Faculty of Engineering and Technology
Electrical and Computer Engineering Department

ENCS3310– Advanced Digital Design

Project Report

Prepared by: Abdalrhman Juber

ID Number: 1211769

Instructor: Dr. **Abdallatif Abuissa**

Section: 1

Date: 1/19/24

Abstract

Through this project, we aim to acquire knowledge pertaining to the implementation of a microprocessor that comprises an ALU and register file which are interconnected. Our focus is on achieving data synchronization, ascertaining accurate calculation of operations and testing the microprocessor for correct output.

Table of Figures

Brief Introduction and Background

In this project, we shall undertake the development of a straightforward microprocessor consisting of three stages. The initial stage involves the creation of an Arithmetic Logic Unit (ALU) that incorporates various operations including addition, subtraction, maximum and minimum computation as well as fundamental bit-wise operators.

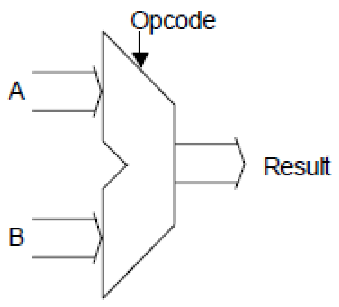


Figure 1: ALU

The second stage is to implement a register file contains an array of 32*32 that stores some data.

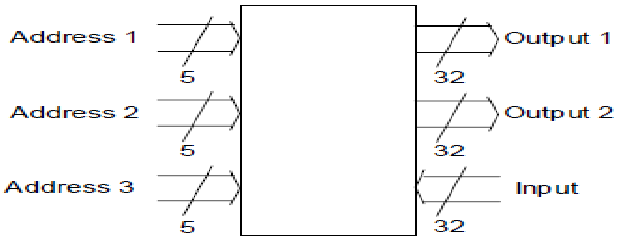


Figure 2: Register file

The last stage is to connect the ALU with RAM as showing in figure 3.

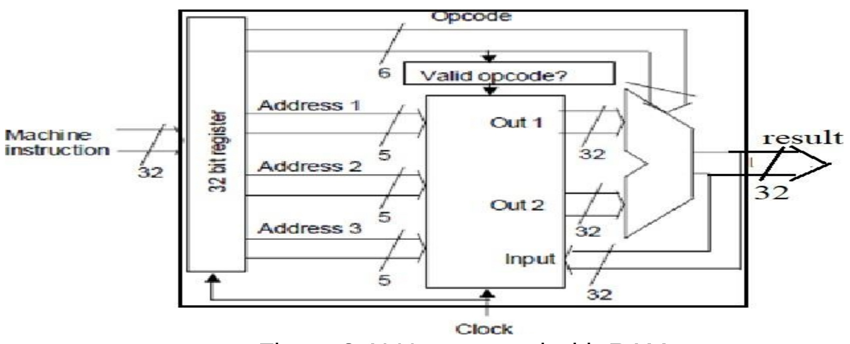
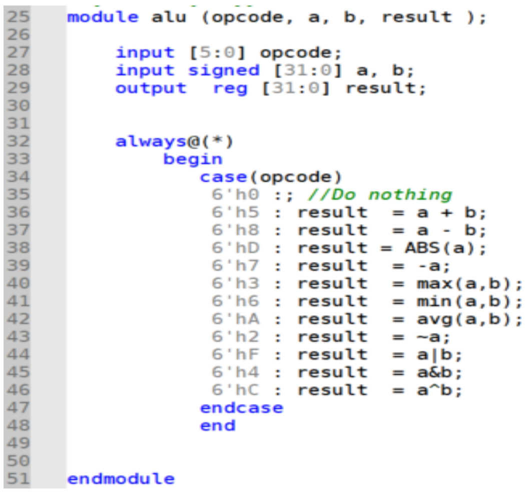


Figure 3:ALU connected with RAM

Design philosophy

Part One : ALU

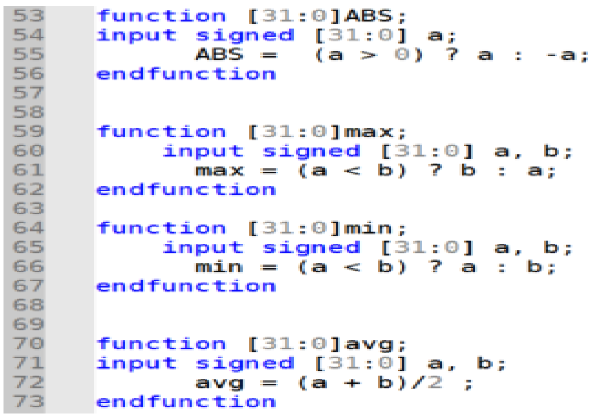
The Arithmetic Logic Unit (ALU) executes a range of operations, including addition, subtraction, absolute value computation, negation, maximum and minimum determination, average calculation, as well as logical functions such as NOT, OR, AND and XOR. Additionally, the No-Opcode operation has been designed to enable the microprocessor to remain idle for an entire cycle. It is important to note that this ALU is implemented using combinational logic circuitry without requiring a clock signal input as shown in Figure 4.



