Faculty of Engineering & Technology
Electrical & Computer Engineering Department

ADVANCED DIGITAL SYSTEMS DESIGN

-ENCS3310-

First semester 2023-2024

ADVANCED DIGITAL SYSTEMS DESIGN Project

Prepared by:	Qusay Taradeh
ID Number:	1212508
Section Number:	3

Date:	21-1-2024

Abstract:

This project endeavors to construct a microprocessor characterized by a register file functioning as a compact and expeditious Random-Access Memory (RAM), complemented by an Arithmetic and Logic Unit (ALU) serving as the computational nucleus for processing operation codes (opcodes) provided to it. The proposed methodology involves the composition of Verilog code for each constituent component or module, subsequently interlinking them in a cohesive manner. Additionally, a comprehensive testbench will be developed to assess the functionality of the microprocessor. This testbench will encompass a series of instructions specifying opcodes and the corresponding addresses for the retrieval and storage of numerical data within the register file. The successful implementation of these procedures will serve as a validation of the constructed microprocessor's robustness and reliability.

Part One:

1.1 Arithmetic And Logic Unit (ALU):

This Unit do various operations based on 6-bit operation code (opcode) given on two 32-bit numbers symbolized by (a) and (b) and output the 32-bit result, the numbers given to it are from register file by addresses of them extracted from 32-bit instruction.

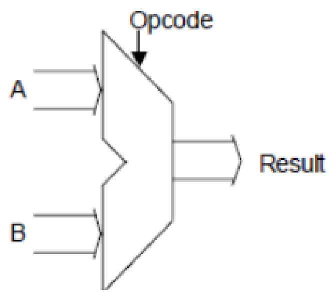


Fig.1.1: ALU

The operations that ALU can do following opcode based on last digit of my ID number 121250**8** in decimal are listed below in Table:

Opcode	**Operation**	**Description**
1	a + b	Addition
6	a - b	Subtraction
13	|a|	Absolute value of ‘a’
8	-a	Negative value of ‘a’
7	max (a, b)	Maximum between ‘a’ and ‘b’
4	min (a, b)	Minimum between ‘a’ and ‘b’
11	avg (a, b)	Average value of ‘a’ and ‘b’
15	not a	reverse all bits of ‘a’
3	a OR b	OR between ‘a’ and ‘b’
5	a AND b	AND between ‘a’ and ‘b’
2	a XOR b	exclusive OR between ‘a’ and ‘b’

How it works: We defined all opcodes as parameters with names associated with operation work, make a case statement to cover all opcodes to check if given opcode equals one of them, and on positive edge result of operation will assigned the value immediately "blocking", so on positive edge of current clock the result will appear.

How we tested it: We defined two numbers and change opcode to fit all opcodes with some delay between each case, to wait for operating the right result, since we define 'a' and 'b' as signed numbers then the result would be signed, from results since 'a' is positive then the absolute value also positive. Therefore, all operations successfully worked as expected in each operation, so our design is right.

The testbench results that verify the unit implemented successfully set in **Appendix 1**

1.2 Register File:

This File as a small fast RAM but it's implemented inside microprocessor, it stores 32 locations of wide 32-bit for each, these locations addresses from 0 to 31 store numbers that be passed to ALU at "out1" and "out2" to do various operations, also it can store the result of the ALU "in" in location addressed by address entered to register file "Address 3" extracted from 32-bit instruction.

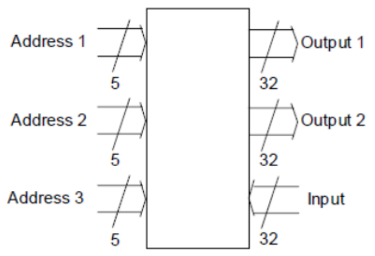
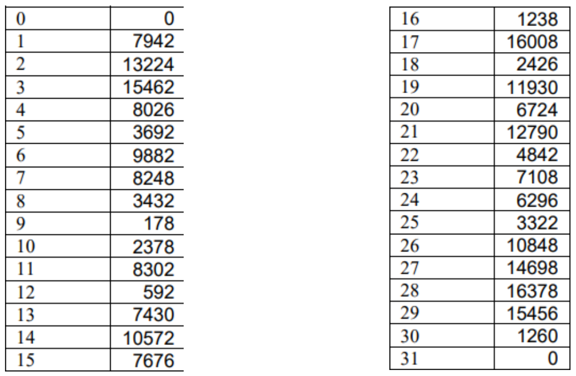


Fig.1.2: Register File

The register file contents based on second last digit of my ID number 121250**8** which is zero got from project file from index 0 to index 31 below in Fig 1.2.1:



0	0
1	7942
2	13224
3	15462
4	8026
5	3692
6	9882
7	8248
8	3432
9	178
10	2378
11	8302
12	592
13	7430
14	10572
15	7676
16	1238
17	16008
18	2426
19	11930
20	6724
21	12790
22	4842
23	7108
24	6296
25	3322
26	10848
27	14698
28	16378
29	15456
30	1260
31	0

Fig.1.2.1: Table of stored values in Register file

Part Two:

2.1 Clocking the Register File:

To make sure that addresses and input to register file in right way, we may clock the register file on the clocks' edge, such that addresses of both numbers should be first passed to get the numbers from the file, then at next edge we got the address of result that should stored in the file at the end of successfully execution, in this case we used non-blocking assignment to put addresses and input value after positive edge of clock.

2.2 Enabling the Register File:

To use register file firstly we should enable it by checking the opcode if valid, if invalid then no matter to do anything; because it isn't enabled.

How it works: when enabled by valid opcode, get 3 addresses from input , first 2 to load 2 output values respectively based on addresses numbers, the third address is the address on which location we will store the value entered in input, load and store will not be on same moment to avoid any error or any unintended value, that is by "non-blocking" such that on positive edge all inputs come and after positive edge in order load before store and loaded value should be real value stored after this the value will be updated by store (just if any address or both first 2 equal last address of store), otherwise store can first normally, but to avoid any error as we mentioned we used non-blocking assignment.

How we tested it: we defined enable register as opcode valid , addresses , input value , input addresses and 2 output values, then initialize 3 addresses such that address two and address three equals each other, to test load and store are work successfully, and make clock, so at positive edge and valid opcode 2 outputs are ready then disabled to show the result with some delay, then again enable to test if new value stored in location of old value.

The testbench result shows that our register file design works right, as out1 loaded value shows the content of location addressed by addr1 and out2 also, but addr1 which equals addr2 store the value "0xF1234567" in same location, so non-blocking is right solution as load from location[addr2] must be before store new value in it. Therefore, from simulation results the load done before store such that old value was "0x00000D68" and new value is "0xF1234567" loaded and shown in last output line, so our design is right. The results are set in **Appendix 1.**

2.3 Creating the core of the microprocessor:

In order to manage ALU and Register file we should implement a simple microprocessor that connect them, and extract parameters of each one from the instruction provided. Therefore, the result of this microprocessor represents the result out from ALU after do operation on numbers.

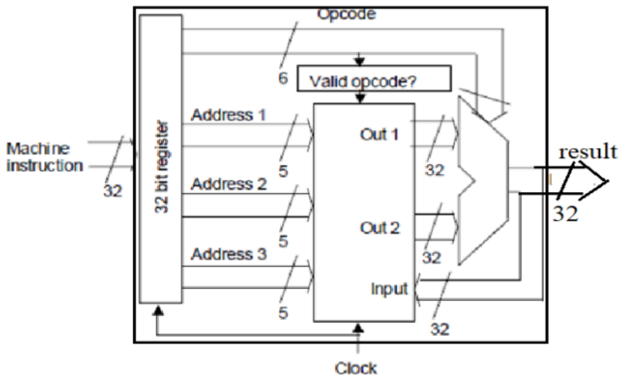


Fig.2.3: Microprocessor

Machine instruction 32-bit format:

|31 (11-bit) 21 | 20 (5-bit) 16 | 15 (5-bit) 11 | 10 (5-bit) 6 | 5 (6-bit) 0|

Unused	Address 3	Address 2	Address 1	Opcode

How it works:

It connects ALU with register file, at positive edge get the 32-bit instruction, then extract all addresses of loaded values and value that should store. Therefore, first test if opcode valid to enable register file before give values from it, if not then not extract and again check till opcode valid or next instruction opcode valid. To add to that, in result of validity opcode giving addresses of input numbers to ALU and address of storing the result, then make some delaying on passing opcode to ALU till numbers come from the register file. After values come from register file result of them will be stored in the location of file addressed by address provided in the instruction, but showing the result of microprocessor after storing to guarantee that result stored correctly.

How we tested it: We initialize all inputs and outputs of other components that connect them, input instruction, clock and output result that obtained from ALU when opcode valid such that all valid opcodes in range determined in “if statements” compare each given opcode with them. To add to that, instantiate components and pass parameters then at positive edge extract opcode from the instruction and check it, if valid then enable register file, make some delay on passing opcode to ALU till values come then calculate the result, store it then output it from micro.
In testbench, we initialized the register file values and valid opcode again to calculate expected value of any given instruction, then we initialize array of 11 instructions to pass them in micro., now at positive edge instruction passed to micro. Calculate expected result by function calculate() then after 2 cycles (to guarantee that result obtained in right way) check if result identical with expected or not, by task called check() that take result and the expected, then print “Pass” if both identical and “Fail” if not.

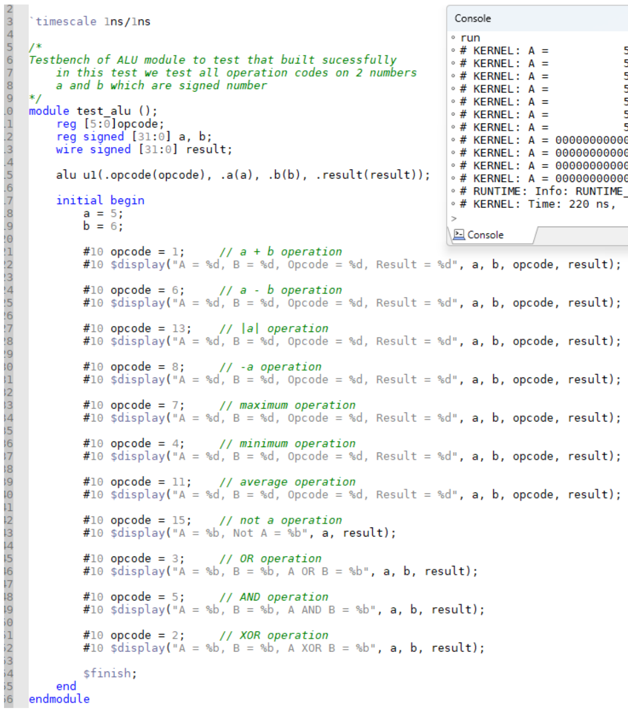
***Note:** Making the array of instructions done by a website that take the format of instruction and make instructions by typing operation as assembly (i.e.: Add r31, r1, r2 that r31 = r1 + r2) make instruction of ADD opcode and addresses depending on numbers after ‘r’.

The testbench results shown in **Appendix 1** approved that our design is correct and works as expected.

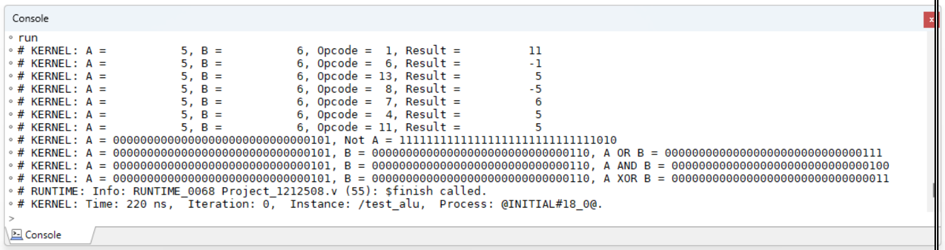
Appendix 1:

ALU:

Testbench code and results:



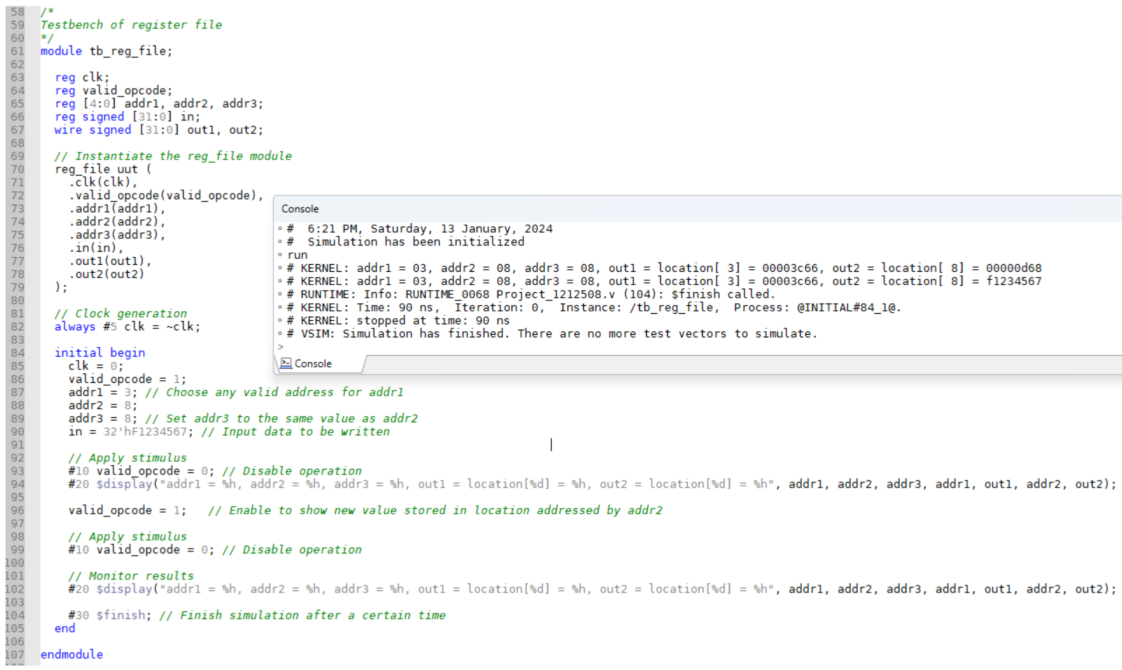
```verilog
`timescale 1ns/1ns

/*
Testbench of ALU module to test that built sucessfully
    in this test we test all operation codes on 2 numbers
    a and b which are signed number
*/
module test_alu ();
    reg [5:0]opcode;
    reg signed [31:0] a, b;
    wire signed [31:0] result;

    alu u1(.opcode(opcode), .a(a), .b(b), .result(result));

    initial begin
        a = 5;
        b = 6;

        #10 opcode = 1;     // a + b operation
        #10 $display("A = %d, B = %d, Opcode = %d, Result = %d", a, b, opcode, result);

        #10 opcode = 6;     // a - b operation
        #10 $display("A = %d, B = %d, Opcode = %d, Result = %d", a, b, opcode, result);

        #10 opcode = 13;    // |a| operation
        #10 $display("A = %d, B = %d, Opcode = %d, Result = %d", a, b, opcode, result);

        #10 opcode = 8;     // -a operation
        #10 $display("A = %d, B = %d, Opcode = %d, Result = %d", a, b, opcode, result);

        #10 opcode = 7;     // maximum operation
        #10 $display("A = %d, B = %d, Opcode = %d, Result = %d", a, b, opcode, result);

        #10 opcode = 4;     // minimum operation
        #10 $display("A = %d, B = %d, Opcode = %d, Result = %d", a, b, opcode, result);

        #10 opcode = 11;    // average operation
        #10 $display("A = %d, B = %d, Opcode = %d, Result = %d", a, b, opcode, result);

        #10 opcode = 15;    // not a operation
        #10 $display("A = %b, Not A = %b", a, result);

        #10 opcode = 3;     // OR operation
        #10 $display("A = %b, B = %b, A OR B = %b", a, b, result);

        #10 opcode = 5;     // AND operation
        #10 $display("A = %b, B = %b, A AND B = %b", a, b, result);

        #10 opcode = 2;     // XOR operation
        #10 $display("A = %b, B = %b, A XOR B = %b", a, b, result);

        $finish;
    end
endmodule
```



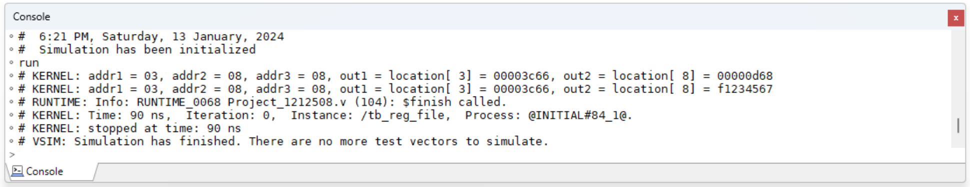
```
Console
◦ run
◦ # KERNEL: A =           5, B =           6, Opcode =  1, Result =          11
◦ # KERNEL: A =           5, B =           6, Opcode =  6, Result =          -1
◦ # KERNEL: A =           5, B =           6, Opcode = 13, Result =           5
◦ # KERNEL: A =           5, B =           6, Opcode =  8, Result =          -5
◦ # KERNEL: A =           5, B =           6, Opcode =  7, Result =           6
◦ # KERNEL: A =           5, B =           6, Opcode =  4, Result =           5
◦ # KERNEL: A =           5, B =           6, Opcode = 11, Result =           5
◦ # KERNEL: A = 00000000000000000000000000000101, Not A = 11111111111111111111111111111010
◦ # KERNEL: A = 00000000000000000000000000000101, B = 00000000000000000000000000000110, A OR B = 00000000000000000000000000000111
◦ # KERNEL: A = 00000000000000000000000000000101, B = 00000000000000000000000000000110, A AND B = 00000000000000000000000000000100
◦ # KERNEL: A = 00000000000000000000000000000101, B = 00000000000000000000000000000110, A XOR B = 00000000000000000000000000000011
◦ # RUNTIME: Info: RUNTIME_0068 Project_1212508.v (55): $finish called.
◦ # KERNEL: Time: 220 ns,  Iteration: 0,  Instance: /test_alu,  Process: @INITIAL#18_0@.
>
Console
```

Register File:

Testbench code and results:



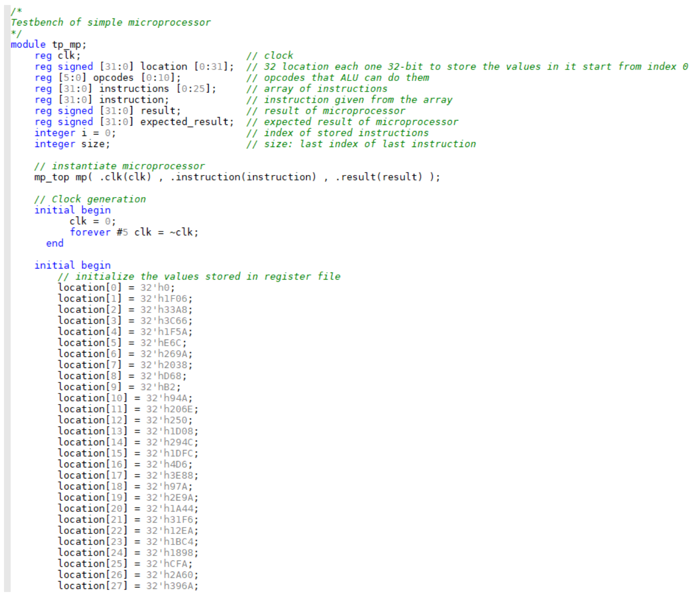
```verilog
/*
Testbench of register file
*/
module tb_reg_file;

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

  // Instantiate the reg_file module
  reg_file uut (
    .clk(clk),
    .valid_opcode(valid_opcode),
    .addr1(addr1),
    .addr2(addr2),
    .addr3(addr3),
    .in(in),
    .out1(out1),
    .out2(out2)
  );

  // Clock generation
  always #5 clk = ~clk;

  initial begin
    clk = 0;
    valid_opcode = 1;
    addr1 = 3; // Choose any valid address for addr1
    addr2 = 8;
    addr3 = 8; // Set addr3 to the same value as addr2
    in = 32'hF1234567; // Input data to be written

    // Apply stimulus
    #10 valid_opcode = 0; // Disable operation
    #20 $display("addr1 = %h, addr2 = %h, addr3 = %h, out1 = location[%d] = %h, out2 = location[%d] = %h", addr1, addr2, addr3, addr1, out1, addr2, out2);

    valid_opcode = 1;   // Enable to show new value stored in location addressed by addr2

    // Apply stimulus
    #10 valid_opcode = 0; // Disable operation

    // Monitor results
    #20 $display("addr1 = %h, addr2 = %h, addr3 = %h, out1 = location[%d] = %h, out2 = location[%d] = %h", addr1, addr2, addr3, addr1, out1, addr2, out2);

    #30 $finish; // Finish simulation after a certain time
  end

endmodule
```



