




# FPGA Processor

## Individual module:

### Instruction decoder:

```
module instruction_decoder(  
    input wire [7:0] instruction_code,  
    output wire [3:0] opcode,  
    output wire [3:0] reg_add);  
    assign opcode = instruction_code[7:4];  
    assign reg_add = instruction_code[3:0];  
endmodule  
  
module tb_instruction_decoder;  
    reg [7:0] instruction;  
    wire [3:0] opcode, reg_add;  
    instruction_decoder uut (instruction,opcode,reg_add);  
    initial begin  
        instruction = 8'b11010010; #10;  
        instruction = 8'b00011110; #10;  
        instruction = 8'b01001101; #10;  
        $finish;  
    end  
endmodule
```

Name	Value	0.000 ns	5.000 ns	10.000 ns	15.000 ns	20.000 ns	25.000 ns
>  instruction[7:0]	01001101	11010010		00011110		01001101	
>  opcode[3:0]	0100	1101		0001		0100	
>  reg_add[3:0]	1101	0010		1110		1101	

### Register file:

```
module register_file(  

```

```

input wire clk,
input wire we,
input wire [3:0] addr,
input wire [7:0] data_in,
output wire [7:0] data_out);
reg [7:0] registers [15:0];
integer i;
initial begin
    for (i = 0; i < 16; i = i + 1)
        registers[i] = 0;    end
always @(posedge clk) begin
    if (we)
        registers[addr] <= data_in;
end
assign data_out = registers[addr];
endmodule

```

```

module tb_register_file;
    reg clk, we;
    reg [3:0] addr;
    reg [7:0] data_in;
    wire [7:0] data_out;
    register_file uut(clk,we,addr,data_in,data_out);
    initial begin
        clk=0;
        forever #5 clk = ~clk;
    end
    initial begin
        clk = 0;
        we = 0;
        addr = 0;
    end
endmodule

```

```

    data_in = 0;

    #10;

    we = 1;//write

    addr = 3;

    data_in = 45;

    #10;

    we = 0;// read

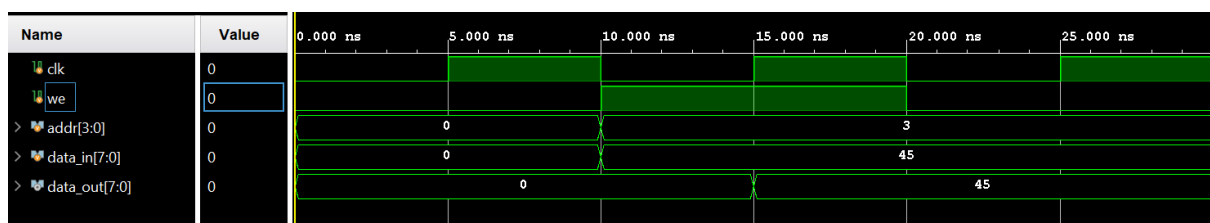
    #10;

    $finish;

end

endmodule

```



### Alu:

```

module alu(
    input wire [3:0] opcode,
    input wire [7:0] acc_in,
    input wire [7:0] operand,
    output reg [7:0] acc_out,
    output reg [15:0] extended_out,
    output reg c_b);
    always @(*) begin
        acc_out = acc_in;
        extended_out = 16'd0;
        c_b = 1'b0;

        case (opcode)

```

```

4'b0001: {c_b, acc_out} = acc_in + operand;    // ADD (9-bit result)
4'b0010: {c_b, acc_out} = acc_in - operand;    // SUB (9-bit result)
4'b0011: extended_out = acc_in * operand;      // MUL (16-bit result)
4'b0100: acc_out = acc_in | operand;           // OR (added)
4'b0101: acc_out = acc_in & operand;           // AND
4'b0110: acc_out = acc_in ^ operand;           // XOR
4'b0111: c_b = (acc_in < operand);             // CMP
4'b1000: acc_out = acc_in << 1;                // SHL (added)
4'b1001: {acc_out, c_b} = {1'b0, acc_in} >> 1; // SHR (added)

default: begin
    acc_out = acc_in;
    extended_out = 16'd0;
    c_b = 1'b0;
end
endcase
end
endmodule

