Digital Systems - processor

Code:

Processor main file:

```
`timescale 1ns / 1ps

module processor(
    input real_clock, reset, En, write2,
    input [4:0] displayAdd,
    output reg CB,
    output reg [7:0] display, writeData2,  // nothing, acc, ext, inst, & all registers
    output wire clock,
    output reg [2:0] pc_out
);

slowing_the_clock slwclk(.clk(real_clock),.clk_out(clock));

reg [2:0] PC;
reg [7:0] ACC, EXT;
reg [7:0] inst;

wire [7:0] tempinst;
wire tempCB,single;
wire [3:0] regAdd,alu_op;
wire [7:0] tempReg, tempACC, tempEXT, tempReg2;

reg [3:0] readAdd, writeAdd, readAdd2, writeAdd2;
reg [7:0] readData, writeData, readData2;
reg  tempPC, write;
reg stop;
initial begin
    PC = 0;
    CB = 0;
    ACC = 0;
    EXT = 0;
    inst = 0;
    readAdd = 0;
    readData = 0;
    writeAdd = 0;
    writeAdd2 = 0;
    readAdd2 = 0;
```

```
    readData2 = 0;
    writeData = 0;
    display = 0;
    stop = 0;
end

and(Enclock,En,clock);

program pgm(.clock(Enclock),
            .pc(PC),
            .inst(tempinst));

InstructionDecoder pm(.inst(inst),
            .operandAdd(regAdd),
            .single(single),
            .alu_op(alu_op));

RegisterFile r(.clock(Enclock),
            .write(write),
            .reset(reset),
            .readAdd(readAdd),
            .writeAdd(writeAdd),
            .writeData(writeData),
            .readData(tempReg));

RegisterFile r2(.clock(clock),
            .write(write2),
            .reset(reset),
            .readAdd(readAdd2),
            .writeAdd(writeAdd2),
            .writeData(writeData2),
            .readData(tempReg2));

ALU alu(.clock(Enclock),
            .operand1(ACC),
            .operand2(readData),
            .single(single),
            .alu_op(alu_op),
            .result(tempACC),
            .EXT(tempEXT),
            .cb(tempCB));

always @(posedge real_clock) begin
    case(displayAdd)
        5'b00001: display = ACC;
        5'b00010: display = EXT;
        5'b00011: display = inst;
        default: display = 0;
    endcase
```

```
    if(displayAdd[4]) begin
        readAdd2 = displayAdd[3:0];
        display = tempReg2;
        if(write) begin
            writeAdd2 = displayAdd[3:0];
        end
    end
end

always @(posedge Enclock) begin
    if(reset) begin
        PC = 0;
        CB = 0;
        ACC = 0;
        EXT = 0;
        readAdd = 0;
        readData = 0;
        writeAdd = 0;
        writeData = 0;
        stop = 0;
    end
    else if (!reset) begin
        if(alu_op == 8 & CB == 1) begin
            tempPC = PC + 1;
            PC = regAdd;
            write = 1;
            writeData = tempPC;
        end
        else if (alu_op == 9) begin
            ACC = readData;
        end
        else if (alu_op == 10) begin
            write = 1;
            writeData = ACC;
        end
        else if(alu_op == 11) begin
            PC = tempPC;
        end
        else if(alu_op == 15) begin
            stop = 1;
        end
        if(!stop) begin
            pc_out = PC;
            PC = PC + 1;
            ACC = tempACC;
            EXT = tempEXT;
            inst = tempinst;
            readData = tempReg;
            readAdd = regAdd;
```

```
            writeAdd = regAdd;
            CB = tempCB;
        end
    end
end
endmodule

module slowing_the_clock(
input clk,
output reg clk_out
);
reg [31:0] counter = 0;
always @(posedge clk) begin
    counter <= counter + 1;
    clk_out <= counter[27];
end
endmodule
```

