Gene Mary Cheruvathur and Ashley Guillard
EE 469
April 5, 2024

Lab 1 Report

Procedure

This lab consisted of 3 main tasks, a warm up exercise, creating a register file, and creating an Arithmetic Logic Unit (ALU). The first task was a refresher on the intricacies and process of creating a project in Quartus, writing SystemVerilog codes, and simulating files using the provided ModelSim feature. The files and results of the first task will not be provided in this report or the lab submission. The second task was to use a 2D array as a register file for memory. The lab specifications required this task to have one synchronous write port and two asynchronous read ports. The third task was to create an ALU that could add, subtract, AND, as well as OR two 32-bit inputs. An in-depth description of each task is provided below.

Task #2 - Register File

Task 2 asked us to create a 16x32 register file with two asynchronous read ports and one synchronous write port. The setup for the register file can be seen in the block diagram below in Figure 1, where we have six inputs (two for the read port address, one for enabling write, one for the write port address, and one for the data to be written to the write address) and two outputs (the data from the read addresses). Since the register file has 16 words, the total address bits for the inputs are 4-bits; while the number of bits for the data is 32-bits.

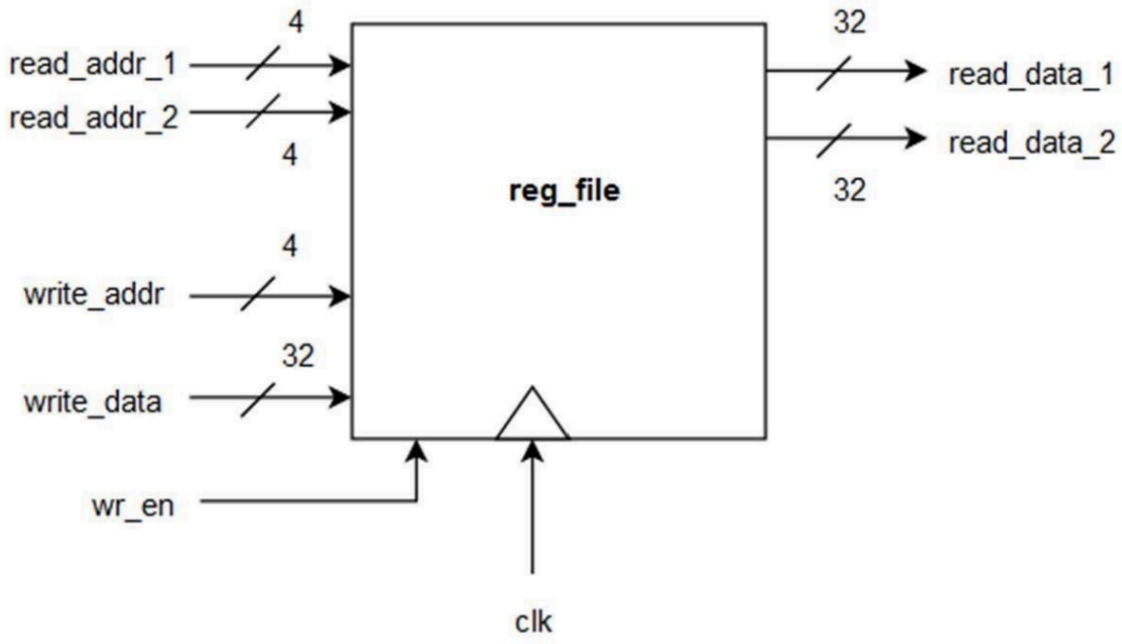

Figure 1: Design for 16x32 register file with two read ports and one write port

When creating the module for the register file, we used sequential logic for writing the data since we knew it was synchronous with the clock cycle. While for the reading, we used the combinational logic for assigning the data from the specified address from the memory. That way it can be an immediate assignment and asynchronous from the clock.

Task #3 - ALU

The ALU built for this lab had the ability to compute addition, subtraction, ANDing, as well as ORing. To compute the addition, a module named "adder" generates a Full Adder for each bit of the 32-bit input. The Full Adder adds two 1-bit inputs together with a carry in bit, if provided. The Full Adder outputs the summation of the inputs as well as a carry out bit, see Table _.

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1: Full Adder Truth Table

The subtraction process used for this ALU is a 2's complement addition. This means that to subtract two binary numbers, we add the first number with the inverse of the second number plus one. Due to this method, the adder module can be used for both the addition and subtraction operation. For the AND/OR operation, the corresponding gate is used in SystemVerilog (& for bitwise AND operation, | for bitwise OR operation).

Using a 2-bit ALUControl input, the user is able to control which operation to compute. As seen in Figure 2, the ALU schematic shows a multiplexer deciphering which result to output depending on the given ALUControl. When ALUControl is 00, the ALU adds. When it is 01, the ALU subtracts. When it is 10, the ALU ANDs. Finally, when the ALUControl is 11, the ALU ORs the inputs.

