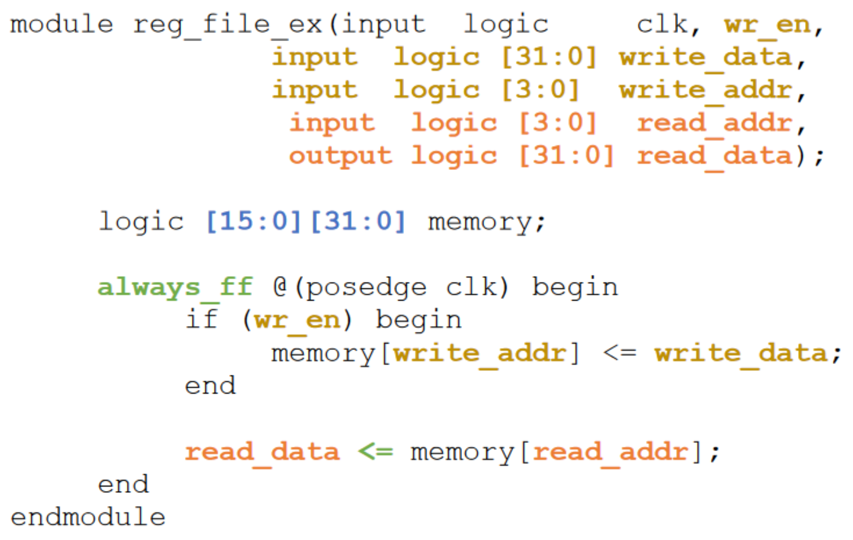
Eugene Ngo
EE 469
4/7/2023
Lab 1 Report

Procedure:

This lab contained 3 tasks. Task 1 was to refresh my digital design memory, task 2 was to design and simulate a register file, and task 3 was to design and simulate an ALU. Once the designs were implemented, they were simulated and tested in Modelsim to demonstrate their functionality.

Task #2:

The second task was to implement a register file, the rapid memory unit on a CPU. This was done by first utilizing the register file given in the lab pdf.



```
module reg_file_ex(input  logic     clk, wr_en,
                   input  logic [31:0] write_data,
                   input  logic [3:0]  write_addr,
                    input  logic [3:0]  read_addr,
                    output logic [31:0] read_data);

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

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

          read_data <= memory[read_addr];
     end
endmodule
```

Figure 2: The base-level register file provided by the lab pdf

This register file implementation was for a 16x32 register file that was synchronous and had one read port and one write port. The task specified a register file to be made that was also 16x32 with one write port, but with two read ports and asynchronous. To achieve this, the given register file had one more read signal added and the always_ff block was changed to always_comb for reading from the file, but the always_ff block was retained for writing to the file since the specification demanded thus.

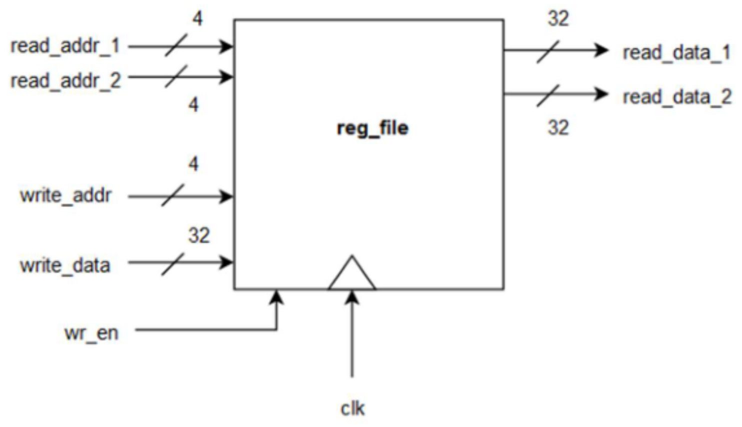


Figure 3: This is a block diagram for the overall register file taken from the lab specification

Task #3:

The third task was to implement an ALU unit from a CPU. This was done by first taking the specifications required by the lab and mapping them out to create a block diagram.