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

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


    always@(*)
        begin
           case(opcode)
            6'h0 :; //Do nothing
            6'h5 : result  = a + b;
            6'h8 : result  = a - b;
            6'hD : result = ABS(a);
            6'h7 : result  = -a;
            6'h3 : result  = max(a,b);
            6'h6 : result  = min(a,b);
            6'hA : result  = avg(a,b);
            6'h2 : result  = ~a;
            6'hF : result  = a|b;
            6'h4 : result  = a&b;
            6'hC : result  = a^b;
           endcase
           end


endmodule
```

Figure4 : ALU code

I did create the max,min and avg as functions as seen in figure 5.

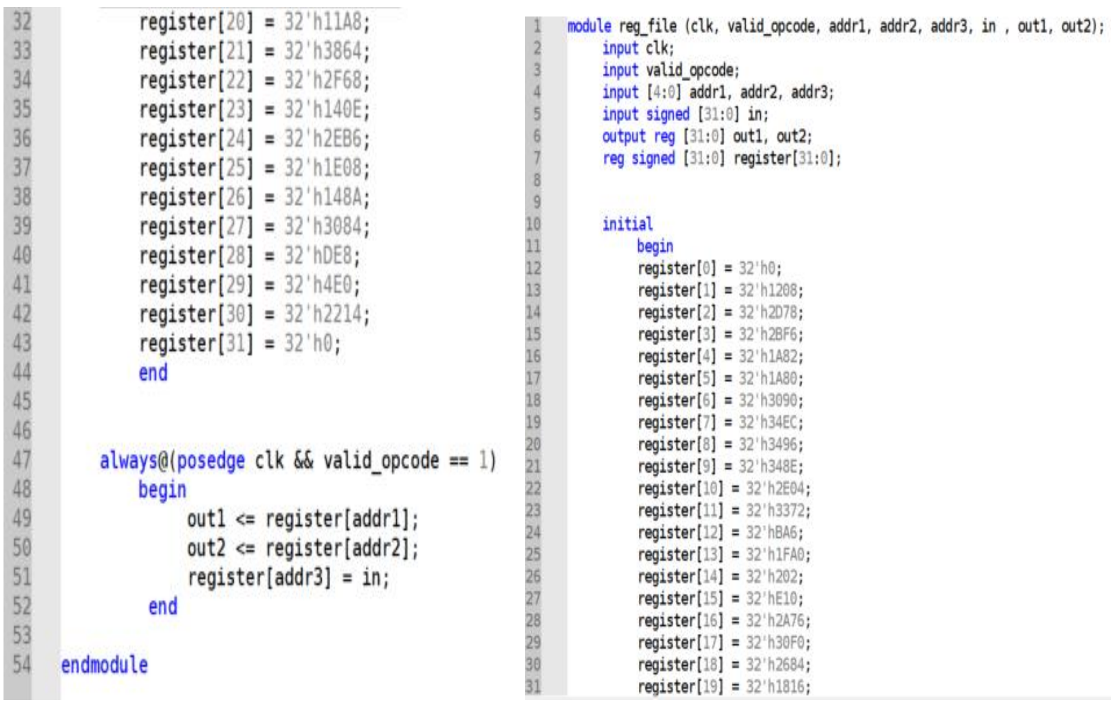


```verilog
function  [31:0]ABS;
input signed  [31:0]  a;
        ABS =    (a > 0)  ? a  :  -a;
endfunction


function  [31:0]max;
     input signed  [31:0]  a,  b;
        max = (a < b)  ? b  :  a;
endfunction

function  [31:0]min;
     input signed  [31:0]  a,  b;
        min = (a < b)  ? a  :  b;
endfunction


function  [31:0]avg;
input signed  [31:0]  a,  b;
        avg = (a + b)/2  ;
endfunction
```

Figure 5: ALU functions

The register file

I stored the data under my identification number, which is 6 as depicted in Figure 6. I opted to convert the values into hexadecimal format for convenience purposes. Additionally, a register with dimensions of 32 columns and 32 rows was established to store said values. The clock mechanism has been set to activate during every rise up, subject to an additional condition stipulating that the opcode must be valid.

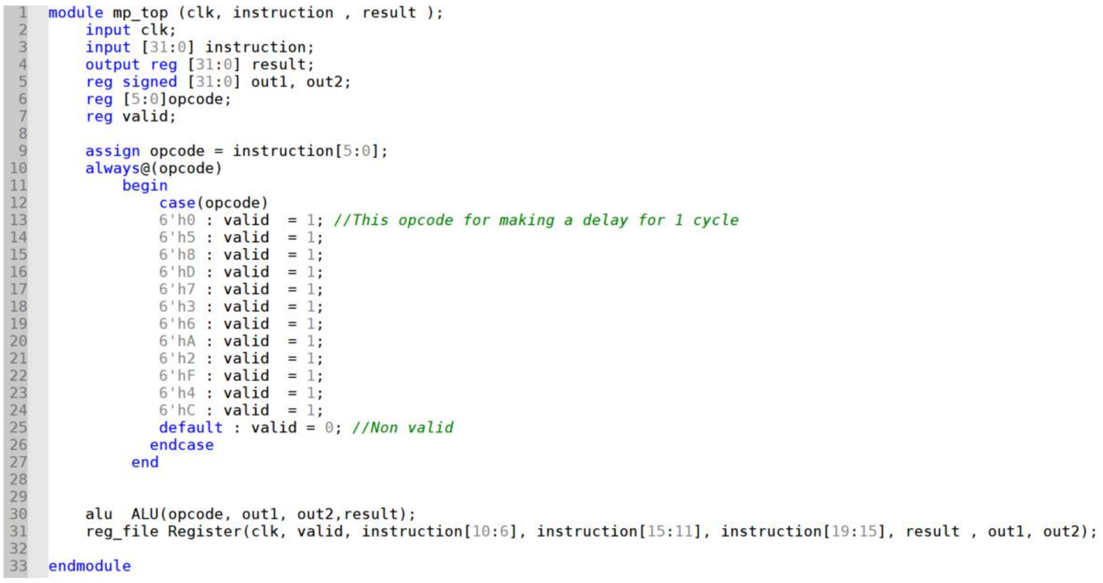


