Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

ENCS3310

Advanced Digital Design

Course Project

Prepared by:

Name: Ali Shaikh Qasem ID: 1212171

Instructor: Dr. Abdallatif Abuissa.

Section: 1.

first-semester

2023/2024

Abstract

This project aims to build a simple model of a microprocessor. specifically, to build an ALU and a simple register file, then connect them together and manage the synchronization between them. In addition, to simulate a simple machine code program and test the design.

Contents

Table of Figures

Theory

Arithmetic Logic Unit (ALU)

Alu is a combinational circuit that performs arithmetic operations like addition and subtraction, and also a logical bitwise-operations like bitwise and. It has two inputs represents the operands and a selection input that determines the operation to perform and a result output.

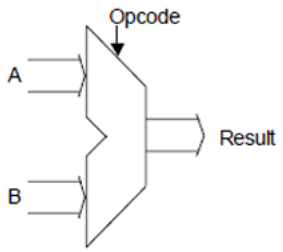

Figure 1: ALU schematic

Register file

Register file is a very small amount of memory inside the processor. It's a clocked circuit used to hold the operands that the processor is currently working on. It has three addresses inputs, two of them represents the address of the required values, and the other represents the address of which the input value will be stored.

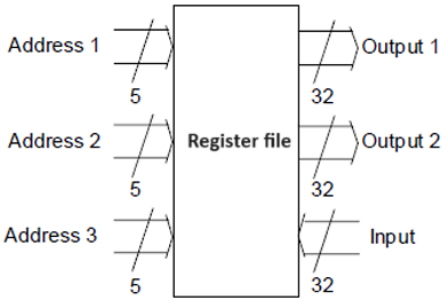

Figure 2: Register file schematic

Microprocessor

Microprocessor is a simple CPU that runs a program by executing the instructions. It consists of the ALU and register file, it uses the register file to fetch the operands and store the results, and it uses the ALU to perform the operation on the operands.

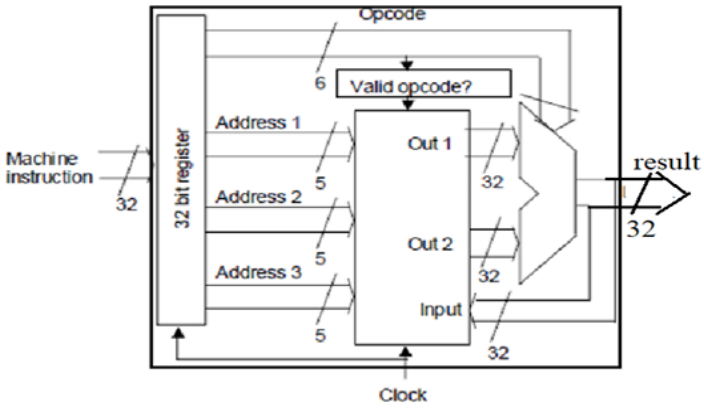

Figure 3: Microprocessor schematic

Procedure & Discussion

ALU design

- The Verilog description of the ALU:

```
module alu (opcode, a, b, result );

    input [5:0]  opcode;
    input signed [31:0] a, b;
    output reg signed [31:0] result;

    always @(*)
        begin
            case (opcode)
                6'b000110 : result = a + b; //a+b
                6'b001000 : result = a - b; //a-b
                6'b001010 : begin           //|a|
                              if(a >= 0)
                                  result = a;
                              else
                                  result = -a;
                            end

                6'b001100 : result = -a; //-a
                6'b001110 : begin       //max(a,b)
                              if(a > b)
                                  result = a;
                              else
                                  result = b;
                            end
                6'b001011 : begin       //min(a,b)
                              if(a < b)
                                  result = a;
                              else
                                  result = b;
                            end

                6'b001101 : result = (a + b) / 2; //avg(a,b)
                6'b001111 : result = ~a;  //not a
                6'b000010 : result = a | b; // a or b
                6'b000011 : result = a & b; // a and b
                6'b001001 : result = a ^ b; // a xor b
                default: result = 32'bx;
            endcase
        end
endmodule
```

Figure 4: Verilog code for the ALU

- As seen above, we have used the behavioral modeling. The operation is determined using case statement based on the values of opcodes that are related to my id.

