Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

ENCS3310

ADVANCED DIGITAL SYSTEMS DESIGN

FIRST SEMESTER PROJECT 2023-2024

Prepared by:**Lana Mahmoud Ali Musaffer**

ID:1210455

Section:2

Instructor:Dr.Abdallatif Abuissa

Abstract

This project aims to learn how to implement ALU and register file modules to build a simple microprocessor, so we can combine what we learned in our verilog chapter in this course. In addition, to gained a practical knowledge of module connections and working with delays and synchronization. Moreover, to learn how to run a program and test it with various cases and scenarios.

Table of Contents

Table of Figures

1. Theory

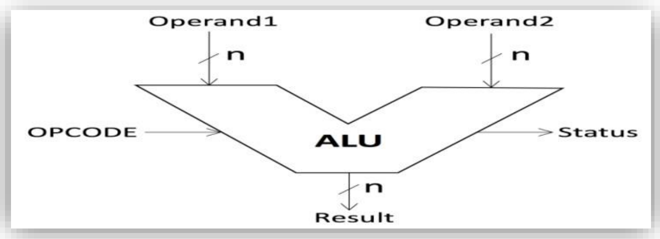
A Microprocessor is an important component of a computer architecture, it is used to perform different tasks on our computers. It is a programmable device that processes input through some arithmetic and logical operations over it and produces the desired output. In simple words, “a Microprocessor is a digital device on a chip that can fetch instructions from memory, decode and execute them, and give results. It Takes a bunch of instructions in machine language and executes them, telling the processor what it has to do”.[1]

In modern microprocessor design, the ALU and register file are two critical blocks that work together to execute instructions. The ALU performs operations on data stored in the register file, and the results are then stored back in the registers. The interaction between these components forms the basis of the microprocessor's ability to execute complex machine code instructions. As technology progresses, designers continue to refine and enhance these components, optimizing their performance and contributing to the ongoing evolution of microprocessor architecture.

1.1 Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is a very essential component of the central processing unit (CPU), it performs arithmetic and logical operations within a computer's processor. The ALU is typically designed in such a way that it has direct input and output access to the random access memory (RAM). In modern computers, the ALU is itself is divided into two categories, Arithmetic Unit (AU), and Logic Unit (LU).

“ALU’s operations are Arithmetic Operators, which refers to bit subtraction and addition, even though it does multiplication and division. Multiplication and division processes, on the other hand, are more expensive to do. Addition can be used in place of multiplication, while subtraction can be used in place of division. **Bit-Shifting Operators**, it is responsible for a multiplication operation, which involves shifting the locations of a bit to the right or left by a particular number of places. **Logical Operations** consist of NOR, AND, NOT, NAND, XOR, OR, and more.” [2]



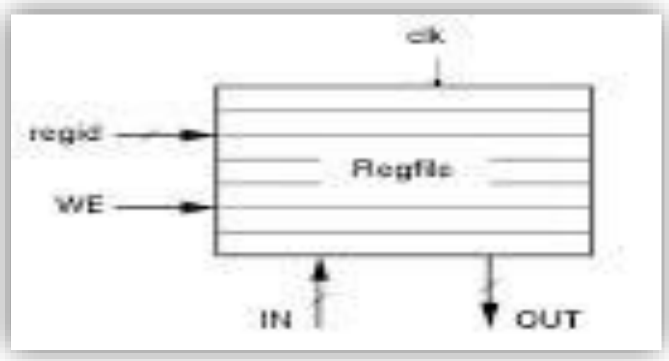
[3]

Figure 1. 1: Arithmetic Logic Unit

1.2 Register File

"Registers are a type of computer memory built directly into the processor or Central Processing Unit (CPU) that is used to store and manipulate data during the execution of instructions. A register may hold an instruction, a storage address, or any kind of data (such as a bit sequence or individual characters)".[4]

A register file is a collection of registers within a microprocessor that serves as fast, small-capacity storage locations for data. Registers are essential for temporarily holding operands and intermediate results during computations. The register file provides quick access to these data storage units, allowing the microprocessor to efficiently manage information during its operations. As technology advanced, the size and organization of register files evolved to meet the increasing demands of computing power and efficiency.



[5]

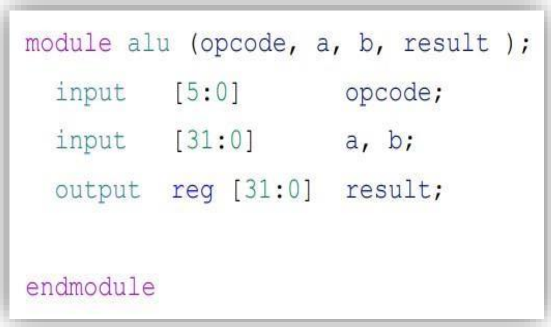
Figure 1. 2: Register File

2. Procedure and Discussion:

2.1 Part One: Main Components

2.2.1 The ALU

In this part, I built an ALU module, with two 32-bit inputs, a and b respectively, and a 32-bit output called result. This ALU module performs various logical and arithmetic operations, based on the value of the 6-bit opcode, and these operations include addition, subtraction, absolute value, inverse, maximum value, minimum value, average value, bit-wise NOT, bit-wise OR, bit-wise AND, and bit-wise XOR.



```
module alu (opcode, a, b, result );
  input   [5:0]       opcode;
  input   [31:0]      a, b;
  output  reg [31:0]  result;

endmodule
```

Figure 2. 1: Basis ALU Module.

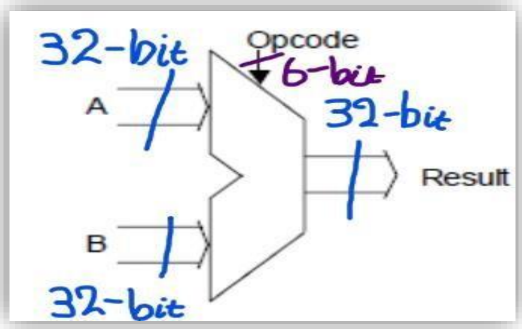
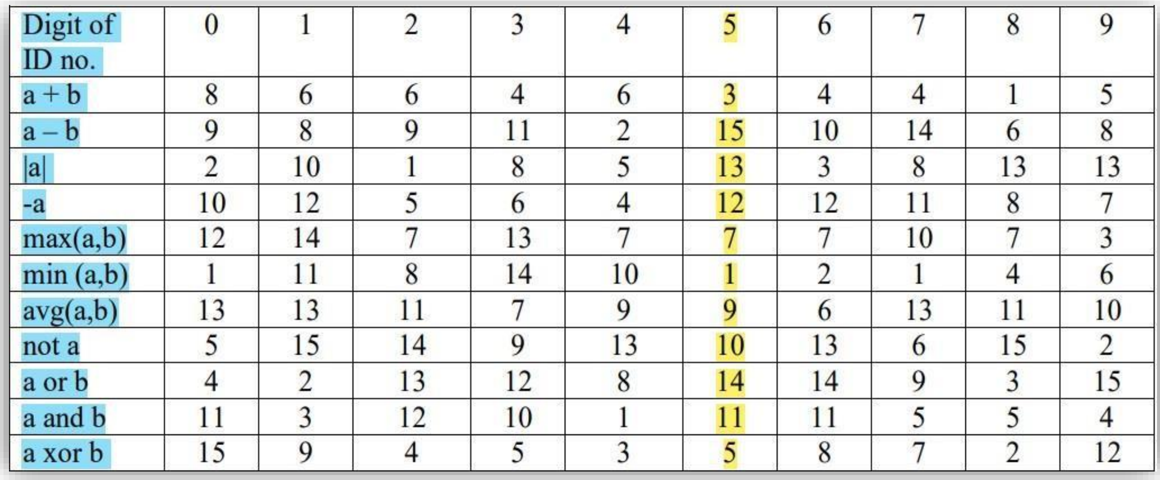


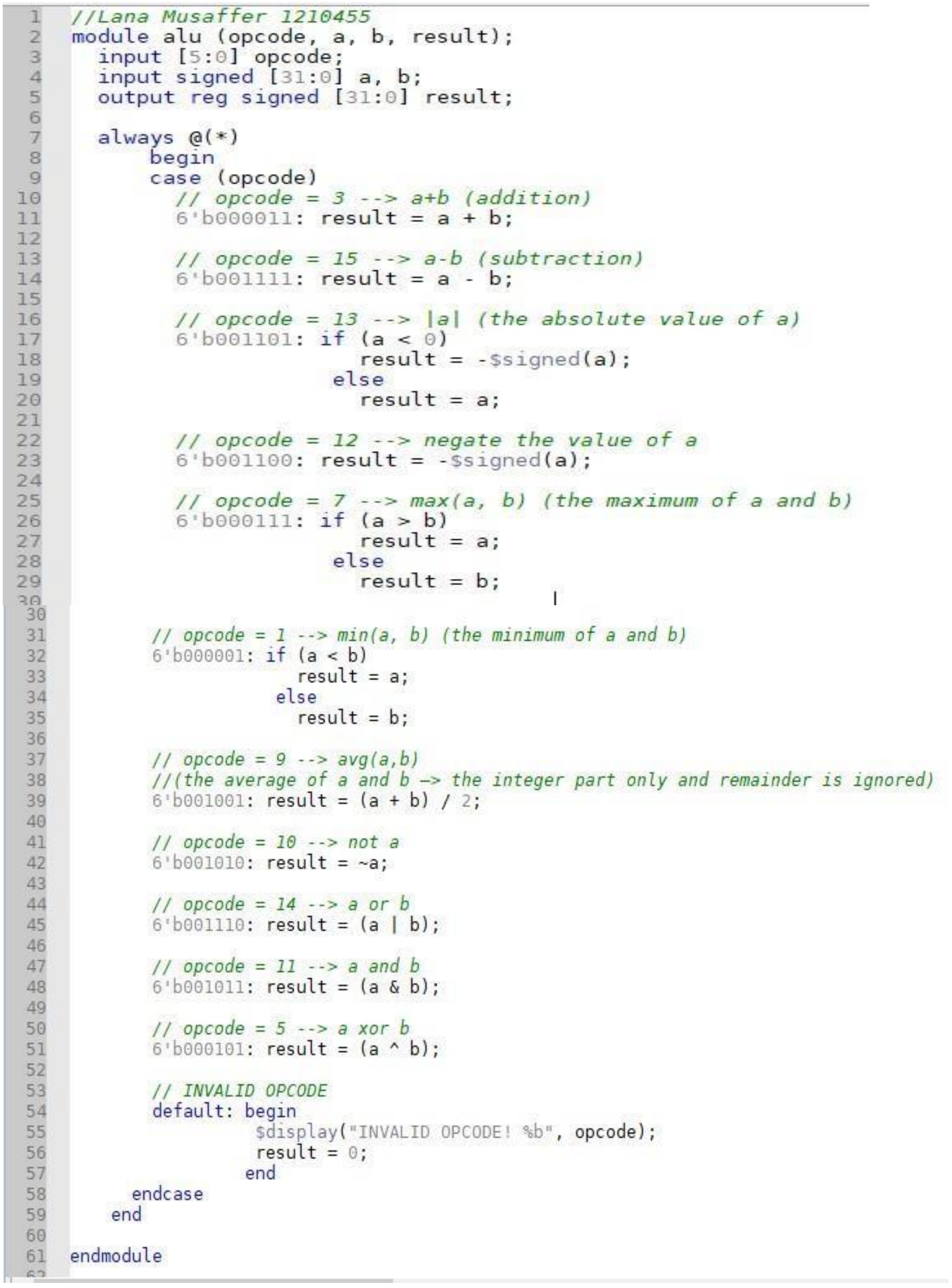
Figure 2. 2: ALU Block Diagram.

