Stephen Macris
Aaron Hong
EE469 Lab 1
4/7/2023

Procedure

Task 2

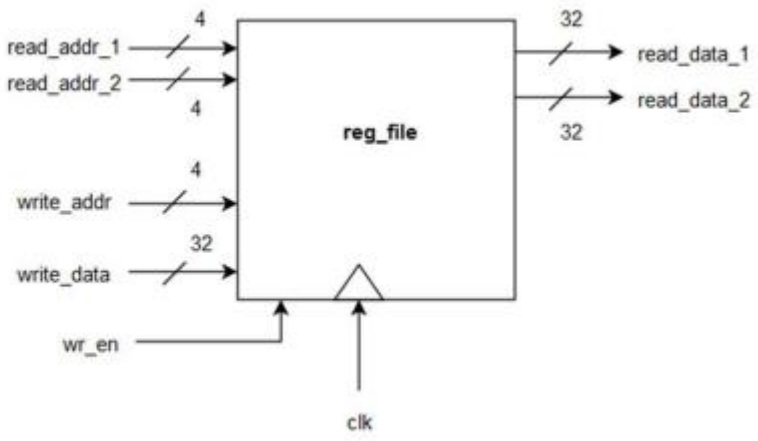

Figure 1: 16x32 register file

In this task, we designed, implemented, and tested a 16x32 register file with two read ports and one write port and asynchronous (pictured above) in System Verilog. For reference, we were given Verilog code for a 16x32 one read, one write, synchronous register file. This largely influenced our implementation and design of our specific register file. The testbench also had specifications such as to verify that write data is written into the register file the clock cycle after wr_en is asserted. Taking all this into account, we constructed our own 16x32 register file and tested it to ensure it made the specifications.

Task 3

This task asked us to design and implement a 32-bit ALU (Arithmetic Logic Unit). The ALU talked about in lecture could add, subtract, do AND, and OR. There were also 4 bits of ALUFlags we had to implement 3 meaning result is negative, 2 result is 0, 1 the adder produces a carry out, and 0 the adder results in overflow. Using these specifications, the ALU was implemented into Verilog. To test it, we used 16 test vectors to ensure the implementation was correct. The test vectors are shown below in Table 1.

Stephen Macris
Aaron Hong
EE469 Lab 1
4/7/2023

Test	**ALUControl[1:0]**	**A**	**B**	**Result**	**ALUFlags**
ADD 0+0	0	000000000	00000000	00000000	4
ADD 0+(-1)	0	FFFFFFFF	FFFFFFFF	FFFFFFFF	8
ADD 1+(-1)	0	FFFFFFFF	FFFFFFFF	00000000	6
ADD FF+1	0	000000FF	00000001	00000100	0
SUB 0-0	1	00000000	00000000	00000000	6
SUB 0-(-1)	1	00000000	FFFFFFFF	00000001	0
SUB 1-1	1	00000001	00000001	00000000	6
SUB 100-1	1	00000100	00000001	000000FF	2
AND FFFFFFFF, FFFFFFFF	2	FFFFFFFF	FFFFFFFF	FFFFFFFF	8
AND FFFFFFFF, 12345678	2	FFFFFFFF	12345678	12345678	0
AND 12345678, 87654321	2	12345678	87654321	02244220	0
AND 00000000, FFFFFFFF	2	00000000	FFFFFFFF	00000000	4
OR FFFFFFFF, FFFFFFFF	3	FFFFFFFF	FFFFFFFF	FFFFFFFF	8
OR 12345678, 87654321	3	12345678	87654321	97755779	8
OR 00000000, FFFFFFFF	3	00000000	FFFFFFFF	FFFFFFFF	8
OR 00000000, 00000000	3	00000000	00000000	00000000	4