- Notice that We have used a signed numbers representation for the operands and results, this is to avoid wrong results especially when dealing with negative numbers and results.

- It's also noticed that in the case the opcode is not valid, the result is assigned to a don't care values.

Register file design

- The Verilog description of the register file:

```
79  module reg_file (clk, valid_opcode, addr1, addr2, addr3, in , out1, out2);
80
81      //defining inputs and outputs
82      input clk;
83      input valid_opcode;
84      input [4:0] addr1, addr2, addr3;
85      input [31:0] in;
86      output reg [31:0] out1, out2;
87      //defininining the register file slots
88      reg [31:0] reg_slots [0:31];
89
90      //initializing the values in the register file
91      initial
92          begin
93              reg_slots [0] = 32'h0; reg_slots [1] = 32'h3ABA;  reg_slots [2] = 32'h2296;  reg_slots [3] = 32'hAA;
94              reg_slots [4] = 32'h1C3A; reg_slots [5] = 32'h1180; reg_slots [6] = 32'h22E0;  reg_slots [7] = 32'h1C86;
95              reg_slots [8] = 32'h22DA; reg_slots [9] = 32'h414; reg_slots [10] = 32'h1A32; reg_slots [11] = 32'h102;
96              reg_slots [12] = 32'h1CBA; reg_slots [13] = 32'hCDE; reg_slots [14] = 32'h3994; reg_slots [15] = 32'h1984;
97              reg_slots [16] = 32'h28C4; reg_slots [17] = 32'h2E7C; reg_slots [18] = 32'h3966; reg_slots [19] = 32'h227E;
98              reg_slots [20] = 32'h2208; reg_slots [21] = 32'h11B4; reg_slots [22] = 32'h237C; reg_slots [23] = 32'h360E;
99              reg_slots [24] = 32'h2722; reg_slots [25] = 32'h500; reg_slots [26] = 32'h16B6; reg_slots [27] = 32'h29E;
100             reg_slots [28] = 32'h2280; reg_slots [29] = 32'h3B52; reg_slots [30] = 32'h11A0; reg_slots [31] = 32'h0;
101         end
102
103     always @ (posedge clk)
104         begin
105             if(valid_opcode)
106                 begin
107                     out1 <= reg_slots [addr1];
108                     out2 <= reg_slots [addr2];
109                     reg_slots [addr3] <= in;
110                 end
111         end
112 endmodule
```

Figure 5: register file code

From the code shown above, we notice the following points:

- The initial values of the register slots are set at the initial block based on the values related to my id.

- An input called “valid opcode” is added to the circuit to work as an enable, when the “valid opcode” bit is high, the register works normally. but when it goes to 0, the register doesn’t work. This ensures that the register avoids undesired (garbage) input values.

- The register file is connected to the clock with a positive edge, this is to avoid over-writing the register values especially when the input and output addresses are the same, and also to synchronize the circuit with the whole system.

- The statements inside the always block are designed in a way that organizes the behavior of the register, we notice that the values of the two outputs (out1 and out2) is brought firstly, and then the input value is stored in the slot addressed by the third address(addr3), we also notice that we have used the non-blocking assignment, thus, in the case the input and output addresses are the same, the value of the output will be fitched before the new value is stored.

