DIGITAL SYSTEM DESIGN APPLICATIONS

(CRN: 11275)

THE REPORT OF THE FINAL PROJECT

Faculty of Electrical and Electronics Engineering

Electronics and Communication Engineering

Yusuf Tekin – 040200043

Bora Kıran – 040210069

Preface

In this project, designing an algorithmic solution to the **modulus function** via using Arithmetic Logic Unit (ALU) is aimed. The top module of the system must have three main parts: ALU, Register Block and Control Unit.

Having the purpose of yielding the remainder when the operand A is divided by the operand B, modulus is a common operator in many languages *(C = A%B in Verilog).* In this project, however, this operation will be done with a single ALU and state machines.

The Algorithmic Logic Unit (ALU) of this project has an AND, a XOR, an ADD, a Circular Left shifter and a Zero Comparator built-in for 8-bit of data each. The modulus operation must be implemented via using these functions. The details of this implementation will be explained in the related section.

Since this project is prepared synchronously and parallelly within the group members (Group Name: 500T), it is not clear which part has been done by which member. So, both members can be accepted as equals. However, if it's necessary, it can be accepted that Bora Kıran has worked more on the control unit and Yusuf Tekin has worked more on the report.

Yusuf Tekin / Bora Kıran
040200043 / 040210069

1. Algorithmic Logic Unit

Since the implementation of the division function in FPGAs is not cost-efficient, the modulus is going to be designed as:

1. $B \rightarrow -B \ \ by\, XOR(B, 0xFF) + 1$
2. $C = ADD(A, -B)$
3. $circularShift(C)$
4. $If(c[0] == 1) \rightarrow output = C + B$
   $else \rightarrow A = C \ and \ repeat \ the \ process$

The ALU design of this project doesn't require all the units however, since the document stated, all the units are placed.

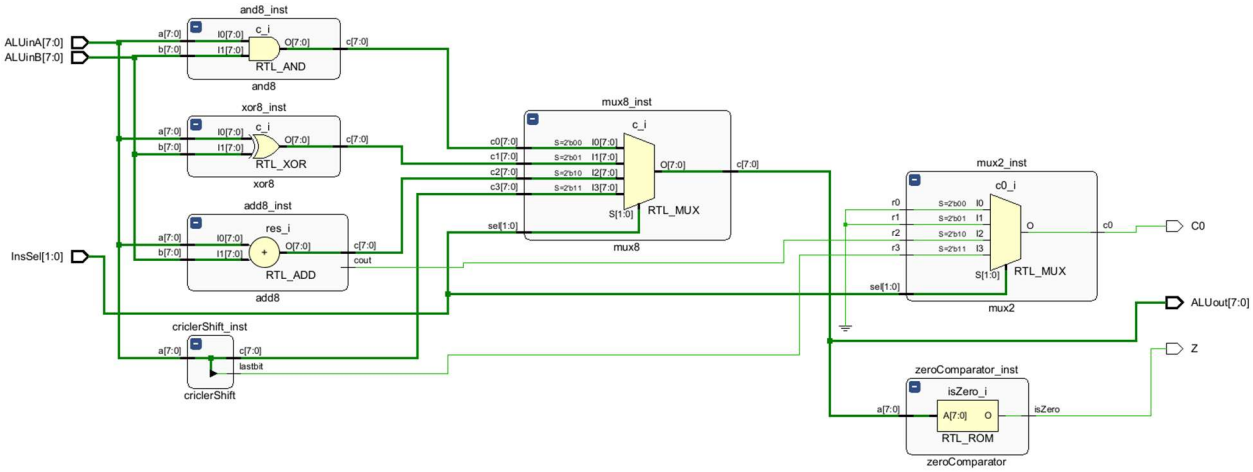

Figure 1 - RTL Schematic of ALU

```
`timescale 1ns / 1ps

module alu(
input [7:0] ALUinA,
input [7:0] ALUinB,
input [1:0] InsSel,
output [7:0]ALUout,
output C0,Z
);
wire[7:0] r0;
wire[7:0] r1;
wire[7:0] r2;
wire[7:0] r3;
wire[7:0] o;
wire output_c;
assign ALUout = o ;

and8 and8_inst (
    .a(ALUinA),
    .b(ALUinB),
    .c(r0)
);

xor8 xor8_inst (
    .a(ALUinA),
    .b(ALUinB),
    .c(r1)
);

add8 add8_inst (
    .a(ALUinA),
    .b(ALUinB),
    .c(r2),
    .cout(output_c)
);

criclerShift criclerShift_inst (
    .a(ALUinA),
    .c(r3),
    .lastbit(last_bit)
);

mux8 mux8_inst (
    .c0(r0),
    .c1(r1),
    .c2(r2),
    .c3(r3),
    .sel(InsSel),
    .c(o)
);

mux2 mux2_inst (
    .r0(1'b0),
    .r1(1'b0),
    .r2(output_c),
    .r3(last_bit),
    .sel(InsSel),
    .c0(C0)
);

zeroComparator zeroComparator_inst (
    .a(o),
    .isZero(Z)
);
endmodule
```

