Junchao Zhou, Chenhan Dai
EE469
April 5, 2023
Lab 1 Report

Procedure

This lab consisted of three tasks: design a simple module using Quartus, and do the same thing for a register file and design an Arithmetic Logic Unit ALU. Then simulate both of these approaches on ModelSim.

Task #1: Warmup Exercise on ModelSim

The first task is to install the Quartus 17.0 and ModelSim, and then apply a simulation on full adder.

Task #2: Register File

In this task, we built a 16 by 32 register file with two read ports, one write port and asynchronous. Which means there will be 16 addresses, each address storing 32 bits value. And the read port will reflect the value of the address without the influence of clock.

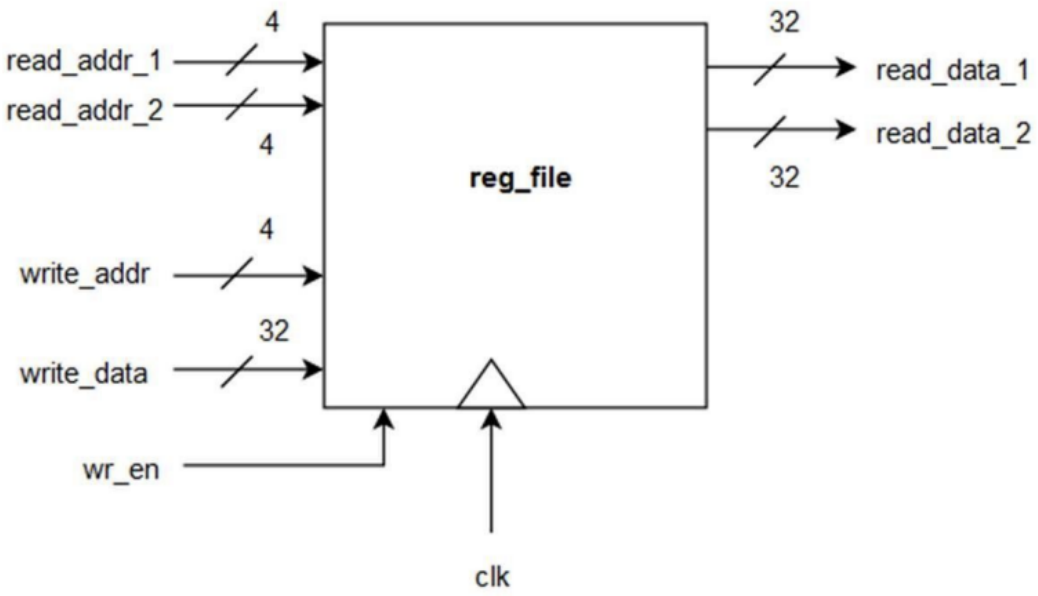

Figure 1: 16 by 32 register with two read port, one write port and asynchronous

Task #3: ALU

This task asks us to build an ALU (Arithmetic Logic Unit) which computes any of four operations: addition, subtraction, ANDing and ORing. It computes every operation and selects one result based on the control signals given in Figure 2. There are other 4-bit flags to see which conditions the result meets in Figure 3. The schematic diagram of ALU with flags is given in Figure 4.

ALU: Arithmetic Logic Unit

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

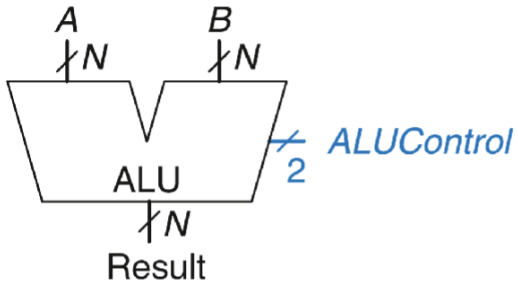

Figure 2: ALUControl and function

The four bits of ALUFlags should be TRUE if a condition is met. The four flags are as follows:

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

Figure 3: ALU flags bit and meaning

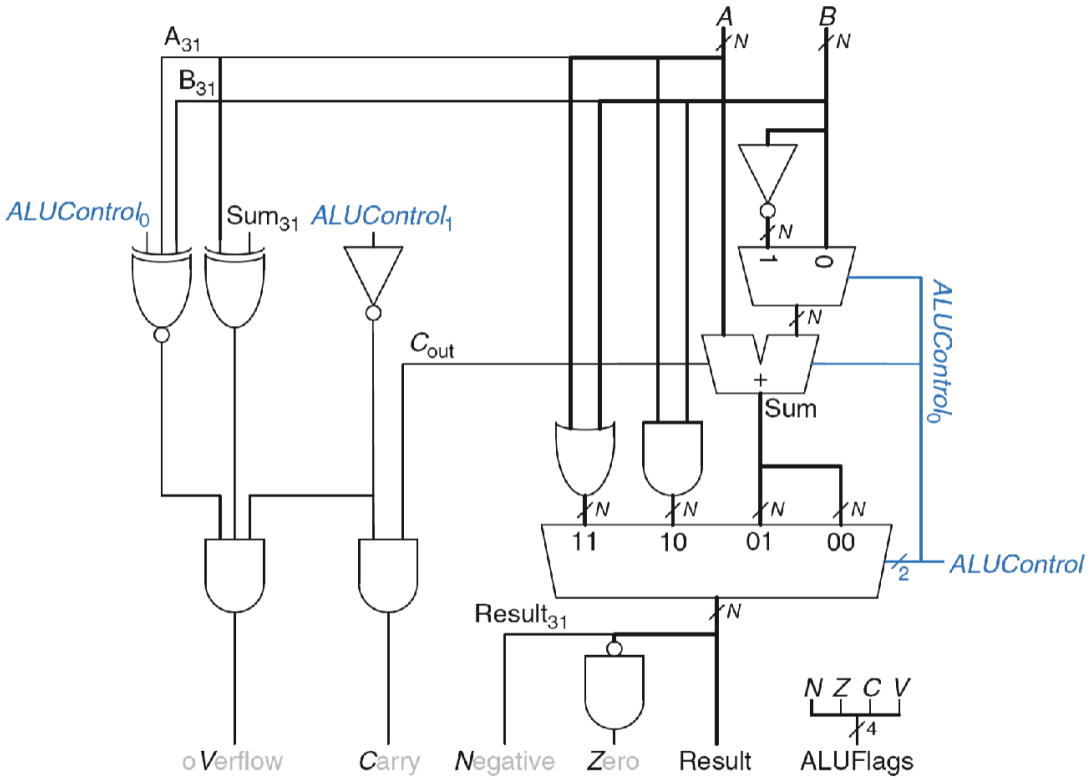

Figure 4: A Schematic Diagram Arithmetic Logic Unit with Status Flags

Result

Task 1:

Finished

Task 2:

After creating the 16 by 32 register file in SystemVerilog, which is 16 *32, two read ports, one write port, and asynchronous. We ran ModelSim to produce the waveform seen in Figure 2.

For the waveform, in the first 20ps, the wr-en is 0, so the write-data is written into the register file the clock cycle after wr-en is asserted (in the 30ps). In addition, the read data is updated to the register data the same cycle the address was provided. Finally, the read data is updated to write data at an address the cycle after the address was provided.

