Faculty of Engineering & Technology
Electrical & Computer Engineering Department

Advanced Digital Design

ENC3310

Course Project

Student Name : Shahd Yahya
Student ID : 1210249

Instructor: Dr.Abdellatif Abu-Issa
Section: 1

Abstract:

This project involves the design and implementation of a microprocessor core, consisting of an Arithmetic Logic Unit (ALU) and a Register File, with specific configurations based on the student's ID. The ALU performs various operations determined by a 6-bit opcode, linked to the last digit of the student ID. The Register File, synchronized to a clock, holds initial values based on the second-from-last digit of the student ID. An enable signal ensures controlled updating of the Register File.

The microprocessor core is assembled by connecting the ALU and Register File. Machine instructions, encoded as 32-bit numbers, dictate operations on registers. The testbench evaluates the correctness and timing of instructions, considering the relationship between clock cycles and result availability. Overall, the project aims to demonstrate a personalized and functional microprocessor design.

Table of content:

Table of figures:

1 Theory:

1.1 Introduction to the problem

Our task is to build a simple part of a microprocessor. Firstly, you will build two main blocks: the ALU and the register file, then you will connect them together and run a simple machine code program on them.After that, we will have to write a testbench to verify that the design is working properly

1.2 Microprocessor

A Microprocessor is an important part of a computer architecture without which you will not be able to perform anything on your computer. It is a programmable device that takes in input, performs 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. [1]

A Microprocessor takes a bunch of instructions in machine language and executes them, telling the processor what it has to do. Microprocessor performs three basic things while executing the instruction:

1 - It performs some basic operations like addition, subtraction, multiplication, division, and some logical operations using its Arithmetic and Logical Unit (ALU). New Microprocessors also perform operations on floating-point numbers also.

2 - Data in microprocessors can move from one location to another.

3 - It has a Program Counter (PC) register that stores the address of the next instruction based on the value of the PC. The Microprocessor jumps from one location to another and makes decisions.

1.2.1 Alu

An arithmetic logic unit (ALU) is a key component of a computer's central processor unit. The ALU performs all arithmetic and logic operations that must be performed on instruction words. The ALU is split into two parts in some microprocessor architectures: the AU and the LU. [2]

1.2.2 register file

A register file is a means of memory storage within a computer's central processing unit (CPU). The computer's register files contain bits of data and mapping locations. These locations specify certain addresses that are input components of a register file. Other inputs include data, a read and write function and execute function. [3]

2 Design Philosophy

2.1 Alu design

Alu is a basic component in a microprocessor .It receives two 32-bit inputs and a 6-bit opcode to specify the operation, producing a 32-bit output that represents the result of a specified operation. The choice of opcode for each operation is determined by the last digit of the student's ID number, as indicated in the table below.(my id is 1210249 so the last digit is 9).

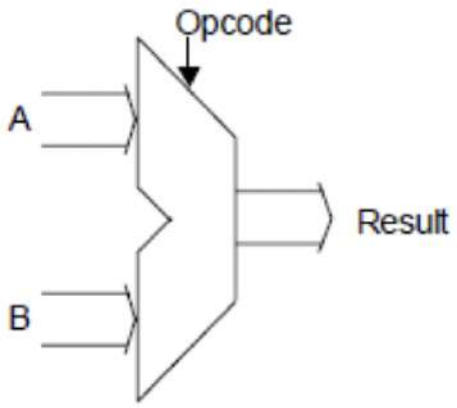

Fig 1 : alu design

Operation	Digit 9 opcodes
a + b	5
a – b	8
|a|	13
-a	7
max(a,b)	3
min (a,b)	6
avg(a,b)	10
~a	2
a or b	15
a and b	4
a xor b	12

Fig 2 : operations opcodes

And here is the code of alu module:

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

    always @(*) begin
            case (opcode) // choose the operation based on the opcode
                6'b000101: result <= a + b;
                6'b001000: result <= a - b;
                6'b001101: result <= (a < 0) ? -a : a;
                6'b000111: result <= -a;
                6'b000011: result <= (a > b) ? a:b;
                6'b000110: result <= (a < b) ? a:b;
                6'b001010: result <= (a + b)/2;
                6'b000010: result <= ~a;
                6'b001111: result <= a | b;
                6'b000100: result <= a & b;
                6'b001100: result <= a ^ b;

            endcase
        end
endmodule
```

Fig 3 : alu module code

The Verilog module alu is a basic Arithmetic Logic Unit (ALU). With a 6-bit opcode and two 32-bit inputs (a and b), it performs operations like addition, subtraction, absolute value, negation, maximum, minimum, average, bitwise NOT, OR, AND, and XOR. The result is output as a 32-bit value.

I built a testbench for alu module:

```
module alu_test;
  reg [5:0] opcode;
  reg signed [31:0] a, b;
  wire signed [31:0] result;
  reg [5:0] opcodeArray [13:0];
  int i;
  reg [31:0] expectedResult;
  string operation, isPassed;
  int passOrFail = 1;

  // ALU instantiation
  alu Alu(opcode, a, b, result);

  initial begin
    $display("opcode      a           b      operation   expected result   result       test");
    $display("-------------------------------------------------------------------------------------");
    i = 0;
    a = 32'h1066;
    b = 32'h15DC;
    opcodeArray[0] = 6'b000101;  // a + b
    opcodeArray[1] = 6'b001000;  // a - b
    opcodeArray[2] = 6'b001101;  // |a|
    opcodeArray[3] = 6'b000111;  // -a
    opcodeArray[4] = 6'b000011;  // max(a,b)
    opcodeArray[5] = 6'b000110;  // min(a,b)
    opcodeArray[6] = 6'b001010;  // avg(a,b)
    opcodeArray[7] = 6'b000010;  // ~a
    opcodeArray[8] = 6'b000001;  // instruction of non-valid opcode
    opcodeArray[9] = 6'b001111; // a | b
    opcodeArray[10] = 6'b001001; // instruction of non-valid opcode
    opcodeArray[11] = 6'b000100; // a & b
    opcodeArray[12] = 6'b001011; // instruction of non-valid opcode
    opcodeArray[13] = 6'b001100; // a ^ b

    for (i = 0; i <= 13; i = i + 1) begin
      #10ns opcode = opcodeArray[i];
    end
    #10ns i = i + 1;
    // Check if the program passes or not
    if (passOrFail == 0) $display("The Program Failed");
    else $display("The Program Passed");
  end

