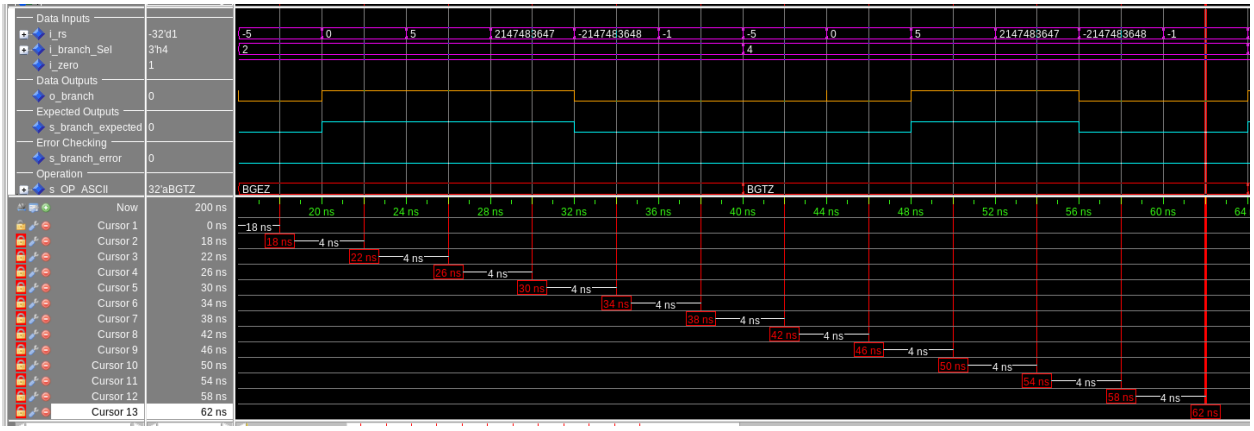
The branch module included three different inputs, including a 32 bit wide input i_rs, which would come from the second operand input of the ALU, in the form of an immediate. The input i_branch_Sel is a 3 bit wide select line that would MUX the correct branch output to o_branch, depending on the current instruction decoded from the control module. This input would be routed through the ALU, into the branch module. The final input was i_zero, which would be 1 if the two ALU operands were equal, and 0 if the two operands were NOT equal. This input signal was used to verify the BEQ and BNE instructions would output the correct branch statements. The output o_branch would output a 1 for each branch conditional operation if the PC should branch for the next instruction. The address for this branch would be summed in the ALU at the same time as this branch output was being found in the ALU, and would be output to the fetch module alongside the output o_branch.

The BEQ and BNE operations each had only two cases tested, since it only depended on the input i_zero, which was one bit wide. Depending on i_zero, the output o_branch would change for the two instructions. The input i_rs did not impact the output of these two instructions.

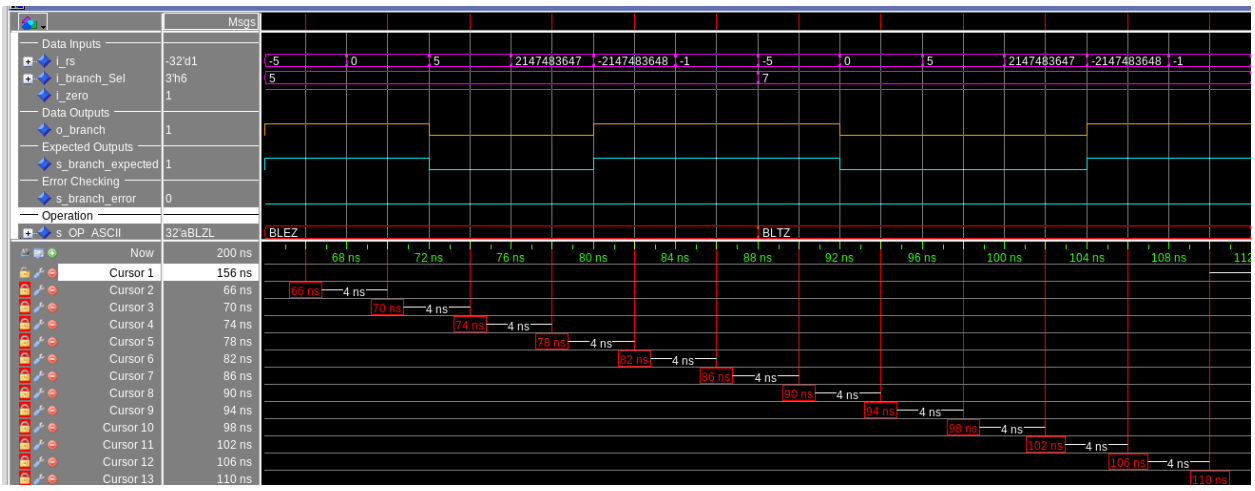


Branch Submodule BEQ/BNE Operations

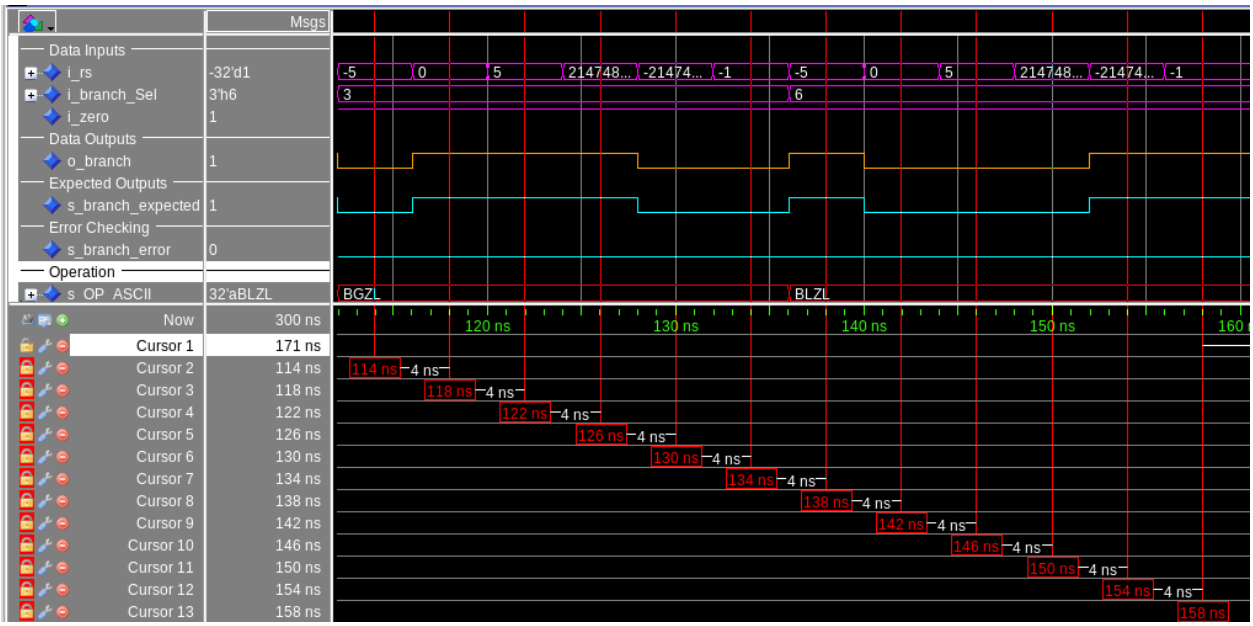
The rest of the conditional instructions done in the branch module, including BGEZ, BGTZ, BLEZ, BLTZ, BGEZAL, and BLTZAL, instead used the input i_rs to determine if a branch should occur. Since each of these conditions compared against being greater than, less than, or equal to 0, only one 32 bit wide operand input was needed, as defined in the instruction. For common cases, we used inputs of -5, 0, and 5 to determine each comparison against 0 asserted o_branch properly. For edge cases, we used the largest positive value and smallest negative value possible with a 32 bit 2's complement value to confirm each output would assert properly. We also included a case to check against -1, since that included all 1's for each 2's complement value, and could lead to some potential hazards.