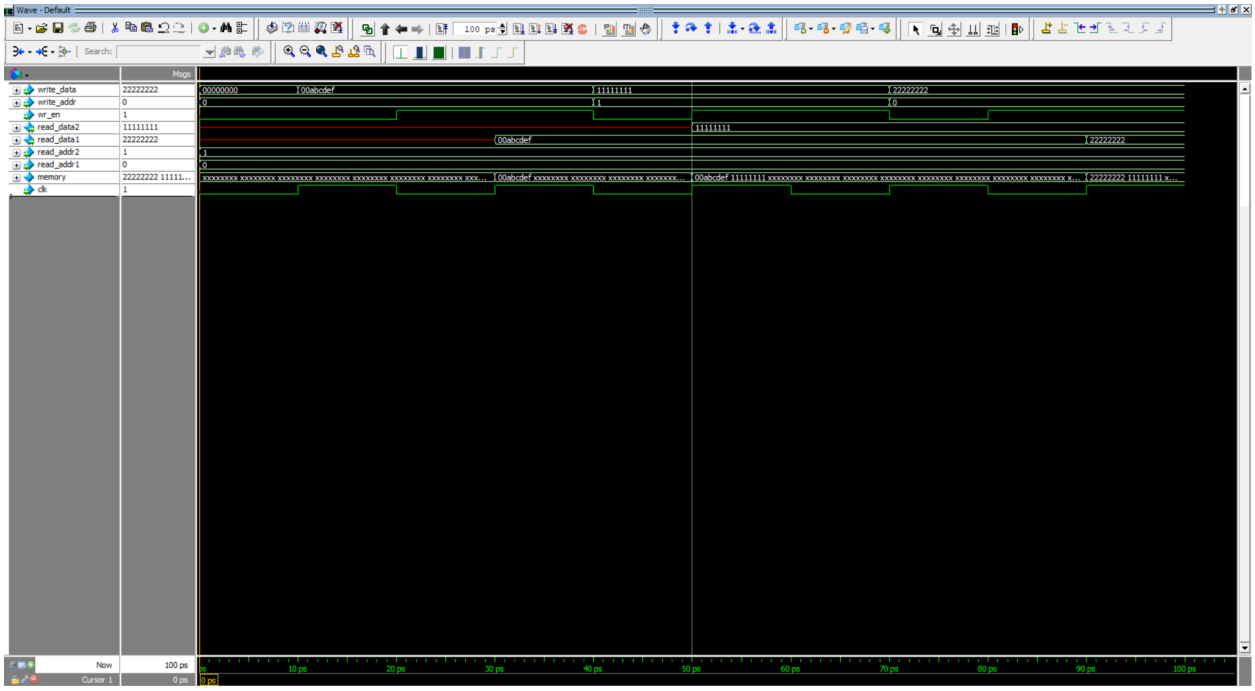

Figure 5: Register file simulation diagram

Task 3:

According to the schematic diagram above, we built the ALU in Verilog. The testbench of ALU is based on the test listed in table 1. The simulation graphs are shown in figure 6 and fill the result from the figure into table 1.