My ID number is 1210455, so the last digit is 5, the below figure shows the opcodes that I used in each instruction. In my code the opcode is represented in **binary** values.



Digit of ID no.	0	1	2	3	4	5	6	7	8	9
a + b	8	6	6	4	6	3	4	4	1	5
a – b	9	8	9	11	2	15	10	14	6	8
|a|	2	10	1	8	5	13	3	8	13	13
-a	10	12	5	6	4	12	12	11	8	7
max(a,b)	12	14	7	13	7	7	7	10	7	3
min (a,b)	1	11	8	14	10	1	2	1	4	6
avg(a,b)	13	13	11	7	9	9	6	13	11	10
not a	5	15	14	9	13	10	13	6	15	2
a or b	4	2	13	12	8	14	14	9	3	15
a and b	11	3	12	10	1	11	11	5	5	4
a xor b	15	9	4	5	3	5	8	7	2	12

Figure 2. 3: Opcode Table.



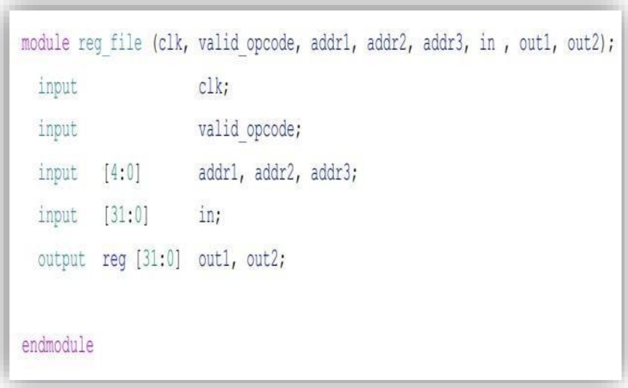
```
//Lana Musaffer 1210455
module alu (opcode, a, b, result);
  input [5:0] opcode;
  input signed [31:0] a, b;
  output reg signed [31:0] result;

  always @(*)
      begin
      case (opcode)
        // opcode = 3 --> a+b (addition)
        6'b000011: result = a + b;

        // opcode = 15 --> a-b (subtraction)
        6'b001111: result = a - b;

        // opcode = 13 --> |a| (the absolute value of a)
        6'b001101: if (a < 0)
                       result = -$signed(a);
                     else
                       result = a;

        // opcode = 12 --> negate the value of a
        6'b001100: result = -$signed(a);

        // opcode = 7 --> max(a, b) (the maximum of a and b)
        6'b000111: if (a > b)
                       result = a;
                     else
                       result = b;

        // opcode = 1 --> min(a, b) (the minimum of a and b)
        6'b000001: if (a < b)
                       result = a;
                     else
                       result = b;

        // opcode = 9 --> avg(a,b)
        //(the average of a and b -> the integer part only and remainder is ignored)
        6'b001001: result = (a + b) / 2;

        // opcode = 10 --> not a
        6'b001010: result = ~a;

        // opcode = 14 --> a or b
        6'b001110: result = (a | b);

        // opcode = 11 --> a and b
        6'b001011: result = (a & b);

        // opcode = 5 --> a xor b
        6'b000101: result = (a ^ b);

        // INVALID OPCODE
        default: begin
                   $display("INVALID OPCODE! %b", opcode);
                   result = 0;
                 end
      endcase
    end

endmodule
```

Figure 2. 4: ALU Module Code

2.2.2 The Register File

In this part, I built a very fast RAM to hold 32 x 32-bit words, with two 32-bit addresses for the read operation, addr1 and addr2 respectively, and a 32-bit address for the write operation, called addr3. This module is implemented to hold the operands that it is currently under processing.

- **Read Operation: On the rising edge of the clock**, output 1 produces the content stored at the memory location addressed by addr1, while Output 2 does the same for addr2
- **Write Operation:** the write operation utilizes the input to supply a value that is written into the memory location addressed by addr3



```
module reg_file (clk, valid_opcode, addr1, addr2, addr3, in , out1, out2);
  input              clk;
  input              valid_opcode;
  input   [4:0]      addr1, addr2, addr3;
  input   [31:0]     in;
  output  reg [31:0] out1, out2;


endmodule
```

Figure 2. 5: Basis regFile Module.

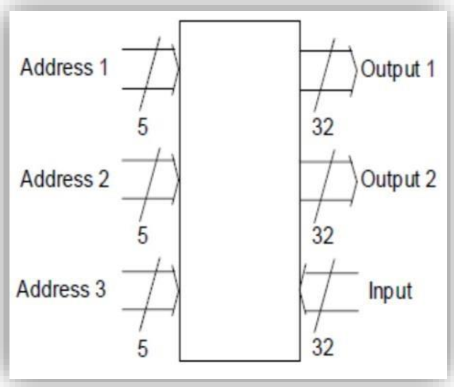
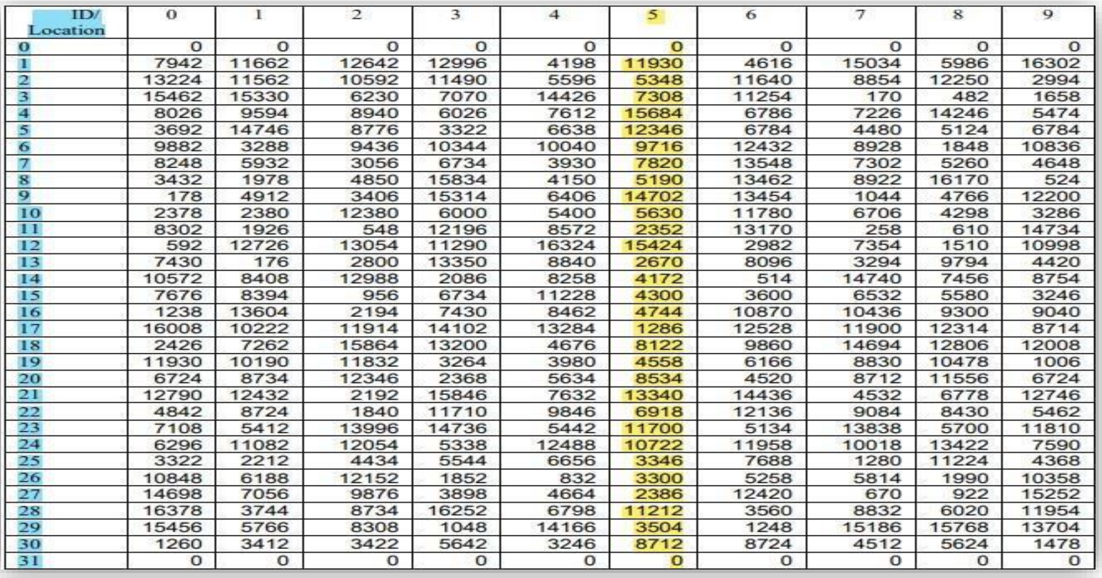


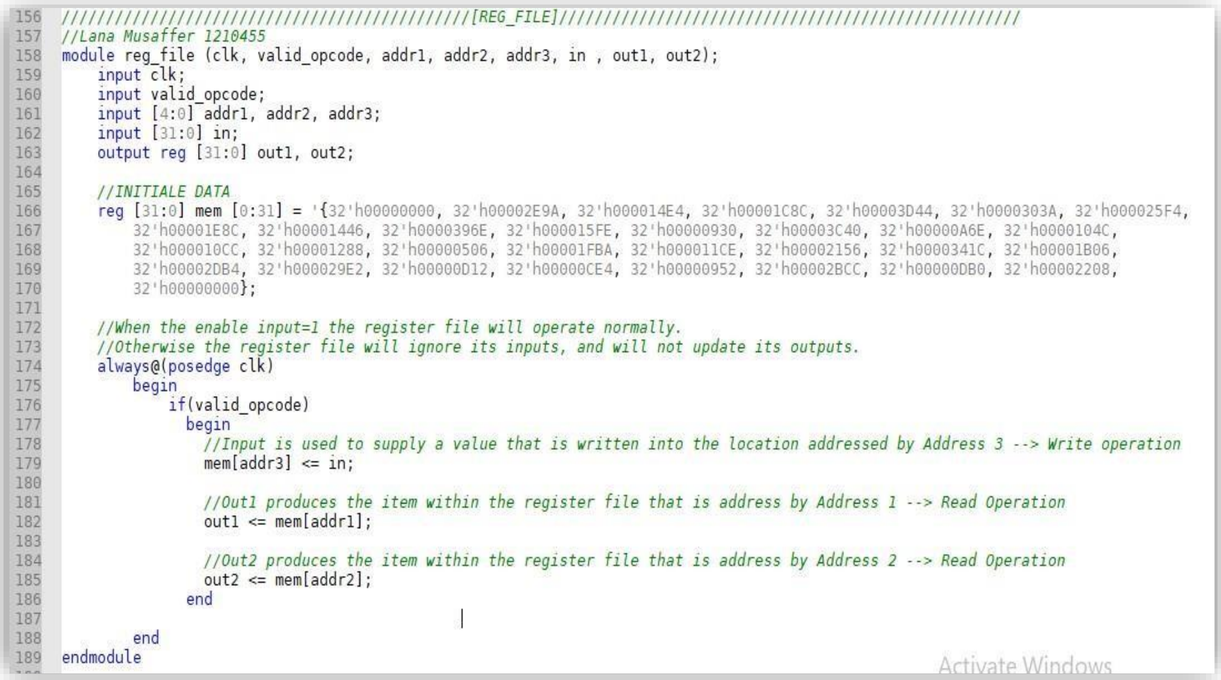
Figure 2. 6: Block Diagram for regFile

The second-from-last digit in my ID number is <u>**5**</u>, the below figure shows the opcodes that I used in each instruction. In my code the opcode is represented in **hexadecimal** values.