Branch Submodule BGEZ/BGTZ Operations



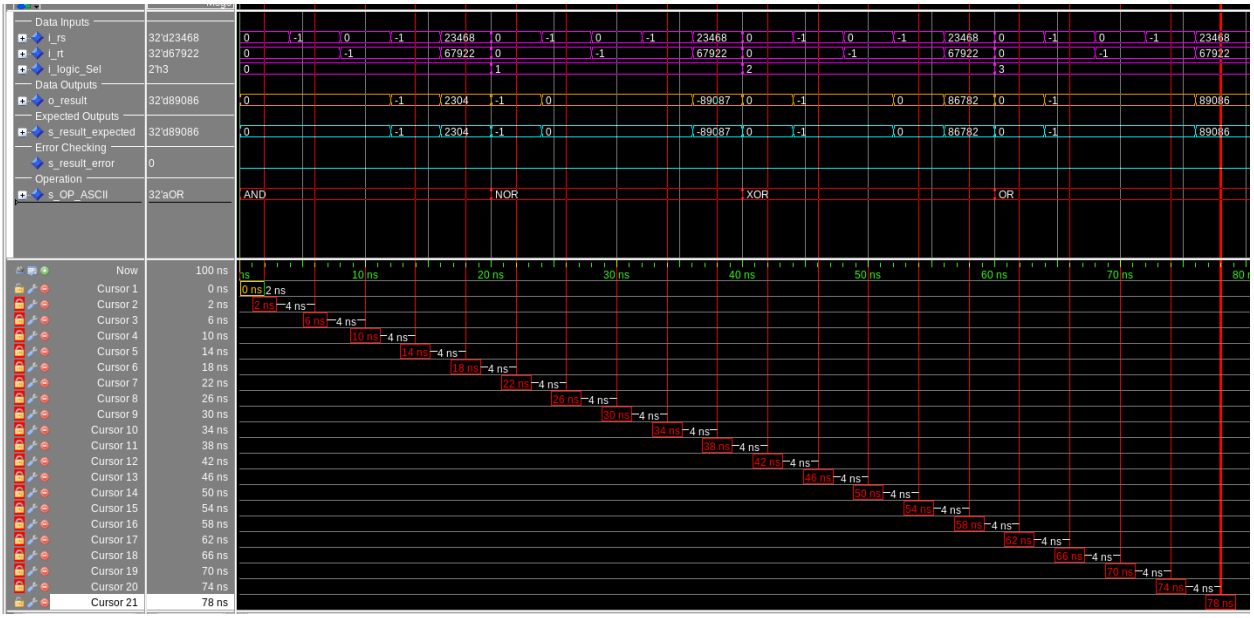
Branch Submodule BLEZ/BLTZ Operations



Branch Submodule BGEZAL/BLTZAL Operations

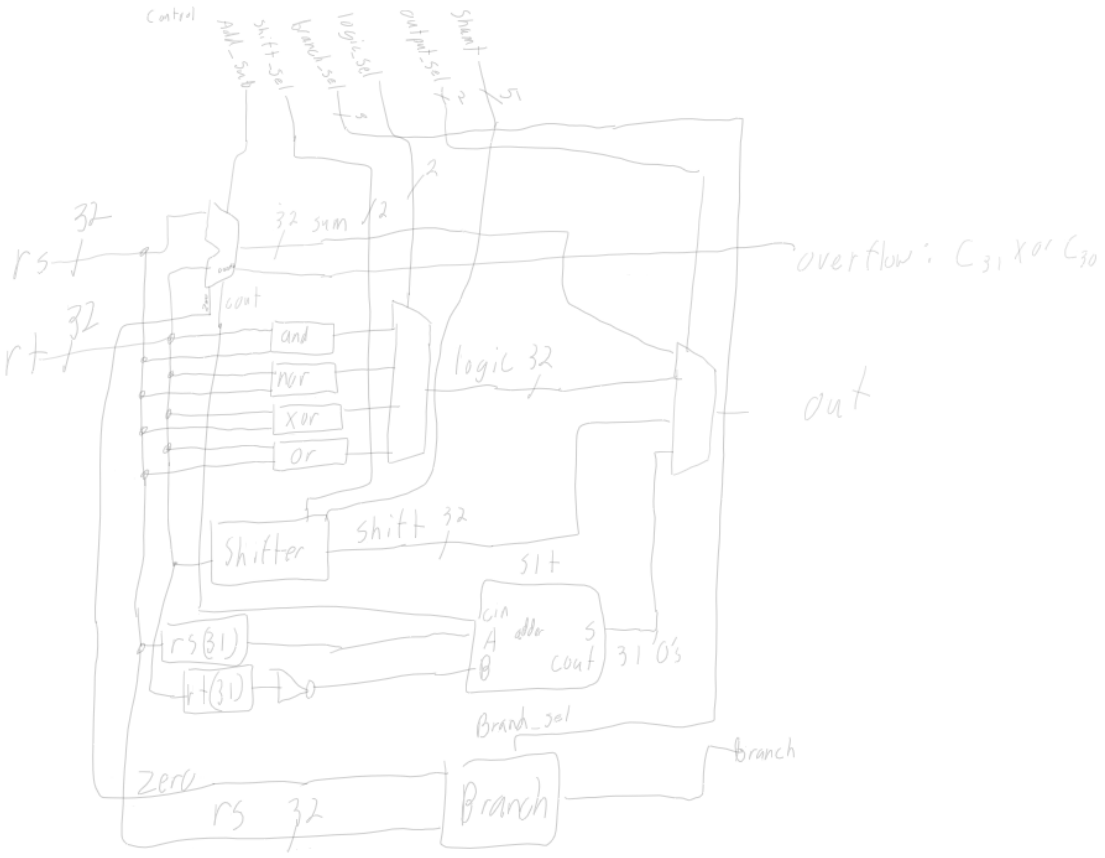
Logic

The final submodule that was included and tested in our ALU was the logic module. Based on our required instructions to implement, we needed to test the bitwise AND, NOR, XOR, and OR operations. Since these were all common bitwise operations, we were able to automate the expected outputs inside of our testbench with a process statement. This made it very easy to test multiple cases, for two operand inputs i_rs and i_rt. To MUX the correct output between each operation, the 2 bit wide i_logic_Sel was used, which would normally come from our Control module and be fed into the ALU. To test each bitwise instruction, we used inputs of 0 and -1 for each possible operand combination to see the effects on each logical operation. The reasoning behind this was because each bit would be covered to be checked for the bitwise operations, and they would all be the same output per bit. We also included another common case with two “random” operand inputs to verify a mixture of 0’s and 1’s would output the correct bitwise logical operation. Similar error flagging was included as above.



Logic Submodule AND/NOR/XOR/OR Operations

[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 is calculated with the last carry bit and the last-1 carry bit xored together

Zero is calculated by oring all the bits of the sum together

Slt is implemented by adding one more digit to the standard adder removing the potential for overflow. To map that properly cout of the main adder has to be cin to the adder, the most significant bit of rs and rt needs to be connected to the two inputs to maintain sign and rt's most significant bit needs to be inverted due to the adder subber

being in subtraction format. The sum bit of this adder can be taken as the least significant bit of the set less than bit with 31 0's concatenated on the left.

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

The different ALU operations in the QuestaSim waveforms can be dictated by the s_OP_ASCII signal, which is a 16 bit standard logic vector with an ASCII radix, describing the ALU operation. The following operations are described with corresponding ASCII values:

1. ADD: add
2. SUB: subtract
3. AND: bitwise AND
4. NOR: bitwise NOR
5. XOR: bitwise XOR
6. OR: bitwise OR
7. SLT: set less than
8. SLL: shift left logical
9. SRL: shift right logical
10. SRA: shift right arithmetic
11. LUI: load upper immediate (shift left 16)
12. BEQ: branch if equal
13. BNE: branch if not equal
14. BGEZ: branch if greater than or equal to zero
15. BGTZ: branch if greater than zero
16. BLEZ: branch if less than or equal to zero
17. BLTZ: branch if less than zero