Figure 2 - ALU Verilog Code (Part 1)

```
module and8(

    input [7:0] a,
    input [7:0] b,
    output [7:0] c
 );
assign c = a & b;
 endmodule

 module xor8 (
    input [7:0] a,
    input [7:0] b,
    output [7:0] c
 );
assign c = a ^ b;
 endmodule

 module add8(
    input [7:0] a,
    input [7:0] b,
    output [7:0] c,
    output cout
 );
 wire [8:0] res;
 assign res = a + b;
 assign c_out = res[8];
 assign c = res[7:0];
 endmodule

 module criclerShift(
    input [7:0] a,
    output [7:0] c,
    output lastbit
 );
 assign c = {a[6:0],a[7]};
 assign lastbit = a[7];
 endmodule

 module zeroComparator(
    input[7:0] a,
    output isZero
 );
assign isZero = (a==0) ? 1 : 0;
endmodule

 module mux8(
input [7:0] c0,
input [7:0] c1,
input [7:0] c2,
input [7:0] c3,
input [1:0] sel,
output reg [7:0] c
 );
 always @(*) begin
    case (sel)
        2'b00: c = c0;
        2'b01: c = c1;
        2'b10: c = c2;
        2'b11: c = c3;
        default: c = 0;
    endcase
 end
 endmodule

module mux2(
input r0,r1,r2,r3,
input [1:0] sel,
output reg c0
);
always @(*) begin
    case (sel)
        2'b00: c0 = r0;
        2'b01: c0 = r1;
        2'b10: c0 = r2;
        2'b11: c0 = r3;
        default: c0 = 0;
    endcase
 end
endmodule
```

Figure 3 - ALU Verilog Code (Part 2)

2. Register Block

The register block consists of an in-MUX, an out-MUX, 16 registers and a decoder. It is the memory of the system, and it has a purpose of containing both A and B inputs, ALU output and control unit constants (all the registered inputs are 8-bit). InMux has a 3-bit selector to choose which input (or register repeat) is going to be chosen for memory write operation. Decoder decides the write operation and which register is going to be used for memory. And the OutMux picks the required registers for repeating process. The first 3 inputs of the 16-input of the OutMux are the outputs of the register block as "**Out**", "**ALUinA**", "**ALUinB**" in the given order.