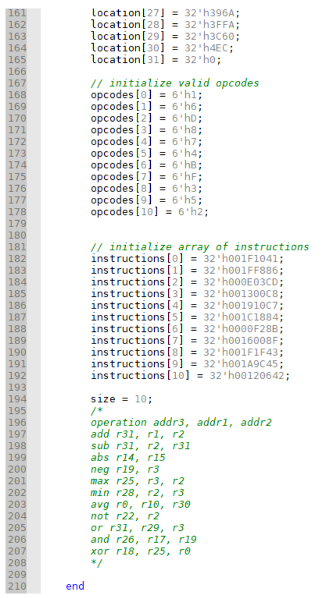
```
Console
◦ #  6:21 PM, Saturday, 13 January, 2024
◦ #  Simulation has been initialized
◦ run
◦ # KERNEL: addr1 = 03, addr2 = 08, addr3 = 08, out1 = location[ 3] = 00003c66, out2 = location[ 8] = 00000d68
◦ # KERNEL: addr1 = 03, addr2 = 08, addr3 = 08, out1 = location[ 3] = 00003c66, out2 = location[ 8] = f1234567
◦ # RUNTIME: Info: RUNTIME_0068 Project_1212508.v (104): $finish called.
◦ # KERNEL: Time: 90 ns,  Iteration: 0,  Instance: /tb_reg_file,  Process: @INITIAL#84_1@.
◦ # KERNEL: stopped at time: 90 ns
◦ # VSIM: Simulation has finished. There are no more test vectors to simulate.
>
Console
```

Microprocessor (Top):

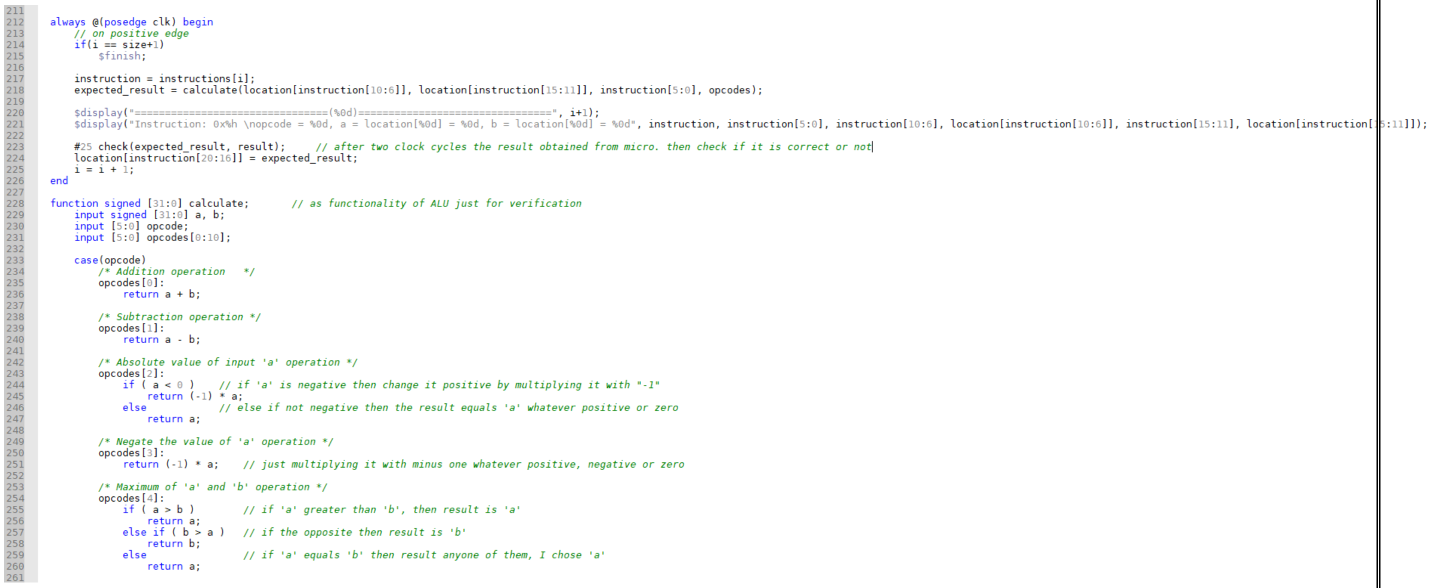
Testbench code and results:



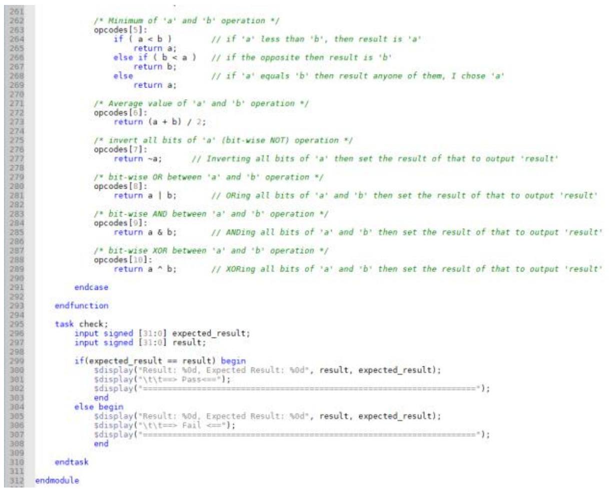
```verilog
/*
Testbench of simple microprocessor
*/
module tp_mp;
    reg clk;                            // clock
    reg signed [31:0] location [0:31];  // 32 location each one 32-bit to store the values in it start from index 0
    reg [5:0] opcodes [0:10];           // opcodes that ALU can do them
    reg [31:0] instructions [0:25];     // array of instructions
    reg [31:0] instruction;             // instruction given from the array
    reg signed [31:0] result;           // result of microprocessor
    reg signed [31:0] expected_result;  // expected result of microprocessor
    integer i = 0;                      // index of stored instructions
    integer size;                       // size: last index of last instruction

    // instantiate microprocessor
    mp_top mp( .clk(clk) , .instruction(instruction) , .result(result) );

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

    initial begin
        // initialize the values stored in register file
        location[0] = 32'h0;
        location[1] = 32'h1F06;
        location[2] = 32'h33A8;
        location[3] = 32'h3C66;
        location[4] = 32'h1F5A;
        location[5] = 32'hE6C;
        location[6] = 32'h269A;
        location[7] = 32'h2038;
        location[8] = 32'hD68;
        location[9] = 32'hB2;
        location[10] = 32'h94A;
        location[11] = 32'h206E;
        location[12] = 32'h250;
        location[13] = 32'h1D08;
        location[14] = 32'h294C;
        location[15] = 32'h1DFC;
        location[16] = 32'h4D6;
        location[17] = 32'h3E88;
        location[18] = 32'h97A;
        location[19] = 32'h2E9A;
        location[20] = 32'h1A44;
        location[21] = 32'h31F6;
        location[22] = 32'h12EA;
        location[23] = 32'h1BC4;
        location[24] = 32'h1898;
        location[25] = 32'hCFA;
        location[26] = 32'h2A60;
        location[27] = 32'h396A;
```



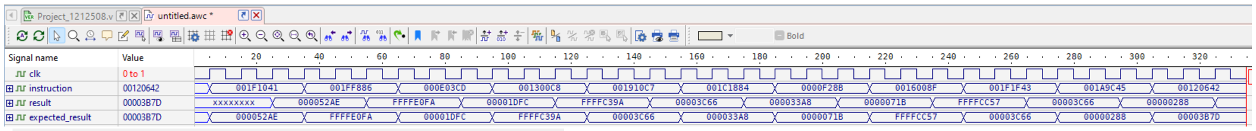
```verilog
        location[27] = 32'h396A;
        location[28] = 32'h3FFA;
        location[29] = 32'h3C60;
        location[30] = 32'h4EC;
        location[31] = 32'h0;

        // initialize valid opcodes
        opcodes[0] = 6'h1;
        opcodes[1] = 6'h6;
        opcodes[2] = 6'hD;
        opcodes[3] = 6'h8;
        opcodes[4] = 6'h7;
        opcodes[5] = 6'h4;
        opcodes[6] = 6'hB;
        opcodes[7] = 6'hF;
        opcodes[8] = 6'h3;
        opcodes[9] = 6'h5;
        opcodes[10] = 6'h2;


        // initialize array of instructions
        instructions[0] = 32'h001F1041;
        instructions[1] = 32'h001FF886;
        instructions[2] = 32'h000E03CD;
        instructions[3] = 32'h001300C8;
        instructions[4] = 32'h001910C7;
        instructions[5] = 32'h001C1884;
        instructions[6] = 32'h0000F28B;
        instructions[7] = 32'h0016008F;
        instructions[8] = 32'h001F1F43;
        instructions[9] = 32'h001A9C45;
        instructions[10] = 32'h00120642;

        size = 10;
        /*
        operation addr3, addr1, addr2
        add r31, r1, r2
        sub r31, r2, r31
        abs r14, r15
        neg r19, r3
        max r25, r3, r2
        min r28, r2, r3
        avg r0, r10, r30
        not r22, r2
        or r31, r29, r3
        and r26, r17, r19
        xor r18, r25, r0
        */

    end
```