ID/ Location	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	7942	11662	12642	12996	4198	11930	4616	15034	5986	16302
2	13224	11562	10592	11490	5596	5348	11640	8854	12250	2994
3	15462	15330	6230	7070	14426	7308	11254	170	482	1658
4	8026	9594	8940	6026	7612	15684	6786	7226	14246	5474
5	3692	14746	8776	3322	6638	12346	6784	4480	5124	6784
6	9882	3288	9436	10344	10040	9716	12432	8928	1848	10836
7	8248	5932	3056	6734	3930	7820	13548	7302	5260	4648
8	3432	1978	4850	15834	4150	5190	13462	8922	16170	524
9	178	4912	3406	15314	6406	14702	13454	1044	4766	12200
10	2378	2380	12380	6000	5400	5630	11780	6706	4298	3286
11	8302	1926	548	12196	8572	2352	13170	258	610	14734
12	592	12726	13054	11290	16324	15424	2982	7354	1510	10998
13	7430	176	2800	13350	8840	2670	8096	3294	9794	4420
14	10572	8408	12988	2086	8258	4172	514	14740	7456	8754
15	7676	8394	956	6734	11228	4300	3600	6532	5580	3246
16	1238	13604	2194	7430	8462	4744	10870	10436	9300	9040
17	16008	10222	11914	14102	13284	1286	12528	11900	12314	8714
18	2426	7262	15864	13200	4676	8122	9860	14694	12806	12008
19	11930	10190	11832	3264	3980	4558	6166	8830	10478	1006
20	6724	8734	12346	2368	5634	8534	4520	8712	11556	6724
21	12790	12432	2192	15846	7632	13340	14436	4532	6778	12746
22	4842	8724	1840	11710	9846	6918	12136	9084	8430	5462
23	7108	5412	13996	14736	5442	11700	5134	13838	5700	11810
24	6296	11082	12054	5338	12488	10722	11958	10018	13422	7590
25	3322	2212	4434	5544	6656	3346	7688	1280	11224	4368
26	10848	6188	12152	1852	832	3300	5258	5814	1990	10358
27	14698	7056	9876	3898	4664	2386	12420	670	922	15252
28	16378	3744	8734	16252	6798	11212	3560	8832	6020	11954
29	15456	5766	8308	1048	14166	3504	1248	15186	15768	13704
30	1260	3412	3422	5642	3246	8712	8724	4512	5624	1478
31	0	0	0	0	0	0	0	0	0	0

Figure 2. 7: Data Table.



```verilog
//////////////////////////////////////////////[REG_FILE]//////////////////////////////////////////////////
//Lana Musaffer 1210455
module reg_file (clk, valid_opcode, addr1, addr2, addr3, in , out1, out2);
    input clk;
    input valid_opcode;
    input [4:0] addr1, addr2, addr3;
    input [31:0] in;
    output reg [31:0] out1, out2;

    //INITIALE DATA
    reg [31:0] mem [0:31] = '{32'h00000000, 32'h00002E9A, 32'h000014E4, 32'h00001C8C, 32'h00003D44, 32'h0000303A, 32'h000025F4,
        32'h00001E8C, 32'h00001446, 32'h0000396E, 32'h000015FE, 32'h00000930, 32'h00003C40, 32'h00000A6E, 32'h0000104C,
        32'h000010CC, 32'h00001288, 32'h00000506, 32'h00001FBA, 32'h000011CE, 32'h00002156, 32'h0000341C, 32'h00001B06,
        32'h00002DB4, 32'h000029E2, 32'h00000D12, 32'h00000CE4, 32'h00000952, 32'h00002BCC, 32'h00000DB0, 32'h00002208,
        32'h00000000};

    //When the enable input=1 the register file will operate normally.
    //Otherwise the register file will ignore its inputs, and will not update its outputs.
    always@(posedge clk)
        begin
            if(valid_opcode)
              begin
                //Input is used to supply a value that is written into the location addressed by Address 3 --> Write operation
                mem[addr3] <= in;

                //Out1 produces the item within the register file that is address by Address 1 --> Read Operation
                out1 <= mem[addr1];

                //Out2 produces the item within the register file that is address by Address 2 --> Read Operation
                out2 <= mem[addr2];
              end

        end
endmodule
```

Figure 2. 8: regFile Module Code.

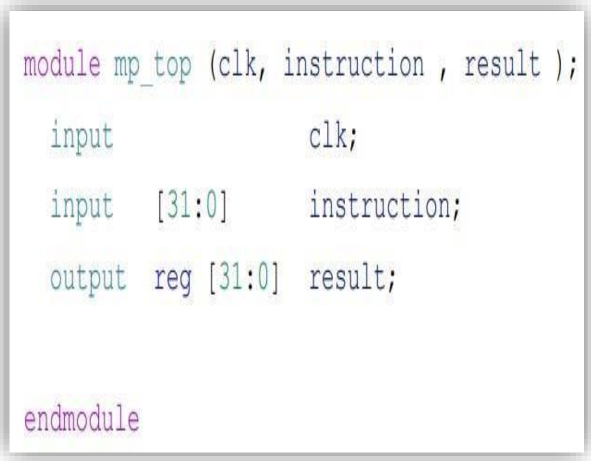
✓ Critical remarks:

A register file is a collection of registers, each capable of storing a hexadecimal value. The enable input (**valid_opcode**)controls whether the register file should respond to, and update its outputs based on the inputs. When the valid_opcode is set to 1 (high), it indicates that the register file is allowed to operate normally. In this state, the register file will accept and process its input signals, updating the values stored in the registers accordingly, so it will operate normally, it actively reads and writes data according to the specified operations. Otherwise, the valid_opcode is set to 0 (low), it signals that the register file should ignore its inputs. In this state, the register file will not update its outputs based on any incoming data. It essentially freezes its state and does not respond to changes in input values, so it will become insensitive to changes in the input signals. The values stored in the registers remain unchanged, and the register file does not output any new data.

Another point is: **Synchronizing the register file** to a clock helps avoid potential problems, such as reading and overwriting occurring simultaneously. I will talk about synchronization later in detail or you can just click to move to this topic.

2.2 Part Two: Simple Microprocessor

In this part, I made machine instructions and they are represented as 32-bit numbers, in a specific format. The opcode, determining the operation to be performed, is denoted by the first 6 bits. The next 5 bits identify the first source register, the subsequent 5 bits identify the second source register, and the next 5 bits designate the destination register. The remaining 11 bits in the instruction format are unused. This structured format ensures a clear and organized representation of instructions, to get efficient decoding and execution by the processor. Then, I connected the main components, ALU and RAM, in order to integrate a simple microprocessor



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

  input              clk;

  input    [31:0]    instruction;

  output   reg [31:0] result;


endmodule
```

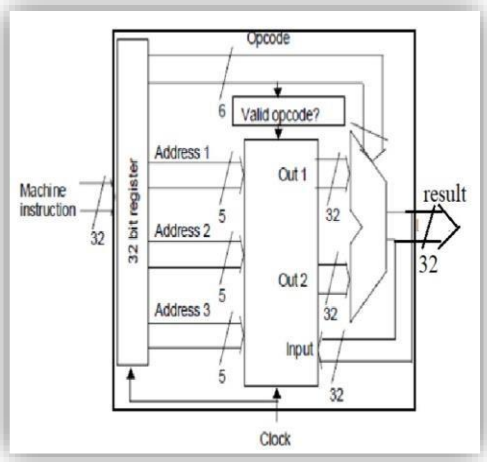
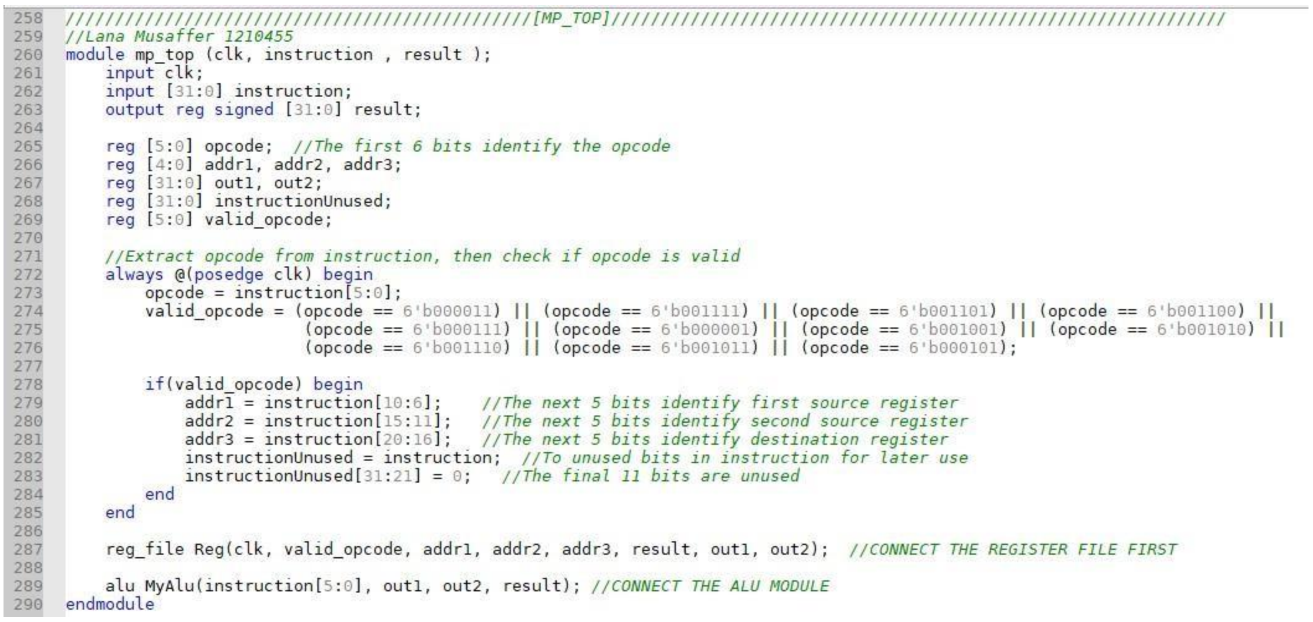


Figure 2. 9: Basis of mpTOP Module

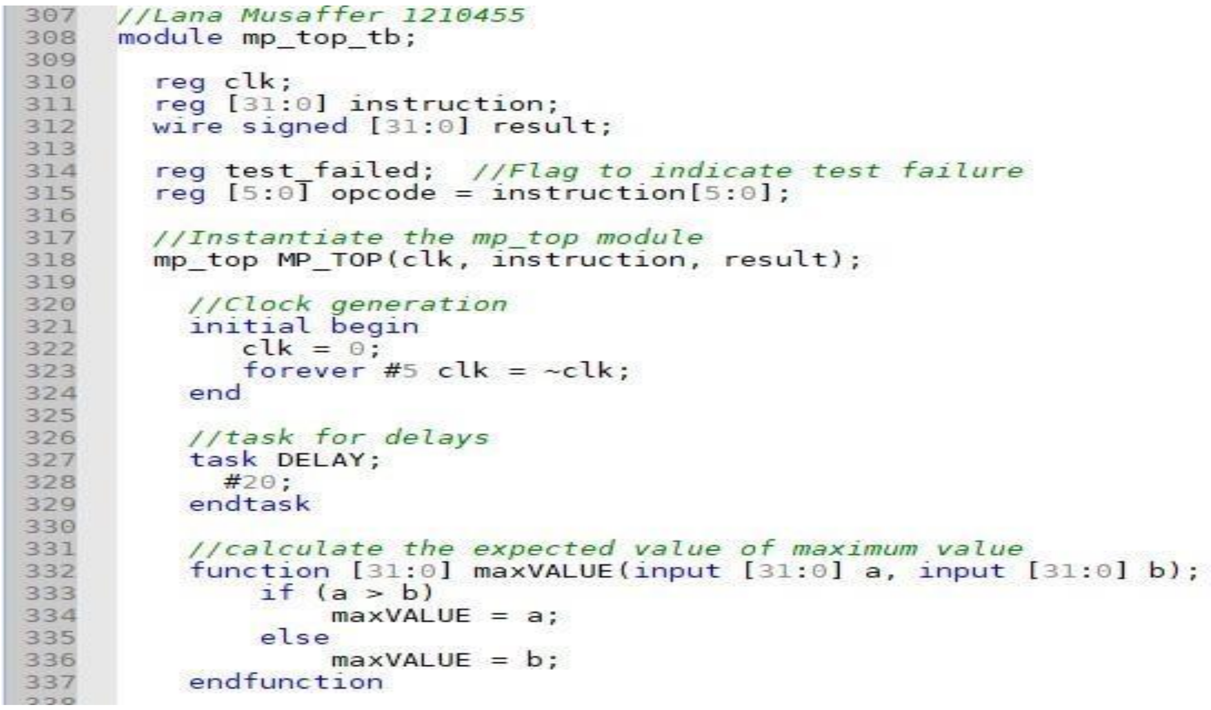
Figure 2. 10: Block Diagram for mpTOP



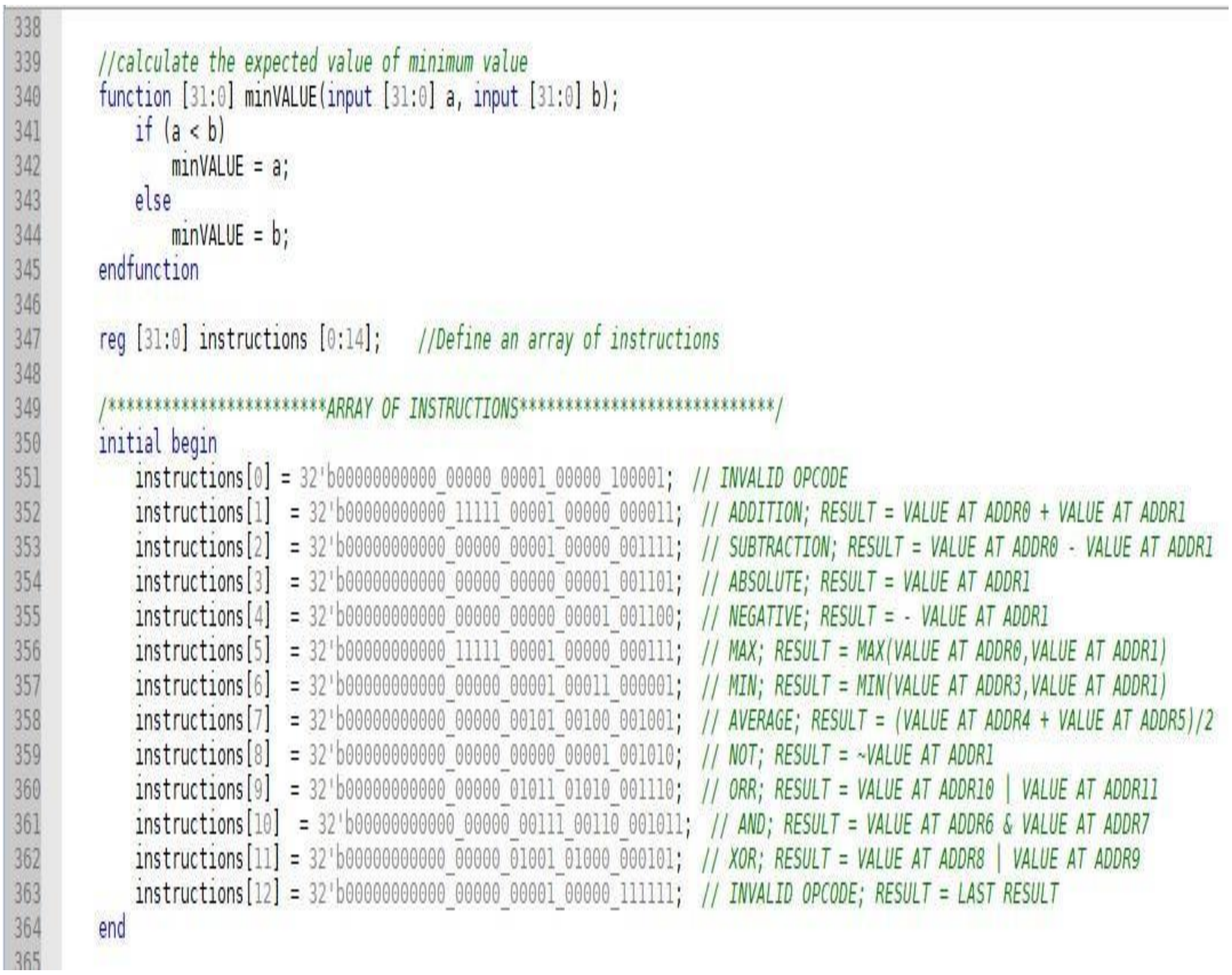
```
/////////////////////////////////////////////////[MP_TOP]/////////////////////////////////////////////////////////////////
//Lana Musaffer 1210455
module mp_top (clk, instruction , result );
    input clk;
    input [31:0] instruction;
    output reg signed [31:0] result;