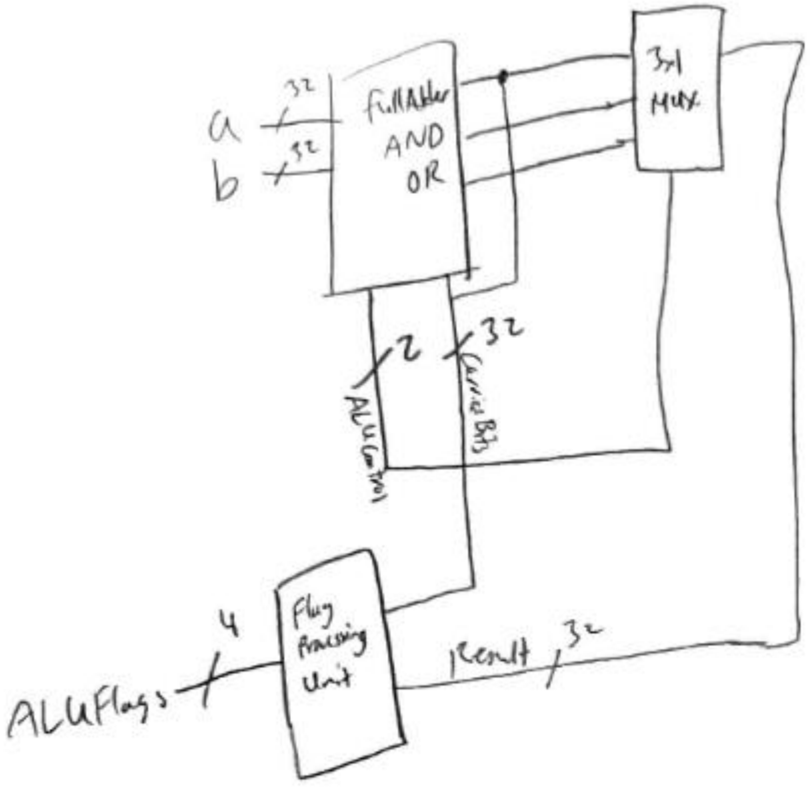
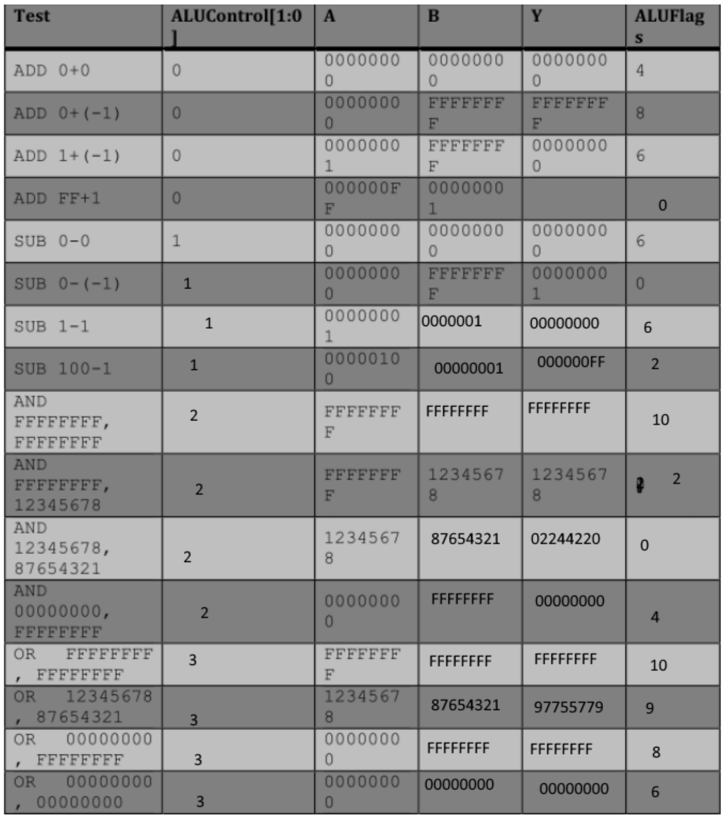