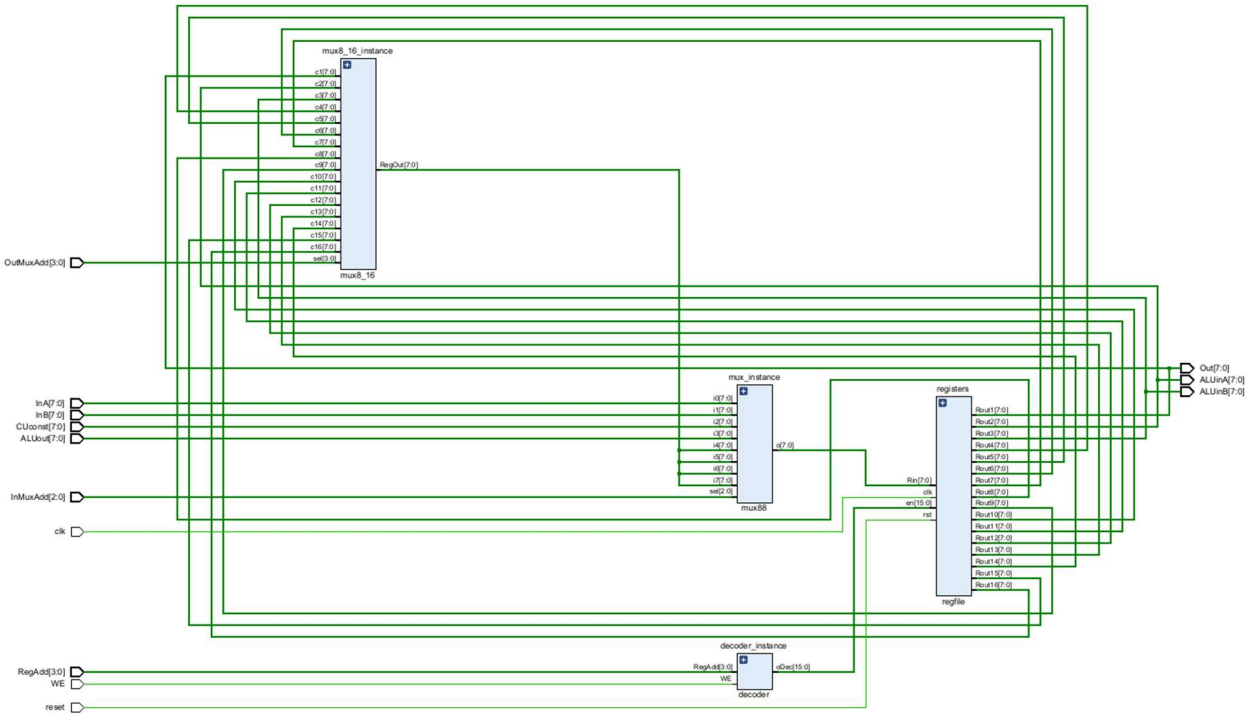

Figure 4 - RTL Schematic of The Register Block

Yusuf Tekin / Bora Kıran
040200043 / 040210069

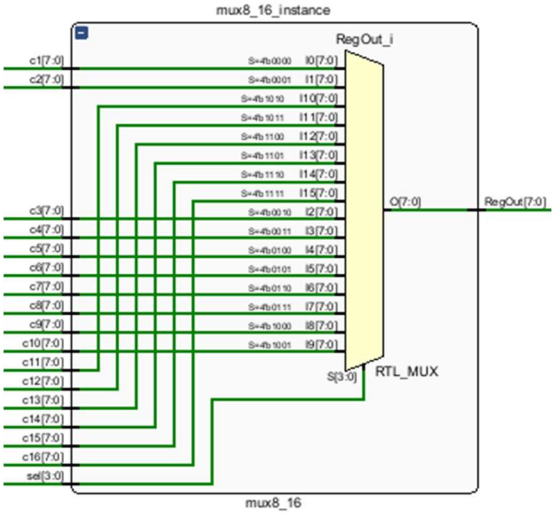

Figure 5 - RTL Schematic of OutMux

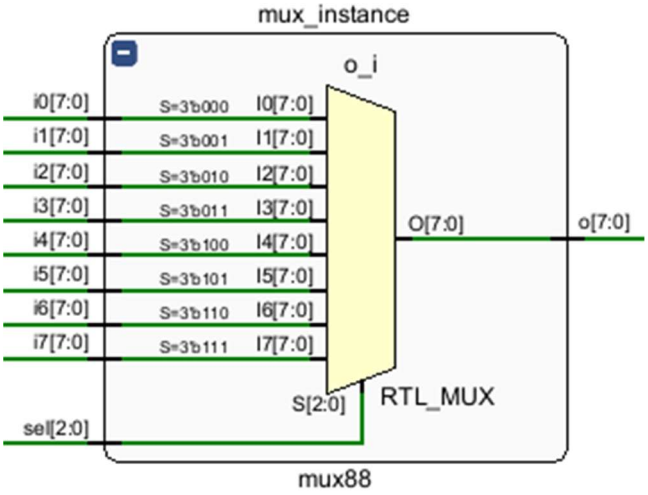

Figure 6 - RTL Schematic of InMux

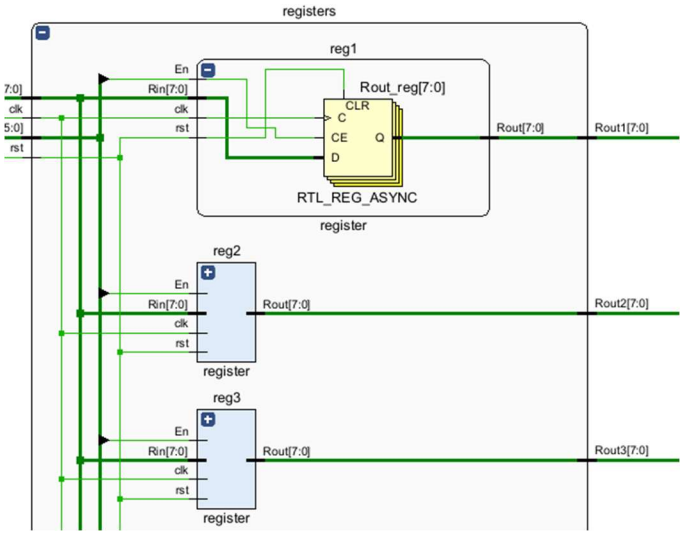

Figure 7 - RTL Schematic of The Registers

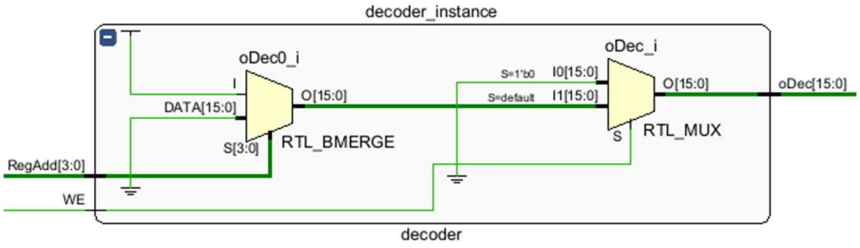

Figure 8 - RTL Schematic of The Decoder

```
`timescale 1ns / 1ps


module rb(
    input [7:0] InA,
    input [7:0] InB,
    input [7:0] CUconst,
    input [2:0] InMuxAdd,
    input WE,reset,clk,
    input [3:0] RegAdd,
    input [3:0] OutMuxAdd,
    input [7:0] ALUout,
    output [7:0] Out,
    output [7:0] ALUinA,
    output [7:0] ALUinB
);
assign Out = R1;
assign ALUinA = R2;
assign ALUinB = R3;
wire [7:0] ALUout;
wire [7:0] R1;
wire [7:0] R2;
wire [7:0] R3;
wire [7:0] R4;
wire [7:0] R5;
wire [7:0] R6;
wire [7:0] R7;
wire [7:0] R8;
wire [7:0] R9;
wire [7:0] R10;
wire [7:0] R11;
wire [7:0] R12;
wire [7:0] R13;
wire [7:0] R14;
wire [7:0] R15;
wire [7:0] R16;
wire [7:0] RegOut;
wire [7:0] Rin;
wire [15:0] enables;

mux88 mux instance (
    .i0(InA),
    .i1(InB),
    .i2(CUconst),
    .i3(ALUout),
    .i4(RegOut),
    .i5(RegOut),
    .i6(RegOut),
    .i7(RegOut),
    .sel(InMuxAdd),
    .o(Rin)
);

decoder decoder instance (
    .RegAdd(RegAdd),
    .WE(WE),
    .oDec(enables)
);

mux8_16 mux8_16_instance (
    .c1(R1),
    .c2(R2),
    .c3(R3),
    .c4(R4),
    .c5(R5),
    .c6(R6),
    .c7(R7),
    .c8(R8),
    .c9(R9),
    .c10(R10),
    .c11(R11),
    .c12(R12),
    .c13(R13),
    .c14(R14),
    .c15(R15),
    .c16(R16),
    .sel(OutMuxAdd),
    .RegOut(RegOut)
);

regfile registers (
    .Rin(Rin),
    .rst(reset),
    .clk(clk),
    .en(enables),
    .Rout1(R1),
    .Rout2(R2),
    .Rout3(R3),
    .Rout4(R4),
    .Rout5(R5),
    .Rout6(R6),
    .Rout7(R7),
    .Rout8(R8),
    .Rout9(R9),
    .Rout10(R10),
    .Rout11(R11),
    .Rout12(R12),
    .Rout13(R13),
    .Rout14(R14),
    .Rout15(R15),
    .Rout16(R16)
);
endmodule

module mux88(
    input[7:0] i0,
    input[7:0] i1,
    input[7:0] i2,
    input[7:0] i3,
    input[7:0] i4,
    input[7:0] i5,
    input[7:0] i6,
    input[7:0] i7,
    input[2:0] sel,
    output reg[7:0] o
);
always @(*) begin
    case (sel)
        3'b000: o = i0;
        3'b001: o = i1;
        3'b010: o = i2;
        3'b011: o = i3;
        3'b100: o = i4;
        3'b101: o = i5;
        3'b110: o = i6;
        3'b111: o = i7;
        default: o = 0;
    endcase
end
endmodule
```

Figure 9 - Register Block Verilog Code (Part 1)

```
module decoder (

input [3:0] RegAdd,
input WE,
output reg [15:0] oDec
);

always @(*) begin

    if (WE == 0) begin
        oDec = 0;
    end else begin
        oDec = 0;
        oDec[RegAdd] = 1;
    end
end
endmodule

module mux8_16(
input [7:0] c1,
input [7:0] c2,
input [7:0] c3,
input [7:0] c4,
input [7:0] c5,
input [7:0] c6,
input [7:0] c7,
input [7:0] c8,
input [7:0] c9,
input [7:0] c10,
input [7:0] c11,
input [7:0] c12,
input [7:0] c13,
input [7:0] c14,
input [7:0] c15,
input [7:0] c16,
input [3:0] sel,
output reg [7:0] RegOut
 );
 always @(*) begin
    case (sel)
        4'b0000: RegOut = c1;
        4'b0001: RegOut = c2;
