Table of Contents

Problem statement:

In this assignment, we are to design a processor to calculate the Greatest Common Divisor of two integers implemented using Euclidian's Algorithm.

Explanation of Approach:

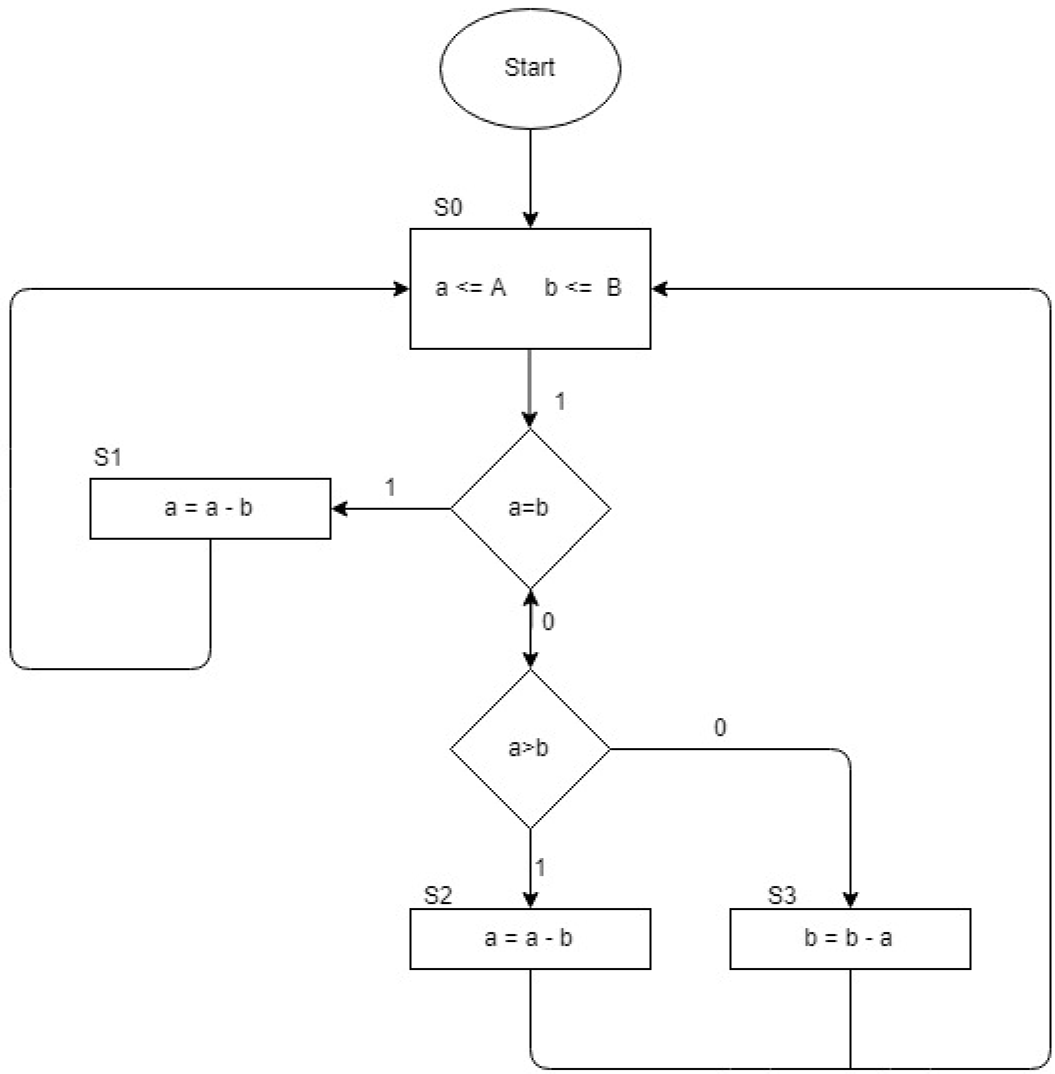
Based on the steps for Euclideans' Algorithm, we knew the processor would need to have a subtractor, comparators, and a few multiplexers. Our first approach was to create the data path to understand the logic of the processor, this consisted of creating two inputs going through a select line and into registers, because the inputs going into the starting registers would be different depending on the state the single bit select line would be necessary. The outputs would feed into either the comparator, subtractor, or the register to output the final value. The output of the subtractor would feed back as an input to the starting input registers and would continue to do so until some point that the two values are equal. Our approach for the data path was to instantiate each component inside as a module and use wires to make the necessary connections to the control unit. Therefore, we created a module for the two input registers, output register, multiplexers as the select line for the inputs, a subtractor, and a comparator. Most of the modules used in this assignment were used in the previous assignment, such as the subtractor and multiplexers.