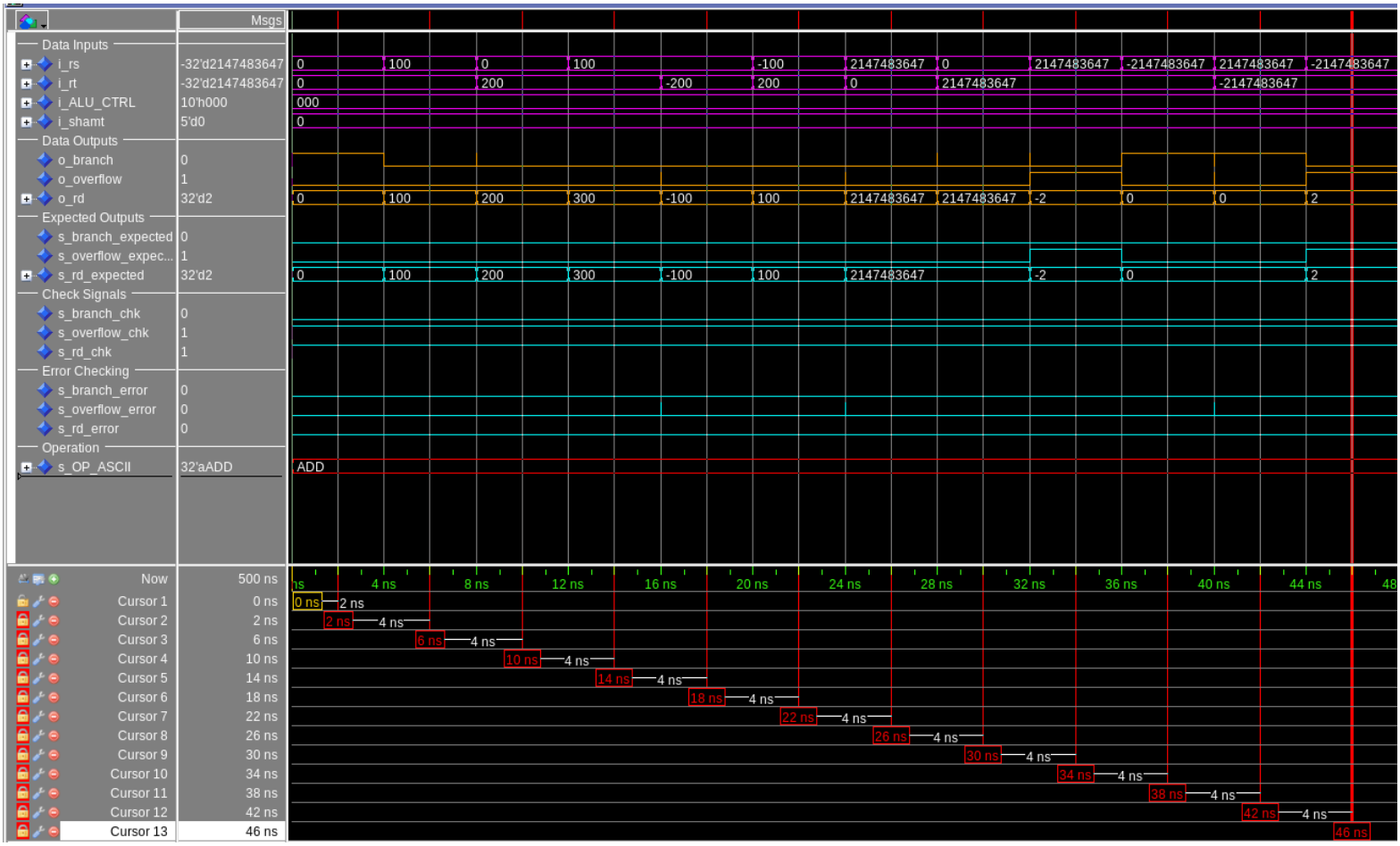
In the testbench, we also included an ALU operation integer, that acts similar to the ALU Control value in the Zybooks examples, with a process statement that dictates the ALU Controls for the ALU Module. This condensed the amount of inputs we needed to update, since the ALU Op integer would handle the controls and ASCII output, allowing us to focus on the error checking and validation of the ALU.

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

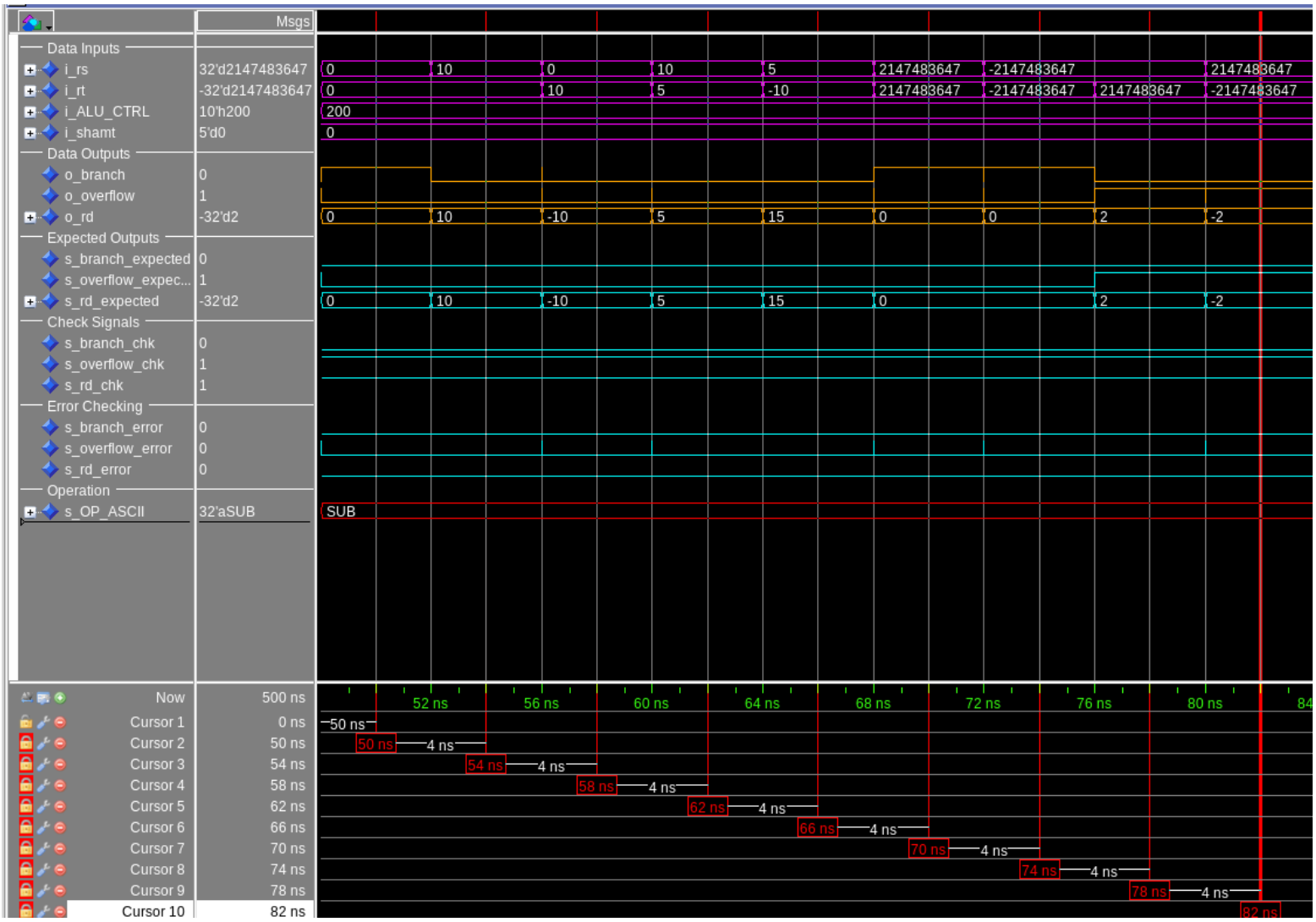
For the total ALU operation, we combined testing for the four different submodules above to drive and test each of those different operations under the top level ALU module. The only instruction that was not included in a submodule that needed to be tested for the first time was the slt instruction. The ALU took in two 32 bit wide operand inputs, where the second input would normally either be the contents of a register or an extended immediate value. These inputs were named i_rs and i_rt. We wrapped the ALU controls into one logic vector for easier control to test, which ran as i_ALU_CTRL, which was decoded to each of the individual ALU control bits listed in our controls sheet. The last input was i_shamt, which was the shift amount given in the shamt field of R-type instructions, intended to be used in the shift module for the sll, srl, and sra instructions.

The outputs of the ALU included the branch indication o_branch, which would be routed from the branch module depending on the 3 bit wide control branch_Sel. The overflow indicator o_overflow would assert if overflow occurred during an add or subtract operation. This only was needed for the add and sub instructions, and could be ignored for the addu and subu instructions. The main output of the ALU is o_rd, which is a 32 bit wide output, which could be written back to an R-type instruction, data memory, or used as an address to update the program counter.