        4'b0010: RegOut = c3;
        4'b0011: RegOut = c4;
        4'b0100: RegOut = c5;
        4'b0101: RegOut = c6;
        4'b0110: RegOut = c7;
        4'b0111: RegOut = c8;
        4'b1000: RegOut = c9;
        4'b1001: RegOut = c10;
        4'b1010: RegOut = c11;
        4'b1011: RegOut = c12;
        4'b1100: RegOut = c13;
        4'b1101: RegOut = c14;
        4'b1110: RegOut = c15;
        4'b1111: RegOut = c16;
        default: RegOut = 0;
    endcase
 end
 endmodule

module register(
    input[7:0] Rin,
    input rst,En,clk,
    output reg [7:0] Rout
);

always @(posedge clk, posedge rst) begin
    if(rst == 1) begin
        Rout <= 0;
    end else begin
        if(En == 1) begin
            Rout <= Rin;
        end

    end
end
endmodule

module regfile(
    input[7:0] Rin,
    input rst,clk,
    input [15:0] en,
    output [7:0] Rout1,
    output [7:0] Rout2,
    output [7:0] Rout3,
    output [7:0] Rout4,
    output [7:0] Rout5,
    output [7:0] Rout6,
    output [7:0] Rout7,
    output [7:0] Rout8,
    output [7:0] Rout9,
    output [7:0] Rout10,
    output [7:0] Rout11,
    output [7:0] Rout12,
    output [7:0] Rout13,
    output [7:0] Rout14,
    output [7:0] Rout15,
    output [7:0] Rout16
);
```

Figure 10 - Register Block Verilog Code (Part 2)

```
register reg1 (


    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[0]),
    .Rout(Rout1)
);
register reg2 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[1]),
    .Rout(Rout2)
);
register reg3 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[2]),
    .Rout(Rout3)
);
register reg4 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[3]),
    .Rout(Rout4)
);
register reg5 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[4]),
    .Rout(Rout5)
);

register reg6 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[5]),
    .Rout(Rout6)
);

register reg7 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[6]),
    .Rout(Rout7)
);

register reg8 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[7]),
    .Rout(Rout8)
);

register reg9 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[8]),
    .Rout(Rout9)
);

register reg10 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[9]),
    .Rout(Rout10)
);

register reg11 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[10]),
    .Rout(Rout11)
);

register reg12 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[11]),
    .Rout(Rout12)
);

register reg13 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[12]),
    .Rout(Rout13)
);

register reg14 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[13]),
    .Rout(Rout14)
);

register reg15 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[14]),
    .Rout(Rout15)
);

register reg16 (
    .Rin(Rin),
    .rst(rst),
    .clk(clk),
    .En(en[15]),
    .Rout(Rout16)
);
endmodule
```

Figure 11 - Register Block Verilog Code (Part 3)

3. Control Unit

ASMs (Algorithmic State Machine) are known to consist of two parts: Control and Datapath. What has been built so far is called the datapath since registers and logic units have a purpose of processing the data. However, the datapath needs to be controlled via a state machine to be able to use the registers. This state machine part of the system is called the Control Unit.