Figure 4: The overall block diagram for the ALU

A and B are fed into fulladders, and and or modules bit-by-bit. The outputs from each are then passed through to a 3x1 MUX and processed to output result. Outputs from the fullAdder are used to determine the carrier inputs in a feedback loop. The ALUControl signal is used as a mux control signal and then the Result combined with the carrier inputs is used to determine the ALUFlags signal.

Following the block diagram, the circuit was implemented in Quartus using System Verilog. It was then tested in Modelsim by varying a ,b, ALUControl, to see the change in Result and ALUFlags. Then a, b and ALUControl were varied using testvectors, see attached alu.tv screenshot in Appendix for details.



Test	ALUControl[1:0]	A	B	Y	ALUFlags
ADD 0+0	0	00000000	00000000	00000000	4
ADD 0+(-1)	0	00000000	FFFFFFFF	FFFFFFFF	8
ADD 1+(-1)	0	00000001	FFFFFFFF	00000000	6
ADD FF+1	0	000000FF	00000001		0
SUB 0-0	1	00000000	00000000	00000000	6
SUB 0-(-1)	1	00000000	FFFFFFFF	00000001	0
SUB 1-1	1	00000001	0000001	00000000	6
SUB 100-1	1	00000100	00000001	000000FF	2
AND FFFFFFFF, FFFFFFFF	2	FFFFFFFF	FFFFFFFF	FFFFFFFF	10
AND FFFFFFFF, 12345678	2	FFFFFFFF	12345678	12345678	[illegible] 2
AND 12345678, 87654321	2	12345678	87654321	02244220	0
AND 00000000, FFFFFFFF	2	00000000	FFFFFFFF	00000000	4
OR FFFFFFFF, FFFFFFFF	3	FFFFFFFF	FFFFFFFF	FFFFFFFF	10
OR 12345678, 87654321	3	12345678	87654321	97755779	9
OR 00000000, FFFFFFFF	3	00000000	FFFFFFFF	FFFFFFFF	8
OR 00000000, 00000000	3	00000000	00000000	00000000	6

Figure 5: The values that were varied for the test vector used and the resulting values

I used this table to verify that all four ALU operations work as intended and filled in the missing values.

Results

Task #2: Register File

After implementing the register file in Quartus, I ran ModelSim to test it.