Furthermore, we designed the control unit with a four state, state machine, with the first state (S0) being the start and to load the input values into the registers. At the first state the decision if the inputs are equal to, greater than or not greater than, and all else would be determined for the next states. The second state (S1) would output the final value should the two starting values already be equal. The third state (S2) would go to the subtractor if the first input (In0) be greater than the second input (In1), and the fourth state (S3) would do the same if the second input were greater than the first. After, the data path module and control unit module, the top module would instantiate the two with wires and output the value by encoding the value onto the board, similar to the past assignment.

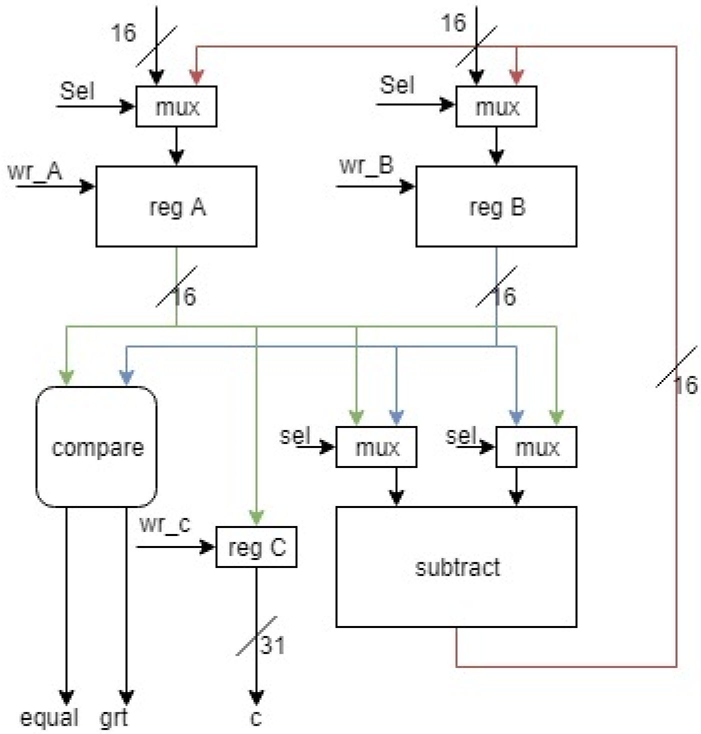
Problems Encountered at Simulation and Implementation:

The output to display onto the board the way we wanted was tedious and challenging. There were things such as getting the counter the right speed to show every state or calculation of the inputs and the resulting output. Initially, the simulation would not display the final output, this was because of wiring issues and some faults in the code for the submodules.

Block Diagram:



Algorithmic State Machine:



Verilog Codes Used:

```
module GCD(
    input  CLK100MHZ, rst,  //start
    input  SW1,
    output reg [7:0] a_to_g,
    output reg [7:0] an
    );

    parameter start      = 1'b1;
    parameter [15:0] In0 = 16'h0321;
```

```
parameter [15:0] In1 = 16'h0123;

reg  [3:0]  LED_BCD;
wire [3:0]  LED_activating_counter;
reg  [19:0] refresh_counter;

wire valid, Wr_In0, Wr_In1, Wr_C, Sel_In0, Sel_In1, Sel_a, Sel_b; //
wire eq, gth;
wire [31:0] C, I0_out, I1_out;
wire [1:0] State_Y;

GCD_datapath    DP(CLK100MHZ, rst, Wr_In0, Wr_In1, Wr_C, Sel_In0, Sel_In1, Sel_a, Sel_b, In0, In1, eq, gth, C, I0_out, I1_out);
GCD_controlunit CU(CLK100MHZ, rst, start, eq, gth, valid, Wr_In0, Wr_In1, Wr_C, Sel_In0, Sel_In1, Sel_a, Sel_b, State_Y);

always @(posedge CLK100MHZ) begin
    if(rst)
        refresh_counter <= 0;
    else
        refresh_counter <= refresh_counter + 1;
end

assign LED_activating_counter = refresh_counter[19:17];
```

```
always @(*) begin
    case(SW1)
        1'b1: begin
                case(LED_activating_counter)
                    3'b000: begin
                            an = 8'b01111111;
                            LED_BCD = I1_out[31:28];
                        end
                    3'b001: begin
                            an = 8'b10111111;
                            LED_BCD = I1_out[27:24];
                        end
                    3'b010: begin
                            an = 8'b11011111;
                            LED_BCD = I1_out[23:20];
                        end
                    3'b011: begin
                            an = 8'b11101111;
                            LED_BCD = I1_out[19:16];
                        end
                    3'b100: begin
                            an = 8'b11110111;
                            LED_BCD = I1_out[15:12];
                        end
```

```
            3'b101: begin
                  an = 8'b11111011;
                  LED_BCD = I1_out[11:8];
                end
            3'b110: begin
                  an = 8'b11111101;
                  LED_BCD = I1_out[7:4];
                end
            3'b111: begin
                  an = 8'b11111110;
                  LED_BCD = I1_out[3:0];
                end
          endcase
        end

    1'b0: begin
        case(LED_activating_counter)
          3'b000: begin
                an = 8'b01111111;
                LED_BCD = In0[15:12];
              end
          3'b001: begin
                an = 8'b10111111;
                LED_BCD = In0[11:8];
```

```
    end
3'b010: begin
      an = 8'b11011111;
      LED_BCD = In0[7:4];
    end
3'b011: begin
      an = 8'b11101111;
      LED_BCD = In0[3:0];
    end
3'b100: begin
      an = 8'b11110111;
      LED_BCD = In1[15:12];
    end
3'b101: begin
      an = 8'b11111011;
      LED_BCD = In1[11:8];
    end
3'b110: begin
      an = 8'b11111101;
      LED_BCD = In1[7:4];
    end
3'b111: begin
      an = 8'b11111110;
      LED_BCD = In1[3:0];
```

```
            end
        endcase
      end
   endcase
end

always @(LED_BCD) begin
   casex(LED_BCD)
      4'b0000: a_to_g = 8'b1_0000001;
      4'b0001: a_to_g = 8'b1_1001111;
      4'b0010: a_to_g = 8'b1_0010010;
      4'b0011: a_to_g = 8'b1_0000110;
      4'b0100: a_to_g = 8'b1_1001100;
      4'b0101: a_to_g = 8'b1_0100100;
      4'b0110: a_to_g = 8'b1_0100000;
      4'b0111: a_to_g = 8'b1_0001111;

      4'b1000: a_to_g = 8'b1_0000000;
      4'b1001: a_to_g = 8'b1_0001100;
      4'b1010: a_to_g = 8'b1_0001000;
      4'b1011: a_to_g = 8'b0_1100000;
      4'b1100: a_to_g = 8'b1_0110001;
      4'b1101: a_to_g = 8'b1_1000010;
      4'b1110: a_to_g = 8'b1_0110000;
```

```
        4'b1111: a_to_g = 8'b1_0111000;

        default: a_to_g = 8'bX_0000000;

      endcase

  end


endmodule