  // Compare results
  always @(i) begin
    if (i <= 14) begin
        #1 // delay
      // Find the expected result
      case (opcodeArray[i - 1])
        6'b000101: begin expectedResult = 32'h00002642; operation = " a + b   "; end
        6'b001000: begin expectedResult = 32'hFFFFFA8A; operation = " a - b   "; end
        6'b001101: begin expectedResult = 32'h00001066; operation = "  |a|    "; end
        6'b000111: begin expectedResult = 32'hFFFFEF9A; operation = "  -a     "; end
        6'b000011: begin expectedResult = 32'h000015DC; operation = "max(a,b) "; end
        6'b000110: begin expectedResult = 32'h00001066; operation = "min(a,b) "; end
        6'b001010: begin expectedResult = 32'h00001321; operation = "avg(a,b) "; end
        6'b000010: begin expectedResult = 32'hFFFFEF99; operation = "  ~a     "; end
        6'b001111: begin expectedResult = 32'h000015FE; operation = "   or    "; end
        6'b000100: begin expectedResult = 32'h00001044; operation = "  and    "; end
        6'b001100: begin expectedResult = 32'h000005BA; operation = "  xor    "; end
         default : begin  operation = "non valid";   end
      endcase

      // Compare the result with the expected result for each opcode
       if (result == expectedResult && operation != "non valid")
        isPassed = "pass";
      else if (result == expectedResult && operation == "non valid")
        isPassed = "not valid(the result remain the same)";
      else begin
        isPassed = "fail";
        passOrFail = 0;
        end

      $display("%b   %h    %h   %s     %h      %h      %s", opcodeArray[i - 1], a, b, operation, expectedResult, result, isPassed);
      isPassed = "";
    end
  end
endmodule
```

Fig 4 : alu testbench code

And then I run the simulation:

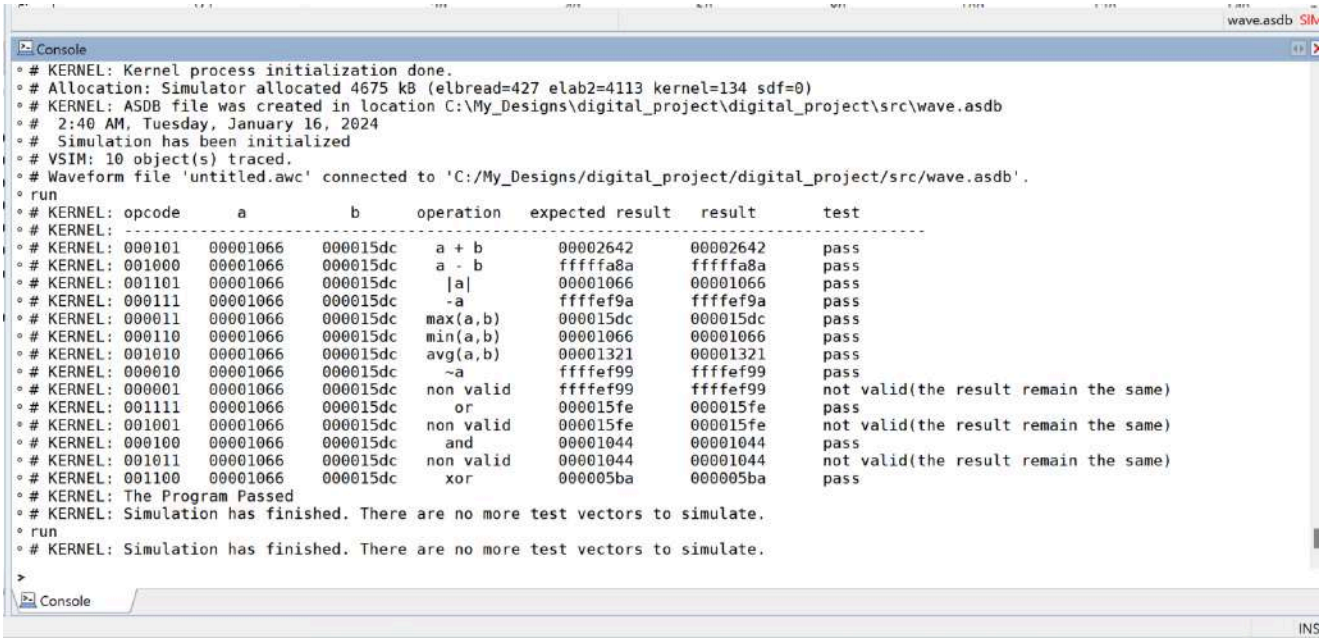

Fig 5 : alu simulation result

And here is the waveform of the alu test bench:

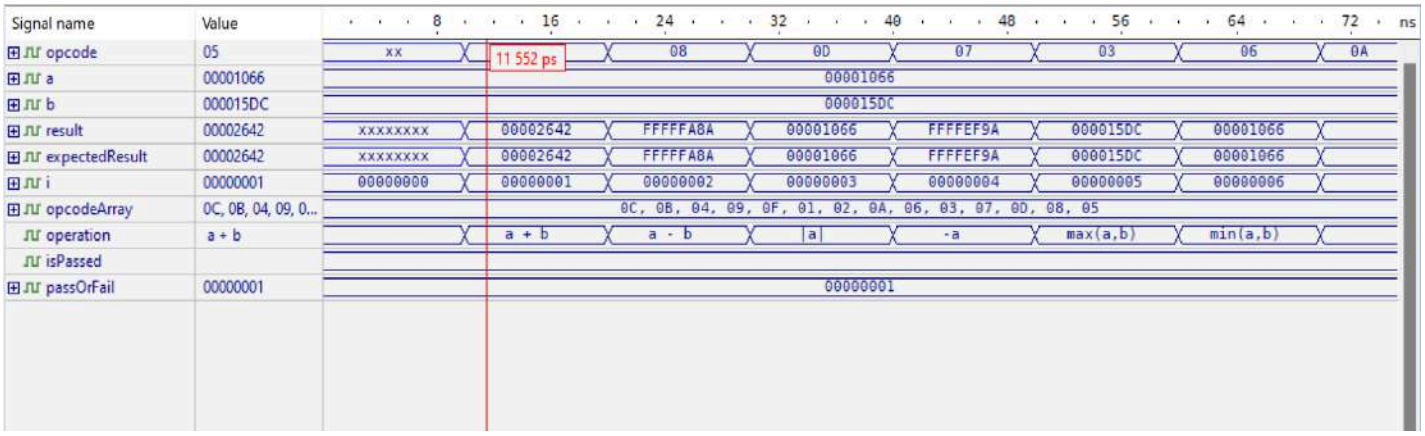

Fig 6 : alu waveform

In the test bench for alu module I created an array of opcodes including at least one opcode for each of the provided opcodes to test them and I calculated the result manually and compared it to the actual result (when the opcode is invalid the result remains the same).

2.2 Register File design

The register file in a modern processor is a small, fast RAM with 32 x 32-bit words, requiring a 5-bit address. It processes three addresses simultaneously (two reads, one write). Output 1 and Output 2 of 32 bit retrieve content based on Address 1 and Address 2, while Input writes a value to the location specified by Address 3.
Initial values in the register file are determined by the second-to-last digit of the student ID. (my id is 1210249 and the second-from-last digit is 4).

ID/ Location	4
0	0
1	4198
2	5596
3	14426
4	7612
5	6638
6	10040
7	3930
8	4150
9	6406
10	5400
11	8572
12	16324
13	8840
14	8258
15	11228
16	8462
17	13284
18	4676
19	3980
20	5634
21	7632
22	9846
23	5442
24	12488
25	6656
26	832
27	4664
28	6798
29	14166
30	3246
31	0

Fig 8 : reg file initial values

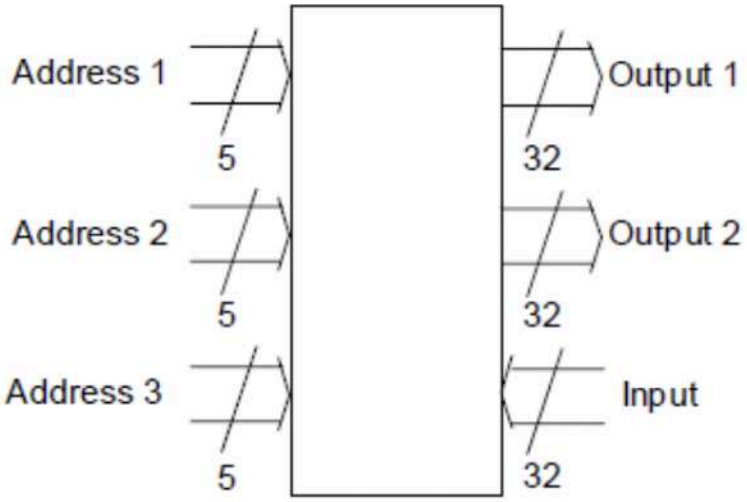

Fig 7 : reg file design

And here is the code of reg file module:

```
module reg_file(clk,valid_opcode,addr1, addr2, addr3, in, out1, out2);
    input clk;
    input [5:0] valid_opcode;
    input [4:0] addr1, addr2, addr3;
    input [31:0] in;
    output reg [31:0] out1,out2;
    reg [31:0]mem [31:0];

    initial begin
        // Fill the initial values of the reg file
            mem[0]  = 32'h0;
            mem[1]  = 32'h1066;
            mem[2]  = 32'h15DC;
            mem[3]  = 32'h385A;
            mem[4]  = 32'h1DBC;
            mem[5]  = 32'h19EE;
            mem[6]  = 32'h2738;
            mem[7]  = 32'hF5A;
            mem[8]  = 32'h1036;
            mem[9]  = 32'h11FE;
            mem[10] = 32'h1518;
            mem[11] = 32'h217C;
            mem[12] = 32'h3FC4;
            mem[13] = 32'h2288;
            mem[14] = 32'h2042;
            mem[15] = 32'h2BDC;
            mem[16] = 32'h210E;
            mem[17] = 32'h33E4;
            mem[18] = 32'h1244;
            mem[19] = 32'hF8C;
            mem[20] = 32'h1602;
            mem[21] = 32'h1DD0;
            mem[22] = 32'h2676;
            mem[23] = 32'h1542;
            mem[24] = 32'h30C8;
            mem[25] = 32'h1A00;
            mem[26] = 32'h340;
            mem[27] = 32'h1238;
            mem[28] = 32'h1ABE;
            mem[29] = 32'h3756;
            mem[30] = 32'hCAE;
            mem[31] = 32'h0;
        end

    always @(posedge clk) begin
    // make sure that the opcode is valid
    if (valid_opcode >= 2 && valid_opcode <= 15 && valid_opcode != 9 && valid_opcode != 11 && valid_opcode != 14) begin
                    out1 <= mem[addr1];
                    out2 <= mem[addr2];
                    mem[addr3] <= in;

    end
end

endmodule
```

Fig 9 : reg file module code

In this code I made sure that the opcode is valid before making any change.(the opcode restricts in numbers 2,3,4,5,6,7,8,10,12,13 and 15).

The register file, initially a combinational circuit, encounters issues when addresses overlap. To address this, synchronization to a clock is introduced. On the clock's rising edge, Output 1 and Output 2 fetch data from addresses specified by Address 1 and Address 2, respectively. Simultaneously, Input writes a value into the location pointed to by Address 3. Outside these rising edges, outputs remain constant based on their values during the last clock rising edge.

I built a testbench for reg file module:

```
module reg_file_test;
    reg clk;
    reg [5:0] validOpcode;
    reg [4:0] addr1,addr2,addr3;
    reg signed [31:0] in;
    wire signed [31:0] regOut1,regOut2;

    reg_file regFile(clk,validOpcode,addr1,addr2,addr3,in,regOut1,regOut2);

   integer i;

    initial begin
    clk = 0;
    for (i = 0; i < 7; i = i + 1) begin
        #5ns;
        clk = ~clk;
    end
  end

    initial begin
        $display("opcode       addr1      addr2          Out1            Out2 ");
        $display("-------------------------------------------------------------");
        clk = 0;
        validOpcode = 6'b000101;
        addr1 = 8;
        addr2 = 10;
        addr3 = 0;
        in = 32'h55;
        #10ns
        validOpcode = 6'b001000;
        addr1 = 0;
        addr2 = 9;
        addr3 = 2;
        in = 32'h42;
        #10ns
        validOpcode = 6'b000010;
        addr1 = 2;
        addr2 = 4;
        addr3 = 31;
        in = 32'h33;
        #10ns
        validOpcode = 6'b011111;
        addr1 = 15;
        addr2 = 31;
        addr3 = 0;
        in = 32'h11;
        end