Table 1: test vectors

Stephen Macris
Aaron Hong
EE469 Lab 1
4/7/2023

Results

Task 2

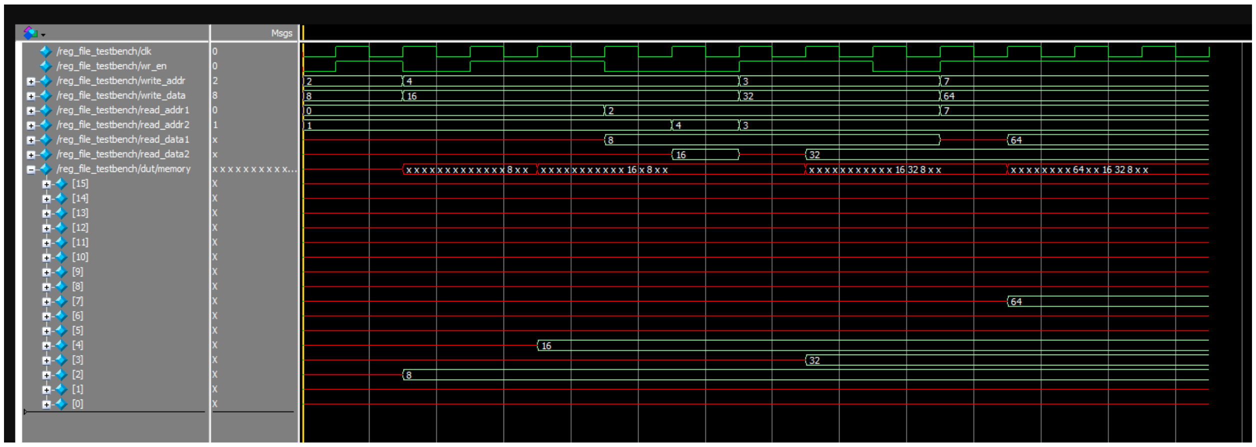
/reg_file_testbench/clk
/reg_file_testbench/wr_en
/reg_file_testbench/write_addr
/reg_file_testbench/write_data
/reg_file_testbench/read_addr1
/reg_file_testbench/read_addr2
/reg_file_testbench/read_data1
/reg_file_testbench/read_data2
/reg_file_testbench/dut/memory
Msgs

Figure 3: Task 2 simulation

The above screenshot shows the testbench for the implementation of the 16x32 register file. The three tests that were asked to verify the implementation was correct was write data is written into the register file the clock cycle after wr_en is asserted, read data is updated to the register data at an address the same cycle the address was provided, and read data is updated to write data at an address the cycle after the address was provided if the write address is the same and wr_en was asserted. Based off the results of the screenshot, the tests did pass and the implementation was indeed correct.

Task 2

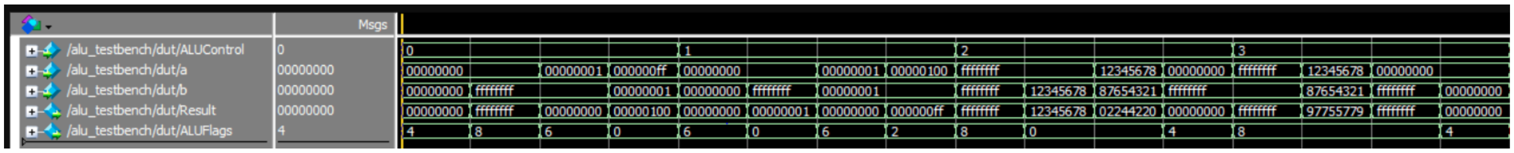
/alu_testbench/dut/ALUControl
/alu_testbench/dut/a
/alu_testbench/dut/b
/alu_testbench/dut/Result
/alu_testbench/dut/ALUFlags
Msgs

Figure 4: Task 3 simulation

The above screenshot shows the ModelSim simulation of the ALU implementation. Here we are looking to see if addition, subtraction, AND, and OR are working successfully based off the test vectors. It is also important to check if the ALUFlags are being flagged appropriately. Based off the results from the screenshot, we can confirm that the ALU is functioning properly, and the implementation was correct.