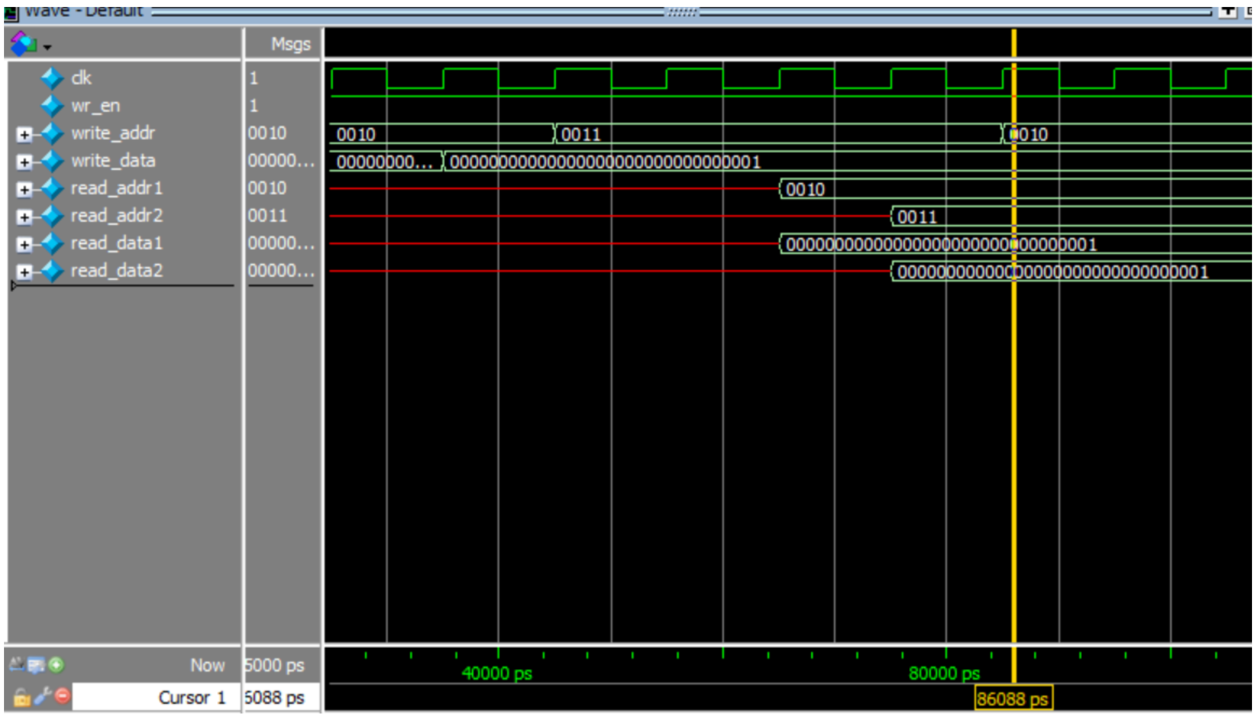


Figure 6: ModelSim waves for testing writes delays (1 cycle)

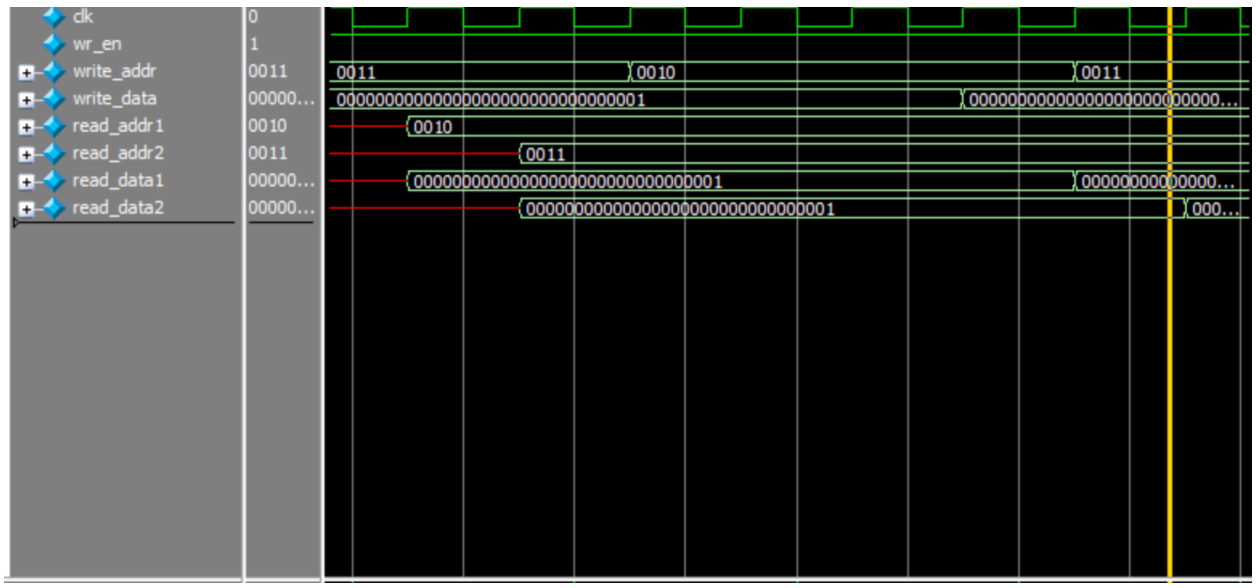


Figure 7: ModelSim waves for testing read delays (0 cycles)