        always@(posedge clk) begin
            #1ns
         $display("%b        %h        %h          %h          %h", validOpcode, addr1, addr2, regOut1,regOut2);
            end
endmodule
```

Fig 10 : reg file testbench code

And then I run the simulation:

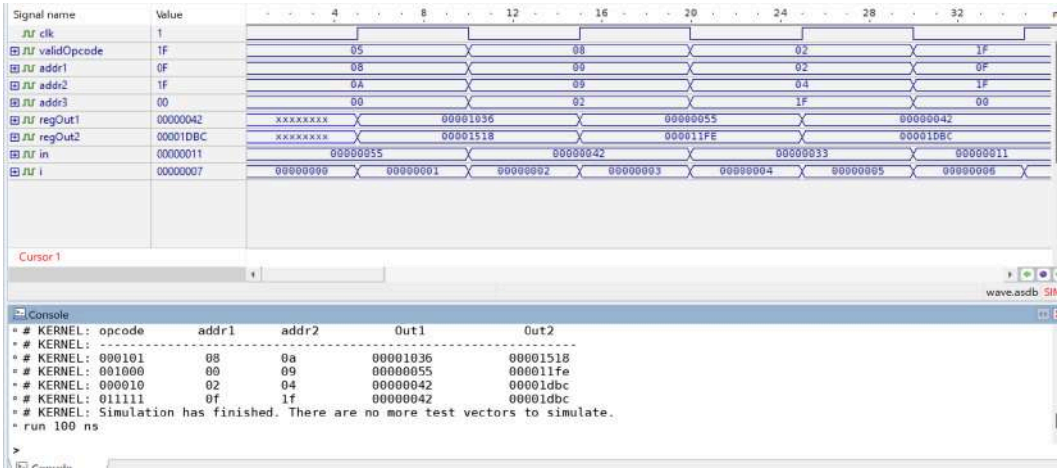

Fig 11 : reg file simulation and waveform

On the initial positive clock edge, the opcode was 5, which is a valid opcode. The values in out1 and out2 correspond to the data located in addresses specified by addr1 and addr2. Additionally, during the first clock cycle, a value of 32'h55 was written to memory location 0, and it was successfully read in the subsequent cycle. Similarly, in the second clock cycle, the value 32'h42 was written, and it was read in the third cycle. The lack of changes in values during the last positive clock edge can be attributed to the presence of a non-valid opcode in that instance.

2.3 other modules design

2.3.1 Instruction register design

```
 //this module to instantiate opcode ,addr1,addr2 and addr3
module instruction_reg(clk,machineInstruction,addr1,addr2,addr3,opcode);
     input clk;
     input [31:0]machineInstruction;
     output reg [4:0] addr1,addr2,addr3;
     output reg [5:0] opcode;
     always@(posedge clk) begin
               opcode <= machineInstruction[5:0];    // opcode is the first 6 bits of the instruction
               addr1 <= machineInstruction[10:6];    // adrr1 is the next 5 bits of the instruction
               addr2 <= machineInstruction[15:11];  // adrr2 is the next 5 bits of the instruction
               addr3 <= machineInstruction[20:16];    // adrr3 is the next 5 bits of the instruction
         end
endmodule
```

Fig 12 : Instruction reg code

The Verilog module instruction_reg takes a 32-bit machine instruction and, on each rising edge of the clock (clk), extracts the opcode, addr1, addr2, and addr3, storing them in separate registers. The opcode is 6 bits, while addr1, addr2, and addr3 are 5 bits each. This module ensures synchronous and reliable extraction of instruction components for further processing in a digital system.

2.3.2 opcode delay design

```

// this module to delay the opcode before entering the alu
module opcode_delay(clk,opcode1,opcode);
     input clk;
     input [5:0] opcode;
     output reg [5:0] opcode1;
     always@(posedge clk) begin
               opcode1 <= opcode;
          end


endmodule

```

Fig 13 : opcode delay code

The Verilog module opcode_delay introduces a one-clock cycle delay to the input opcode before it is output as opcode1. This delay is synchronized with the positive edge of the clock (clk). The module is designed to align the opcode timing as it enters the Arithmetic Logic Unit (ALU) in a digital system.

2.4 Microprocessor design

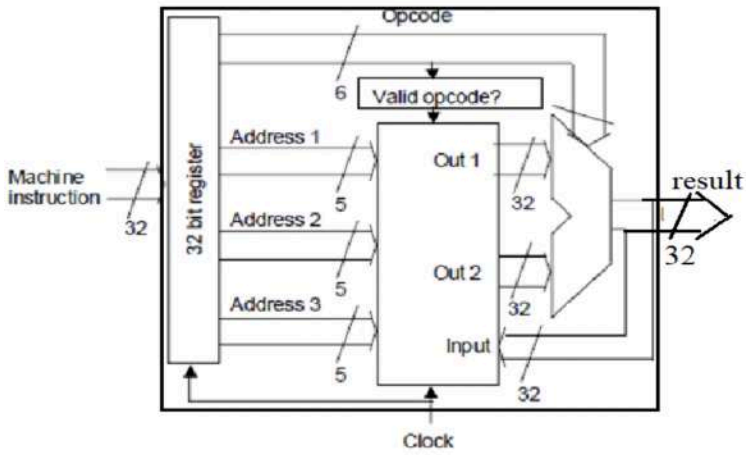

Fig 14 : microprocessor design

```
// module of microprocesses
module mp_top (clk,instruction, result);
    input clk;
    input [31:0] instruction;
    output reg [31:0] result;
    reg [4:0] addr1,addr2,addr3;
    reg [31:0] regFile_out1,regFile_out2;
    reg[5:0] opcode,opcode1; // opcode1 is the delayed version of opcode that enters alu


    instruction_reg instructionReg(.clk(clk),.machineInstruction(instruction),.addr1(addr1),.addr2(addr2),.addr3(addr3),.opcode(opcode));
    reg_file regFile(.clk(clk),.valid_opcode(opcode),.addr1(addr1),.addr2(addr2),.addr3(addr3),.in(result),.out1(regFile_out1),.out2(regFile_out2));
    opcode_delay opcode_delay1(clk,opcode1,opcode);
    alu Alu(.opcode(opcode1),.a(regFile_out1),.b(regFile_out2),.result(result));

endmodule
```

Fig 15 : microprocessor code

The mp_top Verilog module represents the top-level design of a microprocessor. It coordinates various components to execute instructions. Here's a breakdown of its functionality:

1-Instruction Processing:
The instructionReg module extracts information from the 32-bit instruction, setting relevant registers (addr1, addr2, addr3, opcode).

2-Register File:
The regFile module manages a register file, reading and writing data based on addresses (addr1, addr2, addr3) and a valid opcode.

3-Opcode Delay:
The opcode_delay1 module introduces a one-clock cycle delay to the opcode (opcode1), facilitating proper synchronization with other components.

4-ALU Operation:
The alu module (Arithmetic Logic Unit) performs operations based on the delayed opcode (opcode1) and data from the register file (regFile_out1, regFile_out2), producing the final result.

3 Result

I built a test bench for the microprocessor module:

```
module mp_top_test;
  reg clk;
  reg [31:0] instruction;
  wire signed [31:0] result;
  reg [31:0] instructionsArray [14:0] ; //array of instruction
  reg signed [31:0] a,b;
  int i;  //array of instruction index
  reg signed [31:0] expectedResult;
  string opration,isPassed;
  int passOrFail = 1;
  initial begin
      $display("          instruction              a          b      operation   expected result    result    test");
      $display("-------------------------------------------------------------------------------------------------------");
      clk = 0;
      i = 0;
      a = 32'h1066;
      b = 32'h15DC;
      // Fill array of instruction
      instructionsArray[0] = 32'b00000000000000011000100000001000101; // a + b
      instructionsArray[1] = 32'b00000000000000011000100000001001000; // a - b
      instructionsArray[2] = 32'b00000000000000011000100000001001101; // |a|
      instructionsArray[3] = 32'b00000000000000011000100000001000111; // -a
      instructionsArray[4] = 32'b00000000000000011000100000001000011; // max(a,b)
      instructionsArray[5] = 32'b00000000000000011000100000001000110; // min(a,b)
      instructionsArray[6] = 32'b00000000000000011000100000001001010; // avg(a,b)
      instructionsArray[7] = 32'b00000000000000011000100000001000010; // ~a
      instructionsArray[8] = 32'b00000000000000011000100000001000001; //instuction of non valid opcode
      instructionsArray[9] = 32'b00000000000000011000100000001001111; // a | b
      instructionsArray[10] = 32'b00000000000000011000100000001001001; //instuction of non valid opcode
      instructionsArray[11] = 32'b00000000000000011000100000001000100; // a & b
      instructionsArray[12] = 32'b00000000000000011000100000001001011; //instuction of non valid opcode
      instructionsArray[13] = 32'b00000000000000011000100000001001100; // a ^ b
      instructionsArray[14] = 32'b00000000000000011000100000001001110; //instuction of non valid opcode

      // Try all instructions
      for(i = 0; i <= 14; i++)begin
          #20ns instruction = instructionsArray[i];
      end
      #20ns i++;

      // check if the program passes or not
        if(passOrFail == 0) $display("The Program Failed");
          else $display("The Program Passed");
      end

  mp_top mp(clk,instruction, result);


  always #5ns begin
      clk = ~clk;
  end

  always@(i) begin
    if(i <= 15) begin
    // find the expected result
    #17ns
     case(instructionsArray[i - 1][5:0])
                6'b000101: begin expectedResult = 32'h00002642;   opration = " a + b    ";   end
                6'b001000: begin expectedResult = 32'hFFFFFA8A;   opration = " a - b    ";   end
                6'b001101: begin expectedResult = 32'h00001066;   opration = "  |a|     ";   end
                6'b000111: begin expectedResult = 32'hFFFFEF9A;   opration = "  -a      ";   end
                6'b000011: begin expectedResult = 32'h000015DC;   opration = "max(a,b) ";   end
                6'b000110: begin expectedResult = 32'h00001066;   opration = "min(a,b) ";   end
                6'b001010: begin expectedResult = 32'h00001321;   opration = "avg(a,b) ";   end
                6'b000010: begin expectedResult = 32'hFFFFEF99;   opration = "  ~a      ";   end
                6'b001111: begin expectedResult = 32'h000015FE;   opration = "   or     ";   end
                6'b000100: begin expectedResult = 32'h00001044;   opration = "  and     ";   end
                6'b001100: begin expectedResult = 32'h000005BA;   opration = "  xor     ";   end
                default : begin  opration = "non valid";   end

     endcase
     // compare the result with the expected result for each instruction
     if (result == expectedResult && opration != "non valid")
        isPassed = "pass";
     else if (result == expectedResult && opration == "non valid")
        isPassed = "not valid(the result remain the same)";
     else begin
        isPassed = "fail";
        passOrFail = 0;
        end

      $display("%b  %h   %h    %s      %h      %h   %s",instructionsArray[i - 1],a,b,opration,expectedResult,result,isPassed);
      isPassed = "";
        end
          end