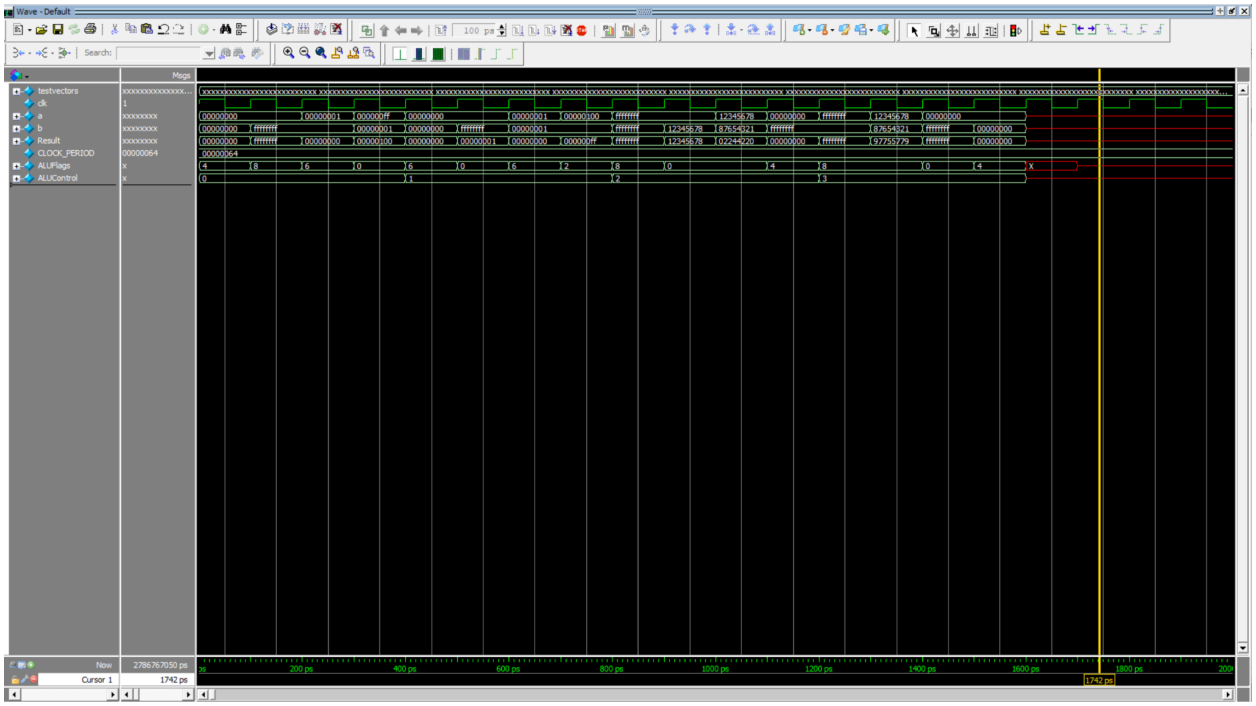

Figure 6: ALU simulation diagram

Test	ALUControl[1:0]	A	B	Result	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	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**	**0**
OR 00000000, 00000000	**3**	**00000000**	**00000000**	**00000000**	**4**

Table 1. ALU operation

Appendix: SystemVerilog code

1) reg_file.sv

```
1  //Junchao Zhou, Chenhan Dai
2  //04/05/2023
3  //EE469
4  //Lab #1, Task2
5
6  //reg_file takes clk, wr_en, 32_bit write_data, 4-bit write_addr, read_addr1, read_addr2 as inputs,
7  // 32-bit read_data1, read_data2 as outputs. And we define a 16 * 32 bit memory.
8  // If write enable, the write data is be written into the write_address of the memory,
9  // and we read the data in read_addr1 and read_addr2 of the memory asynchronously.
10
11 module reg_file(input logic clk, wr_en,
12                  input logic [31:0] write_data,
13                  input logic [3:0] write_addr,
14                  input logic [3:0] read_addr1, read_addr2,
15                  output logic [31:0] read_data1, read_data2);
16
17       //logic [15:0][31:0] memory;
18       logic [31:0] memory [0:15];
19
20
21       // Write port
22       always_ff @(posedge clk) begin
23          if (wr_en)
24             memory[write_addr] <= write_data;
25       end
26
27       // Read Port 1
28       assign read_data1 = memory[read_addr1];
29
30       // Read Port 2
31       assign read_data2 = memory[read_addr2];
32 endmodule
33
34
35
36
```

```
36
37  // reg_file_testbench tests three cases
38  // 1. the write data is written into the register file the clock cycle after wr_en is asserted
39  // 2. Read data is updated to the register at an address the same cycle the address was
40  // provided
41  // 3. Read data is updated to write data at an address the cycle after the address was provided
42  // if the write address is the same and wr_en was asserted
43
44  module reg_file_testbench();
45     //Inputs
46     logic clk, wr_en;
47     logic [31:0] write_data;
48     logic [3:0] write_addr, read_addr1, read_addr2;
49
50     //Outputs
51     logic [31:0] read_data1, read_data2;
52
53     reg_file dut(.clk, .wr_en, .write_data, .write_addr,
54                  .read_addr1, .read_addr2, .read_data1, .read_data2);
55
56     always #10 clk = ~clk;
57
58     // Initialize inputs
59     initial begin
60        clk = 0;
61        wr_en = 0;
62        write_data = 0;
63        write_addr = 0;
64        read_addr1 = 0;
65        read_addr2 = 1;
66        #10;
67
68        // Wite data is written the cycle after wr-en is asserted
69        wr_en = 0;
70        write_addr = 0;
71        write_data = 32'h00abcdef;
72        #10;
73
74        wr_en = 1; #20;
75
76        // Read data is updated the same cycle the address is provided
77        wr_en = 0;
78        write_data = 32'h11111111;
79        write_addr = 1;
80        #10;
81
82        wr_en = 1;
83        #10;
84
85
86        read_addr1 = 0;
87        read_addr2 = 1;
88        #10;
89
```

```
90
91        // Read data is updated the cycle after the address is provided and
92        // wr_en is asserted
93        write_data =32'h22222222;
94        write_addr =0;
95        wr_en = 0;
96        #10;
97
98        wr_en = 1;
99        #10;
100
101       read_addr1= 0;
102       read_addr2 =1;
103       #10;
104       $stop;
105    end
106
107 endmodule
```