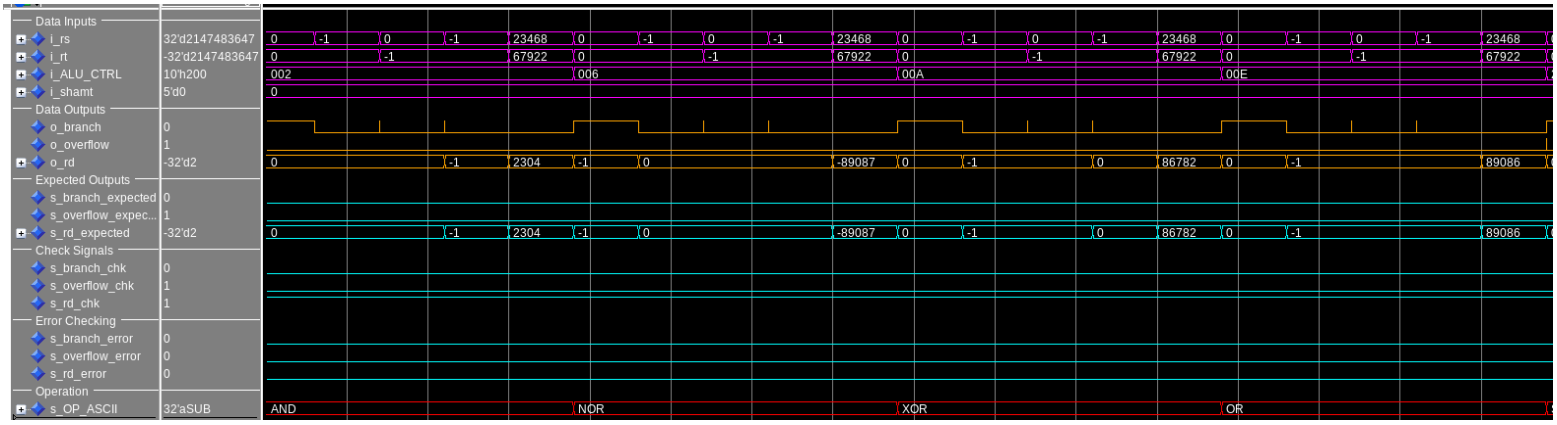
The goal of this top level ALU testbench was to make the testing as seamless and fast as possible. The expected values would either be set with a process statement or manually, depending on the operation type. The chk values determine if an error should be asserted or not if the expected and actual ALU output were not equal. These check signals were included since not every single operation cared about the branch output, since it had a separate control signal to determine if the branch should be checked or not based on the decoded instruction. While many of the waveforms from the submodule tests were included, some of the lengthier ones with many test cases were condensed. The main focus was to test that the final output was MUX'ed correctly between each submodule, and that each of the internal signals were wired correctly together. We also had to test the slt instruction for the first time.



ALU Add Operation Waveform

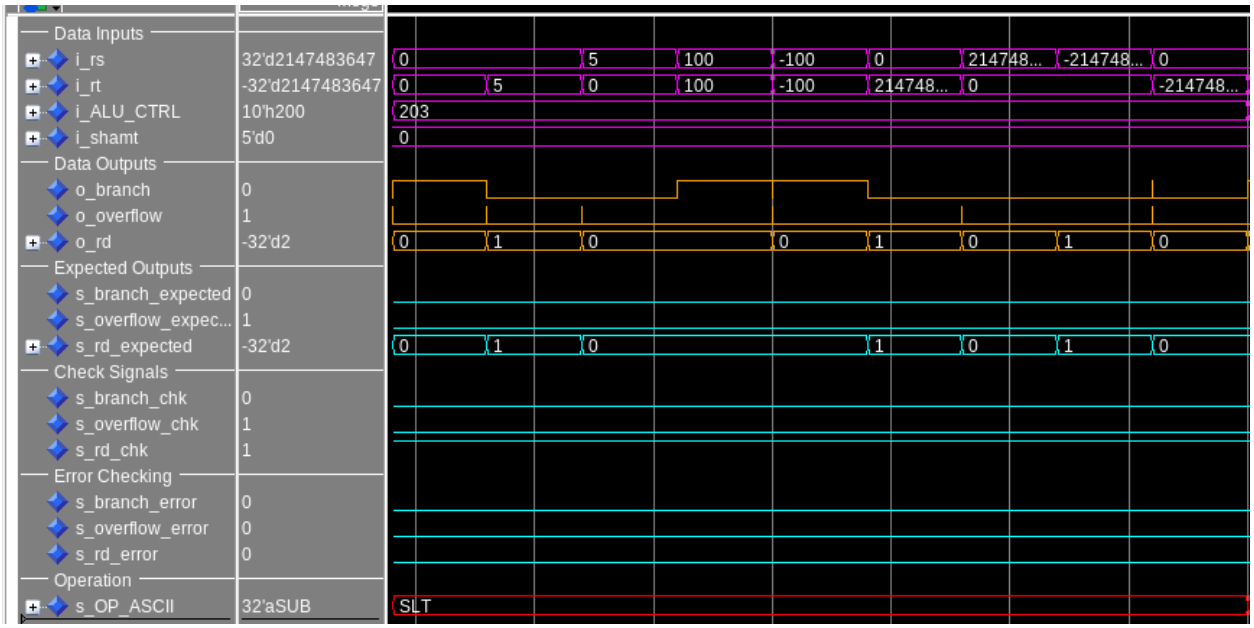


ALU Sub Operation Waveforms

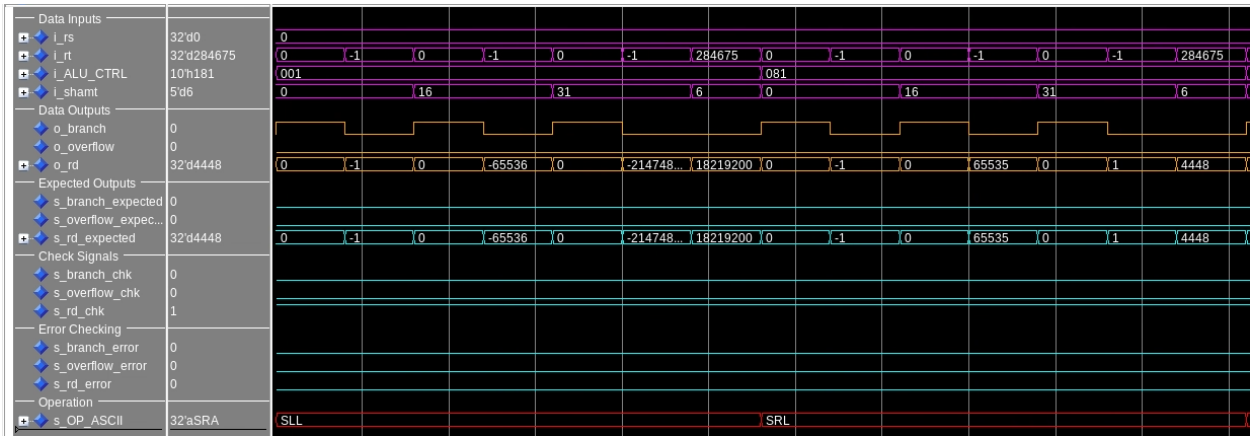


ALU Logic Operation Waveforms

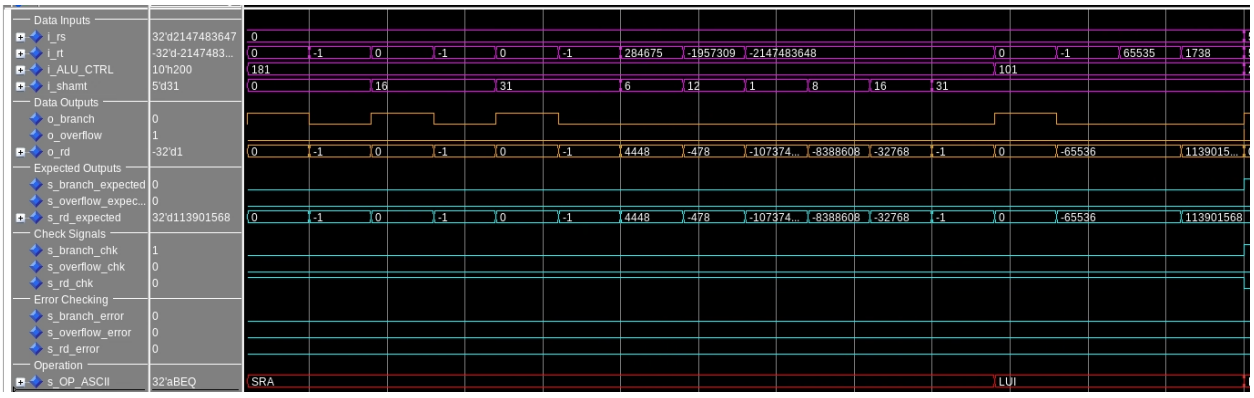
As mentioned before, the slt operation was implemented in the top level ALU module, so it could not be tested before. The SLT instruction would set the output o_RD to one if i_rs was strictly less than i_rt, not including equal. We implemented three common cases with small values, comparing values between 0 and 5 in each order, of equal, less than, and greater than. We also included some more common cases comparing negative values. Finally, we compared the largest and smallest possible 2's complement values to verify slt was asserted correctly.