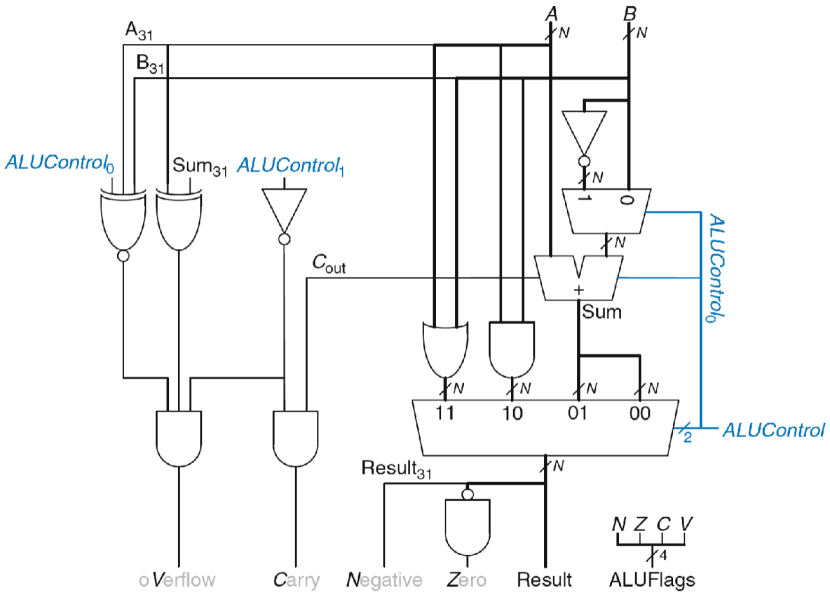

Figure 2: Provided ALU Schematic

This schematic also shows the ALUFlags that our module outputs. These flags use combinational logic to determine if the result is zero, negative, produces a carry out, or produces overflow, see Figure 2. The carry out and overflow flags only are needed when adding and subtracting. For this reason and the fact that the ALU is only adding/subtracting when the first bit of the ALU control is off, the inverse of the ALUControl[1] must be true for either flag to be considered.

ALUFlag bit	Meaning
3	Result is negative
2	Result is 0
1	The adder produces a carry out
0	The adder results in overflow

Table 2: ALUFlags' Meaning

When the main ALU files had been written, the module was simulated in SystemVerilog. To test the ALU, a truth table was written in a text file and read within the testbench module for the ALU. This testbench also consisted of assert statements to ensure that the result and ALUFlags matched the below truth table. All values within the table are in hexadecimal format and represent 32-bit data, except for the 4-bit ALUFlags and 2-bit ALUControl. These flags have also been provided in binary format for quicker comparison to their meanings in Table 2.

Test	ALUControl	A	B	Result	ALUFlags
ADD 0+0	0	00000000	00000000	00000000	4 (0100)
ADD 0+(-1)	0	00000000	FFFFFFFF	FFFFFFFF	8 (1000)
ADD 1+(-1)	0	00000001	FFFFFFFF	00000000	6 (0110)
ADD FF+1	0	000000FF	00000001	00000100	0 (0000)
SUB 0-0	1	00000000	00000000	00000000	6 (0110)
SUB 0-(-1)	1	00000000	FFFFFFFF	00000001	0 (0000)
SUB 1-1	1	00000001	00000001	00000000	6 (0110)
SUB 100-1	1	00000100	00000001	000000FF	2 (0010)
AND FFFFFFFF, FFFFFFFF	2	FFFFFFFF	FFFFFFFF	FFFFFFFF	8 (1000)
AND FFFFFFFF, 12345678	2	FFFFFFFF	12345678	12345678	0 (0000)
AND 12345678, 87654321	2	12345678	87654321	02244220	0 (0000)
AND 00000000, FFFFFFFF	2	00000000	FFFFFFFF	00000000	4 (0100)
OR FFFFFFFF, FFFFFFFF	3	FFFFFFFF	FFFFFFFF	FFFFFFFF	8 (1000)
OR 12345678, 87654321	3	12345678	87654321	97755779	8 (1000)
OR 00000000, FFFFFFFF	3	00000000	FFFFFFFF	FFFFFFFF	8 (1000)
OR 00000000, 00000000	3	00000000	00000000	00000000	4 (0100)

Table 3: Test vectors for ALU

Results

Task #2 - Register File

For this task, we were given three test cases to write in our testbench and the full waveform can be seen in Figure 3:

- Write data is written into the register file at the positive edge of the clock after the wr_en is asserted
- Read data is updated the same clock cycle the register data is updated and the address is provided to the read port
- Read data is updated to the write data the cycle after the address was provided if the write address is the same and wr_en is asserted