Microprocessor design

- The Verilog description of the microprocessor:

```
module mp_top (clk, instruction , result );

    input clk;
    input [31:0] instruction;
    output signed [31:0] result;
    wire valid_opcode_result;
    wire [31:0] out1,out2;

    //defining the registers that hold the instruction and the opcode
    reg [31:0] reg_32_out;
    reg [5:0] opcode;

    //clocking the instruction register and opcode, this is to ensure the opcode reaches the alu with the register values.
    always @(posedge clk)
        begin
          reg_32_out <= instruction;
          opcode <= reg_32_out [5:0];
        end

    // instantiate the required modules
    valid_opcode v1 ( reg_32_out [5:0], valid_opcode_result);
    reg_file rg1 (clk, valid_opcode_result, reg_32_out [10:6] , reg_32_out [15:11], reg_32_out [20:16], result , out1, out2);
    alu a1 (opcode, out1, out2, result );

    task valid_opcode ; // defining a task to check if the provided opcode is valid or not
       input opcode;
       output reg valid;

       //checking if the opcode is valid based on my id corresponding opcodes
       if(opcode==6 || opcode== 8 ||opcode==10||opcode==12 || opcode==14 || opcode==11 || opcode==13 || opcode==15 || opcode==2 || opcode==3 || opcode== 9)
                valid = 1;
            else
                valid = 0;
    endtask

endmodule
```

Figure 6: microprocessor code

- As seen above, the microprocessor is designed using structural modeling, so that it's possible to instantiate the ALU and register files implemented previously.

- It's noticed that we have also defined a 32-bit register called "reg_32_out" to store the instruction, so that the instruction will be available in the microprocessor after a clock cycle.

- We have also defined a 6-bit register to store the opcode which is the first 6 bits of the instruction (instruction[5:0]) and we connected it to the clock, this is to apply a one cycle delay before the opcode reaches the ALU, notice that we have used the non-blocking assignment inside always block, so that the target opcode value will reach the ALU exactly after two cycles at the same moment the register value arrives.

- The overall design shows that the instruction will be performed in a total of two clock cycles, one for fetching the instruction and the other one for fetching the operands from the register file, while the ALU will produce the result at the same moment it's inputs changes. Additionally, it's crucial to notice that storing the result in the destination register will also spend a clock cycle, so we need to provide an additional cycle in the test

to ensure the result has been stored or we use a no opcode instruction that instructs the CPU to do nothing while the value is stored.

- We have also added a task called “valid opcode” to test whether the provided opcode in the instruction is valid or not, the validity here depends on the opcodes related to my id. If the opcode is not valid then, the task will return 0 , thus, the register file will not work, otherwise, everything works normally.

Microprocessor testing

- In this part, we are going to create a program consists of multiple instructions, and execute them using the designed microprocessor.

- The table below describes the created instructions:

Instruction	opcode	Src1	Src2	Destination	Description
0X142006	6	R[0]	R[4]	R[20]	Add R[0] to R[4] and store the result in R[20].
0X1508C8	8	R[3]	R[1]	R[21]	Subtract R[3] - R[1] and store the result in R[21].
0X16054A	10	R[21]	x	R[22]	Get the absolute value of R[21] and store it in R[22].
0X17014C	12	R[5]	x	R[23]	Get the value of -R[5] and store it in R[23].
0X18398E	14	R[6]	R[7]	R[24]	Get the maximum value of R[6] and R[7] and store it in R[24].
0X192B4B	11	R[13]	R[5]	R[25]	Get the minimum value of R[13] and R[5] and store it in R[25].
0X1A7B8D	13	R[14]	R[15]	R[26]	Get the average value of R[14] and R[15] and store it in R[26]
0X1B044F	15	R[17]	x	R[27]	Get the value of not R[17] and store it in R[27]
0X281842	2	R[1]	R[3]	R[28]	Get the bitwise or of R[1] and R[3] and store it in R[28].
0X1D4B03	3	R[12]	R[9]	R[29]	Get the bitwise and of R[12] and R[9] and store it in R[29].
0X1E9089	9	R[2]	R[18]	R[30]	Get the bitwise xor of R[2] and R[18] and store it in R[30].