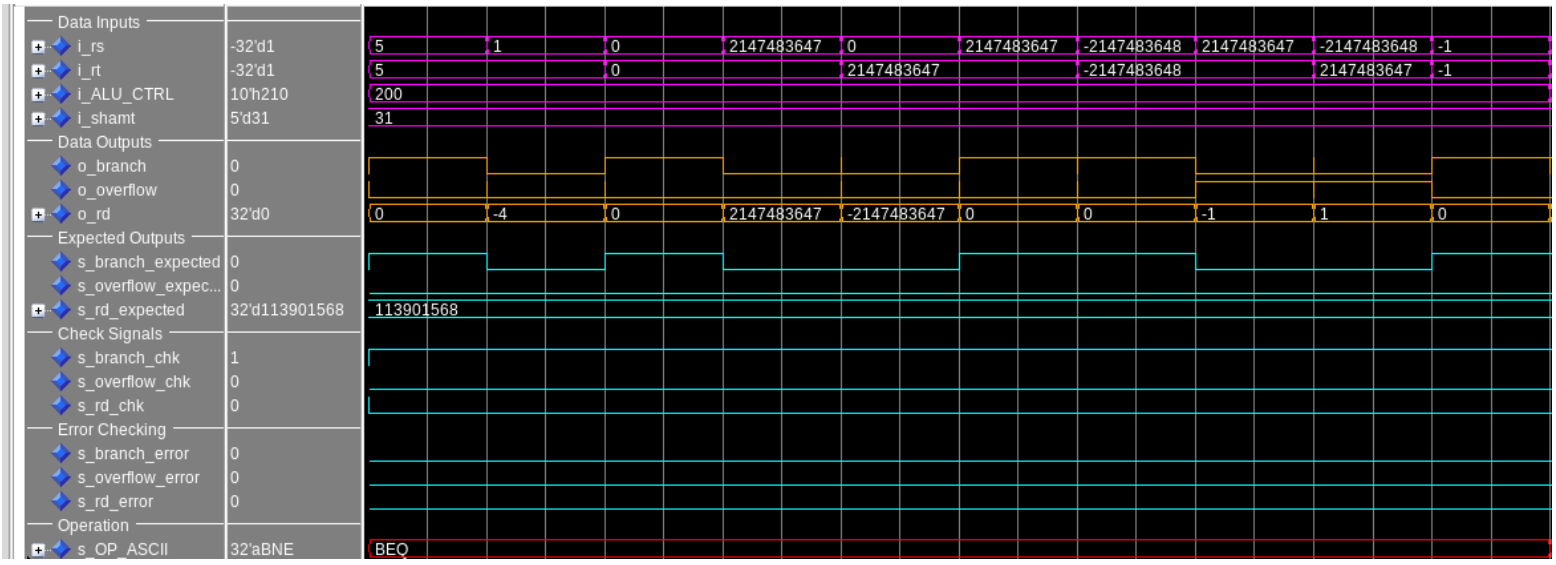
ALU slt Operation Waveforms



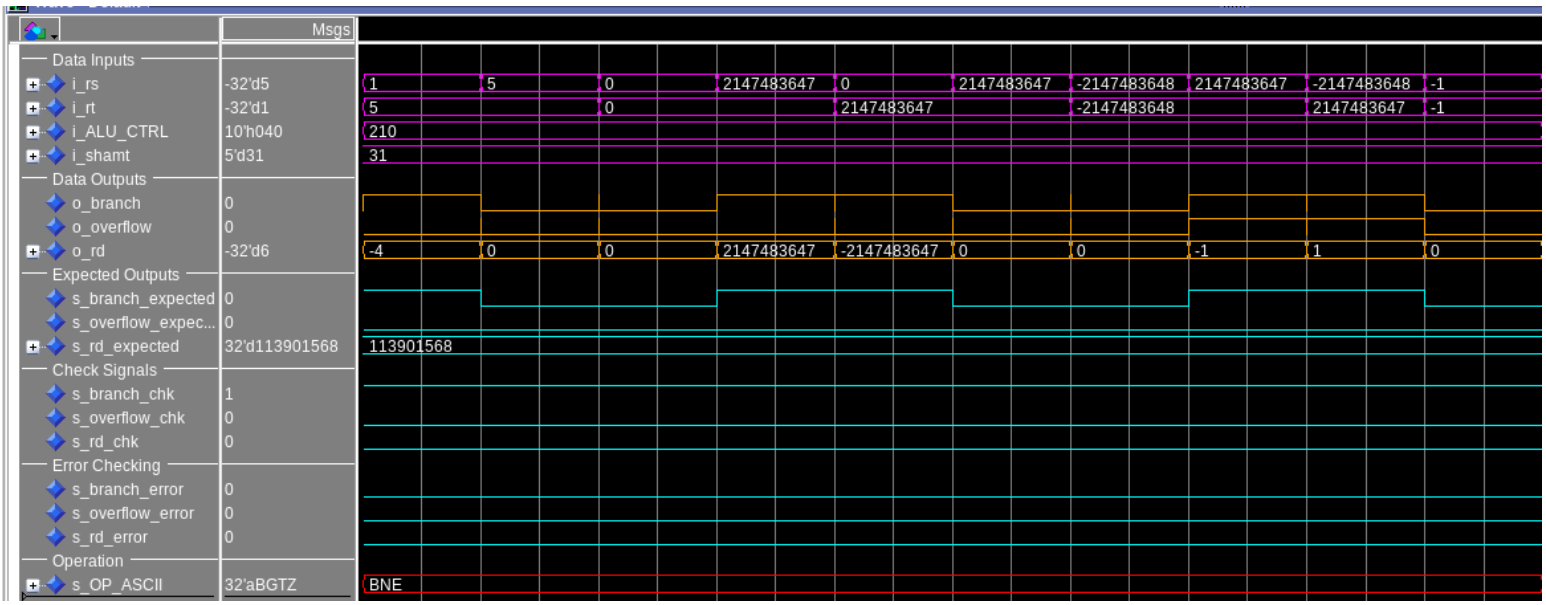
ALU sll/srl Operation Waveforms



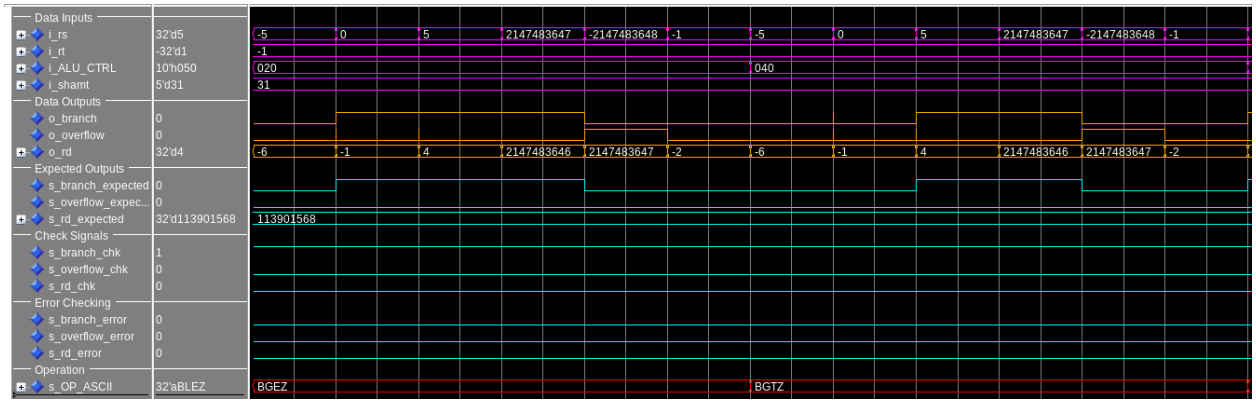
ALU sra/lui Operation Waveforms



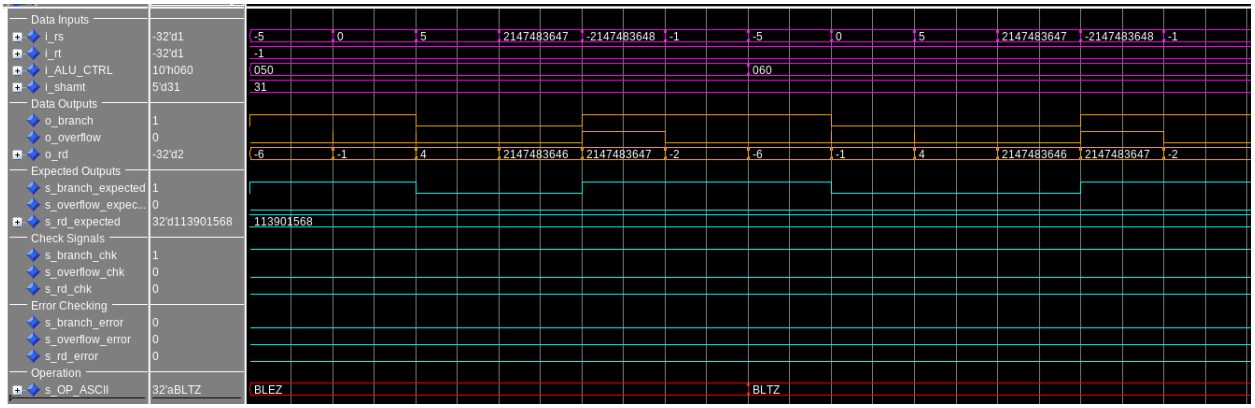
ALU beq Operation Waveforms



ALU bne Operation Waveforms