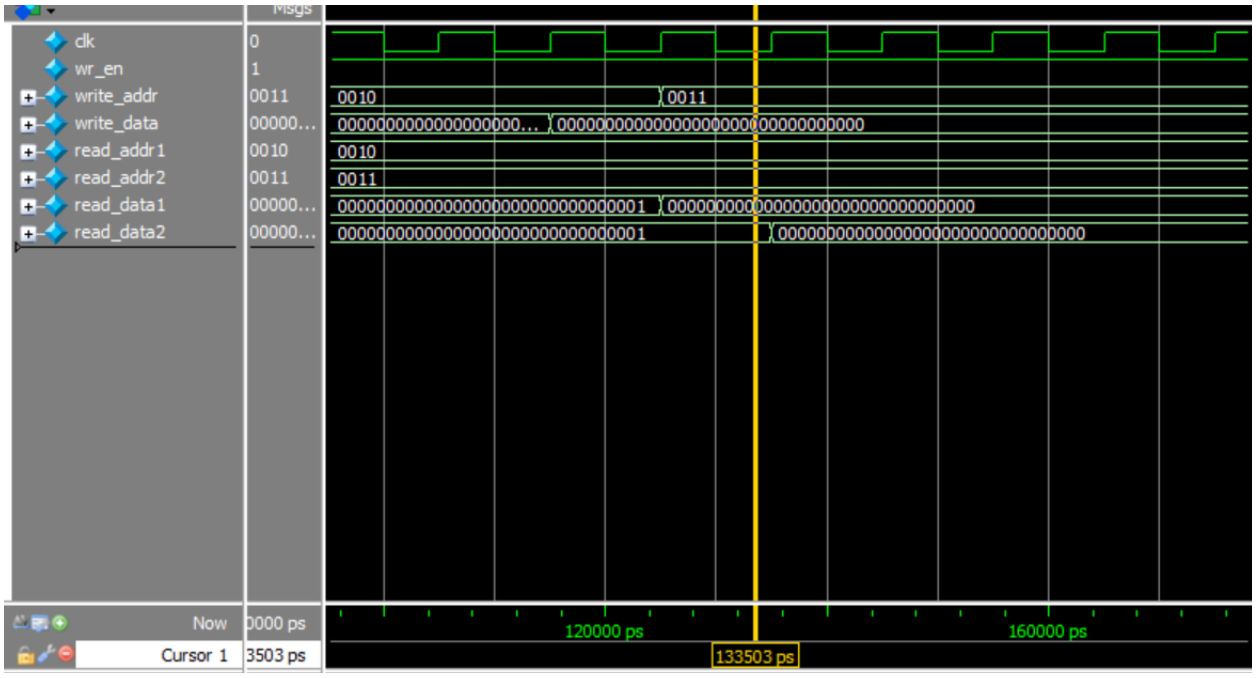


Figure 8: ModelSim waves for testing read update delays after a write (1 cycle after the write)

The three functionalities that are being tested are given by the lab specification:

1. Write data is written into the register file the clock cycle **after** wr_en is asserted.
2. Read data is updated to the register data at an address the **same** cycle the address was provided. Do this for both read addresses and data outputs.
3. 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 asserted. Do this for both read addresses and data outputs.

As seen in the waveforms above, data is written only 1 cycle after 'wr_en' is equal to 1. Read data pushes through in the very same cycle since it is done using combinational logic. Read data is updated 1 cycle after if a write is performed at the same address as the read.

Task #3: ALU

After implementing the ALU in Quartus, I ran ModelSim to test it.