The control unit responses to clock and asynchronous reset and operates when Start = 1. While working, the "Busy" output goes high. The constant output is "**CUconst <= 8'b00000001"** for this design. **InMuxAdd** is the address selector for the InMux multiplexer in the Register Block and the **OutMuxAdd** is the address selector for the OutMux multiplexer. **RegAdd** is the register address selector of the same block. All the address outputs are 8-bit. On the other hand, **WE** (Write Enable) output controls the enable pin of the decoder in the Register Block which decides whether to write the data to the registers is 1-bit output. **InsSel** is the only output that goes to ALU instead of the Register Block. It is 2-bit output and acts as a selector of the operation (opcode) for the ALU (AND, XOR, ADD, SHIFT). C0 input is the mux output which is connected to the adder and shifter of the ALU, and Z input is the zero-comparator output of the ALU. However, the zero-comparator is not required in this calculation, like AND gate so, they are ignored.

In this design, since the system has no counter due to the document's referred to only allow state and output registers, there are 13 states.

IDLE: Waits until Start = 1 and writes register[1] = constant.

INITIAL0: Writes register[2] = B.

INITIAL1: Writes register[3] = A.

INITIAL2: Writes register[4] = B *(memory: 0 , const , B , A , B).*

NEGATIVEB0: Writes the result of the XOR(B) to the register[2] *(memory: 0, const, exor(const, B), A, B).*

NEGATIVEB1: Writes the result of the XOR operation to the place of the constant *(memory: 0, const<<1, exor (const, B), A, B).*

NEGATIVEB2: Repeats B0 and B1 (WE = 0) until (C0 = 1) because the shift of the constant is required to be 8 until comes back to 0000_0001. When (C0 = 1) add constant with B to complement B and write it on register[2]. *(memory: 0, const<<1, exor(const, B), A , B).*

NEGATIVEB3: Replace constant with A (to register[1]) to make A – B.

A_MINUS_B0: Write the result of add operation of A and -B to register[1].

A_MINUS_B1: To check whether A – B is negative in next state, A is shifted (WE = 0).

A_MINUS_B2: Repeat A_MINUS_B0 and A_MINUS_B1 until the A – B is negative (WE = 0). When A – B is negative, bring +B to its origin (register[2]).

OUT0: Do the add to the Prepare the result to be able to give in the next state by saving the results to the main output (register[0]) *(memory before: 0,[(A mod B)-B], A, B; memory after: (A mod B),[(A mod B)-B], A, B).*

OUT1: Busy = 0 and go back to the idle state.

Note: There are explanations within the code to explain the states in more details (Figure 14-15).

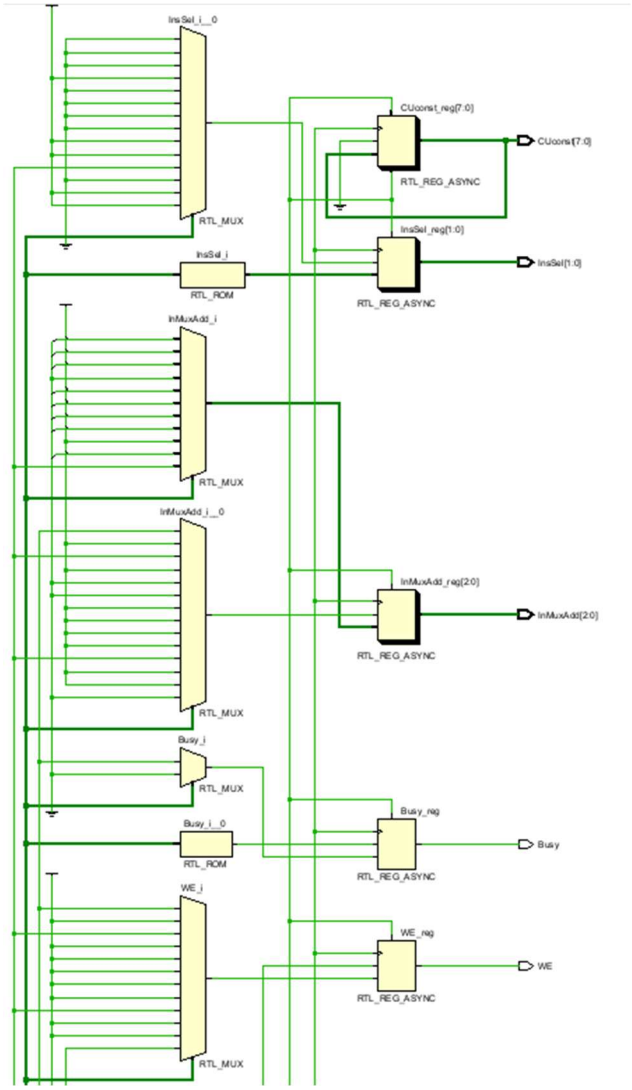

Figure 12 - Control Unit RTL Schematic (Part 1)