    reg [5:0] opcode;  //The first 6 bits identify the opcode
    reg [4:0] addr1, addr2, addr3;
    reg [31:0] out1, out2;
    reg [31:0] instructionUnused;
    reg [5:0] valid_opcode;

    //Extract opcode from instruction, then check if opcode is valid
    always @(posedge clk) begin
        opcode = instruction[5:0];
        valid_opcode = (opcode == 6'b000011) || (opcode == 6'b001111) || (opcode == 6'b001101) || (opcode == 6'b001100) ||
                        (opcode == 6'b000111) || (opcode == 6'b000001) || (opcode == 6'b001001) || (opcode == 6'b001010) ||
                        (opcode == 6'b001110) || (opcode == 6'b001011) || (opcode == 6'b000101);

        if(valid_opcode) begin
            addr1 = instruction[10:6];    //The next 5 bits identify first source register
            addr2 = instruction[15:11];   //The next 5 bits identify second source register
            addr3 = instruction[20:16];   //The next 5 bits identify destination register
            instructionUnused = instruction;  //To unused bits in instruction for later use
            instructionUnused[31:21] = 0;   //The final 11 bits are unused
        end
    end

    reg_file Reg(clk, valid_opcode, addr1, addr2, addr3, result, out1, out2);  //CONNECT THE REGISTER FILE FIRST

    alu MyAlu(instruction[5:0], out1, out2, result); //CONNECT THE ALU MODULE
endmodule
```

Figure 2. 11: mpTOP Module Code

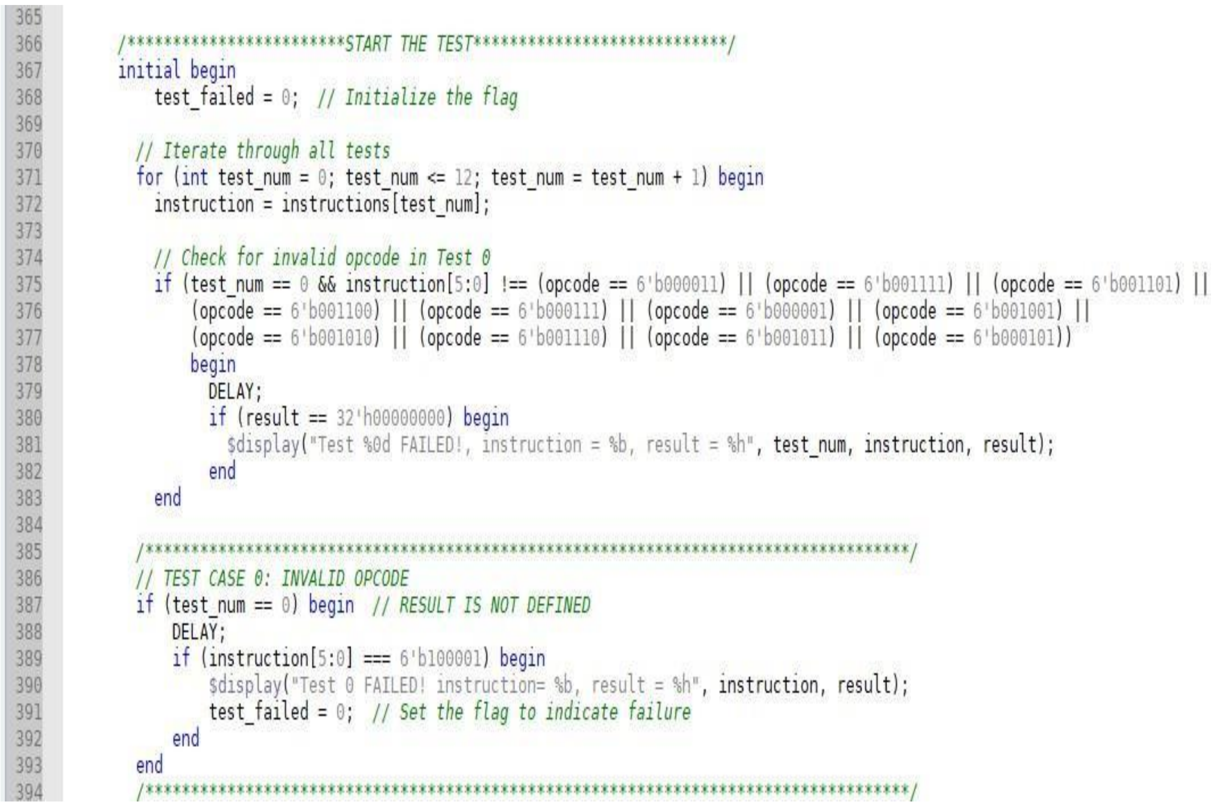
Test Bench For mp_top:



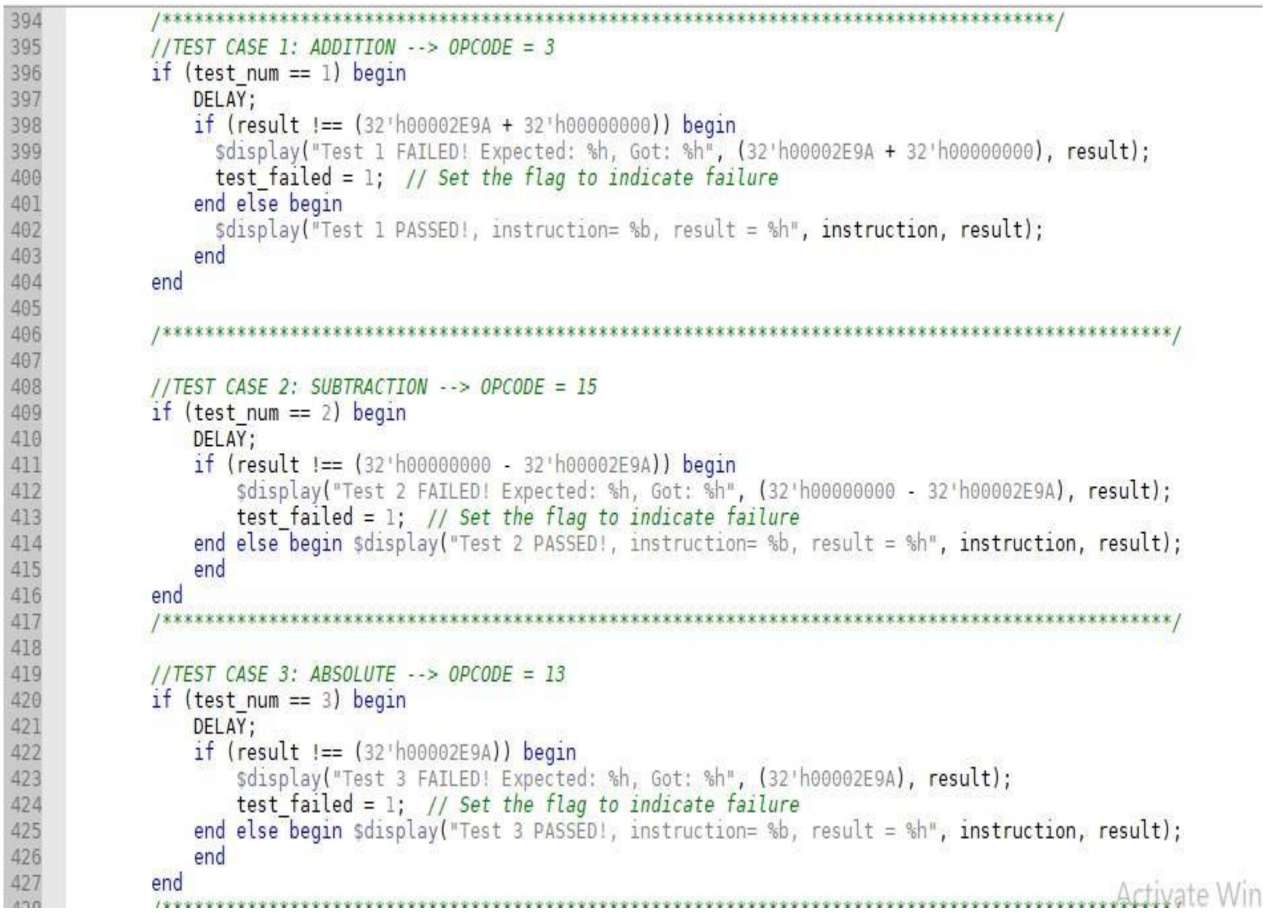
```verilog
//Lana Musaffer 1210455
module mp_top_tb;

  reg clk;
  reg [31:0] instruction;
  wire signed [31:0] result;

  reg test_failed;  //Flag to indicate test failure
  reg [5:0] opcode = instruction[5:0];