```verilog
module reg_file (clk, valid_opcode, addr1, addr2, addr3, in , out1, out2);
    input clk;
    input valid_opcode;
    input [4:0] addr1, addr2, addr3;
    input signed [31:0] in;
    output reg [31:0] out1, out2;
    reg signed [31:0] register[31:0];


    initial
        begin
        register[0] = 32'h0;
        register[1] = 32'h1208;
        register[2] = 32'h2D78;
        register[3] = 32'h2BF6;
        register[4] = 32'h1A82;
        register[5] = 32'h1A80;
        register[6] = 32'h3090;
        register[7] = 32'h34EC;
        register[8] = 32'h3496;
        register[9] = 32'h348E;
        register[10] = 32'h2E04;
        register[11] = 32'h3372;
        register[12] = 32'hBA6;
        register[13] = 32'h1FA0;
        register[14] = 32'h202;
        register[15] = 32'hE10;
        register[16] = 32'h2A76;
        register[17] = 32'h30F0;
        register[18] = 32'h2684;
        register[19] = 32'h1816;
        register[20] = 32'h11A8;
        register[21] = 32'h3864;
        register[22] = 32'h2F68;
        register[23] = 32'h140E;
        register[24] = 32'h2EB6;
        register[25] = 32'h1E08;
        register[26] = 32'h148A;
        register[27] = 32'h3084;
        register[28] = 32'hDE8;
        register[29] = 32'h4E0;
        register[30] = 32'h2214;
        register[31] = 32'h0;
        end


    always@(posedge clk && valid_opcode == 1)
        begin
            out1 <= register[addr1];
            out2 <= register[addr2];
            register[addr3] = in;
         end

endmodule
```

Figure 6: Register file code

Part Two : Connect ALU to RAM

In this section, I successfully established a connection between the ALU and RAM. Additionally, I verified the validity of the opcode by comparing it to its valid values. The results are displayed in Figure 7, where a value of 1 represents validity and a value of 0 indicates invalidity.



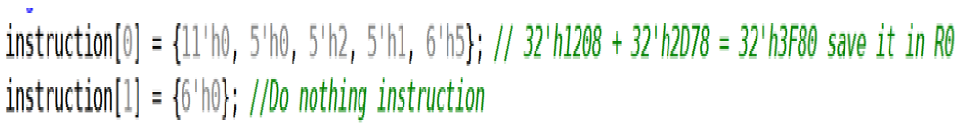
```verilog
module mp_top (clk, instruction , result );
    input clk;
    input [31:0] instruction;
    output reg [31:0] result;
    reg signed [31:0] out1, out2;
    reg [5:0]opcode;
    reg valid;

    assign opcode = instruction[5:0];
    always@(opcode)
        begin
            case(opcode)
            6'h0 : valid  = 1; //This opcode for making a delay for 1 cycle
            6'h5 : valid  = 1;
            6'h8 : valid  = 1;
            6'hD : valid  = 1;
            6'h7 : valid  = 1;
            6'h3 : valid  = 1;
            6'h6 : valid  = 1;
            6'hA : valid  = 1;
            6'h2 : valid  = 1;
            6'hF : valid  = 1;
            6'h4 : valid  = 1;
            6'hC : valid  = 1;
            default : valid = 0; //Non valid
           endcase
         end


    alu  ALU(opcode, out1, out2,result);
    reg_file Register(clk, valid, instruction[10:6], instruction[15:11], instruction[19:15], result , out1, out2);

endmodule
```

Figure 7: connect alu with ram code

Test Bench:

- what happens on what clock cycle and why?

For each clock cycle, the instruction is introduced to the mp_top and the register file retrieves addresses that are subsequently returned as out1 and out2 to alu for opcode calculation. An additional cycle is necessary for obtaining results. To obtain a result value, I developed an instruction with no opcode which serves as a placeholder. This process is illustrated in Figure 8.



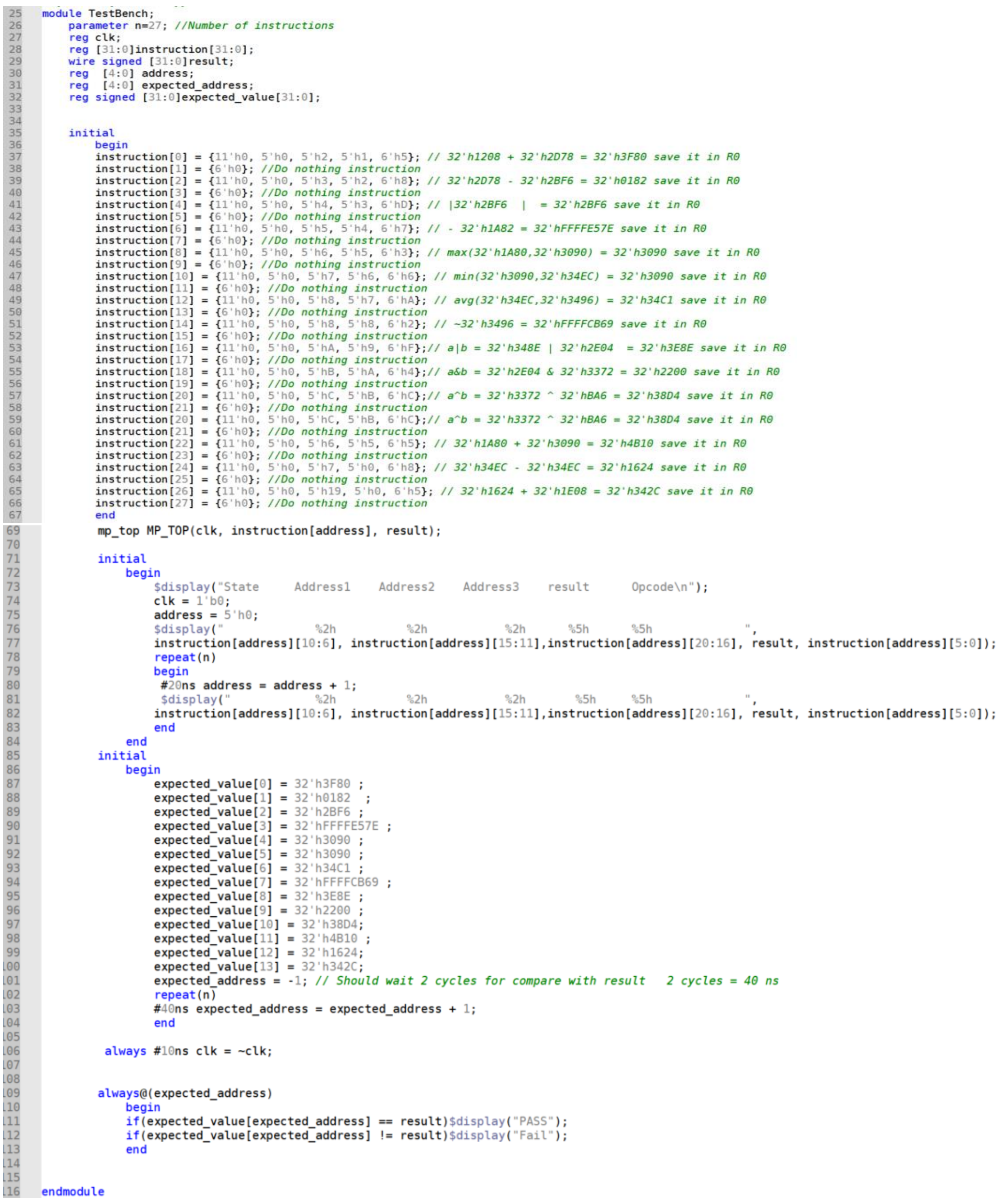
```verilog
instruction[0] = {11'h0, 5'h0, 5'h2, 5'h1, 6'h5}; // 32'h1208 + 32'h2D78 = 32'h3F80 save it in R0
instruction[1] = {6'h0}; //Do nothing instruction
```

Figure 8: Instructions

We require two cycles to accomplish all tasks.

- How you tested it and how you interpreted the results of your tests?

I have composed 27 instructions, half of which do not contain an opcode instruction as illustrated in figure 9. I have calculated the anticipated value for each individual instruction and after a duration of 40 nanoseconds, I will compare the outcome with the projected result. The reasoning behind this time frame is that each cycle requires 20 nanoseconds to complete and we require two cycles to obtain the actual outcome, hence necessitating two cycles for comparison purposes. The results are documented in appendix A.



```verilog
module TestBench;
    parameter n=27; //Number of instructions
    reg clk;
    reg [31:0]instruction[31:0];
    wire signed [31:0]result;
    reg  [4:0] address;
    reg  [4:0] expected_address;
    reg signed [31:0]expected_value[31:0];


    initial
        begin
        instruction[0] = {11'h0, 5'h0, 5'h2, 5'h1, 6'h5}; // 32'h1208 + 32'h2D78 = 32'h3F80 save it in R0
        instruction[1] = {6'h0}; //Do nothing instruction
        instruction[2] = {11'h0, 5'h0, 5'h3, 5'h2, 6'h8}; // 32'h2D78 - 32'h2BF6 = 32'h0182 save it in R0
        instruction[3] = {6'h0}; //Do nothing instruction
        instruction[4] = {11'h0, 5'h0, 5'h4, 5'h3, 6'hD}; // |32'h2BF6  |  = 32'h2BF6 save it in R0
        instruction[5] = {6'h0}; //Do nothing instruction
        instruction[6] = {11'h0, 5'h0, 5'h5, 5'h4, 6'h7}; // - 32'h1A82 = 32'hFFFFE57E save it in R0
        instruction[7] = {6'h0}; //Do nothing instruction
        instruction[8] = {11'h0, 5'h0, 5'h6, 5'h5, 6'h3}; // max(32'h1A80,32'h3090) = 32'h3090 save it in R0
        instruction[9] = {6'h0}; //Do nothing instruction
        instruction[10] = {11'h0, 5'h0, 5'h7, 5'h6, 6'h6}; // min(32'h3090,32'h34EC) = 32'h3090 save it in R0
        instruction[11] = {6'h0}; //Do nothing instruction
        instruction[12] = {11'h0, 5'h0, 5'h8, 5'h7, 6'hA}; // avg(32'h34EC,32'h3496) = 32'h34C1 save it in R0
        instruction[13] = {6'h0}; //Do nothing instruction
        instruction[14] = {11'h0, 5'h0, 5'h8, 5'h8, 6'h2}; // ~32'h3496 = 32'hFFFFCB69 save it in R0
        instruction[15] = {6'h0}; //Do nothing instruction
        instruction[16] = {11'h0, 5'h0, 5'hA, 5'h9, 6'hF};// a|b = 32'h348E | 32'h2E04  = 32'h3E8E save it in R0
        instruction[17] = {6'h0}; //Do nothing instruction
        instruction[18] = {11'h0, 5'h0, 5'hB, 5'hA, 6'h4};// a&b = 32'h2E04 & 32'h3372 = 32'h2200 save it in R0
        instruction[19] = {6'h0}; //Do nothing instruction
        instruction[20] = {11'h0, 5'h0, 5'hC, 5'hB, 6'hC};// a^b = 32'h3372 ^ 32'hBA6 = 32'h38D4 save it in R0
        instruction[21] = {6'h0}; //Do nothing instruction
        instruction[20] = {11'h0, 5'h0, 5'hC, 5'hB, 6'hC};// a^b = 32'h3372 ^ 32'hBA6 = 32'h38D4 save it in R0
        instruction[21] = {6'h0}; //Do nothing instruction
        instruction[22] = {11'h0, 5'h0, 5'h6, 5'h5, 6'h5}; // 32'h1A80 + 32'h3090 = 32'h4B10 save it in R0
        instruction[23] = {6'h0}; //Do nothing instruction
        instruction[24] = {11'h0, 5'h0, 5'h7, 5'h0, 6'h8}; // 32'h34EC - 32'h34EC = 32'h1624 save it in R0
        instruction[25] = {6'h0}; //Do nothing instruction
        instruction[26] = {11'h0, 5'h0, 5'h19, 5'h0, 6'h5}; // 32'h1624 + 32'h1E08 = 32'h342C save it in R0
        instruction[27] = {6'h0}; //Do nothing instruction
        end
        mp_top MP_TOP(clk, instruction[address], result);

        initial
            begin
                $display("State     Address1    Address2    Address3    result      Opcode\n");
                clk = 1'b0;
                address = 5'h0;
                $display("            %2h          %2h           %2h      %5h      %5h             ",
                instruction[address][10:6], instruction[address][15:11],instruction[address][20:16], result, instruction[address][5:0]);
                repeat(n)
                begin
                 #20ns address = address + 1;
                 $display("           %2h          %2h           %2h       %5h     %5h             ",
                instruction[address][10:6], instruction[address][15:11],instruction[address][20:16], result, instruction[address][5:0]);
                end
            end
        initial
            begin
                expected_value[0] = 32'h3F80 ;
                expected_value[1] = 32'h0182  ;
                expected_value[2] = 32'h2BF6 ;
                expected_value[3] = 32'hFFFFE57E ;
                expected_value[4] = 32'h3090 ;
                expected_value[5] = 32'h3090 ;
                expected_value[6] = 32'h34C1 ;
                expected_value[7] = 32'hFFFFCB69 ;
                expected_value[8] = 32'h3E8E ;
                expected_value[9] = 32'h2200 ;
                expected_value[10] = 32'h38D4;
                expected_value[11] = 32'h4B10 ;
                expected_value[12] = 32'h1624;
                expected_value[13] = 32'h342C;
                expected_address = -1; // Should wait 2 cycles for compare with result   2 cycles = 40 ns
                repeat(n)
                #40ns expected_address = expected_address + 1;
                end

         always #10ns clk = ~clk;


        always@(expected_address)
            begin
            if(expected_value[expected_address] == result)$display("PASS");
            if(expected_value[expected_address] != result)$display("Fail");
            end



endmodule
```

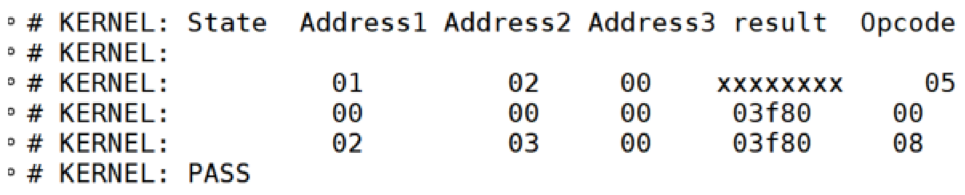
Figure 9 : Test Bench

Results :

All Instruction and results are shown in appendix A.

The first instruction is {11'h0, 5'h0, 5'h2, 5'h1, 6'h5} and it translate as (32'h1208 + 32'h2D78 = **32'h3F80** save it in R0).And the result are correct so we print PASS as shown in figure 10.

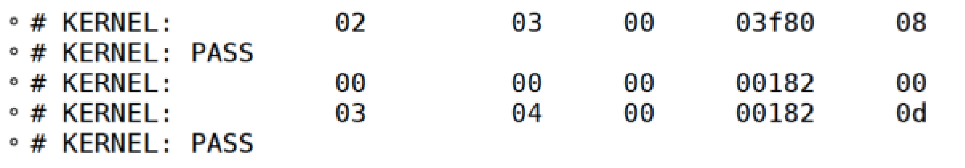
The 0 opcode instruction will store the result of the previous instruction.