Appendix

```
//Aaron Hong (ahong02)
//Stephen Macris (smacris)
//3/29/23
//EE469 Lab1

//This module creates an asynchronous, two read port, one write port, 16x32 register file.

//Inputs: Two 1-bits clk (clock signal), wr_en (write enable), three 4-bits write_addr, read_addr1,
//read_addr2 (write and read addresses), one 32-bit write_data (data to be written at given write address).
//Outputs: Two 32-bits read_data1, read_data2 (data read from given read addresses).
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 [3:0] write_addr, read_addr1, read_addr2;
   input  logic [31:0] write_data;
   output logic [31:0] read_data1, read_data2;

   logic [15:0][31:0] memory;

   always_ff @(posedge clk) begin
      if(wr_en) begin
         memory[write_addr] <= write_data;
      end
   end

   always_comb begin
      read_data1 <= memory[read_addr1];
      read_data2 <= memory[read_addr2];
   end

endmodule
```

```
//Aaron Hong (ahong02)
//Stephen Macris (smacris)
//3/29/23
//EE469 Lab1

//This testbench tests reg_file to ensure that it functions properly: mostly testing asynchronous read.

module reg_file_testbench();

	logic clk, wr_en;
	logic [31:0] write_data, read_data1, read_data2;
	logic [3:0] write_addr, read_addr1, read_addr2;

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

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

	end //initial

	initial begin

	   wr_en <= 1'b0; write_addr <= 4'b0010; write_data <= 32'd8; read_addr1 <= 4'b0000; read_addr2 <= 4'b0001;  @(posedge clk);
	   wr_en <= 1'b1;                                                                   @(posedge clk);
	   wr_en <= 1'b0; write_addr <= 4'b0100; write_data <= 32'd16; read_addr1 <= 4'b0000; read_addr2 <= 4'b0001; @(posedge clk);
	   wr_en <= 1'b1;                                                                   @(posedge clk);
	                    @(posedge clk);
	   wr_en <= 1'b0; read_addr1 <= 4'b0010;                                            @(posedge clk);
	   wr_en <= 1'b0; read_addr2 <= 4'b0100;                                            @(posedge clk);
	   wr_en <= 1'b1; write_addr <= 4'b0011; write_data <= 32'd32; read_addr2 <= 4'b0011; @(posedge clk);
	                    @(posedge clk);
	   wr_en <= 1'b0;                                                                   @(posedge clk);
	   wr_en <= 1'b1; write_addr <= 4'b0111; write_data <= 32'd64; read_addr1 <= 4'b0111; @(posedge clk);
	                    @(posedge clk);
	                    @(posedge clk);
	                    @(posedge clk);

	   $stop; //end simulation

	end //initial

endmodule
```

```
//Aaron Hong (ahong02)
//Stephen Macris (smacris)
//3/29/23
//EE469 Lab1

//This module creates a basic arithmetic logic unit capable of addition, subtraction,
//ANDing, and ORing.

//Inputs: Two 32-bits a, b (inputs to be processed), one 2-bit ALUControl (controls which operation).
//Outputs: One 32-bit Result (result of chosen operation), one 4-bit ALUFlags (flags thrown depending on the result).
module alu (a, b, ALUControl, Result, ALUFlags);

   input  logic  [31:0] a, b;
   input  logic [1:0] ALUControl;
   output logic [31:0] Result;
   output logic [3:0] ALUFlags;
   logic cout, x, diff_sign;
   logic [31:0] sum, b_mod;

   assign x = ~ALUControl[0] & ~ALUControl[1] & (a[31] == b[31]);
   assign diff_sign = a[31] ^ sum[31];

   //Instantiates a 32-bit full adder to do addition and subtraction operations.
   fulladder32 FA(.A(a), .B(b_mod), .cin(ALUControl[0]), .sum(sum), .cout(cout));

   //Determines what operation to perform depending on ALUControl.
   always_comb begin
      case (ALUControl[0])
         1'b0: b_mod = b;
         1'b1: b_mod = ~b;
      endcase;

      case (ALUControl)
         2'b00: Result = sum;
         2'b01: Result = sum;
         2'b10: Result = a & b;
         2'b11: Result = a | b;
      endcase

      ALUFlags[3] = Result[31];
      ALUFlags[2] = (Result == 32'b0);
      ALUFlags[1] = ~ALUControl[1] & cout;
      ALUFlags[0] = x & diff_sign & ~ALUControl[1];
   end

endmodule
```

```
//Aaron Hong (ahong02)
//Stephen Macris (smacris)
//3/29/23
//EE469 Lab1

//This module tests the ALU module against a given set of test vectors.

module alu_testbench();
   logic [103:0] testvectors [1000:0];
   logic [31:0] a, b, Result;
   logic [3:0] ALUFlags;
   logic [1:0] ALUControl;
   logic clk;

   alu dut (.a(a), .b(b), .ALUControl(ALUControl), .Result(Result), .ALUFlags(ALUFlags));

   parameter CLOCK_PERIOD = 100;

   //clock setup
   initial clk = 1;
   always begin
      #(CLOCK_PERIOD/2);
      clk = ~clk;
   end

   initial begin
      $readmemh("alu.tv", testvectors);

      for(int i = 0; i < 20; i = i + 1) begin
         {ALUControl, a, b, Result, ALUFlags} = testvectors[i]; @(posedge clk);
      end //end simulation

   end //initial

endmodule
```

```
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_2
1_00000001_00000001_00000000_4
1_00000100_00000001_00000011_0
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
```

```
//Aaron Hong (ahong02)
//Stephen Macris (smacris)
//EE469 Lab1

//This module adds two 32-bit numbers and a given carry-in value.

//Inputs: Two 32-bits A, B (inputs to be added), one 1-bit cin (carry-in value).
//Outputs: One 32-bit sum (sum), one 1-bit cout (carry-out value).
module fulladder32 (A, B, cin, sum, cout);

   input logic [31:0] A;
   input logic [31:0] B;
   input logic cin;

   output logic [31:0] sum;
   output logic cout;

   logic c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16, c17, c18
, c19, c20, c21, c22, c23, c24, c25, c26, c27, c28, c29, c30;

   //Chains together 32 instantiations of a 1-bit adder.
   fulladder FA0(.A(A[0]), .B(B[0]), .cin(cin), .sum(sum[0]), .cout(c0));
   fulladder FA1(.A(A[1]), .B(B[1]), .cin(c0), .sum(sum[1]), .cout(c1));
   fulladder FA2(.A(A[2]), .B(B[2]), .cin(c1), .sum(sum[2]), .cout(c2));
   fulladder FA3(.A(A[3]), .B(B[3]), .cin(c2), .sum(sum[3]), .cout(c3));
   fulladder FA4(.A(A[4]), .B(B[4]), .cin(c3), .sum(sum[4]), .cout(c4));
   fulladder FA5(.A(A[5]), .B(B[5]), .cin(c4), .sum(sum[5]), .cout(c5));
   fulladder FA6(.A(A[6]), .B(B[6]), .cin(c5), .sum(sum[6]), .cout(c6));
   fulladder FA7(.A(A[7]), .B(B[7]), .cin(c6), .sum(sum[7]), .cout(c7));
   fulladder FA8(.A(A[8]), .B(B[8]), .cin(c7), .sum(sum[8]), .cout(c8));
   fulladder FA9(.A(A[9]), .B(B[9]), .cin(c8), .sum(sum[9]), .cout(c9));
   fulladder FA10(.A(A[10]), .B(B[10]), .cin(c9), .sum(sum[10]), .cout(c10));
   fulladder FA11(.A(A[11]), .B(B[11]), .cin(c10), .sum(sum[11]), .cout(c11));
   fulladder FA12(.A(A[12]), .B(B[12]), .cin(c11), .sum(sum[12]), .cout(c12));
   fulladder FA13(.A(A[13]), .B(B[13]), .cin(c12), .sum(sum[13]), .cout(c13));
   fulladder FA14(.A(A[14]), .B(B[14]), .cin(c13), .sum(sum[14]), .cout(c14));
   fulladder FA15(.A(A[15]), .B(B[15]), .cin(c14), .sum(sum[15]), .cout(c15));
   fulladder FA16(.A(A[16]), .B(B[16]), .cin(c15), .sum(sum[16]), .cout(c16));
   fulladder FA17(.A(A[17]), .B(B[17]), .cin(c16), .sum(sum[17]), .cout(c17));
   fulladder FA18(.A(A[18]), .B(B[18]), .cin(c17), .sum(sum[18]), .cout(c18));
   fulladder FA19(.A(A[19]), .B(B[19]), .cin(c18), .sum(sum[19]), .cout(c19));
   fulladder FA20(.A(A[20]), .B(B[20]), .cin(c19), .sum(sum[20]), .cout(c20));
   fulladder FA21(.A(A[21]), .B(B[21]), .cin(c20), .sum(sum[21]), .cout(c21));
   fulladder FA22(.A(A[22]), .B(B[22]), .cin(c21), .sum(sum[22]), .cout(c22));
   fulladder FA23(.A(A[23]), .B(B[23]), .cin(c22), .sum(sum[23]), .cout(c23));
   fulladder FA24(.A(A[24]), .B(B[24]), .cin(c23), .sum(sum[24]), .cout(c24));
   fulladder FA25(.A(A[25]), .B(B[25]), .cin(c24), .sum(sum[25]), .cout(c25));
   fulladder FA26(.A(A[26]), .B(B[26]), .cin(c25), .sum(sum[26]), .cout(c26));
   fulladder FA27(.A(A[27]), .B(B[27]), .cin(c26), .sum(sum[27]), .cout(c27));
   fulladder FA28(.A(A[28]), .B(B[28]), .cin(c27), .sum(sum[28]), .cout(c28));
   fulladder FA29(.A(A[29]), .B(B[29]), .cin(c28), .sum(sum[29]), .cout(c29));
   fulladder FA30(.A(A[30]), .B(B[30]), .cin(c29), .sum(sum[30]), .cout(c30));
   fulladder FA31(.A(A[31]), .B(B[31]), .cin(c30), .sum(sum[31]), .cout(cout));

endmodule

//Tests fulladder32.
module fulladder32_testbench();

   logic [31:0] A, B, sum;
   logic cin, cout;

   fulladder32 dut (A, B, cin, sum, cout);


   integer i;
   initial begin


      A = 32'd15; B = 32'd16; cin = 1'b0; #10;
      A = 32'd15; B = 32'd16; cin = 1'b1; #10;
      A = 32'd15; B = 32'd163; cin = 1'b0; #10;
      A = 32'd15; B = 32'd216; cin = 1'b0; #10;
      A = 32'd15; B = 32'd146; cin = 1'b0; #10;

   end //initial
```

```

endmodule
```

```
//Aaron Hong (ahong02)
//Stephen Macris (smacris)
//EE469 Lab1

//Adds two 1-bit values and a carry-in value.

//Inputs: Three 1-bits A, B (inputs to be added), cin (carry-in value).
//Outputs: Two 1-bits sum (sum), cout (carry-out value).
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

//Tests module fulladder by simulating all 2^3 input bit combinations
module fulladder_testbench();

   logic A, B, cin, sum, cout;

   fulladder dut (A, B, cin, sum, cout);


   integer i;
   initial begin

      for(i=0; i<2**3; i++) begin
       {A, B, cin} = i; #10;
      end //for loop

   end //initial

endmodule

```