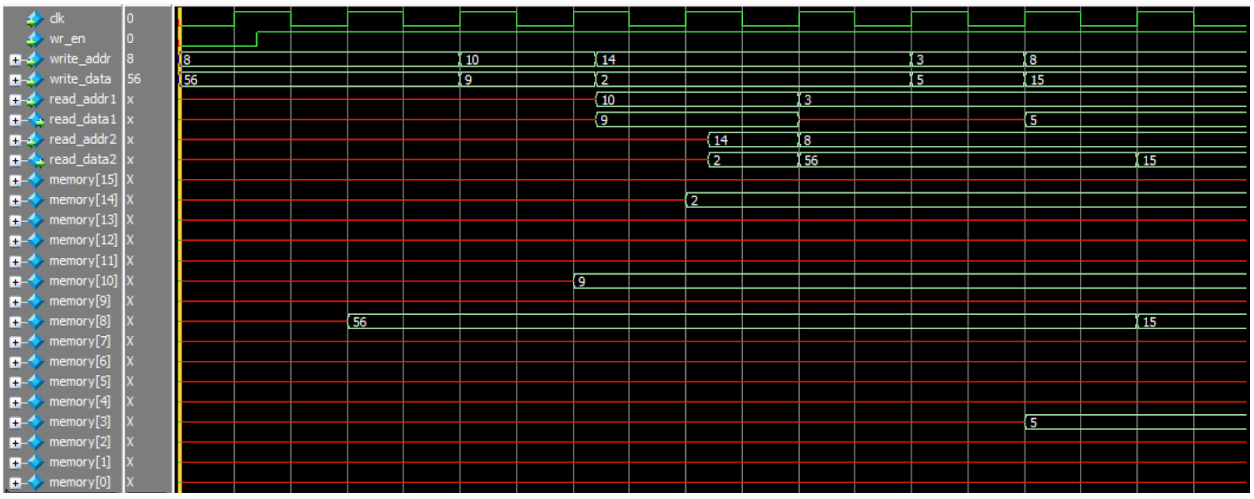

Figure 3: 16x32 register file module full waveform in ModelSIM

For the first test case, we start our clock cycle and set up our write address with the write data. You can see in the zoomed in waveform in Figure 4, no data is written to the memory location since wr_en has not been enabled. In the testbench we give a delay and assert wr_en, even then no data has been written to the memory because we are still in the first clock cycle. After the positive edge of the next clock cycle, the data is written to memory[8]. This passes the first test case, which was to make the write synchronous with the clock.

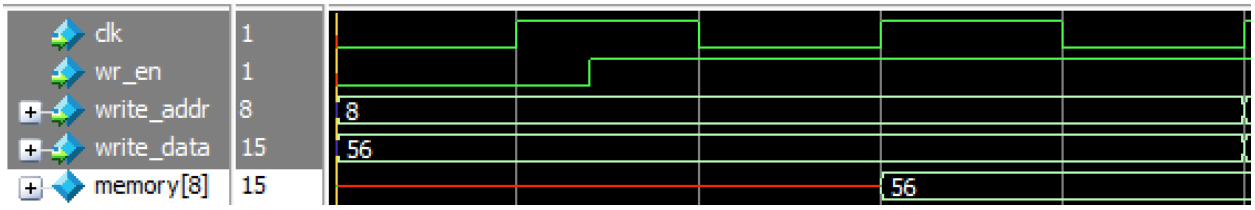

Figure 4: Test case 1 of RegFile

The second test case asked to make asynchronous reads for both ports, which was verified in the waveform in Figure 5. In this, we start off writing to address 10 in the first cycle, which finally writes to the memory at the next clock cycle. And a bit after the second cycle, we ask to read from address 10 and output to the first read port; that way we can test when reading is asynchronous with the clock. When observing the waveform, it can be seen that the read data is immediately outputted, since the data existed in the memory. We do a similar test to the second read port and receive the same results. From this, it can be concluded that the read ports are asynchronous.

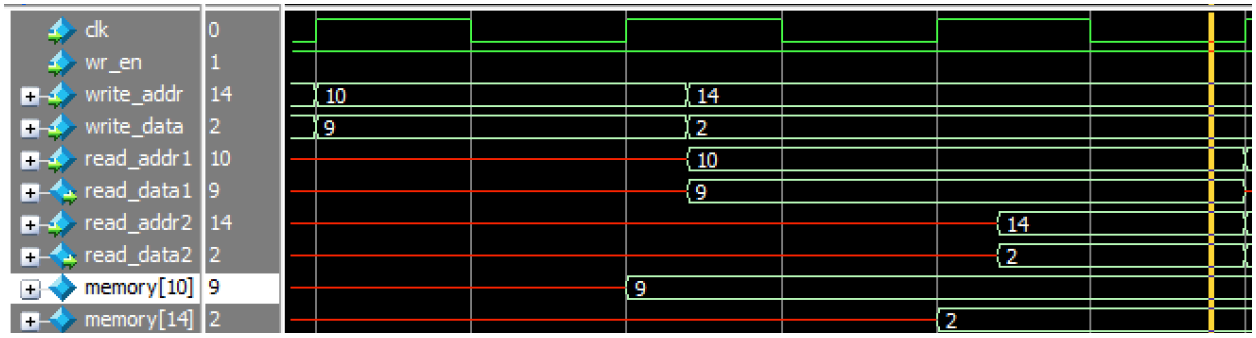

Figure 5: Test case 2 of RegFile

Finally the third task has us write data to an address while reading from there to see that the data in the memory is updating. Initially, we set the first read address to read from address 3 and second from address 8, the first output port outputs nothing since there is nothing stored at memory[3] and the second output port outputs the current data stored. Afterwards, we write to address 3 and the moment it's written to the memory, the data is outputted to the first read port. We follow the same steps and write to address 8, immediately when written, the output changes to the new data at the second read port.