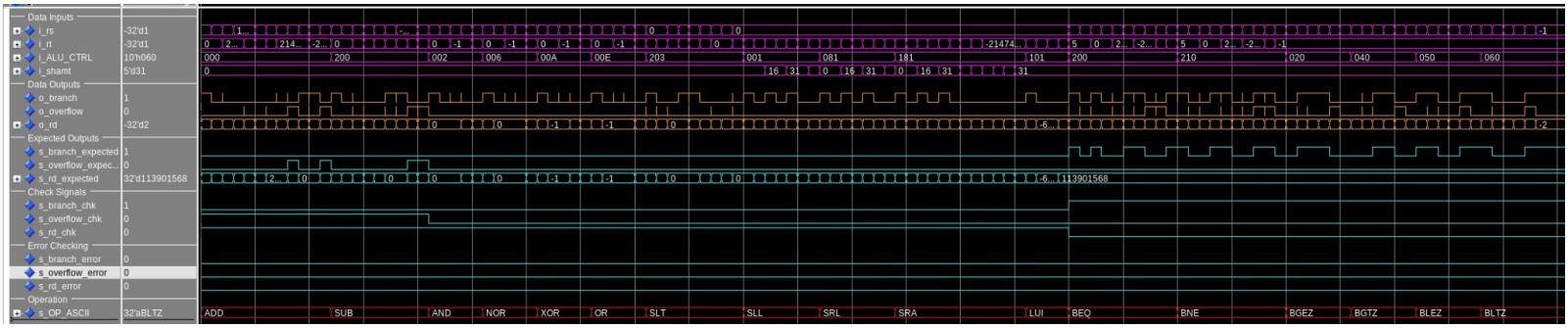


ALU bgez and bgtz Operation Waveforms



ALU blez and bltz Operation Waveforms

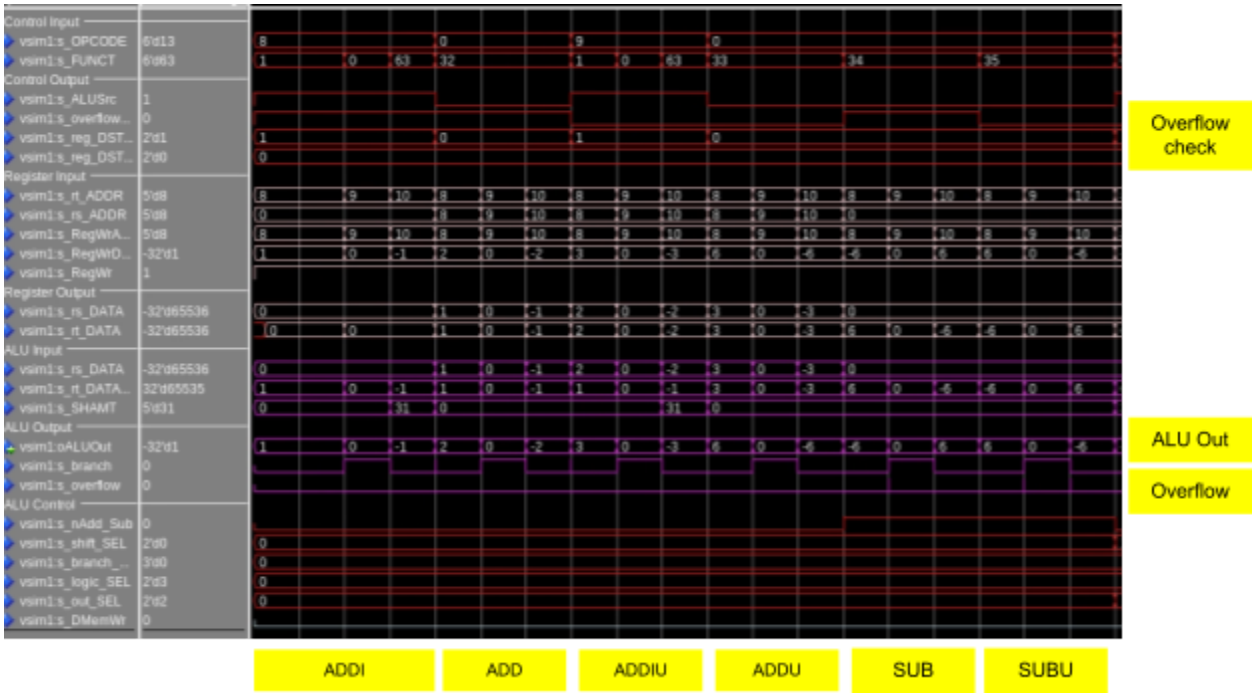
Below is the final ALU waveform output, which is not easy to read at all. But, it does have flat lines for all the error flags, meaning we are ready to implement our top level design!