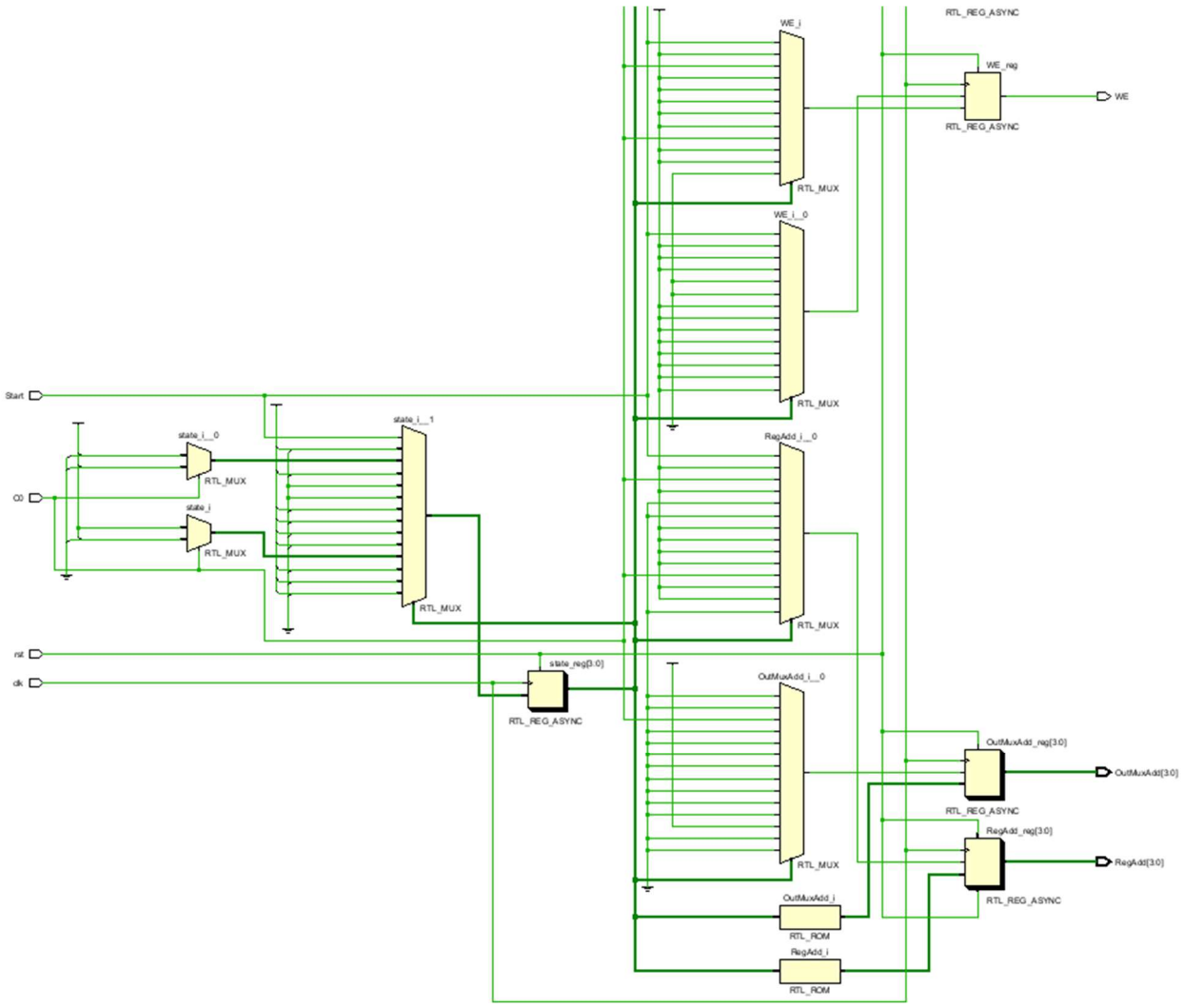

Figure 13 - Control Unit RTL Schematic (Part 2)

```
`timescale 1ns / 1ps

module cu(
input clk,rst,Start,
input C0,Z,
output reg WE,Busy,
output reg [7:0] CUconst,
output reg [2:0] InMuxAdd, // which input
output reg [3:0] RegAdd, // which register
output reg [3:0] OutMuxAdd,
output reg [1:0] InsSel // ALU opcode
    );
// we have to use many states because the document only allows state regs and output regs not counter and stuff
localparam idle = 4'b0000;
localparam initial0 = 4'b0001;
localparam initial1 = 4'b0010;
localparam initial2 = 4'b0011;
localparam negativeB0 = 4'b0100; // xor(b,const) = k
localparam negativeB1 = 4'b0101; // b <= k
localparam negativeB2 = 4'b0110; // const <= const << 1; branch if const<<1 [0] == 1
localparam negativeB3 = 4'b0111; // loading A to reg1

localparam A_minus_B0 = 4'b1000;
localparam A_minus_B1 = 4'b1001;
localparam A_minus_B2 = 4'b1010;
localparam out0 = 4'b1011;
localparam out1 = 4'b1100;
reg [3:0] state;

parameter a = 3'b000 ;
parameter b = 3'b001 ;
parameter const = 3'b010;
parameter aluout = 3'b011;
parameter regout = 3'b111;

// and is not used
parameter exor = 2'b01;
parameter add = 2'b10;
parameter shift = 2'b11;

always @(posedge clk, posedge rst) begin
    if(rst == 1) begin
        WE <= 0;
        Busy <= 0;
        CUconst <= 8'b00000001;
        InMuxAdd <= 3'b000;
        RegAdd <= 4'b0011;
        OutMuxAdd <= 4'b0000;
        InsSel <= 0;
        state <= 0;
    end else begin
```

Figure 14 - Control Unit Verilog Code (Part 1)

```
        case (state)
            idle: begin
            if (Start == 1) begin // it says on the document we should only use C0 and Z but we have to use start signal to ensure correct behaviour
                Busy <= 1;
                RegAdd <= 4'b0001; // on reg2 (aka registers[1])
                InMuxAdd <= const; // write Const
                WE <= 1;
                state <= initial0;
            end else begin
                Busy <= 0;
                state <= idle;
            end
            end
            initial0 : begin
                RegAdd <= 4'b0010; // on reg3
                InMuxAdd <= b; // write B
                WE <= 1; // on the next clock edge (when we transition to initial2) the register will be updated
                state <= initial1;
            end
            initial1 : begin
                RegAdd <= 4'b0011; // on reg4
                InMuxAdd <= a; // write A
                WE <= 1;
                state <= initial2;
            end
            initial2 : begin // to make memory: 0 , const , B , A , B
                RegAdd <= 4'b0100; // on reg5
                InMuxAdd <= b; // write B
                WE <= 1;
                state <= negativeB0;
            end

            negativeB0 : begin // to make memory: 0 , const , exor(const,B) , A , B
                InsSel <= exor;
                RegAdd <= 4'b0010; // on reg3 ( in the place of the first B)
                InMuxAdd <= aluout; // write the result of xor operation on B
                WE <= 1;
                state <= negativeB1;
            end
            negativeB1 : begin // to make memory: 0 , const<<1 , exor(const,B) , A , B
                InsSel <= shift;
                RegAdd <= 4'b0001; // on reg2 where the const is
                InMuxAdd <= aluout; // write the result of xor operation in the place of the const
                WE <= 1;
                state <= negativeB2;
            end
            negativeB2 : begin // to make memory: 0 , const<<1 , exor(const,B) , A , B
                //C0 == 1 means that we complemented B, now to find minus B we need to add const(+1) to B.
                if (C0 == 1) begin
                    state <= negativeB3;
                    InsSel <= add; // we add const (which left shifted 8 times to become 0000_0001 again) to complemented B
                    WE <= 1;
                    InMuxAdd <= aluout; // we place -B in the place of ~B
                    RegAdd <= 4'b0010; // on reg3 (where the ~B is)
                end else begin
                    WE <= 0; // we don't write anything and turn back to xor'in and shifting
                    state <= negativeB0;
                end
            end
            negativeB3 : begin //we don't need const now we load A to its place to make A-B
                OutMuxAdd <= 4'b0011; //from where A is
                RegAdd <= 4'b0001; // to reg2 where the const is
                InMuxAdd <= regout; //to make A-B
                WE <= 1;
                state <= A minus B0;
            end

            A minus B0 : begin
                InsSel <= add;
                WE <= 1;
                RegAdd <= 4'b0001; // on reg2 ( in the place of the A)
                InMuxAdd <= aluout; // write the result of add operation of A and -B
                state <= A minus B1;
            end
            A minus B1 : begin
                InsSel <= shift;
                WE <= 0; // we don't write the result of the shift operation we only check it in the next state to see if A-B is negative
                state <= A minus B2;
            end
            A_minus_B2 : begin
                if(C0 == 1) begin // A-B has reach negative
                    OutMuxAdd <= 4'b0100; //where +B is
                    RegAdd <= 4'b0010; // on reg2 where the -B is
                    InMuxAdd <= regout;
                    WE <= 1;
                    state <= out0;
                end else begin
                    WE <= 0;
                    state <= A_minus_B0;
                end
            end
            out0 : begin // memory when commands of this state will be implemented: 0,[(A mod B)-B], A, B
                InsSel <= add;
                WE <= 1;
                RegAdd <= 4'b0000; // on reg0 (the output)
                InMuxAdd <= aluout; // write the result of add operation of A and -B
                state <= out1;
            end
            out1 : begin
                Busy <= 0;
                state <=idle;
            end
            default: state <= idle;
        endcase
    end
    end
    endmodule
```

Figure 15 - Control Unit Verilog Code (Part 2)