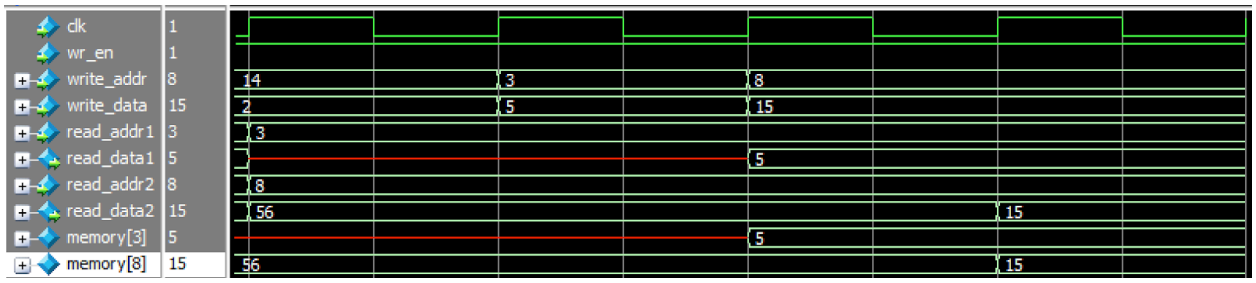

Figure 6: Test case 3 of RegFile

Since all the tests behaved the way we expected the 16x32 register file to behave, we can confirm that our register file works. To summarize, the write port is synchronized with the clock and the two read ports are asynchronous.

Task #3 - ALU

The overall ALU was tested using test vectors as described in the procedure section, however, the individual adder and FullAdder modules were tested separately to ensure that each section of the ALU worked. The starting base for the addition and subtraction section of the ALU was a simple Full Adder which was thoroughly tested below.

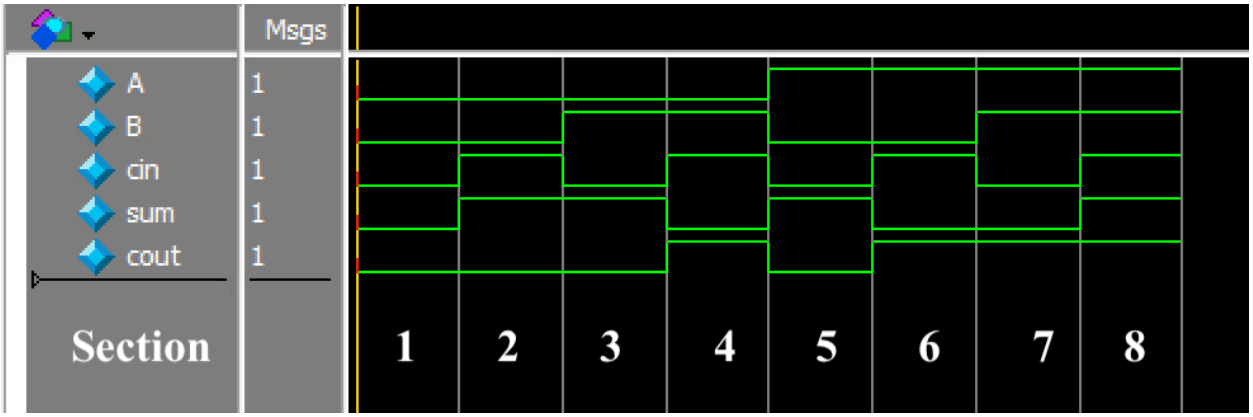

Figure 7: FullAdder module ModelSim

The above simulation screenshot shows every possible combination of the inputs A, B, and cin starting from 000 and incrementing until 111. When all of the inputs are zero, the output for sum and cout are both zero, which can be seen in the first section. When only one of the variables is on, the outputs produced are; sum=1 and cout=0. This can be seen in the second, third, and fifth section of Figure 7. When two of the inputs are on, sum outputs a 0 and cout outputs a 1. The reason is because 1 + 1 in binary is 10, with the 1 representing a carry out and the zero representing the 1-bit sum. This result can be seen in sections four, six, and seven. When all three inputs are on, section eight, the sum and cout outputs a 1. Since 1 + 1 + 1 in binary would be 11, both the sum and cout will be outputted. The outputs show that this module works as expected.

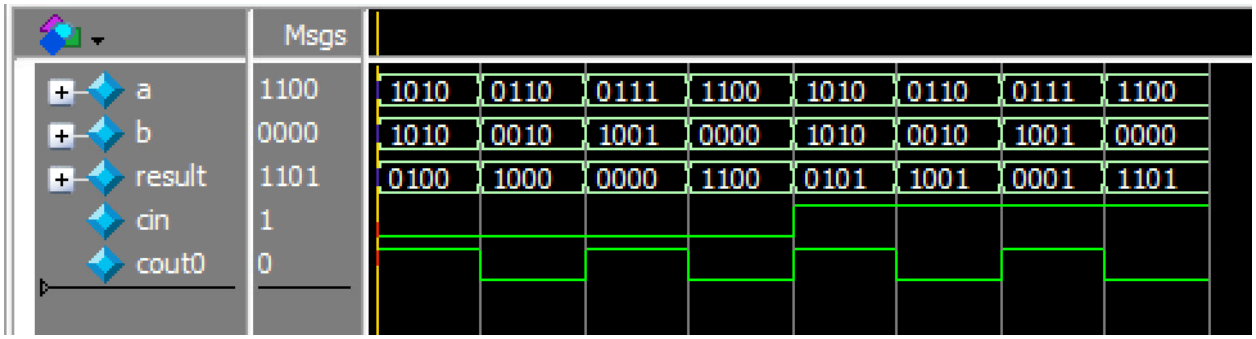

Figure 8: adder module ModelSim

The next module to test would be the adder module which creates multiple instantiations of the Full Adder module equal to the number of bits in the inputs. For clarity, this module was tested