  //Instantiate the mp_top module
  mp_top MP_TOP(clk, instruction, result);

    //Clock generation
    initial begin
       clk = 0;
       forever #5 clk = ~clk;
    end

    //task for delays
    task DELAY;
      #20;
    endtask

    //calculate the expected value of maximum value
    function [31:0] maxVALUE(input [31:0] a, input [31:0] b);
        if (a > b)
            maxVALUE = a;
        else
            maxVALUE = b;
    endfunction
```



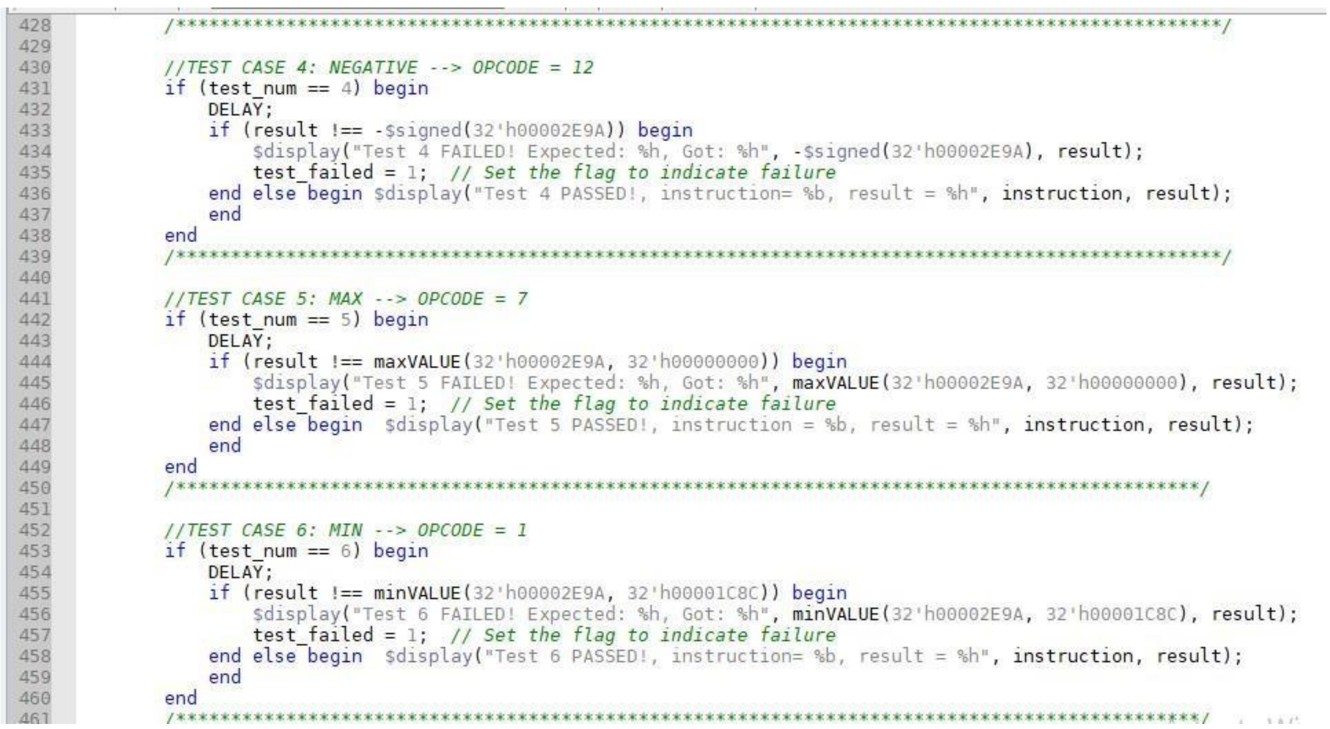
```verilog

//calculate the expected value of minimum value
function [31:0] minVALUE(input [31:0] a, input [31:0] b);
    if (a < b)
        minVALUE = a;
    else
        minVALUE = b;
endfunction

reg [31:0] instructions [0:14];    //Define an array of instructions

/************************ARRAY OF INSTRUCTIONS***************************/
initial begin
    instructions[0] = 32'b00000000000_00000_00001_00000_100001;  // INVALID OPCODE
    instructions[1]  = 32'b00000000000_11111_00001_00000_000011;  // ADDITION; RESULT = VALUE AT ADDR0 + VALUE AT ADDR1
    instructions[2]  = 32'b00000000000_00000_00001_00000_001111;  // SUBTRACTION; RESULT = VALUE AT ADDR0 - VALUE AT ADDR1
    instructions[3]  = 32'b00000000000_00000_00000_00001_001101;  // ABSOLUTE; RESULT = VALUE AT ADDR1
    instructions[4]  = 32'b00000000000_00000_00000_00001_001100;  // NEGATIVE; RESULT = - VALUE AT ADDR1
    instructions[5]  = 32'b00000000000_11111_00001_00000_000111;  // MAX; RESULT = MAX(VALUE AT ADDR0,VALUE AT ADDR1)
    instructions[6]  = 32'b00000000000_00000_00001_00011_000001;  // MIN; RESULT = MIN(VALUE AT ADDR3,VALUE AT ADDR1)
    instructions[7]  = 32'b00000000000_00000_00101_00100_001001;  // AVERAGE; RESULT = (VALUE AT ADDR4 + VALUE AT ADDR5)/2
    instructions[8]  = 32'b00000000000_00000_00000_00001_001010;  // NOT; RESULT = ~VALUE AT ADDR1
    instructions[9]  = 32'b00000000000_00000_01011_01010_001110;  // ORR; RESULT = VALUE AT ADDR10 | VALUE AT ADDR11
    instructions[10]  = 32'b00000000000_00000_00111_00110_001011;  // AND; RESULT = VALUE AT ADDR6 & VALUE AT ADDR7
    instructions[11] = 32'b00000000000_00000_01001_01000_000101;  // XOR; RESULT = VALUE AT ADDR8 | VALUE AT ADDR9
    instructions[12] = 32'b00000000000_00000_00001_00000_111111;  // INVALID OPCODE; RESULT = LAST RESULT
end
```



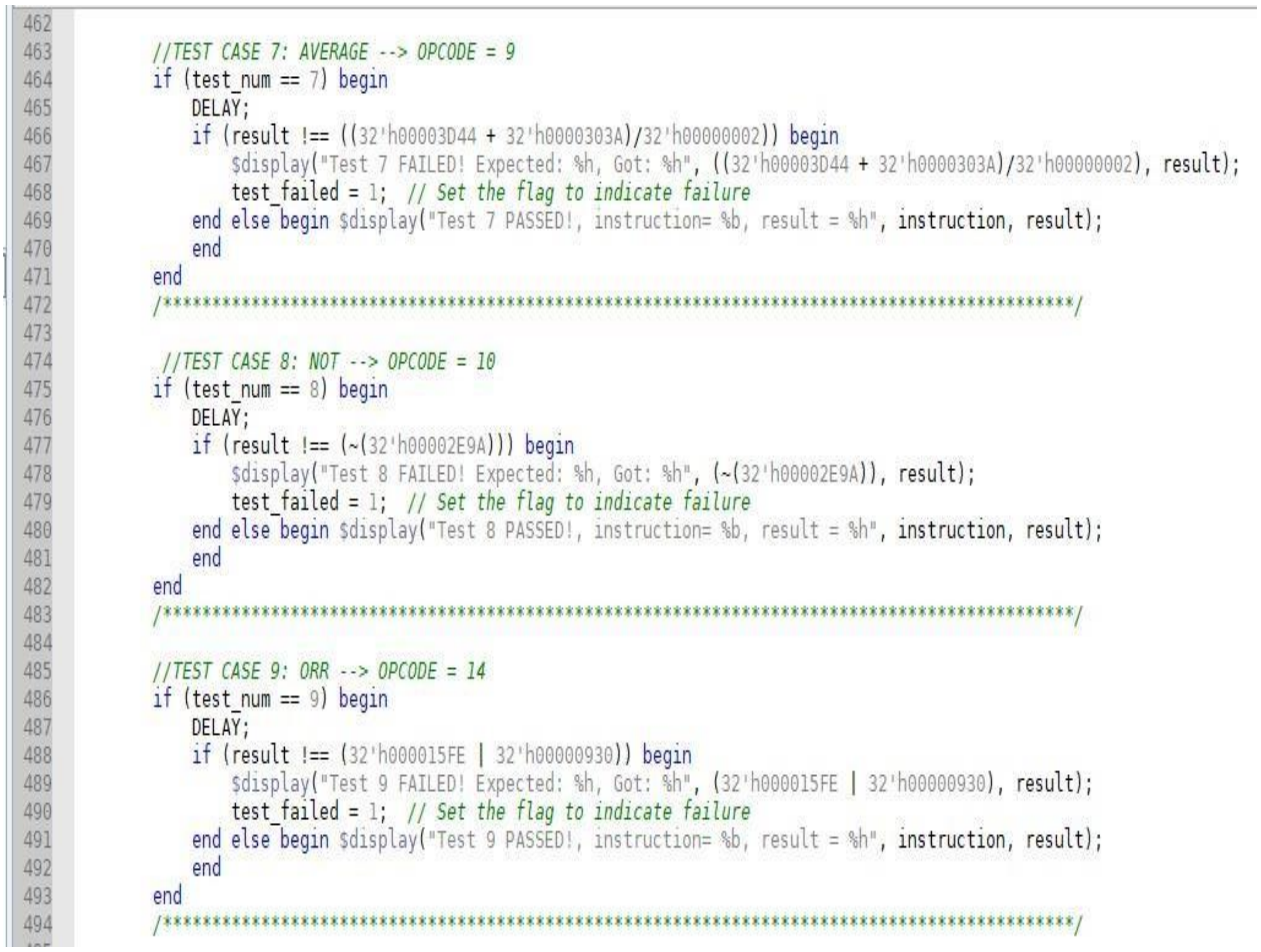
```
/************************START THE TEST*****************************/
initial begin
    test_failed = 0;  // Initialize the flag

  // Iterate through all tests
  for (int test_num = 0; test_num <= 12; test_num = test_num + 1) begin
    instruction = instructions[test_num];