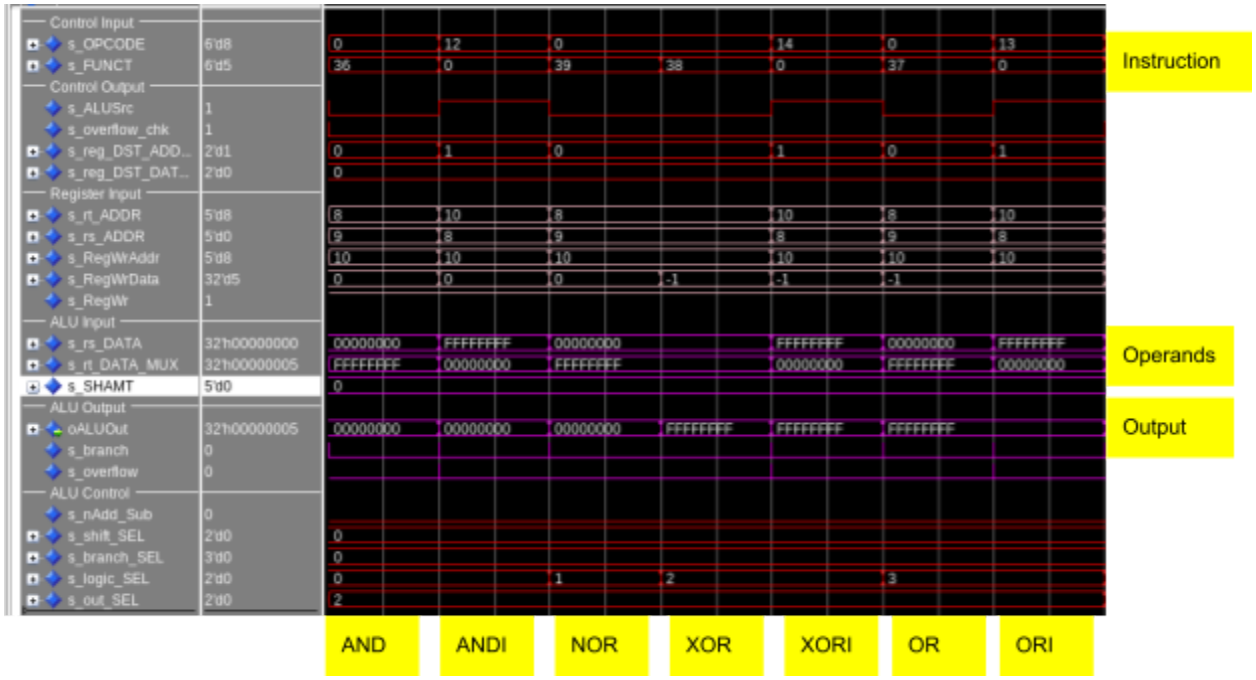
Final ALU waveform

[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.

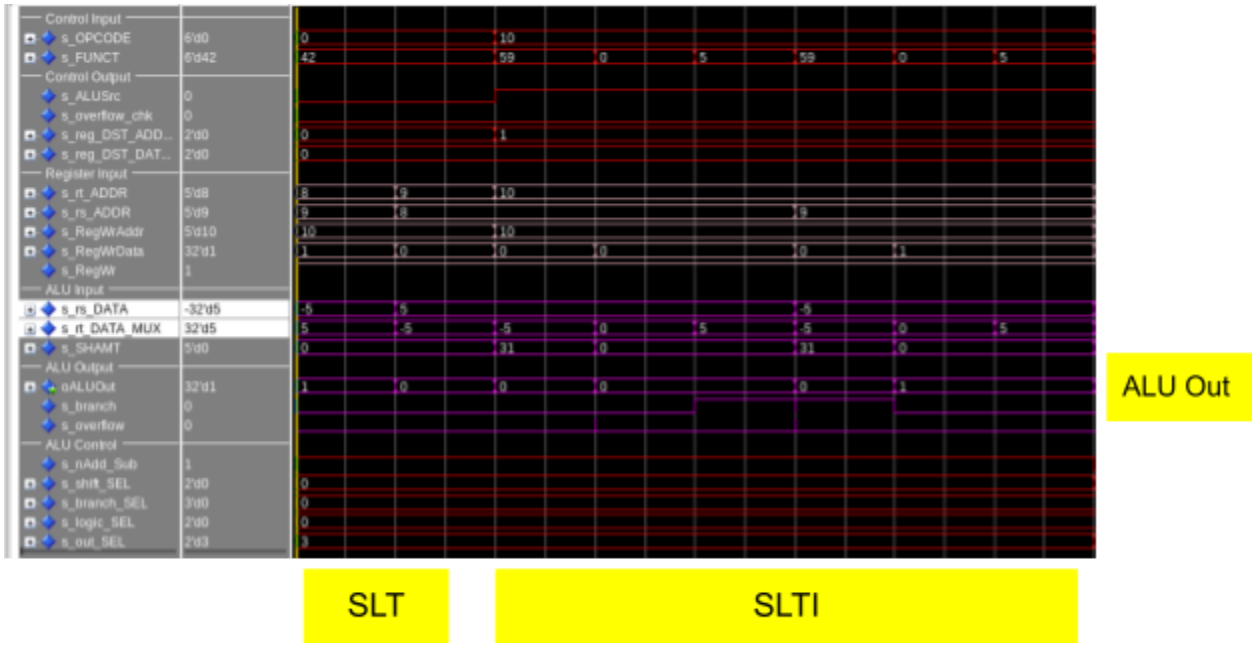
In the first waveform below, we tested in order the addi, add, addiu, addu, sub, and subu instructions. The instructions can be verified with the s_OPCODE and s_FUNCT inputs. Each instruction is called three times, with an input involving 1, 0, or -1 for the immediate inputs. For the R-type instructions, the same register is used as both operand inputs and is written back to the same register. To see the outputs for each instruction, we monitored the oALUOut signal.



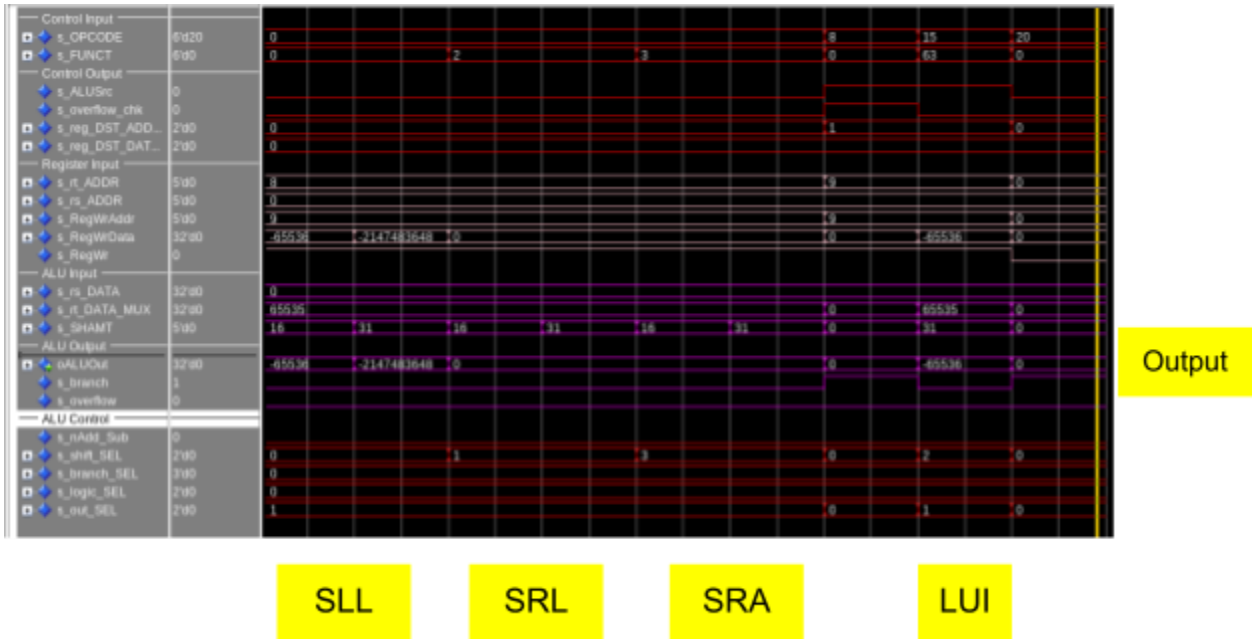
Base Test Addition and subtraction instructions



Base Test Logic Instructions



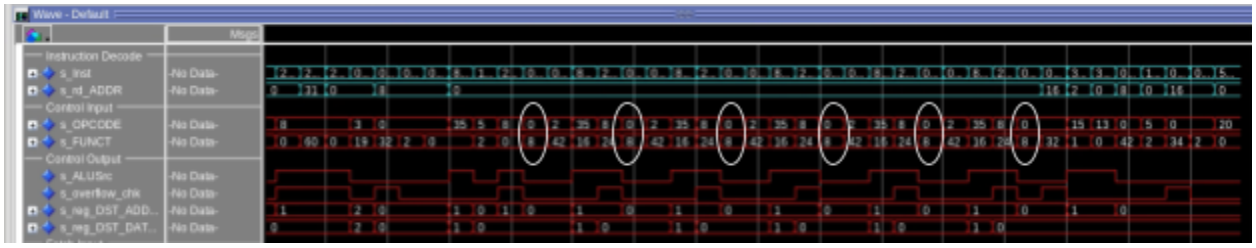
Base Test Set Less Than Instructions



Base Test Shift Instructions

[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.