using 4 bits and the first four tests were chosen arbitrarily. The first four tests had an initial carry in value of zero. When adding the first inputs, 1010 (a) and 1010 (b), the result would be 10100, however, the bit size of the output is only 4 bits meaning that this combination produces a carry out and a sum of 0100. The same logic can be applied to the next three tests, 0110+0010=1000, 0111+1001=(1)0000, and 1100+0000=1100. For the remaining four tests, a carry in of 1 was added. The tests were then repeated which resulted in a summation that was one higher than the previous ones. This proves that this module works as expected.

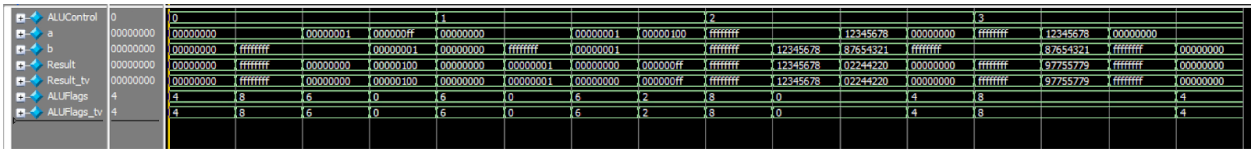

Figure 9: alu module ModelSim

As mentioned in the Procedure section, this module was tested using a text file with the test vectors shown in Table 3. Figure 9 shows the truth table result (row 5) and actual result (row 4) which were compared with assert statements within the testbench module. The ALUFlags (row 6 and 7) were also compared using assert statements. All of the assert statements resulted in the correct information appearing without any errors, see Appendix B - Task #3 for the assert and error messages. The section below goes over each section of the testbench and describes what each hexadecimal output represents.

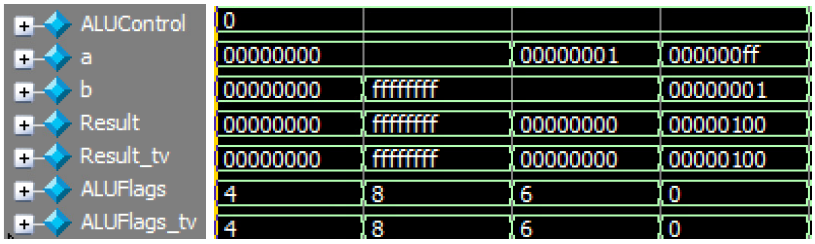

Figure 10: Part 1 of alu ModelSim

This section focused on the addition of the variables a and b. The first test was 0 + 0 which results in a 0. As can be seen the results from the truth table and module are both 0 and produce the ALUFlags 4 (0100) which means the result is equal to zero. The second test was to add 0 and -1 which results in -1. The ALUFlags is 8 (1000) which means that the result (-1) is negative. The next test was 1 plus -1, this results in a 0. Since we are adding 0x00000001 and 0xffffffff the actual result is 0x(1)00000000 with a carry out variable. This is not overflow since the actual answer can be found in the last 32 bits (0x00000000). In the final addition test, we add 0xff and 0x1. This results in an output of 0x100 and no flags.

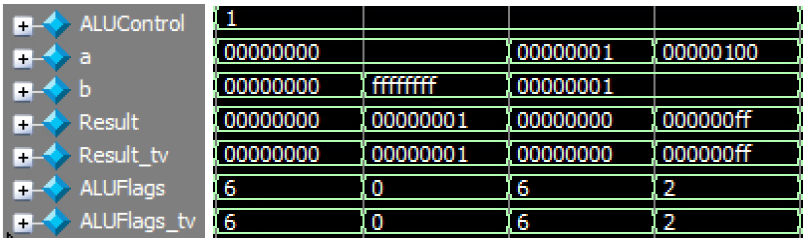

Figure 11: Part 2 of alu ModelSim

The second section focuses on the subtraction of a and b (a-b). The first test is 0-0 which results in a 0. This operation outputs two flags (0110), the result is zero and a carry out was created. The carry out was created due to using a 2's complement addition to create subtraction as described in the Procedure. When the adder adds 0x00000000 (0) and 0xffffffff+1 (2's complement of 0) the result is 0x(1)00000000 similar to the third addition test shown in Figure 11. The second test is 0-(-1) which can be written as the addition of 0x00000000 + 0x00000001 (2's complement of -1). This results in a 1 and does not produce any flags. The third test 1-1 results in a 0 and produces the zero and carry out flags for the same reason as subtraction test one. The last subtraction test is 0x100 minus 0x1 which results in 0xff and a flag for producing a carry out. Since we are adding 0x00000100 with 0xffffffff (2's complement) the result is 0x(1)000000ff.