```

```

module tb_alu;
    reg [3:0] opcode;
    reg [7:0] acc_in, operand;
    wire [7:0] acc_out;
    wire [15:0] extended_out;
    wire c_b;

    alu uut (opcode, acc_in, operand, acc_out, extended_out, c_b);

    initial begin
        acc_in = 8'd10; operand = 8'd3;

        opcode = 4'b0001; #10; // ADD (10 + 3 = 13)
    end
endmodule

```

```

opcode = 4'b0010; #10; // SUB (10 - 3 = 7)
opcode = 4'b0011; #10; // MUL (10 * 3 = 30)
opcode = 4'b0100; #10; // OR (10 | 3)
opcode = 4'b0101; #10; // AND (10 & 3)
opcode = 4'b0110; #10; // XOR (10 ^ 3)
opcode = 4'b0111; #10; // CMP (10 < 3?)
opcode = 4'b1000; #10; // SHL (10 << 1)
opcode = 4'b1001; #10; // SHR (10 >> 1)

```

```

acc_in = 8'd255; operand = 8'd1;

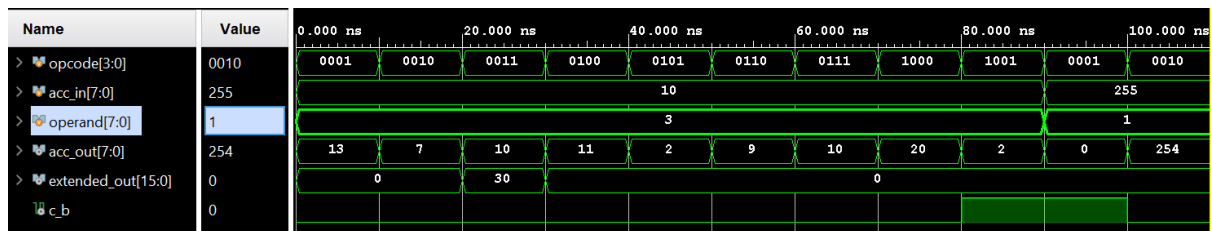
opcode = 4'b0001; #10; // ADD with carry
opcode = 4'b0010; #10; // SUB with borrow

```

```
$finish;
```

```
end
```

```
endmodule
```



## Accumulator:

```

module accumulator_module(
    input wire clk,
    input wire rst,
    input wire [7:0] acc_in,
    input wire load,
    output reg [7:0] acc_out);
    always @(posedge clk or posedge rst) begin
        if (rst)

```

```

        acc_out <= 8'd0;
    else if (load)
        acc_out <= acc_in;
    end
endmodule

module tb_accumulator_module;

    reg clk, rst, load;
    reg [7:0] acc_in;
    wire [7:0] acc_out;

    accumulator_module uut(clk, rst, acc_in, load, acc_out);

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

    initial begin
        rst = 1;
        load = 0;
        acc_in = 0;

        // reset
        #10;
        rst = 0;

        // load
        acc_in = 8'd42;
        load = 1;
        #10;
    end
endmodule

```

```

// hold

load = 0;

acc_in = 8'd99;

#10;

// new load

load = 1;

#10;

// reset while loaded

load = 0;

rst = 1;

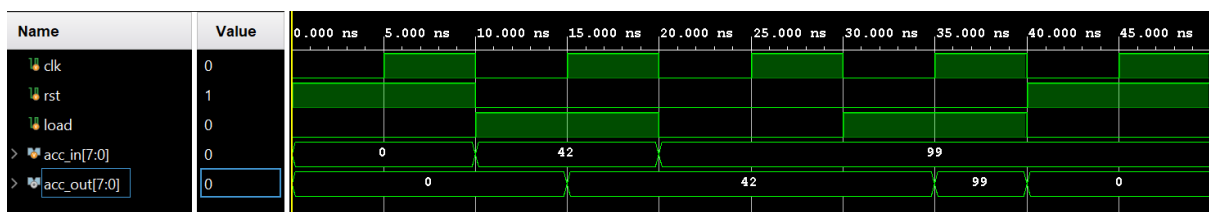
#10;

$finish;

end

endmodule