- The test bench code is shown below:

```
module mp_test ();

    reg [31:0] instruction;
    reg clk;
    wire signed [31:0] result;
    // definig the array of instructions, it includes 11 instructions, that means an instruction for every opcode.
    reg [31:0] instructions [0:10]  = '{32'h00142006, 32'h001508C8, 32'h0016054A, 32'h0017014C, 32'h0018398E, 32'h00192B4B,
    32'h001A7B8D, 32'h001B044F, 32'h00281842, 32'h001D4B03, 32'h001E9089};
    // defining the array pf expected values for the written instructions
    reg [31:0] expected_values [0:10] = '{32'h1C3A, 32'hFFFFC5F0, 32'h3A10, 32'hFFFFEE80, 32'h22E0, 32'h0CDE, 32'h298C, 32'hFFFFD183,
    32'h3ABA, 32'h0410, 32'h1BF0};

    int flag = 0;  // 0 means pass and 1 means fail.


    mp_top m1 (clk, instruction , result );

    initial
        begin
            $display("   Instruction        Result       status ");
            $display("   -------------------------------------------");

            clk = 0;
            instruction = instructions[0];
            #20ns // to execute the first instruction
            if(result == expected_values[0])
                $display("    0x%0h           %0d          pass ", instruction, result);
            else
                begin
                    $display(" 0x%0h         %0d          fail ", instruction, result);
                    flag = 1;
                end
```

Figure 7: microprocessor test part1

```
            for(int i=1 ; i<=10 ; i=i+1)
                begin
                    instruction = instructions[i];
                    #35ns; // 3 cycles, two for executing the instruction and one for storing the result in the destination
                    if(result == expected_values[i])
                       $display("    0x%0h           %0d          pass ", instruction, result);
                    else
                        begin
                            $display("    0x%0h           %0d          fail ", instruction, result);
                            flag = 1;
                        end

                end
            // determine if the test fails or passes
            if(flag == 1)
                $display("The test fails");
            else
                $display("The test passes");

        end


     always #5ns
         clk = ~clk;

endmodule
```

Figure 8: microprocessor test part2

- As shown above, both the instructions and the expected values are stored in an array of registers, to be compared with the calculated values.

- We have defined a flag variable called “flag” to check whether the test passes or fails, as long as the flag equals 0 the test passes, but once it reaches 1 the test fails.

- At the beginning of the initial block, the clock is initialized to zero and the instruction takes the value of the first instruction in the program. The clock cycle is set to 10ns also.

Notice that we have a 20ns delay after the first instruction, first 5 seconds for waiting the first positive edge to occur, and the remaining 15ns to execute the instruction.

- We have defined a loop to execute the remaining 10 instructions. Notice that each instruction must wait 35ns which is 3 cycles to ensure the instruction completes the execution and the result is stored in the destination register before going to the next instruction.

- The console result of the test bench is shown below:

```
run 700 ns
# KERNEL:    Instruction        Result       status
# KERNEL:    ----------------------------------------
# KERNEL:    0x142006           7226          pass
# KERNEL:    0x1508c8           -14864          pass
# KERNEL:    0x16054a           14864          pass
# KERNEL:    0x17014c           -4480          pass
# KERNEL:    0x18398e           8928          pass
# KERNEL:    0x192b4b           3294          pass
# KERNEL:    0x1a7b8d           10636          pass
# KERNEL:    0x1b044f           -11901          pass
# KERNEL:    0x281842           15034          pass
# KERNEL:    0x1d4b03           1040          pass
# KERNEL:    0x1e9089           7152          pass
# KERNEL: The test passes
# KERNEL: stopped at time: 700 ns
```

Figure 9: microprocessor test results

- As results above indicates that all calculated results are equivalent to expected values, thus, the test passes.

- Notice that the third instruction gets the absolute value of R[21] which has been modified in the second instruction, the test shows that it actually gets the absolute value of the previous result which is (-14864), so that we ensure the process of storing results in the register file has been done successfully.

Note: see appendix 1 to view the simulation results for each instruction.