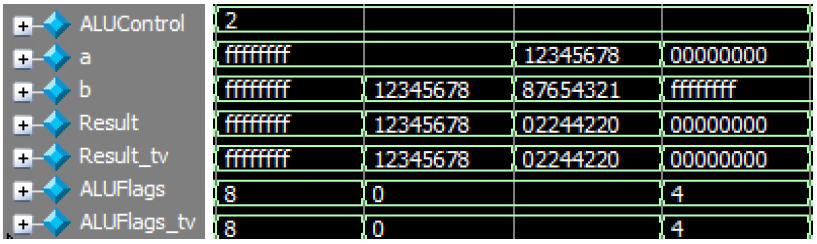

Figure 12: Part 3 of alu ModelSim

The third section of the ModelSim focuses on the ANDing operation. This operation will not produce a carry out or overflow flag since we will not need to worry about adding or subtracting. Most of these results are straightforward, for example, 0xffffffff (each 0xf is 1111 in binary) AND 0xffffffff is 0xffffffff since all bits are on for both inputs. The second test shows that ANDing a variable will all ones (a) and a variable with only some bits on will result in only the matching bits to be carried to the result. In the third test we can compare the bits required for 0x8 (1000) and 0x1 (0001). From their binary representations, we can see that there are no ones that match within the first byte of the inputs. This results in a 0x0 (0000) for the first byte of the answer. The rest of the inputs can be compared the same way resulting in 0x02244220. The last test ANDs 0 with 0xffffffff which will result in a 0 since one of the inputs does not have any bits that are on.

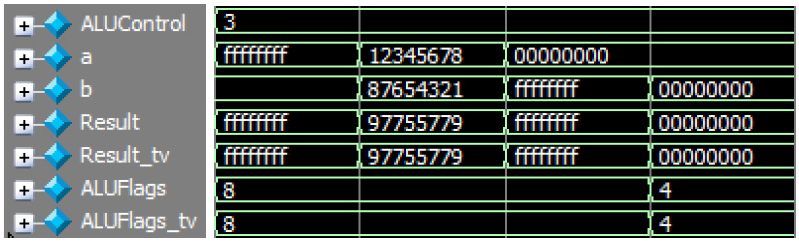

Figure 13: Part 4 of alu ModelSim

The last section of the testbench was the OR operation. The first test is 0xffffffff ORed with itself (cut off in the screenshot). The second test starts by comparing 0x1 (0001) and 0x8 (1000), the result is 0x9 (1001). With this logic, the rest of the second test can be determined to be 0x97755779. The third test ORs and input of all ones which automatically results in an output of all ones. The last test ORs two 0's which results in a zero since no bits were on in either input. After analyzing each test, we can confirm that the modules for Task 3 performed as expected with the correct resulting outputs.

Final Product

The goal of this lab was to refresh our knowledge on using Quartus to write SystemVerilog files, simulate a register file, and create an ALU. The first task of this lab acted as a warm up to Quartus in case we had not used this program in a while. This task was only mentioned a few times in this report and did not require any submissions.

For Task 2, we were expected to create a 16x32 register file that had two asynchronous read ports and one write synchronous port. Our final system implemented the write in sequential logic and the reads as combinational logic to achieve this. We later tested these cases and confirmed that our design behaved the way we were expected to.

For the final task, we created an ALU that could add, subtract, AND, as well as, OR two 32-bit inputs. The lab specifications said that this ALU needed to work for 32-bits, but we thought it would be good to extend this design to any number of bits by using a parameter "N", see Appendix B - Task #3. Our final design was simulated using a text file holding the truth table found in Table 3 from the Procedure section. Due to some intricacies of Quartus, the testbenches worked best when written in the same file as the module to be tested. The final product worked as expected producing the same results as the truth table.

Appendix A - Task #2

1) reg_file.sv

```
//This module is created for Lab 1 Task 2, where we design a 16x32
//register file with 2 asynchronous read ports and 1 synchronous
//write port. The inputs are the write and two read addresses,
//the write data and if write is enabled. The outputs are the data
//at the specified read addresses.
//
//Inputs: clk, wr_en, write_data, write_addr, read_addr1, read_addr2
//Outputs: read_data1, read_data2
module reg_file(clk, wr_en, write_data, write_addr, read_addr1,
                                          read_addr2, read_data1, read_data2);
      input logic clk, wr_en;
      input logic [31:0] write_data;
      input logic [3:0] write_addr;
      input logic [3:0] read_addr1, read_addr2;
      output logic [31:0] read_data1, read_data2;

      //16x32 registers holding data as memory
      logic [15:0][31:0] memory;

      //writes data to specified address on memory at positive edge of
      //clock cycles (synchronous)
      always_ff @(posedge clk) begin
            //only write if wr_en is true
            if (wr_en) begin
                  memory[write_addr] <= write_data;
            end
      end

      //gives the data immediately when both read data is specified (asynch)
      assign read_data1 = memory[read_addr1];
      assign read_data2 = memory[read_addr2];
endmodule
```