    // Check for invalid opcode in Test 0
    if (test_num == 0 && instruction[5:0] !== (opcode == 6'b000011) || (opcode == 6'b001111) || (opcode == 6'b001101) ||
        (opcode == 6'b001100) || (opcode == 6'b000111) || (opcode == 6'b000001) || (opcode == 6'b001001) ||
        (opcode == 6'b001010) || (opcode == 6'b001110) || (opcode == 6'b001011) || (opcode == 6'b000101))
        begin
          DELAY;
          if (result == 32'h00000000) begin
            $display("Test %0d FAILED!, instruction = %b, result = %h", test_num, instruction, result);
          end
    end

  /*************************************************************************************/
  // TEST CASE 0: INVALID OPCODE
  if (test_num == 0) begin  // RESULT IS NOT DEFINED
      DELAY;
      if (instruction[5:0] === 6'b100001) begin
          $display("Test 0 FAILED! instruction= %b, result = %h", instruction, result);
          test_failed = 0;  // Set the flag to indicate failure
      end
  end
  /*************************************************************************************/
```



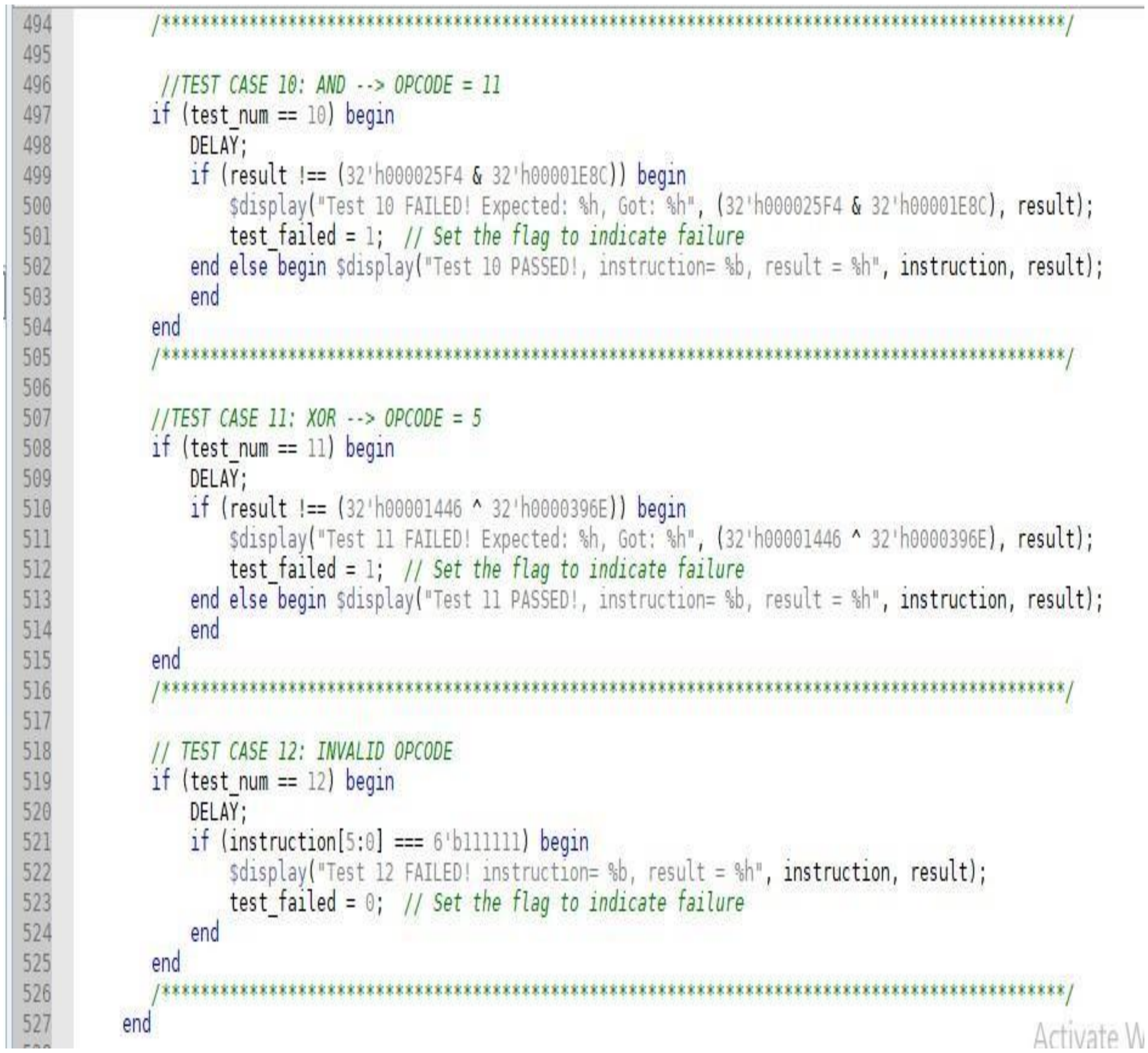
```
  /*******************************************************************************************/
  //TEST CASE 1: ADDITION --> OPCODE = 3
  if (test_num == 1) begin
      DELAY;
      if (result !== (32'h00002E9A + 32'h00000000)) begin
        $display("Test 1 FAILED! Expected: %h, Got: %h", (32'h00002E9A + 32'h00000000), result);
        test_failed = 1;  // Set the flag to indicate failure
      end else begin
        $display("Test 1 PASSED!, instruction= %b, result = %h", instruction, result);
      end
  end

  /*******************************************************************************************************/

  //TEST CASE 2: SUBTRACTION --> OPCODE = 15
  if (test_num == 2) begin
      DELAY;
      if (result !== (32'h00000000 - 32'h00002E9A)) begin
          $display("Test 2 FAILED! Expected: %h, Got: %h", (32'h00000000 - 32'h00002E9A), result);
          test_failed = 1;  // Set the flag to indicate failure
      end else begin $display("Test 2 PASSED!, instruction= %b, result = %h", instruction, result);
      end
  end
  /*******************************************************************************************************/

  //TEST CASE 3: ABSOLUTE --> OPCODE = 13
  if (test_num == 3) begin
      DELAY;
      if (result !== (32'h00002E9A)) begin
          $display("Test 3 FAILED! Expected: %h, Got: %h", (32'h00002E9A), result);
          test_failed = 1;  // Set the flag to indicate failure
      end else begin $display("Test 3 PASSED!, instruction= %b, result = %h", instruction, result);
      end
  end
  /*******************************************************************************************************
```



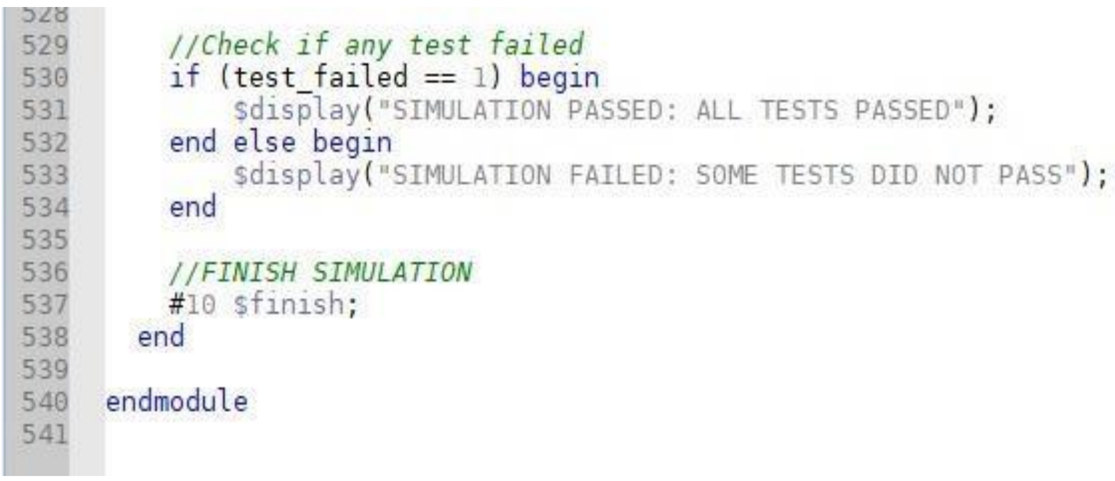
```
        /*********************************************************************************************/

        //TEST CASE 4: NEGATIVE --> OPCODE = 12
        if (test_num == 4) begin
            DELAY;
            if (result !== -$signed(32'h00002E9A)) begin
                $display("Test 4 FAILED! Expected: %h, Got: %h", -$signed(32'h00002E9A), result);
                test_failed = 1;  // Set the flag to indicate failure
            end else begin $display("Test 4 PASSED!, instruction= %b, result = %h", instruction, result);
            end
        end
        /*********************************************************************************************/

        //TEST CASE 5: MAX --> OPCODE = 7
        if (test_num == 5) begin
            DELAY;
            if (result !== maxVALUE(32'h00002E9A, 32'h00000000)) begin
                $display("Test 5 FAILED! Expected: %h, Got: %h", maxVALUE(32'h00002E9A, 32'h00000000), result);
                test_failed = 1;  // Set the flag to indicate failure
            end else begin  $display("Test 5 PASSED!, instruction = %b, result = %h", instruction, result);
            end
        end
        /*******************************************************************************************/

        //TEST CASE 6: MIN --> OPCODE = 1
        if (test_num == 6) begin
            DELAY;
            if (result !== minVALUE(32'h00002E9A, 32'h00001C8C)) begin
                $display("Test 6 FAILED! Expected: %h, Got: %h", minVALUE(32'h00002E9A, 32'h00001C8C), result);
                test_failed = 1;  // Set the flag to indicate failure
            end else begin  $display("Test 6 PASSED!, instruction= %b, result = %h", instruction, result);
            end
        end
        /*******************************************************************************************/
```



```

        //TEST CASE 7: AVERAGE --> OPCODE = 9
        if (test_num == 7) begin
            DELAY;
            if (result !== ((32'h00003D44 + 32'h0000303A)/32'h00000002)) begin
                $display("Test 7 FAILED! Expected: %h, Got: %h", ((32'h00003D44 + 32'h0000303A)/32'h00000002), result);
                test_failed = 1;  // Set the flag to indicate failure
            end else begin $display("Test 7 PASSED!, instruction= %b, result = %h", instruction, result);
            end
        end
        /*******************************************************************************************/

         //TEST CASE 8: NOT --> OPCODE = 10
        if (test_num == 8) begin
            DELAY;
            if (result !== (~(32'h00002E9A))) begin
                $display("Test 8 FAILED! Expected: %h, Got: %h", (~(32'h00002E9A)), result);
                test_failed = 1;  // Set the flag to indicate failure
            end else begin $display("Test 8 PASSED!, instruction= %b, result = %h", instruction, result);
            end
        end
        /*******************************************************************************************/

        //TEST CASE 9: ORR --> OPCODE = 14
        if (test_num == 9) begin
            DELAY;
            if (result !== (32'h000015FE | 32'h00000930)) begin
                $display("Test 9 FAILED! Expected: %h, Got: %h", (32'h000015FE | 32'h00000930), result);
                test_failed = 1;  // Set the flag to indicate failure
            end else begin $display("Test 9 PASSED!, instruction= %b, result = %h", instruction, result);
            end
        end
        /*******************************************************************************************/
```



```
        /*****************************************************************************************/