4. Top Module and Simulations

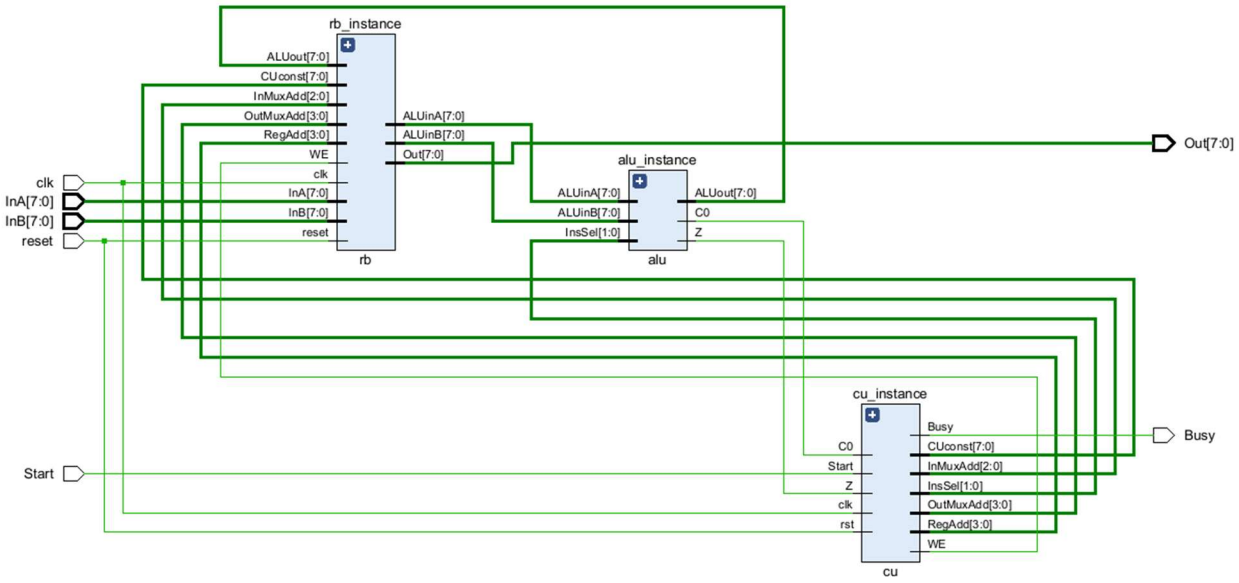

Figure 16 - Top Module RTL Schematic

```
`timescale 1ns / 1ps

module top_tb();

reg clk,reset,start;
reg[7:0] a;
reg[7:0] b;
wire busy;
wire[7:0] out;
wire[7:0] shouldBeResult;

top DUT(
    .clk(clk),
    .reset(reset),
    .Start(start),
    .Busy(busy),
    .InA(a),
    .InB(b),
    .Out(out)
);

assign shouldBeResult = a%b;

always begin
    clk = ~clk;
    #5;
end

initial begin
    clk = 0;
    reset = 1;
    start = 0;
    a = 8'b0111_0110; //118
    b = 8'b0000_0101; //5
    #20;
    reset = 0;
    start = 1;
    #10400;
    a = 8'b0111_0111; //119
    b = 8'b0000_0101; //5
    #10400;
    a = 8'd45;
    b = 8'd30;
    #10400;
    a = 8'd3;
    b = 8'd120;
    #10400;
    a = 8'd73;
    b = 8'd19;
    #10400;
    a = 8'b0000_0000; //255
    b = 8'b0111_1111; //255
    #10400;
    a = 8'b0111_1111; //255
    b = 8'b0000_0000; //0
    #10400;
    a = 8'b0011_1110; //62
    b = 8'b0011_1110; //62
    #10400;
    $finish();
end
endmodule
```

Figure 17 - Top Module Testbench Code

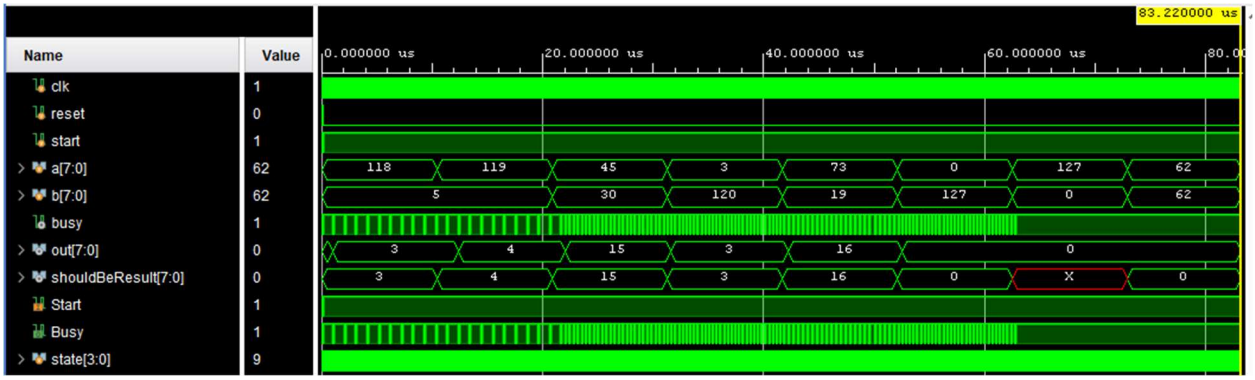

Figure 18 - Top Module Behavioral Simulation

In the simulation, the values tested are checked with the **modulus operator (%)** with "**shouldBeResult**" parameter. As it can be seen in the simulation that the design works as intended and the results are parallel with the "shouldBeResult" parameter. However, the parameter calculated in the simulation goes "**X**" when trying to divide 127 with 0. The reason behind this is basic math, a number cannot be divided with zero. Since the design does not have an error output the module gives zero instead of "x". Besides that, all the entries and edge values resulted in correctly. Also, a delay has been observed due to the time requirement for subtraction computations.

Moreover, if the busy output is observed, the rate of time spent on each calculation depends on the number of subtractions that have been made which grows larger when either the input A is getting larger, or the input B is getting lower.