endmodule
```

Fig 16 : microprocessor testbench code

This module is the test bench for mp_top and created an array of instructions including at least one instruction for each of the provided opcodes to test them and I calculated the result manually and compared it to the actual result.

At t = 25 ns the clk moves from 0 to 1 (positive edge) and after a 2 clock cycle at t = 35ns the result is available.

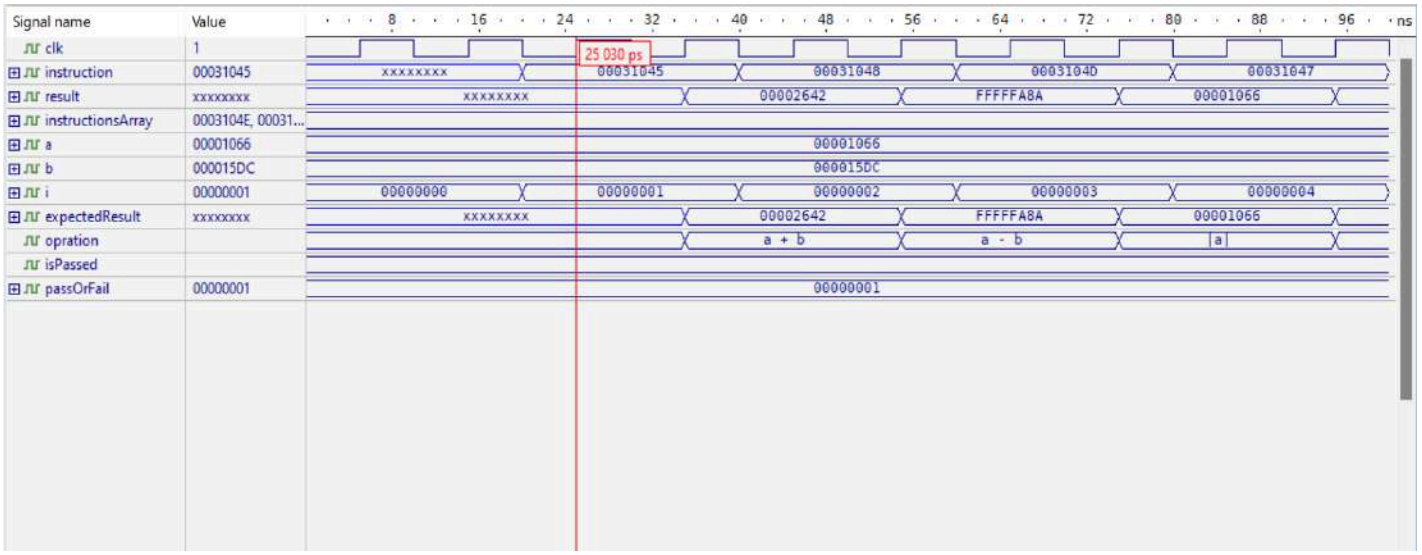

Fig 17 : microprocessor waveform1

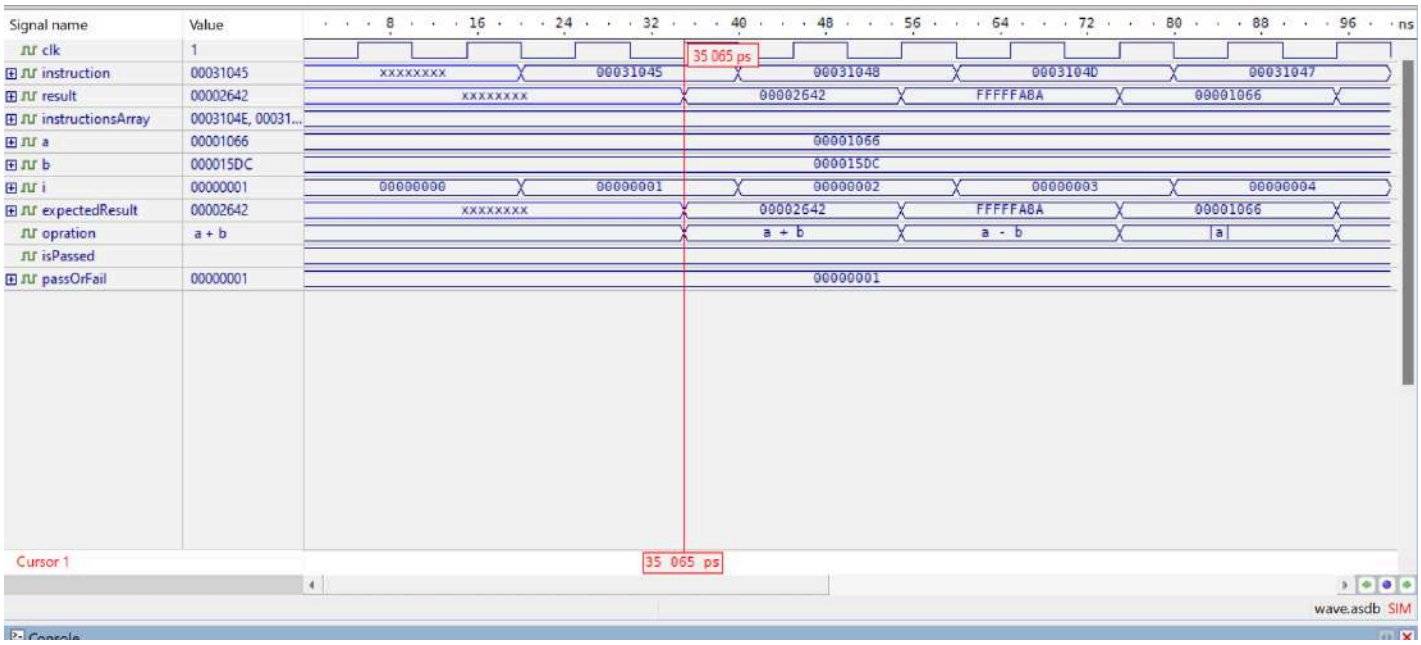

Fig 18 : microprocessor waveform2

Then printed values of the results:

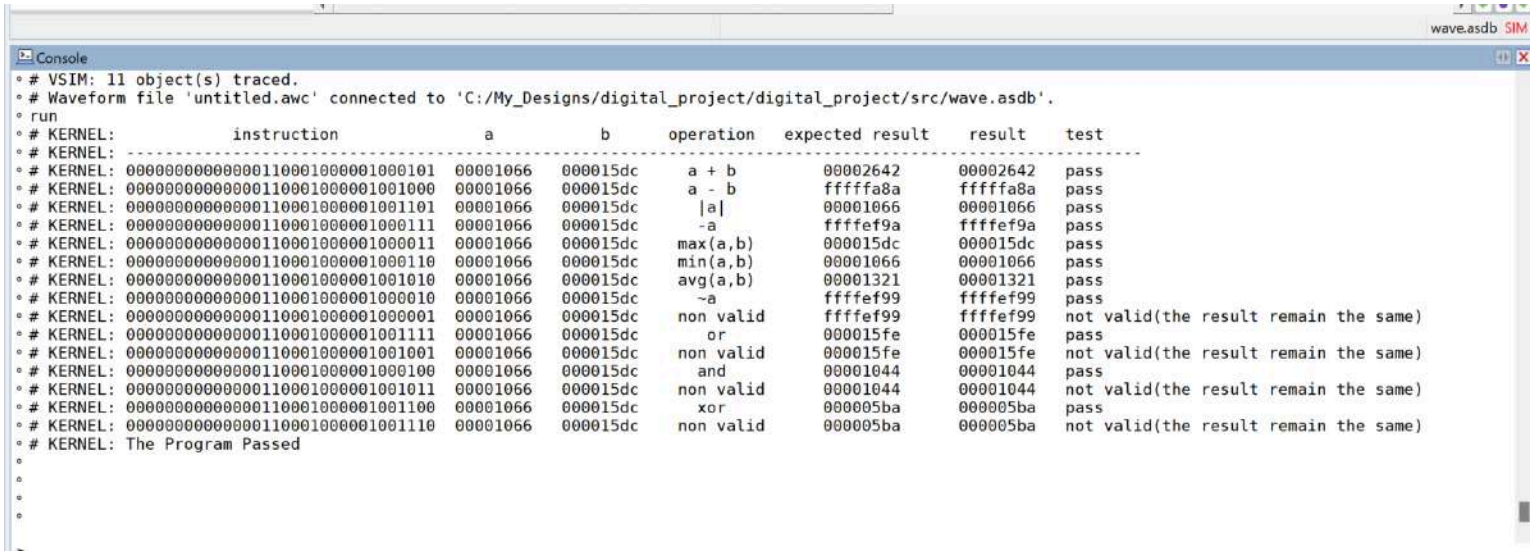

Fig 19 : microprocessor simulation

4 Conclusion

To sum it up, this project successfully created a simple computer chip called a microprocessor. We built important parts like the Arithmetic Logic Unit (ALU) and the register file. These components were carefully connected to perform tasks like addition, subtraction, and other operations.

We tested each part to make sure everything works as expected. The project also involved creating a microprocessor that coordinates these parts to execute instructions. We used a systematic approach to design and thoroughly checked each module to ensure everything runs smoothly.

The testing process involved running simulations to see how each part behaves and making sure they all work well together. This project is not just about writing code; it's about making sure the microprocessor works correctly by testing it thoroughly.

In conclusion, this project helps us understand how microprocessors work and provides a good foundation for more advanced projects in computer design. The systematic approach and testing methods used here are important for building reliable and functional computer systems

5 Reference

[1] https://www.geeksforgeeks.org/introduction-of-microprocessor/

[2] https://byjus.com/gate/alu-notes/#:~:text=An%20arithmetic%20logic%20unit%20(ALU,the%20AU%20and%20the%20LU.

[3] https://www.easytechjunkie.com/what-is-a-register-file.htm

6 Appendix

(Alu module code):

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

	always @(*) begin
		case (opcode)  // choose the operation based on the opcode
			6'b000101: result <= a + b;
			6'b001000: result <= a - b;
			6'b001101: result <= (a < 0) ? -a : a;
			6'b000111: result <= -a;
			6'b000011: result <= (a > b) ? a:b;
			6'b000110: result <= (a < b) ? a:b;
			6'b001010: result <= (a + b)/2;
			6'b000010: result <= ~a;
			6'b001111: result <= a | b;
			6'b000100: result <= a & b;
			6'b001100: result <= a ^ b;

		endcase
	end
endmodule
```