2) reg_file_testbench.sv

```
//This is the testbench module for the the reg_file for Lab 1 Task 2.
//This testbench tests out the specified 3 test cases, which is if
//the writing to the reg file is synchronous, reading from 2 address
//is asynchronous, and if the read data updates immediately when the
//data at the next clock-cycle-for-writing is updated.
module reg_file_testbench();
      logic clk, wr_en;
      logic [31:0] write_data;
      logic [3:0] write_addr;
```

```
	logic [3:0] read_addr1, read_addr2;
	logic [31:0] read_data1, read_data2;

	//instantiates reg_file
	reg_file dut(.clk, .wr_en, .write_data, .write_addr, .read_addr1,
                                          .read_addr2, .read_data1,
.read_data2);

	//clock
	parameter clock_period = 100;
	initial begin
		clk <= 0;
		forever #(clock_period/2) clk <= ~clk;
	end

	//starts testing the 3 important cases
	initial begin
		//testing if the write only happens in on posedge of clock while
wr_en = 1 (synch)
		write_data <= 32'd56; write_addr <= 4'd8; wr_en <= 0; @(posedge
clk);
                                                                #20; //add a
delay to enable write during a clock cycle

                wr_en <= 1; @(posedge clk);

                                          @(posedge clk);


		//testing read address at the same clock cycle write (asynch)
		//read port 1
		write_data <= 32'd9; write_addr <= 4'd10;
@(posedge clk);
                                                                #20; //add a
delay to read data during a clock cycle
		read_addr1 <= 4'd10;
		//read port 2
		write_data <= 32'd2; write_addr <= 4'd14;
@(posedge clk);
                                                                #20; //add a
delay to read data during a clock cycle
		read_addr2 <= 4'd14;

                                          @(posedge clk);

		//testing the the read addresses immediate updates when data
written
		read_addr1 <= 4'd3; read_addr2 <= 4'd1;
@(posedge clk);
```

```
            write_data <= 32'd5; write_addr <= 4'd3;
@(posedge clk);
            write_data <= 32'd15; write_addr <= 4'd1;
@(posedge clk);

                                                @(posedge clk);
            $stop;
      end
endmodule
```

Appendix B - Task #3

1) FullAdder.sv

```
/****************************** FullAdder.sv ******************************

      Ashley Guillard and Gene Mary Cheruvathur
      EE 469 Sp 24: Professor Hussein
      April 5, 2024
      Lab 1: Full Adder

*/

// Adds three 1-bit inputs, A, B, and cin. Produces the sum and carry out.
//
// Inputs: A, B, cin (carry in)
// Outputs: sum, cout (carry out)
module FullAdder (A, B, cin, sum, cout);

      input logic A, B, cin;
      output logic sum, cout;

      assign sum = A ^ B ^ cin;
      assign cout = A&B | cin & (A^B);

endmodule


// TESTBENCH for FullAdder
// Tests all possible values for the inputs
module FullAdder_testbench();

      logic A, B, cin, sum, cout;

      // Instantiate DUT (Device Under Test)
      FullAdder dut (.A, .B, .cin, .sum, .cout);

      // Tests all possible values
      initial begin

            A <= 0; B <= 0; cin <= 0; #10;
            A <= 0; B <= 0; cin <= 1; #10;
            A <= 0; B <= 1; cin <= 0; #10;
            A <= 0; B <= 1; cin <= 1; #10;
            A <= 1; B <= 0; cin <= 0; #10;
            A <= 1; B <= 0; cin <= 1; #10;
            A <= 1; B <= 1; cin <= 0; #10;
            A <= 1; B <= 1; cin <= 1; #10;
```

```
	end

endmodule
```

2) adder.sv

```
/****************************** adder.sv ******************************

	Ashley Guillard and Gene Mary Cheruvathur
	EE 469 Sp 24: Professor Hussein
	April 5, 2024
	Lab 1: adder module

*/

// Adds two N-bit numbers and produces the output and carry out.
// Generates an instantiations of the FullAdder module for each
// of the N bits. The initial carry in is used for the first
// Full Adder.
//
// Inputs: a, b, cin (initial carry in)
// Outputs: result, cout0 (final carry out)
module adder #(parameter N = 3) (
	input logic [N-1:0] a, b,
	input logic cin,
	output logic [N-1:0] result,
	output logic cout0
	);

	// cin and cout
	wire [N-1:0]cout;
	assign cout0 = cout[N-1];

	genvar i;
	generate
		for(i=0; i<N; i++) begin : Adders
			if (i==0) FullAdder FA0 (.A(a[i]), .B(b[i]), .cin(cin),
.sum(result[i]), .cout(cout[i]));
			else FullAdder FA (.A(a[i]), .B(b[i]), .cin(cout[i-1]),
.sum(result[i]), .cout(cout[i]));
		end
	endgenerate

endmodule


// TESTBENCH for adder
// Tests various 4-bit input values
```

```
module adder_testbench();

	parameter N = 4;
	logic [N-1:0] a, b, result;
	logic cin, cout0;

	// Instantiate DUT (Device Under Test)
	adder #(.N(N)) dut (.a, .b, .cin, .result, .cout0);

	initial begin

		// Without carry in
		cin <= 0;
		a <= 10; b = 10; #10; // 10 + 10 = 20 (overflow)
		a <= 6; b = 2; #10;   // 6 + 2 = 8
		a <= 7; b = 9; #10;   // 7 + 9 = 16 (overflow)
		a <= 12; b = 0; #10;  // 12 + 0 = 12

		// With carry in
		cin <= 1;
		a <= 10; b = 10; #10; // 1 + 10 + 10 = 21 (overflow)
		a <= 6; b = 2; #10;   // 1 + 6 + 2 = 9
		a <= 7; b = 9; #10;   // 1 + 7 + 9 = 17 (overflow)
		a <= 12; b = 0; #10;  // 1 + 12 + 0 = 13

	end

endmodule
```