To test this portion of our processor out we wrote the binary search algorithm in MIPS with a test array of 64 integers with the key being the first one making it call binary seach log_2_(64)= 6 times. It then saved the location in the array in this case 0 to the $s0. Below the circled areas are the times it returns from a function with jr which has a opcode of 0 and function of 8. This is after the 6th call finds the number and they cascade returning the value

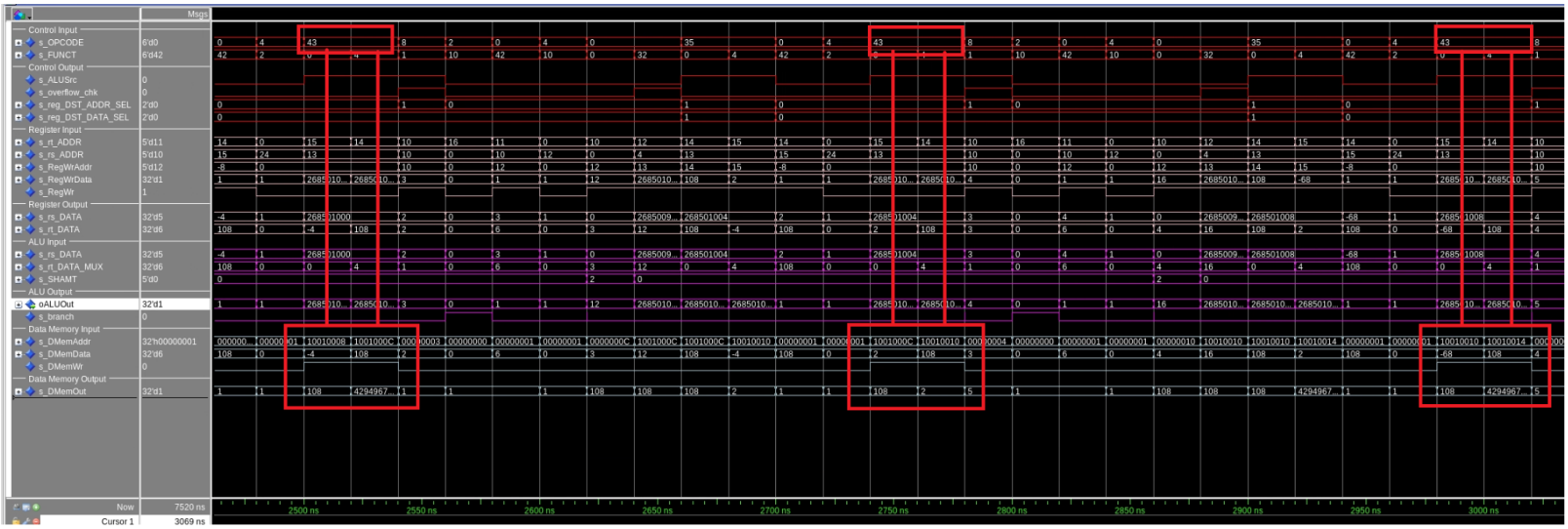


[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file Proj1_bubblesort.s.

We tested our bubble sort assembly file with N=8 elements in an integer array titled array. The function is implemented with two for loops, and tests our branch conditions and arithmetic through these. This test also covers the load word and store word instructions, since for every loop we needed to read and compare two array values, and load them back into memory swapped if the lower address array was larger. In the table below, the initial values are given in the null step. Each step shows the output after one full iteration of the outer loop ran, leading to a sweep of compares through the array. This sweep would occur N times in the same fashion.

	0x0000	0x0004	0x0008	0x000c	0x000f	0x0014	0x0018	0x001c
step	array[0]	array[1]	array[2]	array[3]	array[4]	array[5]	array[6]	array[7]
null	1	5	108	3105	-4	2	-68	19
0	1	5	108	-4	2	-68	19	3105
1	1	5	-4	2	-68	19	108	3105
2	1	-4	2	-68	5	19	108	3105
3	-4	1	-68	2	5	19	108	3105
4	-4	-68	1	2	5	19	108	3105
5	-68	-4	1	2	5	19	108	3105
6	-68	-4	1	2	5	19	108	3105
7	-68	-4	1	2	5	19	108	3105

To verify the correct array values were being swapped, we decided to inspect the memory data. To do this, we found when the control signal to write to the data memory, s_DMemWr, was asserted. Whenever data was written, it had to happen over two instructions, since both array[i] and array[i + 1] had to be swapped. This ended up being the easiest way to verify this test, since we could see that the smaller number would be written first into a lower memory address, and then the larger value would be swapped to the next address in the array. In the image below, the lower squares represent the data signals, and show that ‘108’ is being propagated up through the array, showing the sorting instructions functioning as expected through one iteration of the inner loop. Other values can be expected in other sections of the waveforms, through the other iterations of the outer loop of the sort. We can also verify that the correct store word operation is done with the opcode given with the s_OPCODE signal decoded from the instructions.



Bubblesort swap in memory data

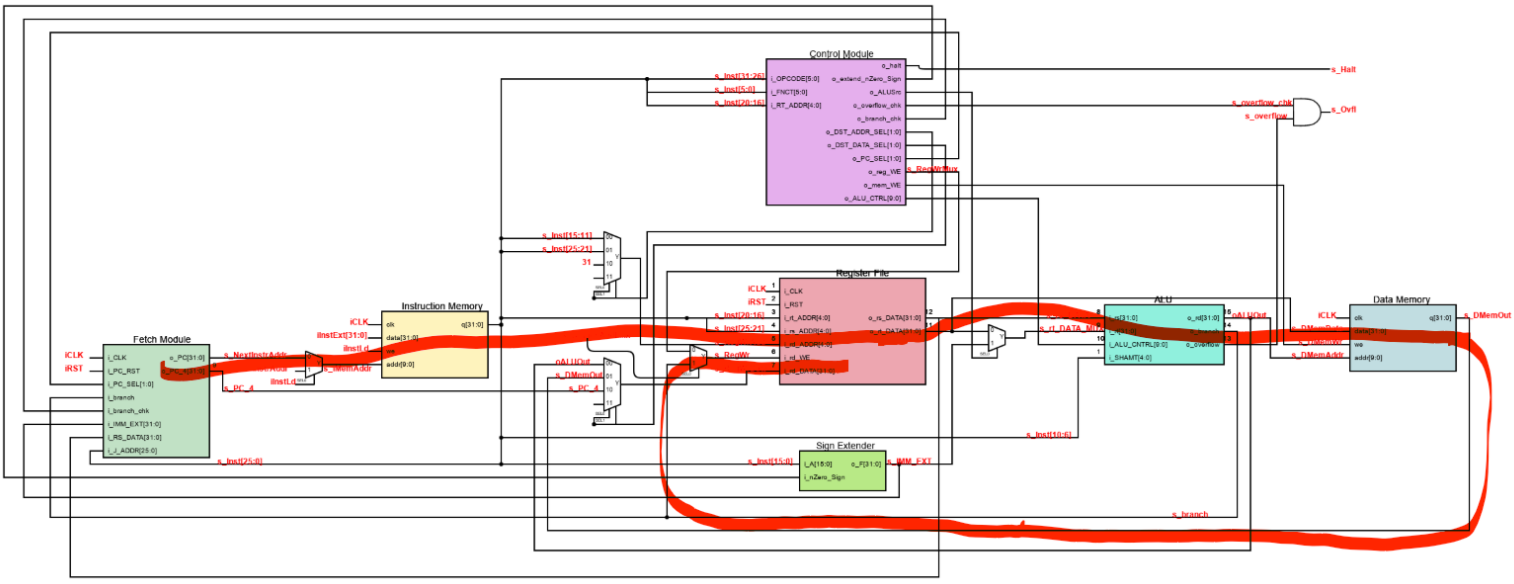
[Part 4] Report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematic from part 1. What components would you focus on to improve the frequency?

The maximum frequency of our processor is 24.71 MHz.

The path given in the timing file was as follows:

1. Fetch module
2. Instruction Memory
3. Register File
4. ALU
5. Data Memory
6. s_RegWr MUX
7. Register File

Based on our timing report, both the instruction memory and data memory took nearly 10 ns to propagate through. This was our largest time difference compared to our other modules, such as the ALU that took around 3 ns for the critical path and the register file for around 4 ns. Looking ahead towards a pipelined design, It would be ideal to reduce the timing of these memory modules down to something closer to 5 ns so that each stage could operate with similar timing for better performance.



Top-level Critical Path