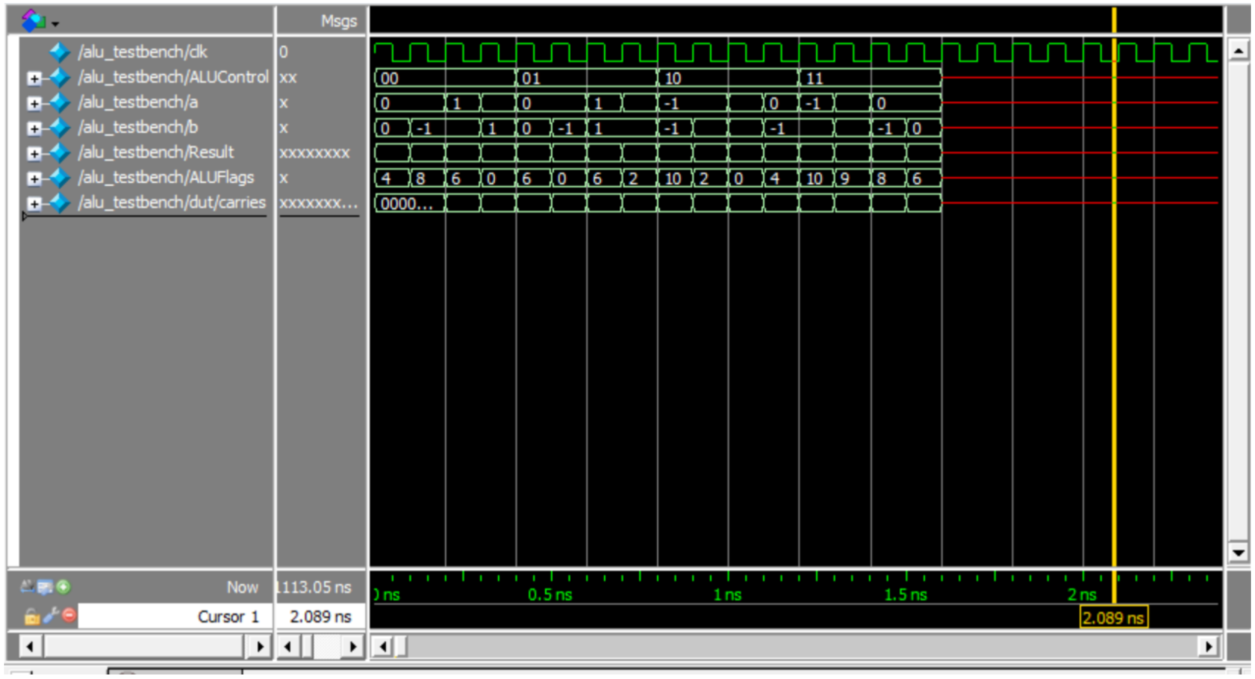


Figure 9: Waveform generated by running a test vector through the ALU

As seen from the figure, Result is the output of performing the function defined by ALUControl and the ALUFlags are updated based on the carriers within the ALU and the Result value.

Appendix

See the following list for the order:

Task 2: reg_file.sv

reg_file_testbench.sv

Task 3: alu.sv

alu_testbench.sv

singleALU.sv

fullAdder.sv

alu.tv

See the attached documents for the code:

```
// Eugene Ngo
// 4/7/2023
// CSE 469
// Lab 1 Task 2

// This is the top level module
// This register file is 16x32, has two read ports, one write port, and is asynchronous
// as specified
module reg_file (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);

   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

// reg_file__testbench tests all expected, unexpected, and edgecase behaviors
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, .wr_en, .write_data, .write_addr, .read_addr1, .read_addr2, .
read_data1, .read_data2);

   parameter clock_period = 10000;

   integer i;

   initial begin // Set up the clock
      clk <= 0;
      for (i=0; i<1000; i++) begin: clockCount
         forever #(clock_period /2) clk <= ~clk;
      end

   end

   initial begin
      $display("%t Behavior check", $time);

            // Testing functionality of
            // 1 cycle delay of writes.
      wr_en = 1'b1;                 @(posedge clk);
      write_data = 32'b0;           @(posedge clk);
      write_addr = 4'b0010;         @(posedge clk);
                                    @(posedge clk);

      write_data = 32'b1;           @(posedge clk);
      write_addr = 4'b0011;         @(posedge clk);
                                    @(posedge clk);
            // Testing functionality of
            // same cycle reads.
      read_addr1 = 4'b0010;         @(posedge clk);
      read_addr2 = 4'b0011;         @(posedge clk);

            // Testing the functionality of
            // 1 cycle delayed reads
            // after an updated write
      write_addr = 4'b0010;         @(posedge clk);
      read_addr1 = 4'b0010;         @(posedge clk);
                                    @(posedge clk);
```

```
        // read_data1 should update to 1 now.

        write_data = 32'b0;           @(posedge clk);
        write_addr = 4'b0011;         @(posedge clk);

        read_addr2 = 4'b0011;         @(posedge clk);
                                      @(posedge clk);
        // read_data2 should update to 0 now.




    end




endmodule