Conclusion

In this project, we have used the knowledge of digital systems design and hardware description language to create a simple meaningful microprocessor system. The design process has started by implementing the individual parts such as ALU and register file based on their behavior, then, the different modules has been connected together forming a perfect microprocessor system. One of the main issues of the design process was the challenge of synchronizing the individual parts, and organizing their timing based on a single clock. Actually, we have realized that the timing and synchronization considerations have a crucial effect on the circuit behavior, so the design procedure must deal with them seriously. Since the design verification is one of the most important steps in the design procedure, we have created a complete test bench represents a simple program that performs every opcode operation. From the test results and simulation, we can conclude that our design works fine.

Appendix 1

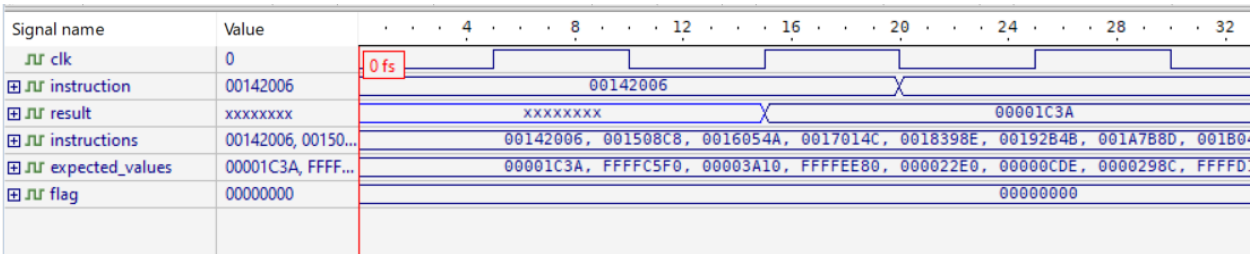

Figure 10: instruction 1 simulation

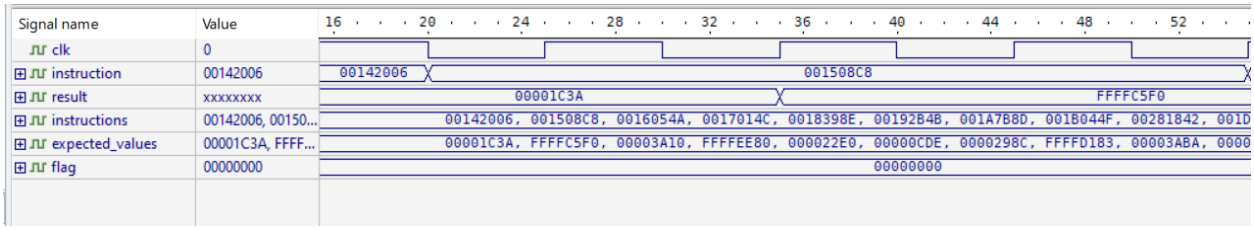

Figure 11: instruction 2 simulation

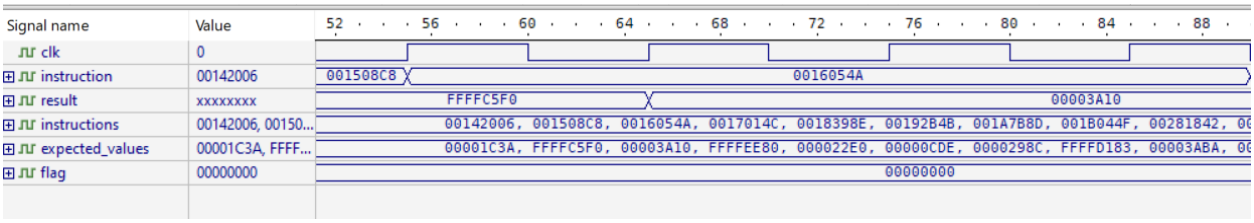

Figure 11: instruction 3 simulation