```
# KERNEL: State  Address1 Address2 Address3 result   Opcode
# KERNEL:
# KERNEL:           01          02      00     xxxxxxxx     05
# KERNEL:           00          00      00      03f80     00
# KERNEL:           02          03      00      03f80     08
# KERNEL: PASS
```

Figure 10: Result for first instruction

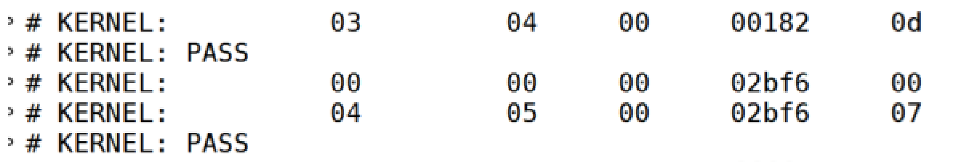
For the second instruction which is value are {11'h0, 5'h0, 5'h3, 5'h2, 6'h8} and it translate as (32'h2D78 - 32'h2BF6 = **32'h0182** save it in R0).And the result are correct so we print PASS as shown in figure 11.



```
# KERNEL:           02          03      00      03f80     08
# KERNEL: PASS
# KERNEL:           00          00      00      00182     00
# KERNEL:           03          04      00      00182     0d
# KERNEL: PASS
```

Figure 11 : Result for second instruction

Third instruction is {11'h0, 5'h0, 5'h4, 5'h3, 6'hD} and it translate as (|32'h2BF6| = 32'h**2BF6** save it in R0).And the result are correct so we print PASS as shown in figure 12.



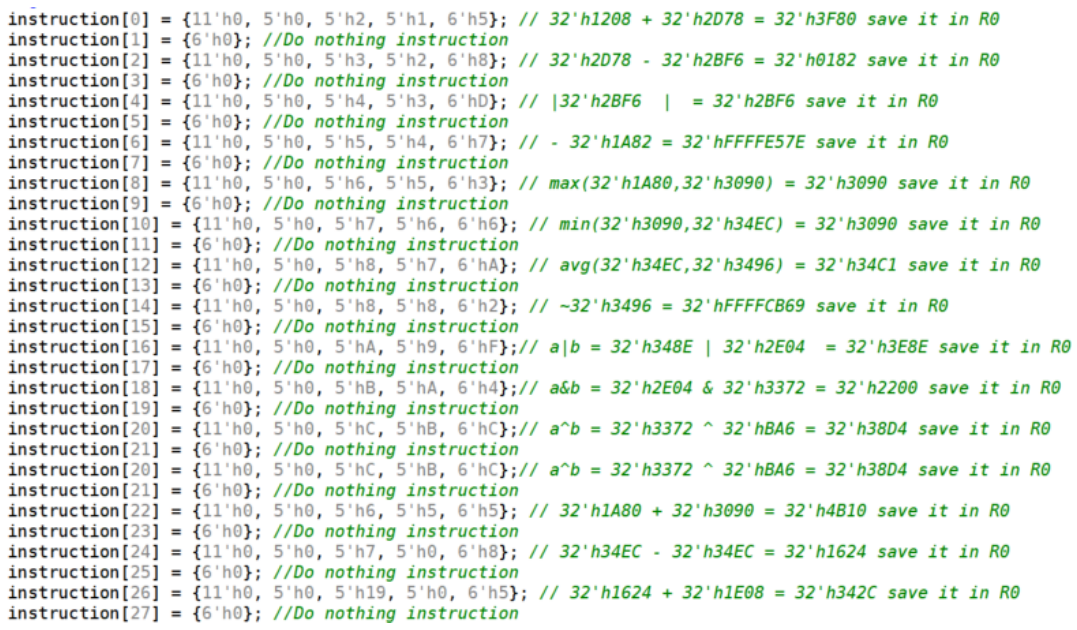
```
# KERNEL:           03          04      00      00182     0d
# KERNEL: PASS
# KERNEL:           00          00      00      02bf6     00
# KERNEL:           04          05      00      02bf6     07
# KERNEL: PASS
```

Figure 12 : Result for third instruction

Conclusion

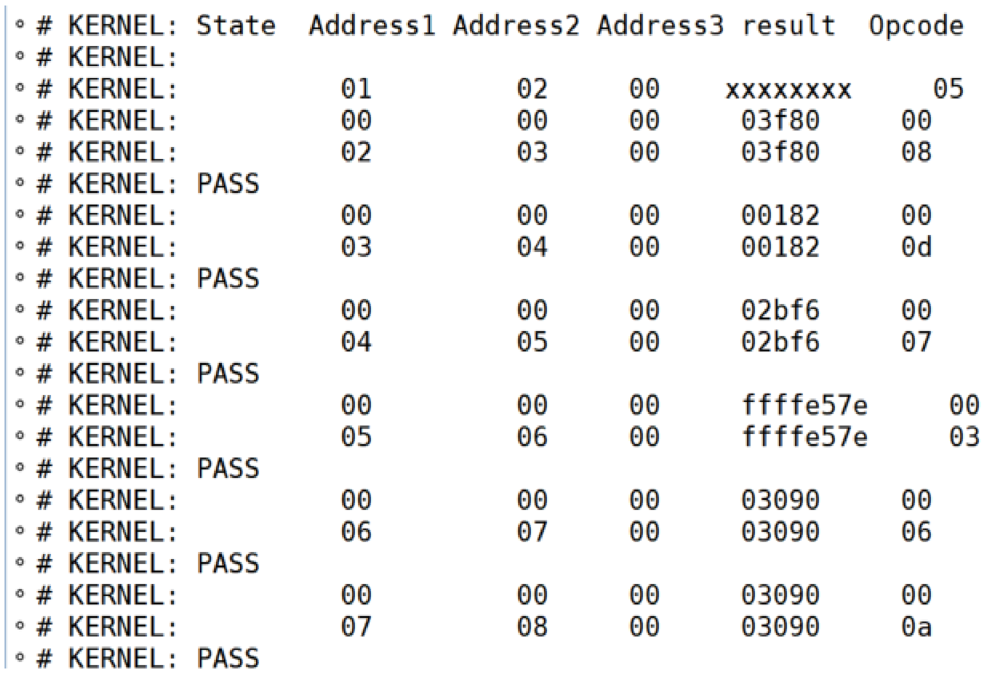
Through this project, we acquired the ability to execute a microprocessor by constructing an ALU and register file, followed by developing interconnectivity between them. Additionally,We achieved synchronization by utilizing a non-opcode instruction to obtain the desired result. Subsequently, we conducted tests on each instruction and labeled them as "PASS" for accurate outcomes and "FAILL" for erroneous ones.

Appendix A



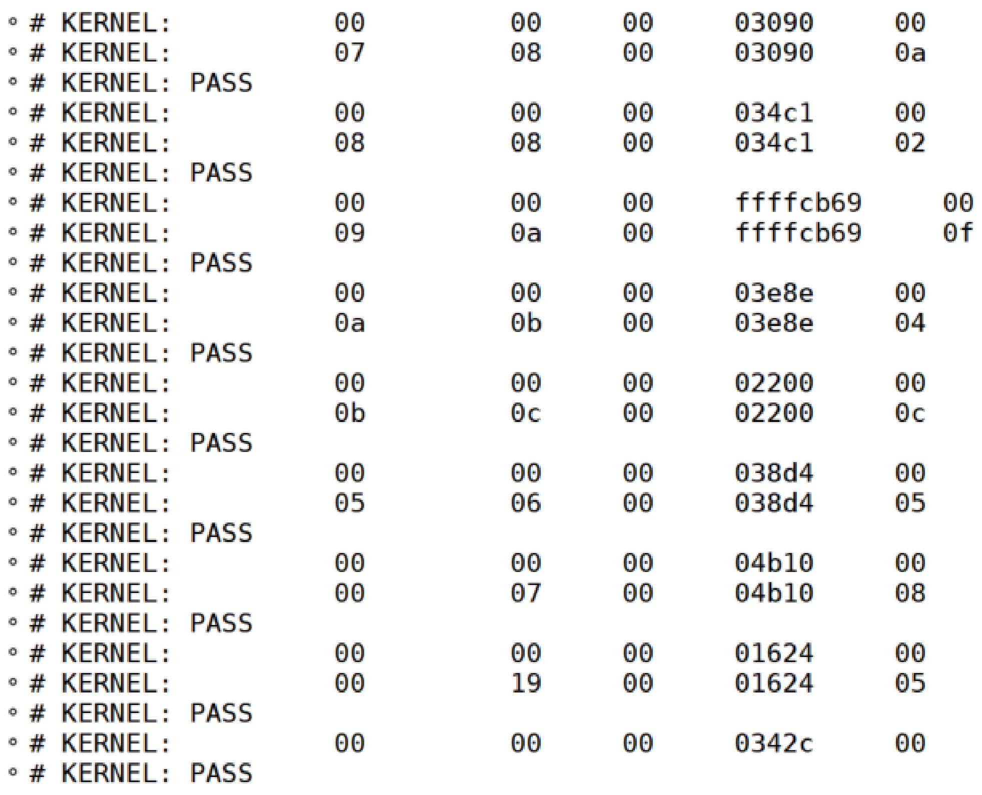
```
instruction[0] = {11'h0, 5'h0, 5'h2, 5'h1, 6'h5}; // 32'h1208 + 32'h2D78 = 32'h3F80 save it in R0
instruction[1] = {6'h0}; //Do nothing instruction
instruction[2] = {11'h0, 5'h0, 5'h3, 5'h2, 6'h8}; // 32'h2D78 - 32'h2BF6 = 32'h0182 save it in R0