3) alu.sv

```
/****************************** alu.sv ******************************

	Ashley Guillard and Gene Mary Cheruvathur
	EE 469 Sp 24: Professor Hussein
	April 5, 2024
	Lab 1: alu (Arithmetic Logic Unit)

*/

// This module holds the ALU for lab 1 Task 3 which operates on two N-bit
inputs.
// This ALU is able to change operations based on the 2-bit "ALUControl"
input. The
// possible operations are: Addition (00), Subtraction (01), AND (10), as well
as
// OR (11). Based on the given input, this module will output the correct
result of
```

```
// the selected operation. This module additionally outputs flags (within the
"ALUFlags"
// variable) when the result is negative (ALUFlags[3]) and result is zero
([2]). When
// adding or subtracting, the ALUFlags[1] and ALUFlags[0] will show whether
the adder
// produced a carry out or experianced an overflow, respectivly.
//
// Inputs: a, b, ALUControl
// Outputs: Result, ALUFlags
module alu #(parameter N = 32)(

	// Inputs a and b (N-bit)
	input logic [N-1:0] a, b,

	// Controls the operation
	input logic [1:0] ALUControl,

	// Output result from operation
	output logic [N-1:0] Result,

	// ALU flags:
	// [0] = Adder results in overflow
	// [1] = Adder produced a carry out
	// [2] = Result is 0
	// [3] = Result is negative
	output logic [3:0] ALUFlags
	);

	// Results from each operation
	logic [N-1:0] sum, AND, OR, b_new;

	// Cout for adder
	logic cout;

	// add/sub b value MUX
	always_comb begin
		if (ALUControl[0]) b_new = ~b; // If subtracting, use ~b
		else b_new = b;                // If adding, use b
	end

	// Instantiate add module
	// Uses Full Adders to add each individual bits together. Produces a
Cout value.
	adder #(.N(N)) adding (.a, .b(b_new), .cin(ALUControl[0]), .result(sum),
.cout0(cout));

	// Store result of ANDing
	assign AND = a & b;
```

```
	// Store result from ORing
	assign OR = a | b;

	// Ending result MUX
	always_comb begin
		case(ALUControl)
			2'b10: Result = AND;
			2'b11: Result = OR;
			default: Result = sum;
		endcase
	end

	// ALU Flags output circuit
	assign ALUFlags[0] = ~( (ALUControl[0]) ^ (a[N-1]) ^ (b[N-1]) ) & (
(a[N-1]) ^ (sum[N-1]) ) & ~( ALUControl[1] );
	assign ALUFlags[1] = ~ALUControl[1] & cout;
	assign ALUFlags[2] = (Result == 0);
	assign ALUFlags[3] = Result[N-1];

endmodule


// TESTBENCH for alu
// Tests various input cases
module alu_testbench();

	parameter N = 32;
	logic [N-1:0] a, b, Result, Result_tv;
	logic [1:0] ALUControl;
	logic [3:0] ALUFlags, ALUFlags_tv;
	logic [103:0] testvectors [1000:0];

	// Instantiate DUT (Device Under Test)
	alu #(.N(N)) dut (.a, .b, .ALUControl, .Result, .ALUFlags);

	// Tests the truth table from the lab specifiations
	initial begin

		// Read file data
		$readmemh("U:/Projects for EE 469/Lab 1/alu.tv", testvectors);

		for(int i = 0; i < 16; i = i + 1) begin

			// Sort the information from the file to the corresponding
variables
			{ALUControl, a, b, Result_tv, ALUFlags_tv} <=
testvectors[i];
```

```
                    #10; // Wait

                    // Check that the results match
                    assert(Result == Result_tv) $display ("Result is correct");
                    else $error ("Incorrect results");

                    // Check that the ALUFlags match
                    assert(ALUFlags == ALUFlags_tv) $display ("ALUFlags are
correct");
                    else $error ("Incorrect ALUFlags");

                    #10; // Wait before repeating

            end //for loop

            $stop; // End waveform

      end //initial

endmodule //alu_testbench
```

4) alu.tv

```
0_00000000_00000000_00000000_4
0_00000000_FFFFFFFF_FFFFFFFF_8
0_00000001_FFFFFFFF_00000000_6
0_000000FF_00000001_00000100_0
1_00000000_00000000_00000000_6
1_00000000_FFFFFFFF_00000001_0
1_00000001_00000001_00000000_6
1_00000100_00000001_000000FF_2
2_FFFFFFFF_FFFFFFFF_FFFFFFFF_8
2_FFFFFFFF_12345678_12345678_0
2_12345678_87654321_02244220_0
2_00000000_FFFFFFFF_00000000_4
3_FFFFFFFF_FFFFFFFF_FFFFFFFF_8
3_12345678_87654321_97755779_8
3_00000000_FFFFFFFF_FFFFFFFF_8
3_00000000_00000000_00000000_4
```