         //TEST CASE 10: AND --> OPCODE = 11
        if (test_num == 10) begin
            DELAY;
            if (result !== (32'h000025F4 & 32'h00001E8C)) begin
                $display("Test 10 FAILED! Expected: %h, Got: %h", (32'h000025F4 & 32'h00001E8C), result);
                test_failed = 1;  // Set the flag to indicate failure
            end else begin $display("Test 10 PASSED!, instruction= %b, result = %h", instruction, result);
            end
        end
        /*****************************************************************************************/

        //TEST CASE 11: XOR --> OPCODE = 5
        if (test_num == 11) begin
            DELAY;
            if (result !== (32'h00001446 ^ 32'h0000396E)) begin
                $display("Test 11 FAILED! Expected: %h, Got: %h", (32'h00001446 ^ 32'h0000396E), result);
                test_failed = 1;  // Set the flag to indicate failure
            end else begin $display("Test 11 PASSED!, instruction= %b, result = %h", instruction, result);
            end
        end
        /*****************************************************************************************/

        // TEST CASE 12: INVALID OPCODE
        if (test_num == 12) begin
            DELAY;
            if (instruction[5:0] === 6'b111111) begin
                $display("Test 12 FAILED! instruction= %b, result = %h", instruction, result);
                test_failed = 0;  // Set the flag to indicate failure
            end
        end
        /*****************************************************************************************/
    end
```



```

        //Check if any test failed
        if (test_failed == 1) begin
            $display("SIMULATION PASSED: ALL TESTS PASSED");
        end else begin
            $display("SIMULATION FAILED: SOME TESTS DID NOT PASS");
        end

        //FINISH SIMULATION
        #10 $finish;
    end

endmodule

```

Figure 2. 12: mp_top Module Code.

How my design works:

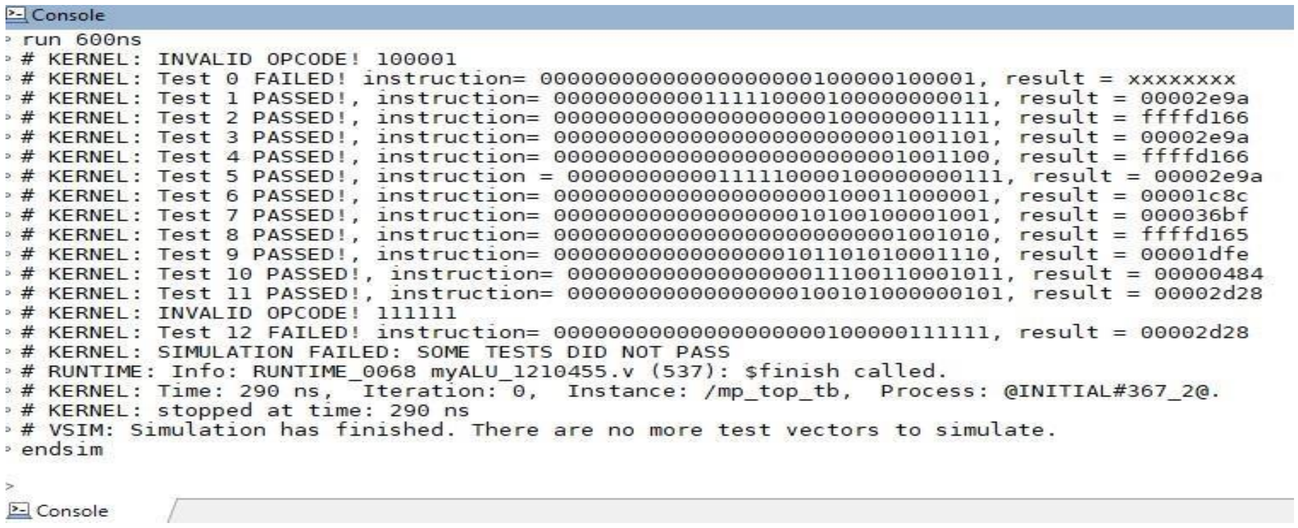
In my designed microprocessor system, the relationship between the clock cycle and instruction execution is crucial for synchronous operation. The clock (CLK) acts as a synchronization mechanism, and the processor responds to the rising edge of the clock. On each rising edge, the processor decodes machine instructions, extracting the opcode and relevant addresses. The subsequent actions, such as enabling the register file and processing operands in the ALU module. In the first cycle, it takes the instruction, and in the second one, it makes the operations.

Notably, the result of an instruction is written to the register file on the same rising edge of the clock. This ensures that the register file is updated synchronously with the instruction execution. The synchronous design guarantees consistency and predictability in the state of the processor at each clock cycle. Therefore, the relationship between the clock cycle and the result being written to the register is immediate, allowing for precise control over the processor's behavior.

For testing, I initialized a flag to keep track of the pass or fail cases, and I made the delay in a task to make it easier for me when changing it to the desired value just ONCE, also I calculated the expected values of the max and min in a function for better readability. I created an array of instructions and ensured the representation for each provided opcode. During testing, I printed both the input instruction and the corresponding output result for thorough examination. In addition, for every instruction, I computed the expected result and compared it with the actual output. If any instruction failed to produce the expected outcome, the test was marked as unsuccessful; otherwise, it was taken as a success.

Results Overview:

Note that I showed in details (in the code) each instruction how it is calculated and also the expected value.



```
Console
run 600ns
# KERNEL: INVALID OPCODE! 100001
# KERNEL: Test 0 FAILED! instruction= 00000000000000000000100000100001, result = xxxxxxxx
# KERNEL: Test 1 PASSED!, instruction= 00000000001111100000100000000011, result = 00002e9a
# KERNEL: Test 2 PASSED!, instruction= 00000000000000000000100000001111, result = ffffd166
# KERNEL: Test 3 PASSED!, instruction= 00000000000000000000000001001101, result = 00002e9a
# KERNEL: Test 4 PASSED!, instruction= 00000000000000000000000001001100, result = ffffd166
# KERNEL: Test 5 PASSED!, instruction = 00000000001111100000100000000111, result = 00002e9a
# KERNEL: Test 6 PASSED!, instruction= 00000000000000000000100011000001, result = 00001c8c
# KERNEL: Test 7 PASSED!, instruction= 00000000000000000101001000001001, result = 000036bf
# KERNEL: Test 8 PASSED!, instruction= 00000000000000000000000001001010, result = ffffd165
# KERNEL: Test 9 PASSED!, instruction= 00000000000000000101101010001110, result = 00001dfe
# KERNEL: Test 10 PASSED!, instruction= 00000000000000000011100110001011, result = 00000484
# KERNEL: Test 11 PASSED!, instruction= 00000000000000000010010100000101, result = 00002d28
# KERNEL: INVALID OPCODE! 111111
# KERNEL: Test 12 FAILED! instruction= 00000000000000000000100000111111, result = 00002d28
# KERNEL: SIMULATION FAILED: SOME TESTS DID NOT PASS
# RUNTIME: Info: RUNTIME_0068 myALU_1210455.v (537): $finish called.
# KERNEL: Time: 290 ns,  Iteration: 0,  Instance: /mp_top_tb,  Process: @INITIAL#367_2@.
# KERNEL: stopped at time: 290 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
endsim

>
Console
```

Figure 2. 13: mp_top Module Results.

CHALLENGES & AMBITIONS

instructions[1] = 32'b00000000000_**11111**_00001_00000_000011;

 // ADDITION; RESULT = VALUE AT ADDR0 + VALUE AT ADDR1

instructions[2] = 32'b00000000000_00000_00001_**11111**_001111;

  // SUBTRACTION; RESULT = VALUE AT ADDR**32** - VALUE AT ADDR1

- When I tried to test this case to make sure that my code worked perfectly and it wrote back the result at each clock cycle, but unfortunately, it didn't work as it was supposed to be; VALUE AT ADDR**32** → Result from the addition operation, instead it just contained the original data.
- I think this issue is because of the delay, and I tried to solve it by using buffer and DFF, but both of them didn't work):
- This is the only problem that I have in my code.

3. Conclusion:

In conclusion, the design and implementation of the microprocessor system involved the creation of two main components, the Arithmetic Logic Unit (ALU) and the Register File. These components function in synchrony within the microprocessor, responding to the rising edge of the clock during each clock cycle. The ALU performs arithmetic and logical operations based on machine instructions, while the Register File manages the storage of the data. During testing, a thorough approach was taken to evaluate the processor's correctness and reliability. The test bench covered various opcodes, with careful examination of each instruction's input and output. So, a very fast and efficient microprocessor is built.

4. References:

[1] Introduction of Microprocessor - GeeksforGeeks

[2] ALU (Arithmetic Logic Unit) | GATE Notes (byjus.com)

[3] https://www.google.com/url?sa=i&url=https%3A%2F%2Fati.ttu.ee%2FIAY0340%2Flabs%2FTutorials%2FSystemC%2FALU.html&psig=AOvVaw0jYyRuJl0G2vNgi0p4RiHU&ust=1705173394640000&source=images&cd=vfe&opi=89978449&ved=0CBMQjRxqFwoTCPito4jI2IMDFQAAAAAdAAAAABAD

[4] https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwi48YDfm-KDAxUyUKQEHTdRAOAQFnoECBIQAw&url=https%3A%2F%2Fwww.totalphase.com%2Fblog%2F2023%2F05%2Fwhat-is-register-in-cpu-how-does-it-work%2F%23%3A~%3Atext%3DRegisters%2520are%2520a%2520type%2520of%2Cbit%2520sequence%2520or%2520individual%2520characters).&usg=AOvVaw3z_zC6FXtRMhrJw79a7Qte&opi=89978449

[5] https://www.google.com/url?sa=i&url=http%3A%2F%2Fwww.ee.ic.ac.uk%2Fpcheung%2Fteaching%2Fee2_digital%2FLecture%252013%2520-%2520FPGA%2520Embedded%2520Memory.pdf&psig=AOvVaw0BtvgGNSx14YYP-NHhAAW_&ust=1705504575536000&source=images&cd=vfe&opi=89978449&ved=0CBMQjRxqFwoTCNjhreaZ4oMDFQAAAAAdAAAAABAH

5. Appendix:

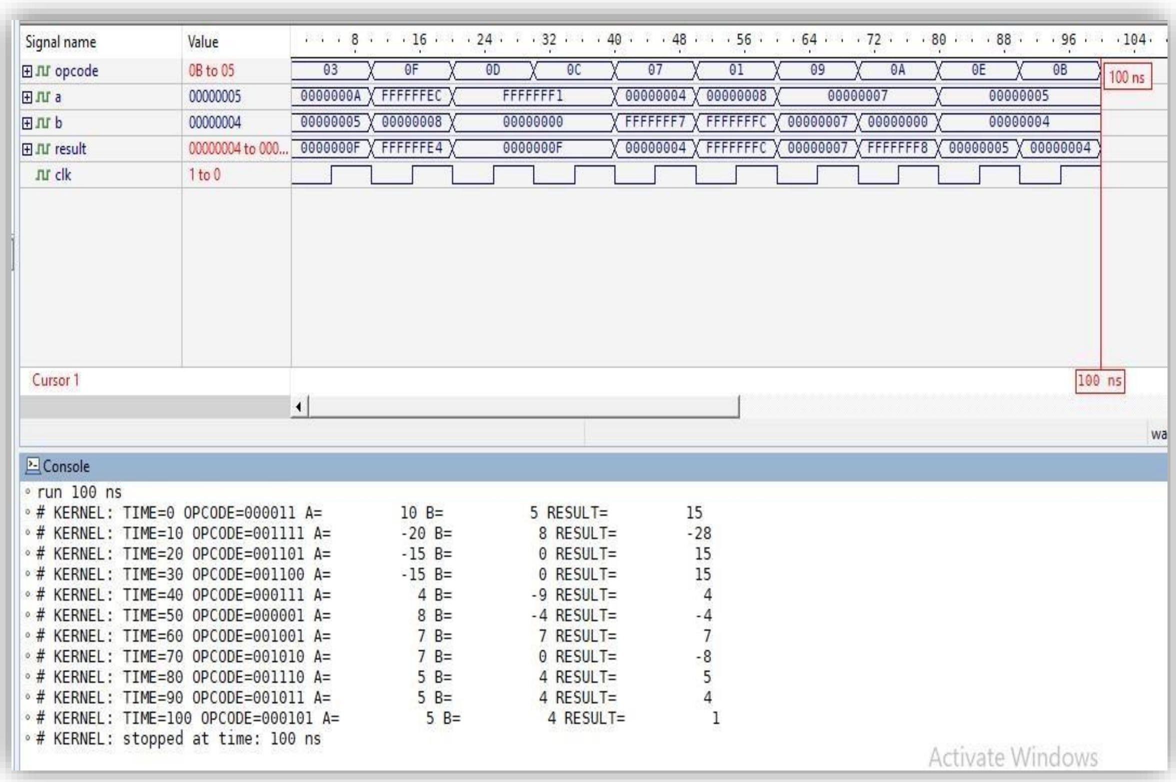
Code Snippet 1: Test Bench for ALU Module