instruction[3] = {6'h0}; //Do nothing instruction
instruction[4] = {11'h0, 5'h0, 5'h4, 5'h3, 6'hD}; // |32'h2BF6  |  = 32'h2BF6 save it in R0
instruction[5] = {6'h0}; //Do nothing instruction
instruction[6] = {11'h0, 5'h0, 5'h5, 5'h4, 6'h7}; // - 32'h1A82 = 32'hFFFFE57E save it in R0
instruction[7] = {6'h0}; //Do nothing instruction
instruction[8] = {11'h0, 5'h0, 5'h6, 5'h5, 6'h3}; // max(32'h1A80,32'h3090) = 32'h3090 save it in R0
instruction[9] = {6'h0}; //Do nothing instruction
instruction[10] = {11'h0, 5'h0, 5'h7, 5'h6, 6'h6}; // min(32'h3090,32'h34EC) = 32'h3090 save it in R0
instruction[11] = {6'h0}; //Do nothing instruction
instruction[12] = {11'h0, 5'h0, 5'h8, 5'h7, 6'hA}; // avg(32'h34EC,32'h3496) = 32'h34C1 save it in R0
instruction[13] = {6'h0}; //Do nothing instruction
instruction[14] = {11'h0, 5'h0, 5'h8, 5'h8, 6'h2}; // ~32'h3496 = 32'hFFFFCB69 save it in R0
instruction[15] = {6'h0}; //Do nothing instruction
instruction[16] = {11'h0, 5'h0, 5'hA, 5'h9, 6'hF};// a|b = 32'h348E | 32'h2E04  = 32'h3E8E save it in R0
instruction[17] = {6'h0}; //Do nothing instruction
instruction[18] = {11'h0, 5'h0, 5'hB, 5'hA, 6'h4};// a&b = 32'h2E04 & 32'h3372 = 32'h2200 save it in R0
instruction[19] = {6'h0}; //Do nothing instruction
instruction[20] = {11'h0, 5'h0, 5'hC, 5'hB, 6'hC};// a^b = 32'h3372 ^ 32'hBA6 = 32'h38D4 save it in R0
instruction[21] = {6'h0}; //Do nothing instruction
instruction[20] = {11'h0, 5'h0, 5'hC, 5'hB, 6'hC};// a^b = 32'h3372 ^ 32'hBA6 = 32'h38D4 save it in R0
instruction[21] = {6'h0}; //Do nothing instruction
instruction[22] = {11'h0, 5'h0, 5'h6, 5'h5, 6'h5}; // 32'h1A80 + 32'h3090 = 32'h4B10 save it in R0
instruction[23] = {6'h0}; //Do nothing instruction
instruction[24] = {11'h0, 5'h0, 5'h7, 5'h0, 6'h8}; // 32'h34EC - 32'h34EC = 32'h1624 save it in R0
instruction[25] = {6'h0}; //Do nothing instruction
instruction[26] = {11'h0, 5'h0, 5'h19, 5'h0, 6'h5}; // 32'h1624 + 32'h1E08 = 32'h342C save it in R0
instruction[27] = {6'h0}; //Do nothing instruction
```

AFigure A1 : Instructions