```

```systemverilog
// Eugene Ngo
// 4/7/2023
// CSE 469
// Lab 1 Task 2

// reg_file__testbench tests all expected, unexpected, and edgecase behaviors
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, .wr_en, .write_data, .write_addr, .read_addr1, .read_addr2, .
read_data1, .read_data2);

   parameter clock_period = 10000;

   integer i;

   initial begin // Set up the clock
      clk <= 0;
      for (i=0; i<1000; i++) begin: clockCount
         forever #(clock_period /2) clk <= ~clk;
      end

   end

   initial begin
      $display("%t Behavior check", $time);

            // Testing functionality of
            // 1 cycle delay of writes.
      wr_en = 1'b1;                  @(posedge clk);
      write_data = 32'b0;            @(posedge clk);
      write_addr = 4'b0010;          @(posedge clk);
                                     @(posedge clk);

      write_data = 32'b1;            @(posedge clk);
      write_addr = 4'b0011;          @(posedge clk);
                                     @(posedge clk);
            // Testing functionality of
            // same cycle reads.
      read_addr1 = 4'b0010;          @(posedge clk);
      read_addr2 = 4'b0011;          @(posedge clk);

            // Testing the functionality of
            // 1 cycle delayed reads
            // after an updated write
      write_addr = 4'b0010;          @(posedge clk);
      read_addr1 = 4'b0010;          @(posedge clk);
                                     @(posedge clk);
      // read_data1 should update to 1 now.

      write_data = 32'b0;            @(posedge clk);
      write_addr = 4'b0011;          @(posedge clk);

      read_addr2 = 4'b0011;          @(posedge clk);
                                     @(posedge clk);
      // read_data2 should update to 0 now.




   end




endmodule

```

```
// Eugene Ngo
// 4/7/2023
// CSE 469
// Lab 1 Task 3

// This is the top level module
// alu is the module used to implement a 32 bit ARM-based ALU which
// contains the central logic for calling on submodules that make up the ALU.
module alu(input logic [31:0] a, b,
           input logic [1:0] ALUControl,
           output logic [31:0] Result,
           output logic [3:0] ALUFlags);
   // 00 = add
   // 01 = subtract
   // 10 = AND
   // 11 = OR

   logic [31:0] carries;

   singleALU setCarries (.a(a[0]), .b(b[0]), .carryIn(ALUControl[0]),
                         .ALUControl(ALUControl), .Result(Result[0]),
                         .carryOut(carries[0]));

   genvar i;
   generate
      for (i = 1; i < 32; i++) begin: ALUPipeline
         singleALU results (.a(a[i]), .b(b[i]), .carryIn(carries[i - 1]),
                            .ALUControl(ALUControl), .Result(Result[i]),
                            .carryOut(carries[i]));
      end // end loop
   endgenerate // end generate

   // Setting flags:
   xor overFlowCheck (ALUFlags[0], carries[31], carries[30]);
   assign ALUFlags[1] = carries[31];

   // Inefficient and bad style. RTL would be better.
   nor zeroChecker
      (ALUFlags[2], Result[31], Result[30], Result[29], Result[28],
       Result[27], Result[26], Result[25], Result[24], Result[23],
       Result[22], Result[21], Result[20], Result[19], Result[18],
       Result[17], Result[16], Result[15], Result[14], Result[13],
       Result[12], Result[11], Result[10], Result[9], Result[8],
       Result[7], Result[6], Result[5], Result[4], Result[3],
       Result[2], Result[1], Result[0]);

   assign ALUFlags[3] = Result[31];

endmodule

// alu_testbench tests all expected, unexpected, and edgecase behaviors
module alu_testbench();
   logic [31:0] a,b;
   logic [1:0] ALUControl;
   logic [31:0] Result;
   logic [3:0] ALUFlags;
   logic clk;
   logic [103:0] testvectors [1000:0];

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

   parameter CLOCK_PERIOD = 100;

   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 loop
   end // end initial
endmodule