```
`timescale 1ns/1ns


module alu_tb;

 reg [5:0] opcode;

 reg signed [31:0] a, b;

 wire signed [31:0] result;


 alu ALU(opcode, a, b, result);


 reg clk = 0;

 always #5 clk = ~clk;


 initial begin

  $monitor("TIME=%0t OPCODE=%b A=%d B=%d RESULT=%d", $time, opcode, a, b, result);


  // ADDITION TEST

  opcode = 6'b000011;

  a = 10;

  b = 5;

  #10;


  // SUBTRACTION TEST

  opcode = 6'b001111;

  a = -20;
```

```
b = 8;

#10;


// ABSOLUTE TEST

opcode = 6'b001101;

a = -15;

b = 0;

#10;


// INVERSE TEST

opcode = 6'b001100;

a = -15;

b = 0;

#10;


// MAX TEST

opcode = 6'b000111;

a = 4;

b = -9;

#10;


// MIN TEST

opcode = 6'b000001;

a = 8;

b = -4;

#10;
```

```
// AVERAGE TEST

opcode = 6'b001001;

a = 7;

b = 7;

#10;


// BIT-WISE NOT TEST

opcode = 6'b001010;

a = 7;

b = 0;

#10;


// BIT-WISE OR TEST

opcode = 6'b001110;

a = 5;

b = 4;

#10;


// BIT-WISE AND TEST

opcode = 6'b001011;

a = 5;

b = 4;

#10;


// BIT-WISE XOR TEST

opcode = 6'b000101;

a = 5;
```

```
    b = 4;

    #10;


            // INVALID TEST

    opcode = 6'b111111;

    a = 0;

    b = 0;

    #10;


    $finish;

  end

  endmodule
```



Signal name	Value
opcode	0B to 05
a	00000005
b	00000004
result	00000004 to 000...
clk	1 to 0

8 16 24 32 40 48 56 64 72 80 88 96 104
03 0F 0D 0C 07 01 09 0A 0E 0B
0000000A FFFFFFEC FFFFFFF1 00000004 00000008 00000007 00000005
00000005 00000008 00000000 FFFFFFF7 FFFFFFFC 00000007 00000000 00000004
0000000F FFFFFFE4 0000000F 00000004 FFFFFFFC 00000007 FFFFFFF8 00000005 00000004
100 ns
Cursor 1
100 ns
Console
Activate Windows

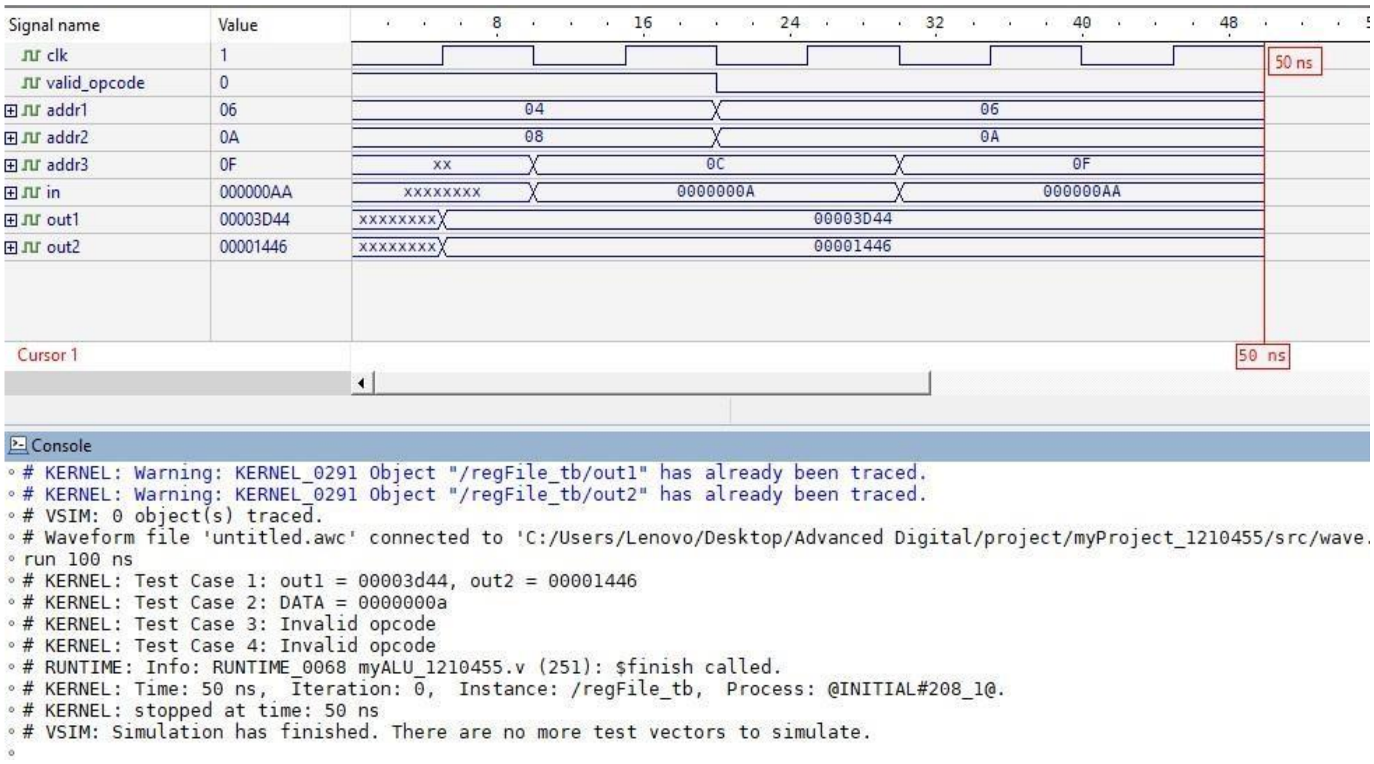
```
run 100 ns
# KERNEL: TIME=0 OPCODE=000011 A=         10 B=          5 RESULT=         15
# KERNEL: TIME=10 OPCODE=001111 A=        -20 B=          8 RESULT=        -28
# KERNEL: TIME=20 OPCODE=001101 A=        -15 B=          0 RESULT=         15
# KERNEL: TIME=30 OPCODE=001100 A=        -15 B=          0 RESULT=         15
# KERNEL: TIME=40 OPCODE=000111 A=          4 B=         -9 RESULT=          4
# KERNEL: TIME=50 OPCODE=000001 A=          8 B=         -4 RESULT=         -4
# KERNEL: TIME=60 OPCODE=001001 A=          7 B=          7 RESULT=          7
# KERNEL: TIME=70 OPCODE=001010 A=          7 B=          0 RESULT=         -8
# KERNEL: TIME=80 OPCODE=001110 A=          5 B=          4 RESULT=          5
# KERNEL: TIME=90 OPCODE=001011 A=          5 B=          4 RESULT=          4
# KERNEL: TIME=100 OPCODE=000101 A=          5 B=          4 RESULT=          1
# KERNEL: stopped at time: 100 ns
```

Figure 4. 1: Results for ALU module.

Code Snippet 2: Test Bench for Register File Module

```
`timescale 1ns/1ns

module regFile_tb;

reg clk;
  reg valid_opcode;
  reg [4:0] addr1, addr2, addr3;
  reg [31:0] in;
  wire [31:0] out1, out2;

  reg_file RF(clk, valid_opcode, addr1, addr2, addr3, in, out1, out2);

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    //          Test
    initial begin
        // Test Case 1: Read operation with valid_opcode
                    valid_opcode = 1;
                    addr1 = 5'b00100; // address 4
                    addr2 = 5'b01000; // address 8
                    #10;
                    if (valid_opcode)
                      $display("Test Case 1: out1 = %h, out2 = %h", out1, out2);  //out1=value at
address 4, out2=value at address 8
                    else
                      $display("Test Case 1: Invalid opcode");

                    // Test Case 2: Write operation with valid_opcode
                    valid_opcode = 1;
                    addr3 = 5'b01100; // address 12
                    in = 32'h0000000A;
                    #10;
                    if (valid_opcode)
                      $display("Test Case 2: DATA = %h", in);  //in=data written in address 12
                    else
                      $display("Test Case 2: Invalid opcode");

                    // Test Case 3: Read operation with invalid_opcode
                    valid_opcode = 0;
                    addr1 = 5'b00110; // address 6
                    addr2 = 5'b01010; // address 10
                    #10;
                    if (valid_opcode)
                      $display("INVALID OPCODE");
                    else
```

```
            $display("Test Case 3: Invalid opcode");

          // Test Case 4: Write operation with invalid_opcode
          valid_opcode = 0;
          addr3 = 5'b01111; // address 15
          in = 32'h000000AA;
          #10;
          if (valid_opcode)
            $display("INVALID OPCODE");
          else
            $display("Test Case 4: Invalid opcode");


    // Finish the simulation
    #10 $finish;
  end

endmodule
```



```
Signal name     Value
clk             1
valid_opcode    0
addr1           06
addr2           0A
addr3           0F
in              000000AA
out1            00003D44
out2            00001446
Cursor 1        50 ns
```

```
Console
# KERNEL: Warning: KERNEL_0291 Object "/regFile_tb/out1" has already been traced.
# KERNEL: Warning: KERNEL_0291 Object "/regFile_tb/out2" has already been traced.
# VSIM: 0 object(s) traced.
# Waveform file 'untitled.awc' connected to 'C:/Users/Lenovo/Desktop/Advanced Digital/project/myProject_1210455/src/wave.
run 100 ns
# KERNEL: Test Case 1: out1 = 00003d44, out2 = 00001446
# KERNEL: Test Case 2: DATA = 0000000a
# KERNEL: Test Case 3: Invalid opcode
# KERNEL: Test Case 4: Invalid opcode
# RUNTIME: Info: RUNTIME_0068 myALU_1210455.v (251): $finish called.
# KERNEL: Time: 50 ns,  Iteration: 0,  Instance: /regFile_tb,  Process: @INITIAL#208_1@.
# KERNEL: stopped at time: 50 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
```

Figure 4. 2: Results for reg_file Module