2) alu.sv

```
1  //Junchao Zhou, Chenhan Dai
2  //04/05/2023
3  //EE469
4  //Lab #1, Task2
5
6  // alu take 32-bit a, b, 2-bit ALUControl as input and return 32-bit Result and 4-bit ALUFlags as
7  // outputs. We define a 32-bit n-b as the inverse of b and a 33-bit temp to find if the ALU has
8  // carryout.
9  module alu(input logic[31:0]a, b,
10            input logic[1:0] ALUControl,
11            output logic[31:0] Result,
12            output logic[3:0] ALUFlags);
13
14            logic[31:0] n_b;
15            logic[32:0] temp;
16            assign n_b = ~b;
17     always_comb begin
18         case(ALUControl)
19            2'b00: begin
20            Result = a + b;
21            temp = a + b;
22            end
23
24            2'b01: begin
25            Result = a - b;
26            temp = a + n_b + 1;
27            end
28
29            2'b10: begin
30            Result = a & b;
31            temp = 0;
32            end
33
34            2'b11: begin
35            Result = a | b;
36            temp = 0;
37            end
38         endcase
39
40         // ALUFlags[0] = 1 when the adder results in overflow
41         ALUFlags[0] = ~(a[31] ^ b[31] ^ ALUControl[0]) & (a[31] ^ Result[31]) & ~ALUControl[1];
42         //ALUFlags[0] = (~Result[31] & a[31] & ~b[31])|(Result[31]&~a[31] &~b[31]);
43
44         // ALUGFlag[1] = 1 when the adder  produces a carry out
45         ALUFlags[1] = temp[32];
46
47         // ALUFlag[2] = 1 when the result is 0
48         ALUFlags[2] = Result == 0;
49
50         // ALUFlag[3] = 1 when the result is negative
51         ALUFlags[3] = Result[31];
52      end
53 endmodule
```

```
54
55 // alu_testbench read the vector file alu.tv and tests all the cases in the file
56 module alu_testbench();
57    logic [31:0]a, b;
58    logic [1:0] ALUControl;
59    logic [31:0] Result;
60    logic [3:0] ALUFlags;
61    logic clk;
62    logic [103:0] testvectors [1000:0];
63
64    alu dut (.a(a), .b(b), .ALUControl(ALUControl), .Result(Result), .ALUFlags(ALUF
65
66    parameter CLOCK_PERIOD = 100;
67
68    initial clk = 1;
69    always begin
70          #(CLOCK_PERIOD/2);
71          clk = ~clk;
72
73    end
74
75    initial begin
76          $readmemh("alu.tv", testvectors);
77
78          for(int i = 0; i < 20; i = i + 1) begin
79             {ALUControl, a, b, Result, ALUFlags} = testvectors[i]; @(posedge clk);
80          end
81
82       end
83 endmodule
84
85
```

3)alu.tv

```
 1    0_00000000_00000000_00000000_4
 2    0_00000000_FFFFFFFF_FFFFFFFF_8
 3    0_00000001_FFFFFFFF_00000000_6
 4    0_000000FF_00000001_00000100_0
 5    1_00000000_00000000_00000000_6
 6    1_00000000_FFFFFFFF_00000001_0
 7    1_00000001_00000001_00000000_6
 8    1_00000100_00000001_000000FF_2
 9    2_FFFFFFFF_FFFFFFFF_FFFFFFFF_8
10    2_FFFFFFFF_12345678_12345678_0
11    2_12345678_87654321_02244220_0
12    2_00000000_FFFFFFFF_00000000_4
13    3_FFFFFFFF_FFFFFFFF_FFFFFFFF_8
14    3_12345678_87654321_97755779_8
15    3_00000000_FFFFFFFF_00000000_0
16    3_00000000_00000000_00000000_4
```