```

```
// Eugene Ngo
// 4/7/2023
// CSE 469
// Lab 1 Task 3

// alu_testbench tests all expected, unexpected, and edgecase behaviors
module alu_testbench();
   logic [31:0] a,b;
   logic [1:0] ALUControl;
   logic [31:0] Result;
   logic [3:0] ALUFlags;
   logic clk;
   logic [103:0] testvectors [1000:0];

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

   parameter CLOCK_PERIOD = 100;

   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 loop
   end // end initial
endmodule
```

```
// Eugene Ngo
// 4/7/2023
// CSE 469
// Lab 1 Task 3

// singleALU implements ALU logic for single bits which can be combined to
// make up large ALUs.
module singleALU (a, b, carryIn, ALUControl, Result, carryOut);
   input logic a, b, carryIn;
   input logic [1:0] ALUControl;
   output logic Result, carryOut;


   logic [2:0] outputs;

   and andValue (outputs[1], b, a);
   or  orValue  (outputs[2], b, a);

   // Selecting B (Addition = A+B+0, Subtraction = A+(~B)+1)
   // logic subtractSelector = ALUControl[0];

   // MUX to select B (ALUControl[0] == 1 = select ~B)
   wire middleValues[1:0];
   and selectB (middleValues[0], ~ALUControl[0], b);
   and selectNotB (middleValues[1], ALUControl[0], ~b);

   logic selectedB;
   or selectedBValue (selectedB, middleValues[0], middleValues[1]);

   fullAdder fullAddedValue (.a(a), .b(selectedB), .carryIn(carryIn),
                             .Result(outputs[0]), .carryOut(carryOut));

   // MUX to select between computed values (add, sub, and, or)

   always_comb begin
      case (ALUControl)
         2'b00: Result = outputs[0];
         2'b01: Result = outputs[0];
         2'b10: Result = outputs[1];
         2'b11: Result = outputs[2];
         default: Result = 1'bX;
      endcase // end case statements
   end // end comb block
   // End of module

endmodule

```

```
// Eugene Ngo
// 4/7/2023
// CSE 469
// Lab 1 Task 3

// A module to implement a fullAdder.
// This file was reused from a given file in EE 371
module fullAdder (a, b, carryIn, Result, carryOut);

   input  logic a, b, carryIn;
   output logic Result, carryOut;

   assign  Result = a ^ b ^ carryIn;
   assign carryOut = (a & b) | (carryIn & (a ^ b));

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_0
1_00000001_00000001_00000000_6
1_00000100_00000001_000000FF_2
2_FFFFFFFF_FFFFFFFF_FFFFFFFF_8
2_FFFFFFFF_12345678_12345678_2
2_12345678_87654321_02244220_0
2_00000000_FFFFFFFF_00000000_4
3_FFFFFFFF_FFFFFFFF_FFFFFFFF_8
3_12345678_87654321_97755779_9
3_00000000_FFFFFFFF_FFFFFFFF_8
3_00000000_00000000_00000000_6
```