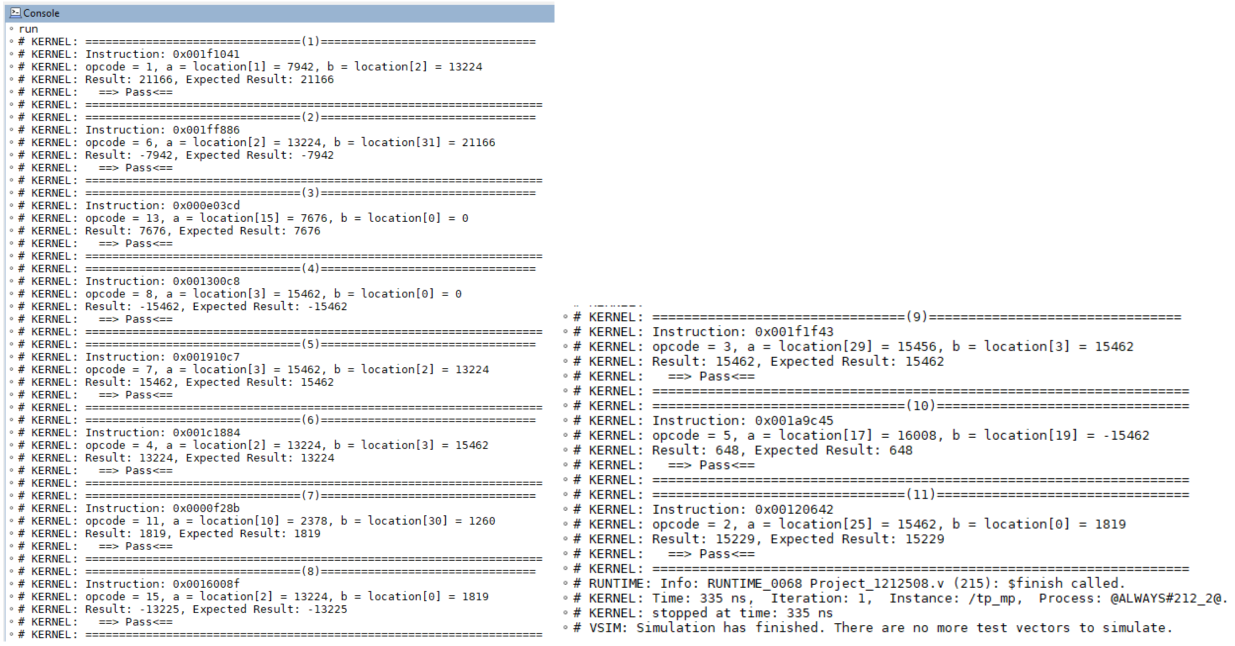
```verilog

    always @(posedge clk) begin
        // on positive edge
        if(i == size+1)
            $finish;

        instruction = instructions[i];
        expected_result = calculate(location[instruction[10:6]], location[instruction[15:11]], instruction[5:0], opcodes);

        $display("===============================(%0d)===============================", i+1);
        $display("Instruction: 0x%h \nopcode = %0d, a = location[%0d] = %0d, b = location[%0d] = %0d", instruction, instruction[5:0], instruction[10:6], location[instruction[10:6]], instruction[15:11], location[instruction[15:11]]);

        #25 check(expected_result, result);     // after two clock cycles the result obtained from micro. then check if it is correct or not
        location[instruction[20:16]] = expected_result;
        i = i + 1;
    end

    function signed [31:0] calculate;       // as functionality of ALU just for verification
        input signed [31:0] a, b;
        input [5:0] opcode;
        input [5:0] opcodes[0:10];

        case(opcode)
            /* Addition operation   */
            opcodes[0]:
                return a + b;

            /* Subtraction operation */
            opcodes[1]:
                return a - b;

            /* Absolute value of input 'a' operation */
            opcodes[2]:
                if ( a < 0 )    // if 'a' is negative then change it positive by multiplying it with "-1"
                    return (-1) * a;
                else            // else if not negative then the result equals 'a' whatever positive or zero
                    return a;

            /* Negate the value of 'a' operation */
            opcodes[3]:
                return (-1) * a;    // just multiplying it with minus one whatever positive, negative or zero

            /* Maximum of 'a' and 'b' operation */
            opcodes[4]:
                if ( a > b )        // if 'a' greater than 'b', then result is 'a'
                    return a;
                else if ( b > a )   // if the opposite then result is 'b'
                    return b;
                else                // if 'a' equals 'b' then result anyone of them, I chose 'a'
                    return a;