```



### Control Logic:

```

module control_logic(
    input wire clk,
    input wire rst,
    input wire branch,
    input wire [3:0] branch_addr,
    output reg [3:0] pc);

```

```

always @(posedge clk or posedge rst) begin
    if (rst)
        pc <= 4'd0;
    else if (branch)
        pc <= branch_addr;
    else
        pc <= pc + 1;
    end
endmodule

module tb_control_logic;
    reg clk, rst, branch;
    reg [3:0] branch_addr;
    wire [3:0] pc;
    control_logic uut(clk,rst,branch,branch_addr,pc);

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

    initial begin
        rst = 1;
        branch = 0;
        branch_addr = 0;

        // reset
        #10;
        rst = 0;

        #20; // normal increment-3 cycles
    end
endmodule

```



```

    branch = 1;// branch
    branch_addr = 4'd5;

    #10;

    branch = 0;

    #190;

    rst = 1;

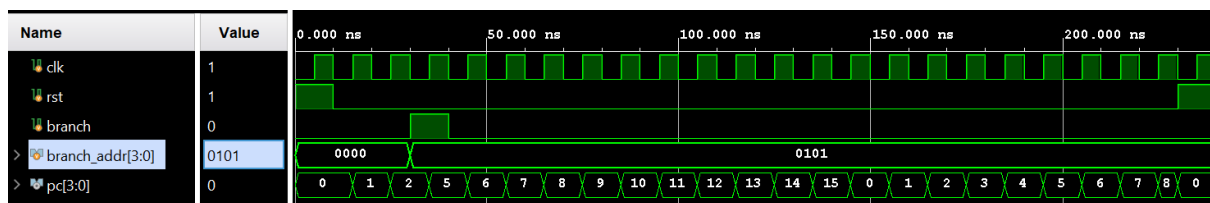
    #10;

    $finish;

end

endmodule

```



## Processor

### Code:

```

`timescale 1ns / 1ps

module processor (
    input clk,
    input en,
    output wire C_B,
    output wire [7:0] accum_wire);
    reg [7:0] accumulator, extended_reg;
    reg [3:0] program_counter;
    reg c_b;

```

```

reg [7:0] registers [0:15];
wire [7:0] instruction_code;
wire [3:0] opcode, reg_add;
assign instruction_code = registers[program_counter];
assign opcode = instruction_code[7:4];
assign reg_add = instruction_code[3:0];
assign accum_wire = accumulator;
assign C_B = c_b;
always @(posedge clk or negedge en) begin
    if (!en) begin
        // defining value
        registers[0] = 8'b00000000;
        registers[1] = 8'b00011111;
        registers[2] = 8'b00011110;
        registers[3] = 8'b01001101;
        registers[4] = 8'b10101100;
        registers[5] = 8'b00111110;
        registers[6] = 8'b00000011;
        registers[7] = 8'b11111111;
        registers[8] = 8'b00000000;
        registers[9] = 8'b00000000;
        registers[10] = 8'b00000000;
        registers[11] = 8'b00000000;
        registers[12] = 8'b00000000;
        registers[13] = 8'd12;
        registers[14] = 8'd24;
        registers[15] = 8'd21;
        // Reset control signals
        program_counter = 4'b0000;
        extended_reg = 8'b00000000;
        accumulator = 8'b00000000;
    end
end