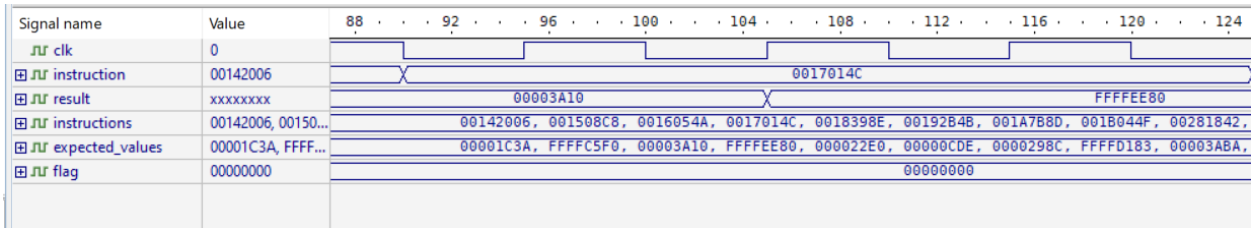

Figure 12: instruction 4 simulation

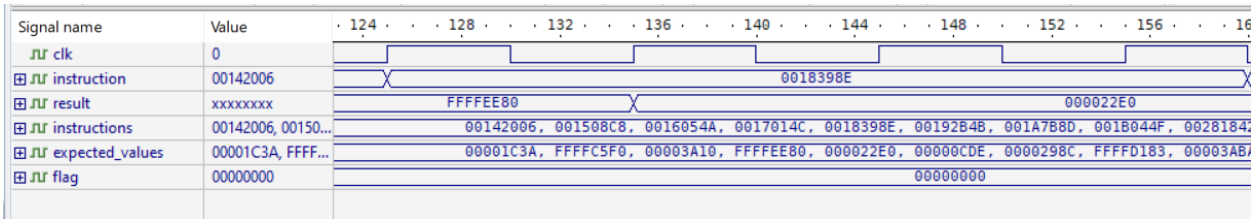

Figure 13: instruction 5 simulation

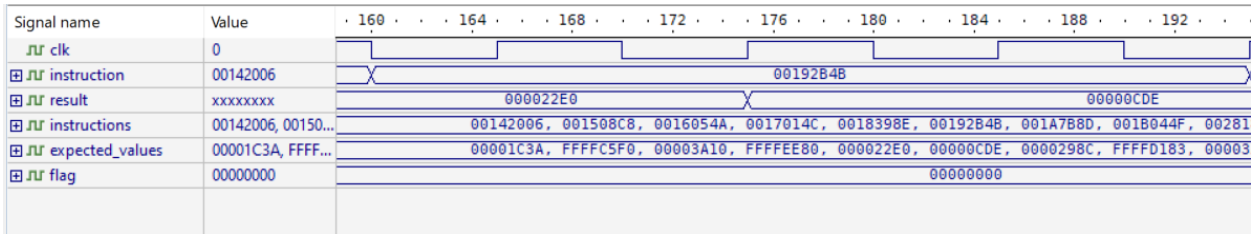

Figure 14: instruction 6 simulation

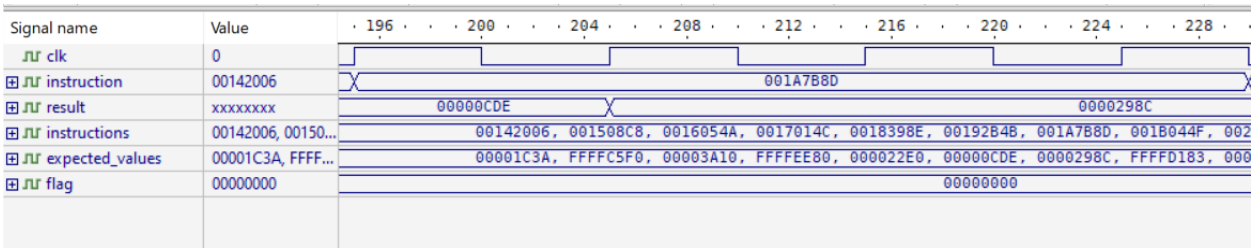

Figure 15: instruction 7 simulation

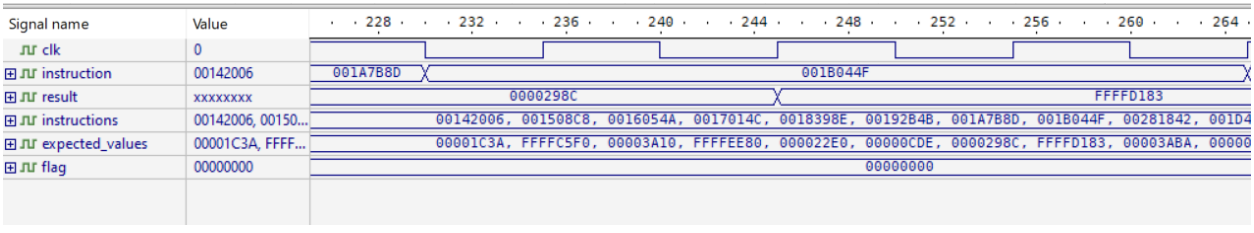

Figure 16: instruction 8 simulation

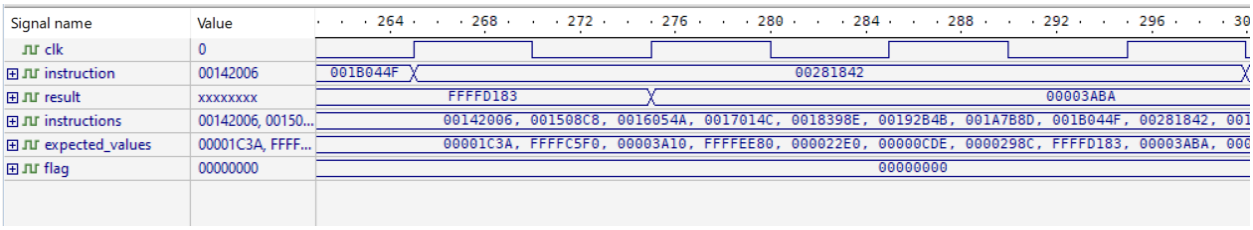

Figure 17: instruction 9 simulation

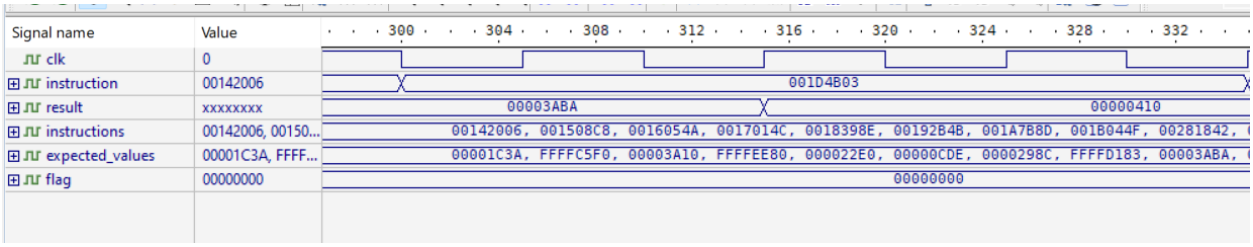

Figure 18: instruction 10 simulation

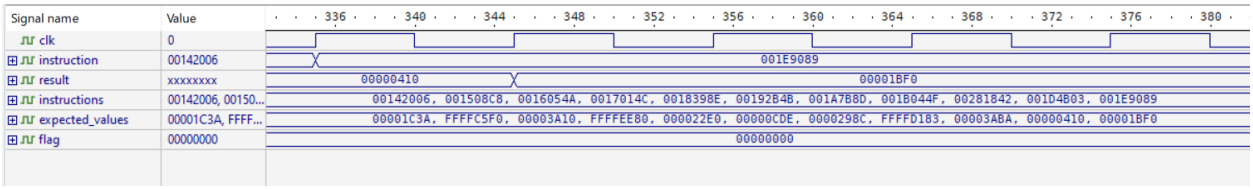

Figure 19: instruction 11 simulation