```



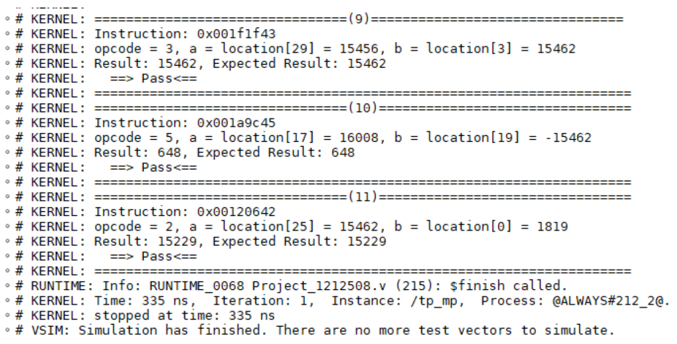
```verilog
            /* Minimum of 'a' and 'b' operation */
            opcodes[5]:
                if ( a < b )        // if 'a' less than 'b', then result is 'a'
                    return a;
                else if ( b < a )   // if the opposite then result is 'b'
                    return b;
                else                // if 'a' equals 'b' then result anyone of them, I chose 'a'
                    return a;

            /* Average value of 'a' and 'b' operation */
            opcodes[6]:
                return (a + b) / 2;

            /* invert all bits of 'a' (bit-wise NOT) operation */
            opcodes[7]:
                return ~a;      // Inverting all bits of 'a' then set the result of that to output 'result'

            /* bit-wise OR between 'a' and 'b' operation */
            opcodes[8]:
                return a | b;       // ORing all bits of 'a' and 'b' then set the result of that to output 'result'

            /* bit-wise AND between 'a' and 'b' operation */
            opcodes[9]:
                return a & b;       // ANDing all bits of 'a' and 'b' then set the result of that to output 'result'

            /* bit-wise XOR between 'a' and 'b' operation */
            opcodes[10]:
                return a ^ b;       // XORing all bits of 'a' and 'b' then set the result of that to output 'result'

        endcase

    endfunction

    task check;
        input signed [31:0] expected_result;
        input signed [31:0] result;

        if(expected_result == result) begin
            $display("Result: %0d, Expected Result: %0d", result, expected_result);
            $display("\t\t==> Pass<==");
            $display("=====================================================================");
            end
        else begin
            $display("Result: %0d, Expected Result: %0d", result, expected_result);
            $display("\t\t==> Fail <==");
            $display("=====================================================================");
            end

    endtask

endmodule
```



```
Signal name       Value      20        40        60        80        100       120       140       160       180       200       220       240       260       280       300       320
clk               0 to 1
instruction       00120642   001F1041  001FF886  000E03CD  001300C8  001910C7  001C1884  0000F28B  0016008F  001F1F43  001A9C45  00120642
result            00003B7D   XXXXXXXX  000052AE  FFFFE0FA  00001DFC  FFFFC39A  00003C66  000033A8  0000071B  FFFFCC57  00003C66  00000288
expected_result   00003B7D   000052AE  FFFFE0FA  00001DFC  FFFFC39A  00003C66  000033A8  0000071B  FFFFCC57  00003C66  00000288  00003B7D
```



```
◦ run
◦ # KERNEL: ================================(1)================================
◦ # KERNEL: Instruction: 0x001f1041
◦ # KERNEL: opcode = 1, a = location[1] = 7942, b = location[2] = 13224
◦ # KERNEL: Result: 21166, Expected Result: 21166
◦ # KERNEL:   ==> Pass<==
◦ # KERNEL: ====================================================================
◦ # KERNEL: ================================(2)================================
◦ # KERNEL: Instruction: 0x001ff886
◦ # KERNEL: opcode = 6, a = location[2] = 13224, b = location[31] = 21166
◦ # KERNEL: Result: -7942, Expected Result: -7942
◦ # KERNEL:   ==> Pass<==
◦ # KERNEL: ====================================================================
◦ # KERNEL: ================================(3)================================
◦ # KERNEL: Instruction: 0x000e03cd
◦ # KERNEL: opcode = 13, a = location[15] = 7676, b = location[0] = 0
◦ # KERNEL: Result: 7676, Expected Result: 7676
◦ # KERNEL:   ==> Pass<==
◦ # KERNEL: ====================================================================
◦ # KERNEL: ================================(4)================================
◦ # KERNEL: Instruction: 0x001300c8
◦ # KERNEL: opcode = 8, a = location[3] = 15462, b = location[0] = 0
◦ # KERNEL: Result: -15462, Expected Result: -15462
◦ # KERNEL:   ==> Pass<==
◦ # KERNEL: ====================================================================
◦ # KERNEL: ================================(5)================================
◦ # KERNEL: Instruction: 0x001910c7
◦ # KERNEL: opcode = 7, a = location[3] = 15462, b = location[2] = 13224
◦ # KERNEL: Result: 15462, Expected Result: 15462
◦ # KERNEL:   ==> Pass<==
◦ # KERNEL: ====================================================================
◦ # KERNEL: ================================(6)================================
◦ # KERNEL: Instruction: 0x001c1884
◦ # KERNEL: opcode = 4, a = location[2] = 13224, b = location[3] = 15462
◦ # KERNEL: Result: 13224, Expected Result: 13224
◦ # KERNEL:   ==> Pass<==
◦ # KERNEL: ====================================================================
◦ # KERNEL: ================================(7)================================
◦ # KERNEL: Instruction: 0x0000f28b
◦ # KERNEL: opcode = 11, a = location[10] = 2378, b = location[30] = 1260
◦ # KERNEL: Result: 1819, Expected Result: 1819
◦ # KERNEL:   ==> Pass<==
◦ # KERNEL: ====================================================================
◦ # KERNEL: ================================(8)================================
◦ # KERNEL: Instruction: 0x0016008f
◦ # KERNEL: opcode = 15, a = location[2] = 13224, b = location[0] = 1819
◦ # KERNEL: Result: -13225, Expected Result: -13225
◦ # KERNEL:   ==> Pass<==
◦ # KERNEL: ====================================================================
```



```
◦ # KERNEL: ================================(9)================================
◦ # KERNEL: Instruction: 0x001f1f43
◦ # KERNEL: opcode = 3, a = location[29] = 15456, b = location[3] = 15462
◦ # KERNEL: Result: 15462, Expected Result: 15462
◦ # KERNEL:   ==> Pass<==
◦ # KERNEL: ====================================================================
◦ # KERNEL: ================================(10)===============================
◦ # KERNEL: Instruction: 0x001a9c45
◦ # KERNEL: opcode = 5, a = location[17] = 16008, b = location[19] = -15462
◦ # KERNEL: Result: 648, Expected Result: 648
◦ # KERNEL:   ==> Pass<==
◦ # KERNEL: ====================================================================
◦ # KERNEL: ================================(11)===============================
◦ # KERNEL: Instruction: 0x00120642
◦ # KERNEL: opcode = 2, a = location[25] = 15462, b = location[0] = 1819
◦ # KERNEL: Result: 15229, Expected Result: 15229
◦ # KERNEL:   ==> Pass<==
◦ # KERNEL: ====================================================================
◦ # RUNTIME: Info: RUNTIME_0068 Project_1212508.v (215): $finish called.
◦ # KERNEL: Time: 335 ns,  Iteration: 1,  Instance: /tp_mp,  Process: @ALWAYS#212_2@.
◦ # KERNEL: stopped at time: 335 ns
◦ # VSIM: Simulation has finished. There are no more test vectors to simulate.
```