(Reg File module code):

```
module reg_file(clk,valid_opcode,addr1, addr2, addr3, in, out1, out2);
	input clk;
	input [5:0] valid_opcode;
	input [4:0] addr1, addr2, addr3;
	input signed [31:0] in;
	output reg signed [31:0] out1,out2;
	reg signed [31:0]mem [31:0];

	initial begin
		// Fill the reg file
			mem[0]  = 32'h0;
			mem[1]  = 32'h1066;
			mem[2]  = 32'h15DC;
			mem[3]  = 32'h385A;
```

```
                mem[4]  = 32'h1DBC;
                mem[5]  = 32'h19EE;
                mem[6]  = 32'h2738;
                mem[7]  = 32'hF5A;
                mem[8]  = 32'h1036;
                mem[9]  = 32'h11FE;
                mem[10] = 32'h1518;
                mem[11] = 32'h217C;
                mem[12] = 32'h3FC4;
                mem[13] = 32'h2288;
                mem[14] = 32'h2042;
                mem[15] = 32'h2BDC;
                mem[16] = 32'h210E;
                mem[17] = 32'h33E4;
                mem[18] = 32'h1244;
                mem[19] = 32'hF8C;
                mem[20] = 32'h1602;
                mem[21] = 32'h1DD0;
                mem[22] = 32'h2676;
                mem[23] = 32'h1542;
                mem[24] = 32'h30C8;
                mem[25] = 32'h1A00;
                mem[26] = 32'h340;
                mem[27] = 32'h1238;
                mem[28] = 32'h1A8E;
                mem[29] = 32'h3756;
                mem[30] = 32'hCAE;
                mem[31] = 32'h0;
        end

        always @(posedge clk) begin
        // make sure that the opcode is valid
   if (valid_opcode >= 2 && valid_opcode <= 15 && valid_opcode != 9 && valid_opcode != 11
&& valid_opcode != 14) begin
                                        out1 <= mem[addr1];
                                        out2 <= mem[addr2];
                                        mem[addr3] <= in;
end
end
        endmodule
```

(Microprocessor module code):

```
module mp_top (clk,instruction, result);
        input clk;
        input [31:0] instruction;
        output reg [31:0] result;
        reg [4:0] addr1,addr2,addr3;
        reg [31:0] regFile_out1,regFile_out2;
        reg[5:0] opcode,opcode1; // opcode1 is the delayed version of opcode that enters alu
        instruction_reg
instructionReg(.clk(clk),.machineInstruction(instruction),.addr1(addr1),.addr2(addr2),.addr3(add
r3),.opcode(opcode));
        reg_file
regFile(.clk(clk),.valid_opcode(opcode),.addr1(addr1),.addr2(addr2),.addr3(addr3),.in(result),.ou
t1(regFile_out1),.out2(regFile_out2));
        opcode_delay opcode_delay1(clk,opcode1,opcode);
        alu Alu(.opcode(opcode1),.a(regFile_out1),.b(regFile_out2),.result(result));
endmodule
```

(Instruction Reg module code):

```
module instruction_reg(clk,machineInstruction,addr1,addr2,addr3,opcode);
        input clk;
        input [31:0]machineInstruction;
        output reg [4:0] addr1,addr2,addr3;
        output reg [5:0] opcode;
        always@(posedge clk) begin
                        opcode <= machineInstruction[5:0];  // opcode is the first 6 bits of the
instruction
                        addr1 <= machineInstruction[10:6];   // adrr1 is the next 5 bits of the
instruction
                        addr2 <= machineInstruction[15:11];  // adrr2 is the next 5 bits of the
instruction
                        addr3 <= machineInstruction[20:16];  // adrr3 is the next 5 bits of the
instruction
                end
endmodule
```

(Opcode delay module code):

```
module opcode_delay(clk,opcode1,opcode);
        input clk;
        input [5:0] opcode;
```

```
        output reg [5:0] opcode1;
        always@(posedge clk) begin
                        opcode1 <= opcode;
                end
endmodule
```

(Microprocessor module testbench code):

```
module mp_top_test;
  reg clk;
  reg [31:0] instruction;
  wire signed [31:0] result;
  reg [31:0] instructionsArray [14:0] ; //array of instruction
  reg signed [31:0] a,b;
  int i;  //array of instruction index
  reg signed [31:0] expectedResult;
  string opration,isPassed;
  int passOrFail = 1;
  initial begin
        $display("        instruction        a        b     operation   expected result    result
test");

$display("----------------------------------------------------------------------------------------------------
-");
        clk = 0;
        i = 0;
        a = 32'h1066;
        b = 32'h15DC;
        // Fill array of instruction
        instructionsArray[0] = 32'b00000000000000011000100000100101; // a + b
        instructionsArray[1] = 32'b00000000000000011000100000101000; // a - b
        instructionsArray[2] = 32'b00000000000000011000100000101101; // |a|
        instructionsArray[3] = 32'b00000000000000011000100000100111; // -a
        instructionsArray[4] = 32'b00000000000000011000100000100011; // max(a,b)
        instructionsArray[5] = 32'b00000000000000011000100000100110; // min(a,b)
        instructionsArray[6] = 32'b00000000000000011000100000101010; // avg(a,b)
        instructionsArray[7] = 32'b00000000000000011000100000100010; // ~a
        instructionsArray[8] = 32'b00000000000000011000100000100001; //instuction of non
valid opcode
        instructionsArray[9] = 32'b00000000000000011000100000101111; // a | b
```

```
        instructionsArray[10] = 32'b00000000000000011000100000100100 1; //instuction of non
valid opcode
        instructionsArray[11] = 32'b00000000000000011000100000100010 0; // a & b
        instructionsArray[12] = 32'b00000000000000011000100000100101 1; //instuction of non
valid opcode
        instructionsArray[13] = 32'b00000000000000011000100000100110 0; // a ^ b
        instructionsArray[14] = 32'b00000000000000011000100000100111 0; //instuction of non
valid opcode

        // Try all instructions
        for(i = 0; i <= 14; i++)begin
                #20ns instruction = instructionsArray[i];
        end
        #20ns i++;

        // check if the program passes or not
          if(passOrFail == 0) $display("The Program Failed");
                else $display("The Program Passed");
        end

 mp_top mp(clk,instruction, result);


 always #5ns begin
        clk = ~clk;
 end
  always@(i) begin
       if(i <= 15) begin
       // find the expected result
       #15001
        case(instructionsArray[i - 1][5:0])
                        6'b000101: begin expectedResult = 32'h00002642;   opration = " a + b
";   end
                                6'b001000: begin expectedResult = 32'hFFFFFA8A;   opration = "
a - b   ";   end
                        6'b001101: begin expectedResult = 32'h00001066;   opration = "  |a|    ";
end
                                6'b000111: begin expectedResult = 32'hFFFFEF9A;   opration = "
-a     ";   end
```

```
                                6'b000011: begin expectedResult = 32'h000015DC;   opration =
"max(a,b) ";   end
                                6'b000110: begin expectedResult = 32'h00001066;   opration =
"min(a,b) ";   end
                                6'b001010: begin expectedResult = 32'h00001321;   opration =
"avg(a,b) ";   end
                                6'b000010: begin expectedResult = 32'hFFFFEF99;   opration = "
~a     ";   end
                                6'b001111: begin expectedResult = 32'h000015FE;   opration = "
or    ";   end
                                6'b000100: begin expectedResult = 32'h00001044;   opration = "
and    ";   end
                                6'b001100: begin expectedResult = 32'h000005BA;   opration = "
xor    ";   end
                                default : begin  opration = "non valid";   end

            endcase
            // compare the result with the expected result for each instruction
            if (result == expectedResult && opration != "non valid")
        isPassed = "pass";
      else if (result == expectedResult && opration == "non valid")
        isPassed = "not valid(the result remain the same)";
            else begin
                    isPassed = "fail";
                    passOrFail = 0;
                    end

             $display("%b  %h   %h    %s      %h      %h   %s",instructionsArray[i -
1],a,b,opration,expectedResult,result,isPassed);
             isPassed = "";
                    end
                     end
endmodule
```