```
# KERNEL: State   Address1 Address2 Address3 result   Opcode
# KERNEL:
# KERNEL:           01          02      00     xxxxxxxx    05
# KERNEL:           00          00      00      03f80     00
# KERNEL:           02          03      00      03f80     08
# KERNEL: PASS
# KERNEL:           00          00      00      00182     00
# KERNEL:           03          04      00      00182     0d
# KERNEL: PASS
# KERNEL:           00          00      00      02bf6     00
# KERNEL:           04          05      00      02bf6     07
# KERNEL: PASS
# KERNEL:           00          00      00      ffffe57e     00
# KERNEL:           05          06      00      ffffe57e     03
# KERNEL: PASS
# KERNEL:           00          00      00      03090     00
# KERNEL:           06          07      00      03090     06
# KERNEL: PASS
# KERNEL:           00          00      00      03090     00
# KERNEL:           07          08      00      03090     0a
# KERNEL: PASS
```

BFigure A2 : Results



```
◦ # KERNEL:          00          00      00      03090      00
◦ # KERNEL:          07          08      00      03090      0a
◦ # KERNEL: PASS
◦ # KERNEL:          00          00      00      034c1      00
◦ # KERNEL:          08          08      00      034c1      02
◦ # KERNEL: PASS
◦ # KERNEL:          00          00      00      ffffcb69      00
◦ # KERNEL:          09          0a      00      ffffcb69      0f
◦ # KERNEL: PASS
◦ # KERNEL:          00          00      00      03e8e      00
◦ # KERNEL:          0a          0b      00      03e8e      04
◦ # KERNEL: PASS
◦ # KERNEL:          00          00      00      02200      00
◦ # KERNEL:          0b          0c      00      02200      0c
◦ # KERNEL: PASS
◦ # KERNEL:          00          00      00      038d4      00
◦ # KERNEL:          05          06      00      038d4      05
◦ # KERNEL: PASS
◦ # KERNEL:          00          00      00      04b10      00
◦ # KERNEL:          00          07      00      04b10      08
◦ # KERNEL: PASS
◦ # KERNEL:          00          00      00      01624      00
◦ # KERNEL:          00          19      00      01624      05
◦ # KERNEL: PASS
◦ # KERNEL:          00          00      00      0342c      00
◦ # KERNEL: PASS
```

CFigure A2 : Results