ALU code:-

```
`timescale 1ns / 1ps

module ALU(
input clock,
input [7:0] operand1,
input [7:0] operand2,
input [3:0] alu_op,
input single,
output reg [7:0] result,
output reg [7:0] EXT,
output reg cb
);

reg [15:0] temp;
always @(posedge clock) begin
    EXT = 0;
    if (single == 1) begin
        if (alu_op==1) begin
            result = operand1 <<< 1;
        end
        else if (alu_op==2) begin
            result = operand1 >>> 1;
        end
        else if (alu_op==3) begin
```

```
        result[7] = operand1[0];
        result[6:0] = operand1[7:1];
    end
    else if (alu_op==4) begin
        result[0] = operand1[7];
        result[7:1] = operand1[6:0];
    end
    else if (alu_op == 5) begin
        result[7] = operand1[7];
        result[6:0] = operand1[7:1];
    end
    else if (alu_op == 6) begin
        if (operand1 == 255) begin
            result = 0;
            cb = 1;
        end
        else begin
            result = operand1 + 1;
        end
    end
    else if (alu_op == 7) begin
        if (operand1 == 0) begin
            result = 255;
            cb = 1;
        end
        else begin
            result = operand1 - 1;
        end
    end
end
else begin
    if (alu_op==1) begin
        temp = operand1 + operand2;
        cb = temp[8];
        result = temp[7:0];
    end
    else if (alu_op==2) begin
        temp = operand1 - operand2;
        cb = temp[8];
        result = temp[7:0];
    end
    else if (alu_op==3) begin
        temp = operand1 * operand2;
        EXT = temp[15:8];
        result = temp[7:0];
    end
    else if (alu_op==4) begin
        EXT = operand1 % operand2;
        result = operand1 / operand2;
```

```
        end
        else if (alu_op == 5) begin
            result = operand1 & operand2;
        end
        else if (alu_op==6) begin
            result = operand1 ^ operand2;
        end
        else if (alu_op==7) begin
            if (operand1 >= operand2)
                cb = 0;
            else
                cb = 1;
        end
        else begin
            result = 0;
            EXT = 0;
        end
    end
end

endmodule
```

Program file that contains the code:-

```
`timescale 1ns / 1ps

module program(
input clock,
input [2:0] pc,
output reg [7:0] inst
);

reg [7:0] programReg [7:0];

initial begin
    programReg[0] = 8'b10010000;
    programReg[1] = 8'b01100000;
    programReg[2] = 8'b00010100;
    programReg[3] = 8'b00010101;
    programReg[4] = 8'b10100110;
    programReg[5] = 8'b11111111;
    programReg[6] = 8'b00011111;
```

```
    programReg[7] = 8'b00010000;
end
initial begin
    inst = 0;
end
always @(posedge clock)
    inst = programReg[pc];

endmodule
```

Instruction decoder:-

```
`timescale 1ns / 1ps

module InstructionDecoder(
input [7:0] inst,
output reg [3:0] operandAdd,
output reg [3:0] alu_op,
output reg single
);

always @* begin
    alu_op = inst[7:4];
    if (inst[7:4] == 4'b0000) begin
        operandAdd = 8'hff;
        single = 1;
    end
    else begin
        operandAdd = inst[3:0];
        single = 0;
    end
end

endmodule
```

Register file code:-

```
`timescale 1ns / 1ps

module RegisterFile (
    input clock, write, reset,
    input [3:0] readAdd, writeAdd,
    input [7:0] writeData,
```

```
    output reg [7:0] readData
);

reg [7:0] registers [15:0];
initial begin           // Initiallizing the registers
    registers[0] = 10;
    registers[1] = 20;
    registers[2] = 30;
    registers[3] = 40;
    registers[4] = 50;
    registers[5] = 60;
    registers[6] = 70;
    registers[7] = 80;
    registers[8] = 90;
    registers[9] = 100;
    registers[10] = 110;
    registers[11] = 120;
    registers[12] = 130;
    registers[13] = 140;
    registers[14] = 150;
    registers[15] = 160;
end
integer j;
always @(posedge clock) begin
    if (reset) begin
        for (j = 0; j < 16; j = j + 1) begin
            registers[j] <= j; // On reseting we are initializing registers with numbers 0 to 15.
        end
    end else if (write) begin
        registers[writeAdd] <= writeData;
    end
    readData <= registers[readAdd];
end

endmodule
```

Test bench:-

Instructor decoder testbench:-

```
`timescale 1ns / 1ps

module Instdecodertb();

reg [7:0] inst;
```

```
wire [3:0] operand2;
wire [3:0] alu_op;
wire single;

InstructionDecoder obj(.single(single),.inst(inst),.operand2(alu_op),.opcode(alu_op));

initial begin
   inst = 8'b0;
   #10 inst = 8'b00101010;
   #10 inst = 8'b00010001;
   #10 $finish;
end

endmodule
```

ALU testbench:-

```
`timescale 1ns / 1ps

module ALU_tb;

   // Define parameters
   parameter CLK_PERIOD = 10; // Clock period in ns

   // Declare signals
   reg [7:0] operand1_tb;
   reg [7:0] operand2_tb;
   reg [2:0] alu_op_tb;
   reg single_tb;
   wire [7:0] result_tb;
   wire [7:0] EXT_tb;
   wire cb_tb;

   // Instantiate the ALU module
   ALU dut (
      .operand1(operand1_tb),
      .operand2(operand2_tb),
      .alu_op(alu_op_tb),
      .single(single_tb),
      .result(result_tb),
      .EXT(EXT_tb),
      .cb(cb_tb)
   );

   // Clock generation
   reg clk = 0;
```

```
always #((CLK_PERIOD / 2)) clk = ~clk;

// Test case
initial begin
    // Test single operations
    operand1_tb = 8'hAA;
    operand2_tb = 8'h55;
    alu_op_tb = 1; // Left shift
    single_tb = 1;
    #10;

    operand1_tb = 8'hAA;
    operand2_tb = 8'h55;
    alu_op_tb = 2; // Right shift
    single_tb = 1;
    #10;

    operand1_tb = 8'hAA;
    operand2_tb = 8'h55;
    alu_op_tb = 3; // Rotate left
    single_tb = 1;
    #10;

    operand1_tb = 8'hAA;
    operand2_tb = 8'h55;
    alu_op_tb = 4; // Rotate right
    single_tb = 1;
    #10;

    operand1_tb = 8'hAA;
    operand2_tb = 8'h55;
    alu_op_tb = 5; // Copy high bit
    single_tb = 1;
    #10;

    operand1_tb = 8'hAA;
    operand2_tb = 8'h55;
    alu_op_tb = 6; // Increment or clear
    single_tb = 1;
    #10;

    operand1_tb = 8'hAA;
    operand2_tb = 8'h55;
    alu_op_tb = 7; // Decrement or set
    single_tb = 1;
    #10;

    // Test multiple operations
    operand1_tb = 8'hAA;
```

```
    operand2_tb = 8'h55;
    alu_op_tb = 1; // Add
    single_tb = 0;
    #10;

    operand1_tb = 8'hAA;
    operand2_tb = 8'h55;
    alu_op_tb = 2; // Subtract
    single_tb = 0;
    #10;

    operand1_tb = 8'hAA;
    operand2_tb = 8'h55;
    alu_op_tb = 3; // Multiply
    single_tb = 0;
    #10;

    operand1_tb = 8'hAA;
    operand2_tb = 8'h55;
    alu_op_tb = 4; // Modulus and division
    single_tb = 0;
    #10;

    operand1_tb = 8'hAA;
    operand2_tb = 8'h55;
    alu_op_tb = 5; // Bitwise AND
    single_tb = 0;
    #10;

    operand1_tb = 8'hAA;
    operand2_tb = 8'h55;
    alu_op_tb = 6; // Bitwise XOR
    single_tb = 0;
    #10;

    operand1_tb = 8'hAA;
    operand2_tb = 8'h55;
    alu_op_tb = 7; // Compare
    single_tb = 0;
    #10;

    // End simulation
    $stop;
end

// Assign inputs
always @(posedge clk) begin
    // Assign inputs here based on test cases
end
```

```
    // Monitor outputs
    always @(posedge clk) begin
        // Monitor outputs here
    end

endmodule
```

Program test bench:-

```
`timescale 1ns / 1ps

module program_tb();

reg clock;
reg [2:0] pc;
wire [7:0] inst;

program pro(.clock(clock),.pc(pc),.inst(inst));

initial begin
    clock = 0;
    pc = 0;
    forever #5 clock = ~clock;
end
always @*
    if(inst == 8'b11111111)
        $finish;

always @(posedge clock) begin
    #5 pc = pc + 1;
    if(pc == 7)
        #5 $finish;
end
endmodule
```

Processor testbench:-

```
`timescale 1s/1ps

module processor_tb;

// Inputs
reg clock;
```

```
reg reset;
reg En;
reg [1:0] displayAdd;

// Outputs
wire CB;
wire [7:0] display;
wire slow_clock;
wire [2:0] pc;
// Instantiate the processor module
processor dut(
    .real_clock(clock),
    .reset(reset),
    .En(En),
    .displayAdd(displayAdd),
    .CB(CB),
    .display(display),
    .clock(slow_clock),  // Use real clock for simulation as you won't be able to see the signals changing
due to very low frequency of slow_clock
    .pc(pc)
);

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

// Initial values
initial begin
    clock = 0;
    reset = 1;
    En = 0;
    displayAdd = 0;

    // You can write your own instructions for checking the simulation for different inputs

    // Wait for a few clock cycles before releasing reset
    #40;
    reset = 0;

    // Enable processor
    En = 1;

    // Test scenario 1: Write data to register file and read from it
//    write = 1;
    displayAdd = 1;
    #20;
//    write = 1;
//    displayAdd = 2;
    #20;
```

```
    // Test scenario 2: Perform arithmetic operation
//    write = 1;
//    displayAdd = 3;
    #20;
//    write = 0;
    displayAdd = 1;
    #20;

    // Test scenario 3: Test branch instruction
//    write = 1;
    displayAdd = 1;
    #20;
//    displayAdd = 2;
    #20;

    // Test scenario 4: Test halt instruction
//    write = 1;
//    displayAdd = 2;
    #20;
//    displayAdd = 3;
    #20;

    // Finish simulation after a while
    #100;
    $finish;
end

endmodule
```

Main test bench:-

```
`timescale 1ns / 1ps

module testbench();

reg [7:0] inst;
wire [7:0] operand2;
wire [7:0] opcode;

InstructionDecoder obj(.inst(inst),.operand2(operand2),.opcode(opcode));

initial begin
    inst = 8'b0;
    #10 inst = 8'b00101010;
    #10 inst = 8'b00010001;
```

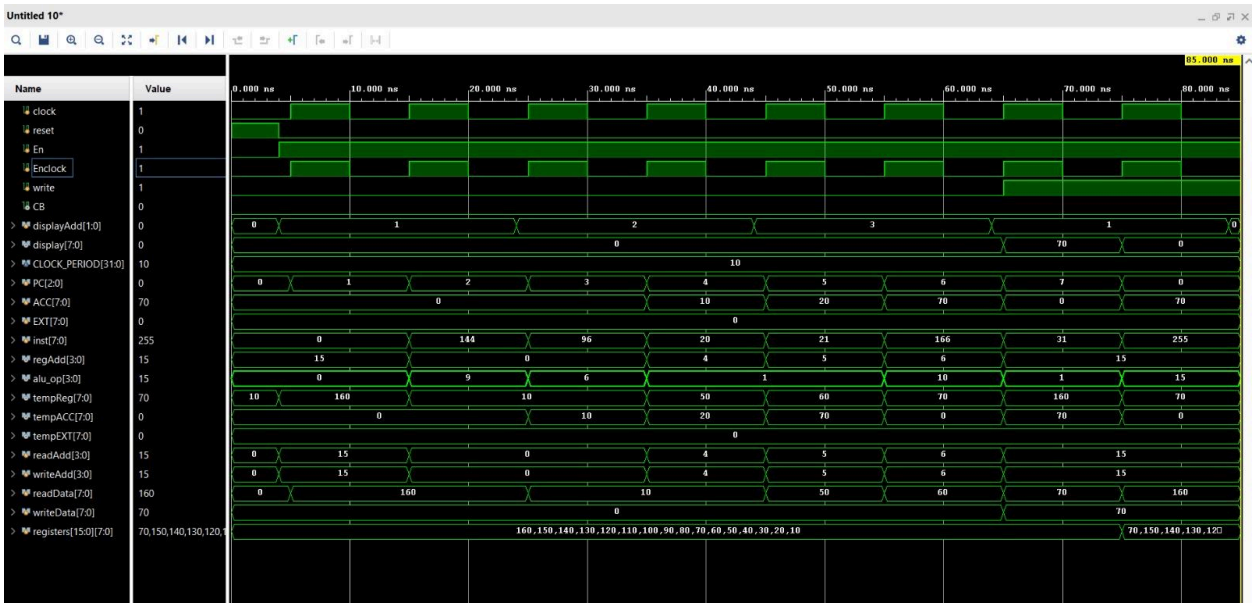
```
    #10 $finish;
end

endmodule
```

Register file test bench:-

```
`timescale 1ns / 1ps

module regfiletb();

reg clock;
reg [3:0] readAdd;
reg [3:0] writeAdd;
reg write, reset;
reg [7:0] writeData;
wire [7:0] acc;

RegisterFile obj (
    .clock(clock),
    .readAdd(readAdd),
    .writeAdd(writeAdd),
    .write(write),
    .reset(reset),
    .writeData(writeData),
    .readData(acc));

initial begin
    clock = 0;
    forever #5 clock = ~clock;
end

initial begin
    reset = 1;
    #10;
    reset = 0;
    write = 1;
    readAdd = 4'b1000;
    writeAdd = 4'b1010;
    writeData = 8'b10101010;
    #10;
    write = 0;
    writeAdd = 4'b1000;
    writeData = 8'b10100000;
    #10;
    write = 1;
    writeAdd = 4'b1110;
```

```
    writeData = 8'b10111110;
    #10;
    write = 0;
    #10 $finish;
end

endmodule
```

Outputs:-



Schematics:-