(Alu module testbench code):

```
module alu_test;
  reg [5:0] opcode;
  reg signed [31:0] a, b;
  wire signed [31:0] result;
  reg [31:0] expectedResult;
```

```
reg [5:0] opcodeArray [13:0];
int i;
string operation, isPassed;
int passOrFail = 1;

// ALU instantiation
alu Alu(opcode, a, b, result);

initial begin
  $display("opcode      a          b      operation   expected result   result       test");
  $display("-------------------------------------------------------------------------------------");
  i = 0;
  a = 32'h1066;
  b = 32'h15DC;
  opcodeArray[0] = 6'b000101;         // a + b
  opcodeArray[1] = 6'b001000;         // a - b
  opcodeArray[2] = 6'b001101;         // |a|
  opcodeArray[3] = 6'b000111;         // -a
  opcodeArray[4] = 6'b000011;         // max(a,b)
  opcodeArray[5] = 6'b000110;         // min(a,b)
  opcodeArray[6] = 6'b001010;         // avg(a,b)
  opcodeArray[7] = 6'b000010;         // ~a
  opcodeArray[8] = 6'b000001;  // instruction of non-valid opcode
  opcodeArray[9] = 6'b001111;         // a | b
  opcodeArray[10] = 6'b001001; // instruction of non-valid opcode
  opcodeArray[11] = 6'b000100; //    a & b
  opcodeArray[12] = 6'b001011; // instruction of non-valid opcode
  opcodeArray[13] = 6'b001100; // a ^ b

  for (i = 0; i <= 13; i = i + 1) begin
    #10ns opcode = opcodeArray[i];
  end
  #10ns i = i + 1;
  // Check if the program passes or not
  if (passOrFail == 0) $display("The Program Failed");
  else $display("The Program Passed");
end

// Compare results
always @(i) begin
```

```
    if (i <= 14) begin
                    #1 // delay
      // Find the expected result
      case (opcodeArray[i - 1])
        6'b000101: begin expectedResult = 32'h00002642; operation = " a + b   "; end
        6'b001000: begin expectedResult = 32'hFFFFFA8A; operation = " a - b   "; end
        6'b001101: begin expectedResult = 32'h00001066; operation = "  |a|    "; end
        6'b000111: begin expectedResult = 32'hFFFFEF9A; operation = "  -a     "; end
        6'b000011: begin expectedResult = 32'h000015DC; operation = "max(a,b) "; end
        6'b000110: begin expectedResult = 32'h00001066; operation = "min(a,b) "; end
        6'b001010: begin expectedResult = 32'h00001321; operation = "avg(a,b) "; end
        6'b000010: begin expectedResult = 32'hFFFFEF99; operation = "  ~a     "; end
        6'b001111: begin expectedResult = 32'h000015FE; operation = "   or    "; end
        6'b000100: begin expectedResult = 32'h00001044; operation = "  and    "; end
        6'b001100: begin expectedResult = 32'h000005BA; operation = "  xor    "; end
         default : begin  operation = "non valid";   end
      endcase

      // Compare the result with the expected result for each opcode
       if (result == expectedResult && operation != "non valid")
        isPassed = "pass";
      else if (result == expectedResult && operation == "non valid")
        isPassed = "not valid(the result remain the same)";
            else begin
                    isPassed = "fail";
                    passOrFail = 0;
                    end

     $display("%b   %h    %h   %s    %h     %h     %s", opcodeArray[i - 1], a, b, operation,
expectedResult, result, isPassed);
      isPassed = "";
    end
  end
endmodule;
```

(Reg File module testbench code):

```
module reg_file_test;
          reg clk;
          reg [5:0] validOpcode;
          reg [4:0] addr1,addr2,addr3;
```

```
        reg signed [31:0] in;
        wire signed [31:0] regOut1,regOut2;

        reg_file regFile(clk,validOpcode,addr1,addr2,addr3,in,regOut1,regOut2);

 integer i;

  initial begin
  clk = 0;
  for (i = 0; i < 7; i = i + 1) begin
     #5ns;
     clk = ~clk;
  end
end

        initial begin
                $display("opcode      addr1      addr2          Out1            Out2 ");
                $display("-----------------------------------------------------------------");
                clk = 0;
                validOpcode = 6'b000101;
                addr1 = 8;
                addr2 = 10;
                addr3 = 0;
                in = 32'h55;
                #10ns
                validOpcode = 6'b001000;
                addr1 = 0;
                addr2 = 9;
                addr3 = 2;
                in = 32'h42;
                #10ns
                validOpcode = 6'b000010;
                addr1 = 2;
                addr2 = 4;
                addr3 = 31;
                in = 32'h33;
                #10ns
                validOpcode = 6'b011111;
                addr1 = 15;
                addr2 = 31;
```

```
                addr3 = 0;
                in = 32'h11;
                end

        always@(posedge clk) begin
                        #1ns
                 $display("%b      %h      %h      %h      %h", validOpcode, addr1, addr2,
regOut1,regOut2);
                        end
endmodule
```