```

```

    c_b = 1'b0;
end else begin
    case (opcode)
        4'b0001: begin // ADD Ri
            {c_b, accumulator} = accumulator + registers[reg_add];
            program_counter = program_counter + 1;
        end
        4'b0010: begin // SUB Ri
            {c_b, accumulator} = accumulator - registers[reg_add];
            program_counter = program_counter + 1;
        end
        4'b0011: begin // MUL Ri
            {extended_reg, accumulator} = accumulator * registers[reg_add];
            program_counter = program_counter + 1;
        end
        4'b0100: begin //No instruction on this opcode
            accumulator = accumulator;
            extended_reg = extended_reg;
            program_counter = program_counter + 1;
        end
        4'b0101: begin // AND Ri
            accumulator = accumulator & registers[reg_add];
            program_counter = program_counter + 1;
        end

        4'b0110: begin // XRA Ri
            accumulator = accumulator ^ registers[reg_add];
            program_counter = program_counter + 1;
        end
        4'b0111: begin // CMP Ri
            c_b = (accumulator < registers[reg_add]);

```

```

    program_counter = program_counter + 1;
end
4'b1001: begin // MOV ACC, Ri
    accumulator = registers[reg_add];
    program_counter = program_counter + 1;
end
4'b1010: begin // MOV Ri, ACC
    registers[reg_add] = accumulator;
    program_counter = program_counter + 1;
end
4'b1000: begin // BR <4-bit address> if c_b
    if (c_b)
        program_counter = reg_add;
    else
        program_counter = program_counter + 1;
    end
end
4'b1011: begin // RET <4-bit address>
    program_counter = reg_add + 1;
end
4'b0000: begin // Misc operations
    case (reg_add)
        4'b0000: accumulator = accumulator; // NOP
        4'b0001: accumulator = accumulator << 1; // LSL ACC
        4'b0010: accumulator = accumulator >> 1; // LSR ACC
        4'b0011: accumulator = {accumulator[0], accumulator[7:1]}; // CIR ACC
        4'b0100: accumulator = {accumulator[6:0], accumulator[7]}; // CIL ACC
        4'b0101: accumulator = accumulator >>> 1; // ASR ACC
        4'b0110: {c_b, accumulator} = accumulator + 1; // INC ACC
        4'b0111: {c_b, accumulator} = accumulator - 1; // DEC ACC
        default: accumulator = accumulator;
    endcase
end

```

```

        program_counter = program_counter + 1;
    end
    4'b1111: begin // HLT
        if (reg_add != 4'b1111)
            accumulator = accumulator;
        // Else, hold PC
    end
    default: begin
        accumulator = accumulator;
    end
endcase
end
end
endmodule

```

### **Testbench :**

```

`timescale 1ns / 1ps
module processor_tb;
    reg clk;
    reg en;
    wire cb;
    wire [7:0] accum;
    processor sm_processor (clk,en,cb,accum);
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
    initial begin
        en = 0;
        #10;
    end
endmodule

```

```

en = 1;

#90;

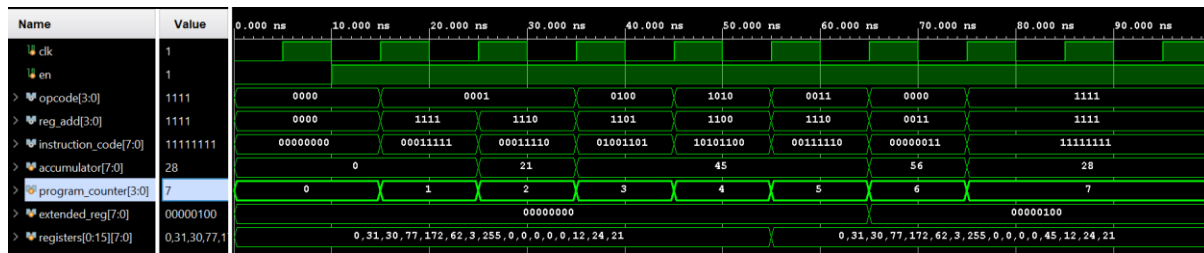
$finish;

end

endmodule

```

### Simulation:



### Block diagram:

