FACULTY OF ENGINEERING AND TECHNOLOGY

ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

ENCS3310

ADVANCED DIGITAL SYSTEMS DESIGN

Course project
Report

Student name: Masa Ahmad Jalamneh

Student Id: 1212145

Section: 3

Instructor: Elias Khalil

Date: 2024-1-15

Table of Contents

1. INTRODUCTION

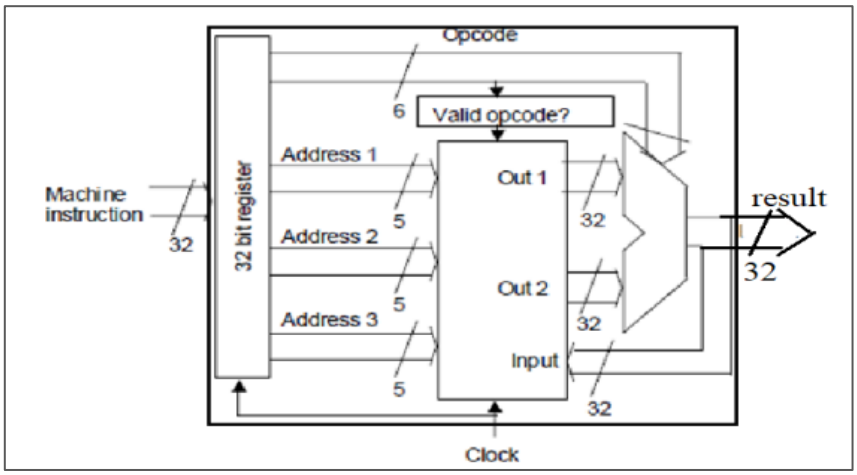
In this project we were asked to build a simple part of a microprocessor.

- First: build two main blocks:

  → the ALU block

  → the register file block

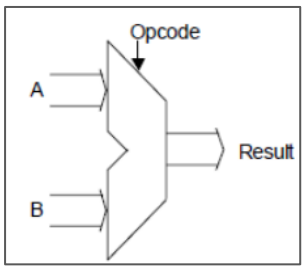
- Second: connect them together (mp_top), and run a simple machine code program on them (testbench).



2. PROCEDURE

2.1 PART 1:

2.1.1 ALU:



THE ALU with:

→ 32 bits inputs (A and B)

→ 32 bits output (Result)

→ 6 bits Opcode

The operation that will be executed depends on the opcode, and the opcodes for the operation based on my id number 121214**5** are**:**

a+b	a-b	|a|	-a	Max(a,b)	Min(a,b)	Avg(a,b)	Not(a)	a OR b	a AND b	a XOR b
3	15	13	12	7	1	9	10	14	11	5

Starting the verilog code by implementing the alu module as

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

Then I followed a different way to execute the operation by implementing a separate module for each arithmetic operation, and for the logic operation I wrote them in the alu module itself as following:

```
// operations results (local registers)
reg [31:0] adder_result, subtractor_result, abs_result, neg_result, max_result, min_result,
avg_result, not_a_result, or_result, and_result, xor_result ;
// operations  :
 adder adder_inst (.X(a), .Y(b), .S(adder_result));              // call adder module
 subtractor sub_inst (.X1(a), .Y1(b), .S1(subtractor_result));   // call sbutractor module
 absolute abs_inst (.in(a),.result(abs_result));                 // call absolute module
 assign neg_result = - a;                                        // negative multiplication
 maximum max (.A(a), .B(b), .result(max_result));                // call maximum module
 minimum min (.A(a), .B(b), .result(min_result));                // call minimum module
 assign avg_result= (adder_result / 2);                          // average operation
 assign not_a_result[31:0]= ~a[31:0];                            // NOT a operation
 assign or_result[31:0] = a[31:0] | b[31:0];                     // OR a,b operation
 assign and_result[31:0]= a[31:0] & b[31:0];                     // AND a,b operation
 assign xor_result[31:0]= a[31:0] ^ b[31:0];                     // XOR a,b operation
```

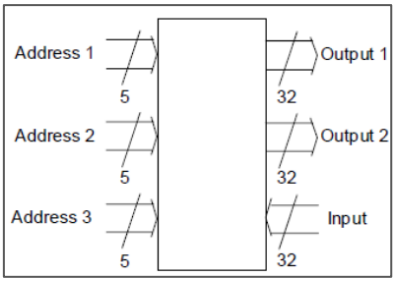
NOTE: all the extra modules are included in the codes file (adder, subtractor, min, max, avg...)

After that, by using the always block, i started implementing the opcode with their operation on (a and b) using (case({opcode})):

```
always @(a, b)
  begin
    case({opcode})
      6'b000011: assign result = adder_result;         // opcode= 3 => a+b
      6'b001111: assign result = subtractor_result;    // opcode=15 => a-b
      6'b001101: assign result = abs_result;           // opcode=13 => |a|
      6'b001100: assign result = neg_result;           // opcode=12 => -a
      6'b000111: assign result = max_result;           // opcode= 7 => max(a,b)
      6'b000001: assign result = min_result;           // opcode= 1 => min(a,b)
      6'b001001: assign result = avg_result;           // opcode= 9 => avg(a,b)
      6'b001010: assign result = not_a_result;         // opcode=10 => not(a)
      6'b001110: assign result = or_result;            // opcode=14 => a / b
      6'b001011: assign result = and_result;           // opcode=11 => a & b
      6'b000101: assign result = xor_result;           // opcode= 5 => a ^ b
      default:   assign result = 32'hxxxxxxxx;         // otherwise => invalid opcode => dont care
      endcase
    end
```

So, this module in general work as combinational circuit and ready to receive 2 inputs and to give an output depending on the opcode.

2.1.2 Reg_file:



The reg_file with:

- → 5 bits inputs (Addr1, Addr2, Addr3)
- → 32 bits outputs (output1, output2)
- → 32 bits input (input)

Output 1 produces the item within the register file that is addressed by Address 1. Similarly Output 2 produces the item within the register file that is addressed by Address 2. Input is used to supply a value that is written into the location addressed by Address 3.

Starting the Verilog code by implementing the reg_file module as:

```
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;

  reg [4:0] ID [31:0];
  reg [31:0] item [31:0];
```

Also implement a local registers (ID and item) to store the values.

The initial values stored in the register file are based on my id number 121214**5**:

```
////// The initial values stored in the reg_file:
 initial
   begin
     ID[0]  = 5'b00000;  item[0]  = 32'h00000000;   //0  -> d: 0     // h:0
     ID[1]  = 5'b00001;  item[1]  = 32'h00001066;   //1  -> d: 4198  // h:1066
     ID[2]  = 5'b00010;  item[2]  = 32'h000015dc;   //2  -> d: 5596  // h:15dc
     ID[3]  = 5'b00011;  item[3]  = 32'h0000385a;   //3  -> d: 14426 // h:385a
     ID[4]  = 5'b00100;  item[4]  = 32'h00001dbc;   //4  -> d: 7612  // h:1dbc
     ID[5]  = 5'b00101;  item[5]  = 32'h000019ee;   //5  -> d: 6638  // h:19ee
     ID[6]  = 5'b00110;  item[6]  = 32'h00002738;   //6  -> d: 10040 // h:2738
     ID[7]  = 5'b00111;  item[7]  = 32'h00000f5a;   //7  -> d: 3930  // h:f5a
     ID[8]  = 5'b01000;  item[8]  = 32'h00001036;   //8  -> d: 4150  // h:1036
     ID[9]  = 5'b01001;  item[9]  = 32'h00001906;   //9  -> d: 6406  // h:1906
     ID[10] = 5'b01010;  item[10] = 32'h00001518;   //10 -> d: 5400  // h:1518
     ID[11] = 5'b01011;  item[11] = 32'h0000127c;   //11 -> d: 8572  // h:217c
```

NOTE: these are the first 12 item only the rest of the item are written in the reg_file code.

NOTE: I converted these values from decimal to hexadecimal, and I used the hexadecimal implementation for the values.

After that, by using the always block that is sensitive on all the inputs and the clock; because this module is a combinational circuit, i started it by checking the validity of the opcode (as an Enable). If valid then search (using for loop) for the items that are addressed by Address 1 and Address2, to produce Output 1 and Output 2. Location addressed by Address 3 to write Input into it (after #10 delay) as follow:

```
always @( posedge clk, valid_opcode, addr1, addr2, addr3, in ) begin

   if (valid_opcode) begin


     for (int i = 0; i <= 31; i = i + 1)begin

           if (addr1 == ID[i]) begin
             out1 <= item[i];
            end
           if (addr2 == ID[i]) begin
             out2 <= item[i];
            end
           if (addr3 == ID[i]) begin
             #10
             item[i] <= in;
           end
    end

  end
  else begin
       out1 <= 32'hxxxxxxxx;
       out2 <= 32'hxxxxxxxx;
  end
```

NOTE that if opcode is invalid there will be no output (Don't care) out of the reg_file.

So, this module in general work as combinational circuit and ready to receive 3 inputs (addresses) and to give 2 outputs then receive an input to write into it.

2.2 PART2:

2.2.1 mp_top:

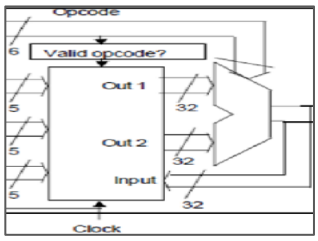
The top module where we connect the two modules (alu and reg_file) together, after making sure that we synchronized the register file to a clock.

So, on the rising edge of the clock:

• Output 1 produces the item within the register file that is addressed by Address 1, and then passes that value to the Alu module as input (a).

• Output 2 produces the item within the register file that is address by Address 2, and then passes that value to the Alu module as input (b).

• Input is used to supply a value (in) that is coming from the Alu module to write it into the location addressed by Address 3 in the reg_file.



Starting the Verilog code by implementing the mp_top module as:

```
module mp_top (clk, instruction , result);
 input clk;
 input [31:0] instruction;
 output reg [31:0] result;
```

And in the process of getting a complete design “core of the microprocessor”, I completed the mp_top by implementing local registers to provide the two modules inputs and outputs (addr1, addr2, addr3, opcode, reg_uot1, reg_out2, reg_in), also the stored_opcodes register to give valid-opcode a value (valid or invalid):

```
reg [5:0] opcode;                // as input for the alu module
reg valid_opcode;                    // as Enable for the reg_file module
reg [4:0] addr1, addr2, addr3; // as inputs for the reg_file module
reg [31:0] reg_out1, reg_out2, reg_in;// as outputs from the reg_file module inputs for the alu module


reg [5:0] stored_opcodes [10:0]; // to check the validity (work as Enable)
initial begin
 stored_opcodes[0]  =  6'b000011 ;
 stored_opcodes[1]  =  6'b001111 ;
 stored_opcodes[2]  =  6'b001101 ;
 stored_opcodes[3]  =  6'b001100 ;
 stored_opcodes[4]  =  6'b000111 ;
 stored_opcodes[5]  =  6'b000001 ;
 stored_opcodes[6]  =  6'b001001 ;
 stored_opcodes[7]  =  6'b001010 ;
 stored_opcodes[8]  =  6'b001110 ;
 stored_opcodes[9]  =  6'b001011 ;
 stored_opcodes[10] =  6'b000101 ;
end
```

Then I called the two modules and connect them:

```
reg_file r_f (
   .clk(clk),
   .valid_opcode(valid_opcode),
   .addr1(addr1),
   .addr2(addr2),
   .addr3(addr3),
   .in(reg_in),
   .out1(reg_out1),
   .out2(reg_out2)
 );

alu alu1 (
   .opcode(opcode),
   .a(reg_out1),
   .b(reg_out2),
   .result(reg_in)
 );
```

Then implement always block that is sensitive on the inputs (instruction and clk), instructions are supplied to this arrangement in the form of 32-bit numbers. The format of these instructions is as follows:

• The first 6 bits identify the opcode, after getting a value to the opcode I compared it with the stored values (using for loop) to check if the entered opcode is valid or not and to store the value in 'valid_opcode' register

• The next 5 bits identify first source register => goes to Address1 in the reg_file

• The next 5 bits identify second source register => goes to Address2 in the reg_file

• The next 5 bits identify destination register => goes to Address3 in the reg_file

• The final 11 bits are unused

• The mp_top result will take result of the alu module that I called (reg_in), but after delay #10 to make sure that the value is calculated.

```
always @(posedge clk, instruction)begin

  opcode <= instruction[5:0];
  valid_opcode <= 1'b0;

  for (int i = 0; i <= 10; i = i + 1)begin

     if (opcode == stored_opcodes[i]) begin
             valid_opcode <= 1'b1;
     end

   end

  addr1 <= instruction[10:6];
  addr2 <= instruction[15:11];

  addr3 <= instruction[20:16];
  #10
  result = reg_in;

end
```

So, this module in general represent the top module where we connect PART1 blocks (alu and reg_file) together, and this module is ready to receive 2 inputs (clock and instruction) and to give one output (result), by sending the two modules the values they need to do operations on some values.

2.2.2 testbench:

Finally finish with the testbench for (module mp_top), which (mp_top) contains the two modules (alu and reg_file):

Implement the wires and registers to call the mp_top module to start testing:

```
//----signals declaration for connecting design module----//
reg clk;
reg [31:0] instruction;
wire [31:0] result;
reg [5:0] opcode;
reg [31:0] expected_result [13:0];
reg valid_opcode;
reg pass;

//----design module----//
mp_top uut (
  .clk(clk),
  .instruction(instruction),
  .result(result)
    );
```

I also implemented some other registers:

- → Opcode: to check if the enterd opcode is valid or not
- → Valid_opcode (by comparing the opcode I get from instructions with the stored opcode (for loop)):
  - 0=> invalid opcode (won't do the comparison with the expected result and will print 'invalid opcode' near the instruction)
  - 1=> valid opcode (will calculate the value and then compare it with the stored expected result (will print correct or incorrect near the instruction))
- → pass to check if the program works correctly or not (at the end of the program will print 'PASS' or 'FAIL' depending on the result we will get from the module (and given instruction))
- → expexted_result to test if the mp_top module gives a correct answer or not (I calculated them and stored their values)

By implementing a list of instruction to test the module (I checked all the cases), and followed by their expected results:

```
//------------------------------//
reg [31:0] instructions_array [13:0];
initial begin
  instructions_array[0]= 32'h00000000; expected_result[0]= 32'hXXXXXXXX;   //invalid
  instructions_array[1]= 32'h001f1043; expected_result[1]= 32'h00002642;   // +
  instructions_array[2]= 32'h00001841; expected_result[2]= 32'h00001066;   // min()
  instructions_array[3]= 32'h00047a85; expected_result[3]= 32'h00003ec4;   // XOR
  instructions_array[4]= 32'h00047a88; expected_result[4]= 32'hXXXXXXXX;   // invalid
  instructions_array[5]= 32'h00001049; expected_result[5]= 32'h00001321;   // avg()
  instructions_array[6]= 32'h0003080d; expected_result[6]= 32'h00001321;   // | |
  instructions_array[7]= 32'h001def0c; expected_result[7]= 32'hfffffe572;  // -(a)
  instructions_array[8]= 32'h00006b0f; expected_result[8]= 32'h00001d3c;   // -
  instructions_array[9]= 32'h001f020a; expected_result[9]= 32'hffffefc9;   // not
  instructions_array[10]= 32'h0013944e; expected_result[10]= 32'h000033e4; // OR
  instructions_array[11]= 32'h00005247; expected_result[11]= 32'h00001906; //max()
  instructions_array[12]= 32'h00047a82; expected_result[12]= 32'hXXXXXXXX; // invalid
  instructions_array[13]= 32'h0000ce0b; expected_result[13]= 32'h00001000; // AND
end
//------------------------------//
```

Then start the clock generation, and initialize inputs:

```
// clock generation
 always begin
   #10 clk = ~clk;
  end

 initial begin
 // Initialize Inputs
 clk = 1;
 pass = 1'b1;

 $display("\n");
 $monitor(" -> instruction=%b, -> result=%h",instruction, result);

 #100
```

For testing all instruction I implemented a for loop [0 →13], between instruction and next instruction added delay #100, then take the first 6 bit from the instruction to identify the opcode, after having a value for the opcode I checked if it's valid or not by comparing it with the stored opcodes (using for loop):

- → if valid_opcode = 1: get the the value (result) and compare it with the expected result stored: if equal then correct, if not equal then incorrect
- → if valid_opcode = 0: won't compare the result and will print 'invalid opcode' near the instruction

at the end of testing all the instructions and based on the value of the 'pass' register the program will print 'PASS' or 'FAIL'

```
 for (int j = 0; j <= 13; j = j + 1)begin

 #100
   instruction = instructions_array[j];
 #100

   opcode = instruction[5:0];
   valid_opcode = 1'b0;

   for (int i = 0; i <=10; i = i + 1)begin

      if (opcode == stored_opcodes[i]) begin
            valid_opcode = 1'b1;
      end

   end
   if(valid_opcode) begin

     if (expected_result[j] == result)
       $display(" correct \n ");
     else
       begin
         pass= 1'b0;
         $display(" incorrect");
       end
     end
   else
      $display(" invalid opcode \n");
 end

  // Add some delay to observe results
  #1000;
 if (pass)
   $display(" ==> PASS ( This program works correctly ) \n");
 else
   $display(" ==> FAIL (This program does NOT work correctly!!! \n )");

  $finish; // Finish simulation
```

3. CALCULATIONS

```
instructions_array[0]= 32'h00000000; expected_result[0]= 32'hXXXXXXXX;   //invalid
instructions_array[1]= 32'h001f1043; expected_result[1]= 32'h00002642;   // +
instructions_array[2]= 32'h00001841; expected_result[2]= 32'h00001066;   // min()
instructions_array[3]= 32'h00047a85; expected_result[3]= 32'h00003ec4;   // XOR
instructions_array[4]= 32'h00047a88; expected_result[4]= 32'hXXXXXXXX;   // invalid
instructions_array[5]= 32'h00001049; expected_result[5]= 32'h00001321;   // avg()
instructions_array[6]= 32'h0003080d; expected_result[6]= 32'h00001321;   // | |
instructions_array[7]= 32'h001def0c; expected_result[7]= 32'hfffffe572;  // -(a)
instructions_array[8]= 32'h00006b0f; expected_result[8]= 32'h00001d3c;   // -
instructions_array[9]= 32'h001f020a; expected_result[9]= 32'hfffffefc9;  // not
instructions_array[10]= 32'h0013944e; expected_result[10]= 32'h000033e4; // OR
instructions_array[11]= 32'h00005247; expected_result[11]= 32'h00001906; //max()
instructions_array[12]= 32'h00047a82; expected_result[12]= 32'hXXXXXXXX; // invalid
instructions_array[13]= 32'h0000ce0b; expected_result[13]= 32'h00001000; // AND
```

0) The first instruction have invalid opcode to start the program

1) Instruction[1]: 0000 0000 0001 1111 0001 0000 0100 0011
=> Opcode: 000011 (+) =>addr1:00001 (item [1] =>addr2:00010 (item [2]) =>addr3:00000
Getting the items and preform addition: 32'h00001066 + 32'h000015dc = **32'h00002642** stored in item [0]

2) Instruction[2]: 0000 0000 0000 0000 0001 1000 0100 0001
=> Opcode: 000001 (MIN.) =>addr1:00001 (item [1]) =>addr2:00011 (item [2]) =>addr3:00000
Getting the items and find min: min (32'h00001066, 32'h0000 385a) = **32'h00001066** stored in item [0]

3) Instruction[3]: 0000 0000 0000 0100 0111 1010 1000 0101
=> Opcode: 000101 (XOR) =>addr1:01010 (item [10]) =>addr2:01111 (item [15]) =>addr3:00100
Getting the items and preform XOR: 32'h00001518 XOR 32'h00002bcd= **32'h00003ec4** stored in item [4]

4) The fifth instruction have invalid opcode

5) Instruction[5]: 0000 0000 0000 0000 0001 0000 0100 1001
=> Opcode: 001001 (avg) =>addr1:00001 (item [1]) =>addr2:00010 (item [2]) =>addr3:00000
Getting the items and preform AVG.: avg(32'h00001066, 32'h000015dc)= **32'h00001321** stored in item [0]

6) Instruction[6]: 0000 0000 0000 0011 0000 1000 0000 1101
=> Opcode: 001101 (| a |) =>addr1:00000 (item [0]) =>addr2:00001 (item [1]) =>addr3:00011
Getting the items and preform abs.: |32'h00001321| = **32'h00001321** stored in item [3]

7) Instruction[7]: 0000 0000 0001 1101 1110 1111 0000 1100
=> Opcode: 001100 (-a) =>addr1:11100 (item [28]) =>addr2:11101 (item [29]) =>addr3:11101
Getting the items and preform -a: -(32'h00001a8e) = **32'hfffffe572** stored in item [29]

8) Instruction[8]: 0000 0000 0000 0000 0110 1011 0000 1111
=> Opcode: 001111 (-) =>addr1:01100 (item [12]) =>addr2:01111 (item [15]) =>addr3:00000
Getting the items and preform subtraction: (32'h00003fc4 – 32'h00002bdc) = **32'h00001d3c** stored in item [0]

9) Instruction[9]: 0000 0000 0001 1111 0000 0010 0000 1010
=> Opcode: 001010(not) =>addr1:001000 (item [8]) =>addr2:01010 (item [10]) =>addr3:11111
Getting the items and preform NOT (a): NOT (32'h00001036) = **32'hfffffefc9** stored in item [31]

10) Instruction[10]: 0000 0000 0001 0011 1001 0100 0100 1110
=> Opcode: 001110 (OR) =>addr1:10001 (item [17]) =>addr2:10010 (item [18]) =>addr3:10011
Getting the items and preform OR: 32'h000033e4 OR 32'h 00001244 = **32'h000033e4** stored in item [19]

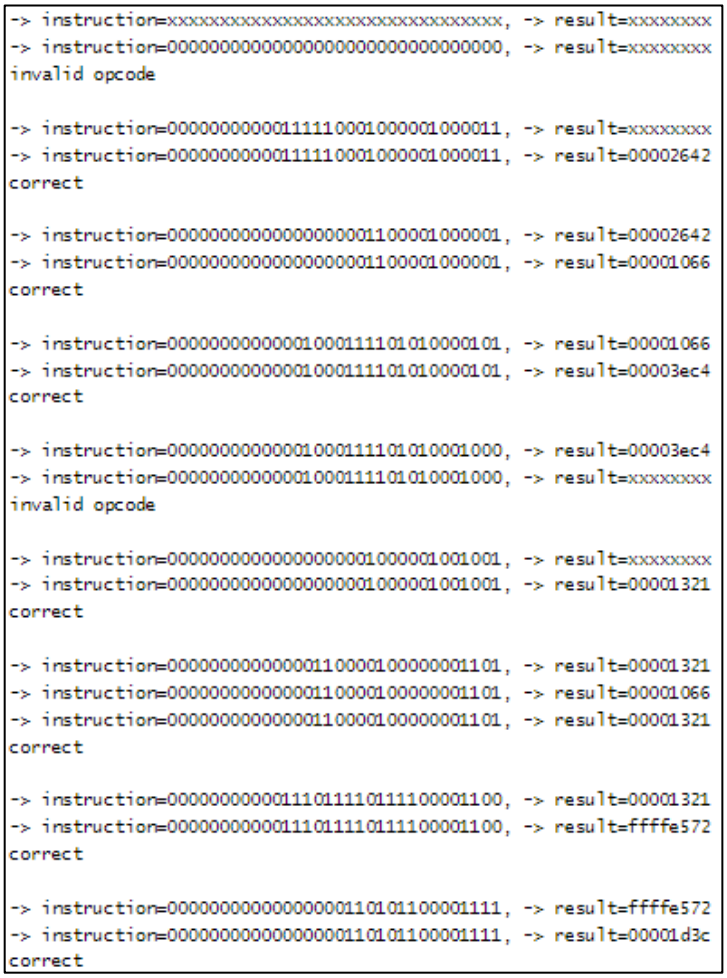
11) Instruction[11]: 0000 0000 0000 0000 0101 0010 0100 0111
=> Opcode: 000111(MAX.) =>addr1:01001 (item [9]) =>addr2:01010 (item [10]) =>addr3:00000
Getting the items and find max.: max(32'h00001906, 32'h00001518) = **32'h00001906** stored in item [0]

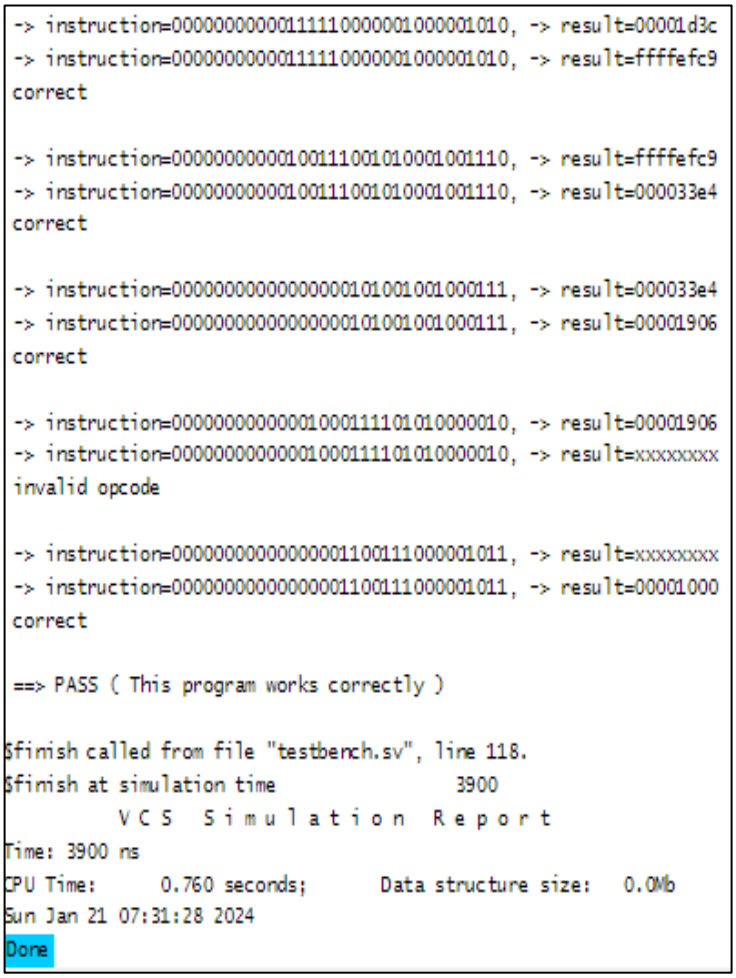
12) The 12th instruction have invalid opcode

13) Instruction[13]: 0000 0000 0000 0100 0111 1010 1000 0010
=> Opcode: 000010(AND) =>addr1:11000 (item [24]) =>addr2:11001 (item [25]) =>addr3:00000
Getting the items and perform AND.: 32'h000030c8 AND 32'h00001a00 = **32'h00001000** stored in item [0]

4. RESULTS AND CONCLUSION (Appendix)

The results I got after running the testbench (14 instructions 3 of them have invalid opcodes and the rest cover all the cases (11)):





As previously shown in the figures, the program will:

→ Take each instruction and calculate it's result
→ The first line, the instruction won't get it's value yet; because it is not calculated yet
→ The instruction will be separated into opcode, addr1, addr2, and addr3 to give them as inputs for reg_file
→ Then search for matched stored items to give values for out1 and out2
→ These values will enter the ALU as inputs (a and b)
→ And based on the opcode the ALU will calculate the result
→ Then the calculated result will be given to the mp_top module result as a final output (shown in the second line for each instruction calculations)

So, in the second line the instruction will have the final value, after considering that I delayed some values to enter modules as inputs or put them as output, to get the program working correctly.

At the end of the program and based on the value of 'pass' register:

- Pass=0; if at least there is one incorrect value and will print: 'FAIL (This program does NOT work correctly!!!'
- Pass=1; if all the results of all the instructions are correct (instructions with invalid opcodes are not involved), and will print: 'PASS (This program works correctly)'

So, as shown I tested all the cases (operations) and i have got all of them correct (the instructions calculated results through the mp_top matches their expected results stores and calculated by me).

As a result: the program is working correctly ⇒ PASS.