Faculty of Engineering and Technology
Electrical and Computer Engineering Department

Advanced Digital Design Project

Name : Osaid Hasan Nur

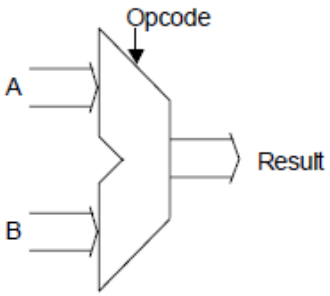
ID : 1210733

Instructor :

Dr. Abdallatif Abuissa

Section : 2

Part 1 : ALU :



The alu takes the opcode , and determine the Operation depending on it .

In the project , we have 11 different opcodes with Different operations :

4 : a+b	5 : a xor b	6 : -a
7 : avg(a,b)	8 : abs(a)	9 : not a
10 : a and b	11 : a-b	12 : a or b
13 : max	14 : min	

Note : The code is supplied in Appendix B

About the Code :

I wrote a simple combinational code , I made a case statement that takes the opcode and outputs the result in a register called “result” in the ALU , and it’s the real output of this Part .

Part 2 : Register File :

The register file acts as internal memory for the microprocessor , it has a read and write ability , and supplies the microprocessor with data in a fast way , and in real circuits , the registers is made of S Ram memory type that have a very high speed in data processing .

Note : The code is supplied in Appendix B

About the code :

When the code first initialize , the values is being stored in the memory as mentioned in the project specification and depending on the last digit of my ID number (3) , after this , the reg_file works as follows :

- We have an enable register called valid_opcode , if it is set to 1 , the reg_file works normally , and if it is set to 0 , the reg_file will have no change status , and no values will be written to it

- After that , the register has two always blocks , one for read , and the other for write , the always block for read don't need to a clock pulse to work , so that it acts as combinational circuit, and if any register of the sensitive list changes , the values of "out1" and "out2" will change immediately .

- On the other hand , the always block for write operation needs a positive edge to work , and this is to prevent any intercect between read and write operation in case the read and write address are the same , and in the write operation , we write the value of "in" to the location passed "addr3"

At the end of clock cycle , we will have "out1" and "out2 " that have the values stored in locations "addr1" and "addr2" in the memory . also we will have a new written value to the memory which is the value of "in" .

Part 3 : mp_top :

This part contains all parts of the microprocessor connected together and acts as one block , and operate as a real microprocessor .

Note : The code is supplied in Appendix B

About the code :

This code contains one always block that work in positive edge of the clock , and the main purpose of this always is to take the instruction and decode it to extract the opcode ,src1 , src2 , dest as the way mentioned in the project describtion , and in this step , I check the opcode if it's valid or not , and put the validity status in a register called "valid" , and this register will be passed to the register file .

After the always block , we have an instance of register file that take the opcode , src1 , src2 and dest produced by the always block , and bring the values from the register file , and put the output on registers "out1" and "out2" . After that we have another instance which is the alu , that takes the ouputs from reg_file module , and give them to the alu ,and produce the output immediately .

To explain things clearly , in my programe I let the read option without any need of clock , so , if any values of inputs of the register changed , the register will read them and pass them to the alu immediately , so , actually , at the first clock cycle ,we will have a result value produced , but it still not written to the memory . At the second clock cycle , it will read another instruction , and produce another result value , and at the same time , when the clock pulse arrives, it will write the value for previous instruction on the memory , and if we focus on the register file code , we will notice that write operation is made blocking while the read operation is make non blocking , so that the write operation will be done before the read operation , this will help that if an instruction contains the same address for read and write , we will not have any conflict .

Part 4 : Test Bench :

I made about 22 instruction in my testbench , and I tried to use all edge cases , including theses cases :

1- We read and write in the same address .
2- We read from the same register and write on it .
3- we use an instruction with invalid opcode .

To test my design , I used a task called expected_result that calculate the expected result for the instructions , and I made an array of instructions , and at the first exectute of the code , I fill it with all expected values , and when I start execute the instructions and get the actual result , I compare every result with the expected result , and determine if the test pass or fail .

Appendix A :

Results for the testbench :

```
run 600ns
# KERNEL: Time : 20000  || Instruction :    2038405  || Result :      13070  || Expected Result :      13070 || Status : PASS
# KERNEL: Time : 40000  || Instruction :    2032262  || Result :       3310  || Expected Result :       3310 || Status : PASS
# KERNEL: Time : 60000  || Instruction :       6791  || Result : 4294961296  || Expected Result : 4294961296 || Status : PASS
# KERNEL: Time : 80000  || Instruction :       1992  || Result :       6535  || Expected Result :       6535 || Status : PASS
# KERNEL: Time : 100000  || Instruction :    2038409  || Result :       6000  || Expected Result :       6000 || Status : PASS
# KERNEL: Time : 120000  || Instruction :    2038410  || Result : 4294961295  || Expected Result : 4294961295 || Status : PASS
# KERNEL: Time : 140000  || Instruction :    2038411  || Result :       4880  || Expected Result :       4880 || Status : PASS
# KERNEL: Time : 160000  || Instruction :    2038412  || Result : 4294966226  || Expected Result : 4294966226 || Status : PASS
# KERNEL: Time : 180000  || Instruction :    2038413  || Result :       8190  || Expected Result :       8190 || Status : PASS
# KERNEL: Time : 200000  || Instruction :    2038414  || Result :       7070  || Expected Result :       7070 || Status : PASS
# KERNEL: Time : 220000  || Instruction :    2070475  || Result :       6000  || Expected Result :       6000 || Status : PASS
# KERNEL: Time : 240000  || Instruction :     328008  || Result : 4294960096  || Expected Result : 4294960096 || Status : PASS
# KERNEL: Time : 260000  || Instruction :     359119  || Result :       3322  || Expected Result :       3322 || Status : PASS
# KERNEL: Invalid opcode !
# KERNEL: Time : 280000  || Instruction :     328008  || Result :       3322  || Expected Result :       3322 || Status : PASS
# KERNEL: Time : 300000  || Instruction :    2033606  || Result :       3322  || Expected Result :       3322 || Status : PASS
# KERNEL: Time : 320000  || Instruction :          6  || Result :       7200  || Expected Result :       7200 || Status : PASS
# KERNEL: Time : 340000  || Instruction :    2033613  || Result : 4294961296  || Expected Result : 4294961296 || Status : PASS
# KERNEL: Time : 360000  || Instruction :         11  || Result :       7200  || Expected Result :       7200 || Status : PASS
# KERNEL: Time : 380000  || Instruction :      49157  || Result :          0  || Expected Result :          0 || Status : PASS
# KERNEL: Time : 400000  || Instruction :    2033158  || Result :       5338  || Expected Result :       5338 || Status : PASS
# KERNEL: Time : 420000  || Instruction :      49165  || Result : 4294961958  || Expected Result : 4294961958 || Status : PASS
# KERNEL: Time : 440000  || Instruction :    2097102  || Result :       5338  || Expected Result :       5338 || Status : PASS
```

Appendix B :

ALU :

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

    // select the operation depending on the value of opcode
    always@(*)begin
    case (opcode)
        //a+b
        6'd4:
        assign result = a+b ;

        //a xor b
        6'd5 :
        assign result = a ^ b ;

        //-a
        6'd6 :
        assign result = -a ;

        //avg
        6'd7 :
        assign result = (a+b)/2 ;

        //abs(a)
        6'd8 :
        assign result = (a > 0) ? a : -a;

        //not a
         //not a
         6'd9 :
         assign result = ~a ;

         // a and b
         6'd10 :
         assign result = a & b ;

         //a-b
         6'd11:
         assign result = a-b ;

         // a or b
         6'd12 :
         assign result = a | b ;

         //max(a,b)
         6'd13 :
         assign result = (a > b) ? a : b;

         //min(a,b)
         6'd14 :
         assign result = (a < b) ? a : b;

         default : result = result ;
     endcase
     end

     endmodule
```

Register File :

```
module reg_file (clk,valid_opcode, addr1, addr2, addr3, in , out1, out2);
    input clk , 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] ;

    // fill the memory with the values given in the specification of project
    initial begin
      mem[0] = 32'd0;
      mem[1] = 32'd12996;
      mem[2] = 32'd11490;
      mem[3] = 32'd7070;
      mem[4] = 32'd6026;
      mem[5] = 32'd3322;
      mem[6] = 32'd10344;
      mem[7] = 32'd6734;
      mem[8] = 32'd15834;
      mem[9] = 32'd15314;
      mem[10] = 32'd6000;
      mem[11] = 32'd12196;
      mem[12] = 32'd11290;
      mem[13] = 32'd13350;
      mem[14] = 32'd2086;
      mem[15] = 32'd6734;
      mem[16] = 32'd7430;
      mem[17] = 32'd14102;
      mem[18] = 32'd13200;
      mem[19] = 32'd3264;
    mem[20] = 32'd2368;
    mem[21] = 32'd15846;
    mem[22] = 32'd11710;
    mem[23] = 32'd14736;
    mem[24] = 32'd5338;
    mem[25] = 32'd5544;
    mem[26] = 32'd1852;
    mem[27] = 32'd3898;
    mem[28] = 32'd16252;
    mem[29] = 32'd1048;
    mem[30] = 32'd5642;
    mem[31] = 32'd0;
  end

  // the read operation don't need clock to work , and this reduces the number of clocks to finish the instruction
  always@(*)begin
  if(valid_opcode)begin
       out1 <= mem[addr1] ;
       out2 <= mem[addr2] ;
      end
  end

  // at the positive edge of the clock , it only writes the value
  // of register "in" to the memory , all previous actions will happen only
  // if the passed opcode is valid , if it isn't , we will have no change of the memory
  always@(posedge clk)begin
      if(valid_opcode)begin
           mem[addr3] = in ;
       end
```

Mp_t op :

```
module mp_top(clk, instruction , result );
    input clk;
    input [31:0] instruction;
    output reg signed [31:0] result ;
    reg [5:0]opcode;
    reg [4:0]src1 , src2 , dest ;
    reg signed [31:0] out1,out2 ;

    // this register works as enable regiester mentioned in the project description
    reg valid ;

    //extract all information and operands from the instruction
    always@(posedge clk)begin
        opcode=0 ; src1=0 ; src2=0 ; dest=0 ;
        opcode = instruction[5:0];
        src1 = instruction[10:6] ;
        src2 = instruction[15:11];
        dest= instruction[20:16];

        if(opcode <4 || opcode>14)begin
            $display("Invalid opcode !") ;
        valid=0;
        end
        else valid=1 ;
        end

       // pass the parameters to the register file , from this instance we will obtain the
       // values of out1 and out2 immediatly which are the output from the read operation , and
       // the value of result will be written on the register file
       reg_file my_reg(clk, valid, src1, src2, dest, result, out1, out2);


       // here , we pass the opcode to the alu with the outputs of register file
       //,and we are sure that the final result will be ready after this step
       alu my_alu(opcode ,out1 , out2, result );

endmodule
```

Test Bench :

```
module Design_tb ;
    reg clk ;
    reg [31:0]instruction ;
    wire [31:0]result ;
    reg signed [31:0]mem [31:0] ;
    reg [31:0]inst[0:22] = '{
    32'b00000000000011111000110101000000100 , //1 add R31,R10,R3
    32'b00000000000011111000110101000000101 , //2 xor R31,R10,R3
    32'b00000000000011111000000101000000110 , //3 neg R31,R10
    32'b00000000000000000000000001101010000111 , //4 avg R0,R10,R3
    32'b00000000000000000000000011111001000 , //5 abs R0,R31
    32'b00000000000011111000110101000001001 , //6 not R31,R10
    32'b00000000000011111000110101000001010 , //7 and R31,R10,R3
    32'b00000000000011111000110101000001011 , //8 sub R31,R10,R3
    32'b00000000000011111000110101000001100 , //9 or  R31,R10,R3
    32'b00000000000011111000110101000001101 , //10 max R31,R10,R3
    32'b00000000000011111000110101000001110 , //11 min R31,R10,R3
    32'b00000000000011111110010111111001011 , //12 sub R31,R31,R18
    32'b00000000000000101000000010100100000 , //13 abs R5,R5
    32'b00000000000000101011110101100111111 , //14 Invalid opcode (15) "we will check if reg file changed or not"
    32'b00000000000000101000000010100100000 , //15 abs R5,R5
    32'b00000000000011111000000111111000110 , //16 neg R31,R31
    32'b00000000000000000000000000000000110 , //17 neg R0,R0
    32'b00000000000011111000001111111001101 , //18 max R31,R31,R0
    32'b00000000000000000000000000000001011 , //19 sub R0,R0,R0
    32'b00000000000000000011000000000000101 , //20 xor R0,R0,R24
    32'b00000000000011111000001100000000110 , //21 neg R31,R24
    32'b00000000000000000011000000000001101 , //22 max R0,R0,R24
    32'b00000000000011111111111111111001110   //23 min R31,R31,R31
    } ;
```

```
// This is task to calculate the expected result for any instruction , and we use it for verification
task expected_result;
input [31:0]cur_instruction ;
output [31:0]res ;
reg [5:0]opcode ;
reg [4:0]src1 , src2 , dest ;
opcode=0 ; src1=0 ; src2=0 ; dest=0 ;
    opcode = cur_instruction[5:0];
    src1 = cur_instruction[10:6] ;
    src2 = cur_instruction[15:11];
    dest= cur_instruction[20:16];

    case (opcode)

    //a+b
    6'd4:begin
    assign res = mem[src1]+mem[src2] ;
    mem[dest]= res ;
    end

    //a xor b
    6'd5 :begin
    assign res = mem[src1] ^ mem[src2] ;
    mem[dest]= res ;
    end
```

```
        //-a
        6'd6 :begin
        assign res = -mem[src1] ;
        mem[dest]= res ;
        end

        //avg
        6'd7 :begin
        assign res = (mem[src1]+mem[src2])/2 ;
        mem[dest]= res ;
        end

        //abs(a)
        6'd8 :begin
        assign res = (mem[src1] > 0) ? mem[src1] : -mem[src1];
        mem[dest]= res ;
        end

        //not a
        6'd9 : begin
        assign res = ~mem[src1] ;
        mem[dest]= res ;
        end

        // a and b
        6'd10 :  begin
        assign res = mem[src1] & mem[src2] ;
        mem[dest]= res ;
        end

        //a-b
        6'd11:begin
        assign res = mem[src1]-mem[src2] ;
        mem[dest]= res ;
        end

        // a or b
        6'd12 :begin
        assign res = mem[src1] | mem[src2] ;
        mem[dest]= res ;
        end

        //max(a,b)
        6'd13 :  begin
        assign res = (mem[src1] > mem[src2]) ? mem[src1] : mem[src2];
        mem[dest]= res ;
        end

        //min(a,b)
        6'd14 :  begin
        assign res = (mem[src1] < mem[src2]) ? mem[src1] : mem[src2];
        mem[dest]= res ;
        end
      endcase

    endtask
```

```
// integer i is to read the instructions from the array of instructions , and j to iterate over the array of instructions
integer i,j ;
reg [31:0] expected_res ;

// this array will contain all expected values from all instructions
reg [31:0]expected[0:22] ;

// this register will hold every expected result that will be compared to the actual result
reg [31:0] final_expected ;

// This instance will be called every time the instruction changed
mp_top tt(clk,instruction,result) ;

// the clock
always #10ns clk = ~clk;

// a flag to determine if the result is same as expected result
    reg accepted ;

// All the work is here
initial begin
    clk=1 ;
    i=1 ;

  // fill the memory with the values given in the specification of project
   mem[0] = 32'd0;
  mem[1] = 32'd12996;
  mem[2] = 32'd11490;
  mem[3] = 32'd7070;
  mem[4] = 32'd6026;
```

```
    mem[0] = 32'd0;
    mem[1] = 32'd12996;
    mem[2] = 32'd11490;
    mem[3] = 32'd7070;
    mem[4] = 32'd6026;
    mem[5] = 32'd3322;
    mem[6] = 32'd10344;
    mem[7] = 32'd6734;
    mem[8] = 32'd15834;
    mem[9] = 32'd15314;
    mem[10] = 32'd6000;
    mem[11] = 32'd12196;
    mem[12] = 32'd11290;
    mem[13] = 32'd13350;
    mem[14] = 32'd2086;
    mem[15] = 32'd6734;
    mem[16] = 32'd7430;
    mem[17] = 32'd14102;
    mem[18] = 32'd13200;
    mem[19] = 32'd3264;
    mem[20] = 32'd2368;
    mem[21] = 32'd15846;
    mem[22] = 32'd11710;
    mem[23] = 32'd14736;
    mem[24] = 32'd5338;
    mem[25] = 32'd5544;
    mem[26] = 32'd1852;
    mem[27] = 32'd3898;
    mem[28] = 32'd16252;
    mem[29] = 32'd1048;
```

```
    // this loop to calulate the expected values for all instructions and store them in the array "expected"
    for(j=0;j<23;j++)begin
         expected_result(inst[j],expected_res) ;
         expected[j]=expected_res ;
       end

    // start our program with the first instruction
    instruction = inst[0] ;
    final_expected = expected[0] ;

    // loop over 22 instructions in my array of instructions
       repeat(22)
        #20ns begin
             // check if the value of result matches the value of expected result ,
             accepted = (result == final_expected)? 1 : 0 ;

             // print all information that we need to the terminal
      $display("Time : %2d  || Instruction : %d  || Result : %d  || Expected Result : %d || Status : %s" , $time,inst[i],result,final_expected,(accepted==1)?"PASS":"FAIL");

             // move to the next instruction
             instruction = inst[i] ;
             final_expected = expected[i] ;

             // increase the value of i to move to the next instrucion
             i++ ;
       end

endmodule
endmodule
```