//------------------------------------------------------------------------------

module display(

  input [15:0] a,b,

  input CLK100MHZ, rst,

  output reg[7:0] a_to_g,

  output reg[7:0] an

  );


  reg  [3:0]  LED_BCD;

  wire [3:0]  LED_activating_counter;

  reg  [19:0] refresh_counter;


  always @(posedge CLK100MHZ) begin

    if(rst)

      refresh_counter <= 0;

     else

      refresh_counter <= refresh_counter + 1;
```

```
end

assign LED_activating_counter = refresh_counter[19:17];

always @(*) begin
    case(LED_activating_counter)
        3'b000: begin
                an = 8'b01111111;
                LED_BCD = a[15:12];
            end
        3'b001: begin
                an = 8'b10111111;
                LED_BCD = a[11:8];
            end
        3'b010: begin
                an = 8'b11011111;
                LED_BCD = a[7:4];
            end
        3'b011: begin
                an = 8'b11101111;
                LED_BCD = a[3:0];
            end
        3'b100: begin
                an = 8'b11110111;
```

```
            LED_BCD = b[15:12];
          end
      3'b101: begin
            an = 8'b11111011;
            LED_BCD = b[11:8];
          end
      3'b110: begin
            an = 8'b11111101;
            LED_BCD = b[7:4];
          end
      3'b111: begin
            an = 8'b11111110;
            LED_BCD = b[3:0];
          end
   endcase
 end

always@(LED_BCD)begin
casex(LED_BCD)
   4'b0000:   a_to_g = 8'b1_0000001;
   4'b0001:   a_to_g = 8'b1_1001111;
   4'b0010:   a_to_g = 8'b1_0010010;
   4'b0011:   a_to_g = 8'b1_0000110;
   4'b0100:   a_to_g = 8'b1_1001100;
```

```
        4'b0101:   a_to_g = 8'b1_0100100;

        4'b0110:   a_to_g = 8'b1_0100000;

        4'b0111:   a_to_g = 8'b1_0001111;


        4'b1000:   a_to_g = 8'b1_0000000;

        4'b1001:   a_to_g = 8'b1_0001100;

        4'b1010:   a_to_g = 8'b1_0001000;

        4'b1011:   a_to_g = 8'b0_1100000;

        4'b1100:   a_to_g = 8'b1_0110001;

        4'b1101:   a_to_g = 8'b1_1000010;

        4'b1110:   a_to_g = 8'b1_0110000;

        4'b1111:   a_to_g = 8'b1_0111000;

        default:   a_to_g = 8'bX_0000000;

    endcase

 end


endmodule


//------------------------------------------------------------------------------

module GCD_controlunit(

    input clk, rst, start, eq, gth,

    output valid, Wr_In0, Wr_In1, Wr_C, Sel_In0, Sel_In1, Sel_a, Sel_b,

    output [1:0] State_Y

    );
```

```
reg [7:0] Control_Variable;

reg [1:0] state, nstate;


//wire eq, gth;

wire valid, Wr_In0, Wr_In1, Wr_C, Sel_In0, Sel_In1, Sel_a, Sel_b;        //

//wire [1:0] State_Y;


parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;


//Reset and Update State

always @(posedge rst or negedge clk) begin

   if(rst) state <= 2'b00;

   else state <= nstate;

end


 //Next State

 always@(state or start or eq or gth) begin

    case(state)

        2'b00: begin if(~start) nstate <= 2'b00;

                    else if(eq) nstate <= 2'b01;

                    else if(gth) nstate <= 2'b10;

                    else nstate <=2'b11;

               end
```

```
      2'b01: nstate <= 2'b00;
      2'b10: begin if(eq) nstate <= 2'b01;
                  else if(gth) nstate<=2'b10;
                  else nstate<=2'b11;
             end
      2'b11: begin if(eq) nstate <= 2'b01;
                  else if(gth) nstate<=2'b10;
                  else nstate<=2'b11;
             end
      default: nstate<=2'b00;
   endcase
end


//Output
always @(state) begin
   case(state)
      2'b00: Control_Variable <= 8'b1100_1100;
      2'b01: Control_Variable <= 8'b0000_0011;
      2'b10: Control_Variable <= 8'b0011_1000;
      2'b11: Control_Variable <= 8'b0000_0100;
   endcase
end


assign Sel_In0 = Control_Variable[7];
```

```
    assign Sel_In1 = Control_Variable[6];
    assign Sel_a   = Control_Variable[5];
    assign Sel_b   = Control_Variable[4];
    assign Wr_In0  = Control_Variable[3];
    assign Wr_In1  = Control_Variable[2];
    assign Wr_C    = Control_Variable[1];
    assign valid   = Control_Variable[0];


    assign State_Y = nstate;


endmodule


//------------------------------------------------------------------------------
module GCD_datapath(
   input  clk, rst, Wr_In0, Wr_In1, Wr_C, Sel_In0, Sel_In1, Sel_a, Sel_b,
   input  [15:0] In0, In1,
   output eq, gth,
   output [15:0] C,
   output [15:0] I0_out, I1_out
   );


   wire [15:0] I0_wire, I1_wire, ALU;
   wire [15:0] A_wire, B_wire;
```

```
    RegisterIn0 Reg0(clk, rst, Wr_In0, Sel_In0, In0, ALU, I0_wire);

    RegisterIn1 Reg1(clk, rst, Wr_In1, Sel_In1, In1, ALU, I1_wire);

    RegisterC RegC(clk, rst, Wr_C, I0_wire, C);

    MuxA        MA(I0_wire, I1_wire, Sel_a, A_wire);

    MuxB        MB(I0_wire, I1_wire, Sel_b, A_wire);

    ALU         A(A_wire, B_wire, ALU);

    Comparator Com(I0_wire, I1_wire, eq, gth);


    assign I0_out = I0_wire;

    assign I1_out = I1_wire;


endmodule


//-----------------------------------------------------------------------------

module RegisterIn0(

    input  clk,

    input  rst,

    input  Wr_In0,

    input  Sel_In0,

    input  [15:0] In0,

    input  [15:0] ALU,

    output [15:0] out

    );
```

```
    reg [15:0] I0_wire;


    always @(posedge rst or posedge clk) begin
        if(rst)
            I0_wire <= 16'h0000;
        else if(Wr_In0) begin
            if(Sel_In0)
                I0_wire <= In0;
            else
                I0_wire <= ALU;
        end
    end


    assign out = I0_wire;


endmodule


//--------------------------------------------------------------------------
module RegisterIn1(
    input  clk,
    input  rst,
    input  Wr_In1,
    input  Sel_In1,
    input  [15:0] In1,
```

```
    input  [15:0] ALU,
    output [15:0] out
    );

    reg [15:0] I1_wire;

    always @(posedge rst or posedge clk) begin
        if(rst)
            I1_wire <= 16'h0000;
        else if(Wr_In1) begin
            if(Sel_In1)
                I1_wire <= In1;
            else
                I1_wire <= ALU;
        end
    end

    assign out = I1_wire;

 endmodule

//----------------------------------------------------------------------------
module RegisterC(
    input  clk,
```

```
    input  rst,

    input  Wr_C,

    input  [15:0] I0_wire,

    output [15:0] out

    );


    reg [15:0] C;


    always @(posedge rst or posedge clk) begin

        if(rst)

            C <= 16'h0000;

        else if (Wr_C)

            C <= I0_wire;

      //  else

       //     C<= 0;     /////

    end


    assign out = C;


endmodule


//--------------------------------------------------------------------------

module MuxA(

    input  [15:0] I0_wire,
```

```
    input  [15:0] I1_wire,
    input  Sel_a,
    output reg [15:0] A_wire
    );

    always @(I0_wire or I1_wire or Sel_a) begin
        if(Sel_a)
            A_wire <= I0_wire;
        else
            A_wire <= I1_wire;
    end

endmodule

//-----------------------------------------------------------------------------
module MuxB(
    input  [15:0] I0_wire,
    input  [15:0] I1_wire,
    input  Sel_b,
    output reg [15:0] B_wire
    );

    always @(I0_wire or I1_wire or Sel_b) begin
        if(Sel_b)
```

```
            B_wire <= I1_wire;
        else
            B_wire <= I0_wire;
    end

 endmodule

//----------------------------------------------------------------------------
module ALU(
    input  [15:0] A_wire,
    input  [15:0] B_wire,
    output reg [15:0] ALU
    );

    always @(A_wire or B_wire) begin
        ALU <= A_wire - B_wire;
    end

endmodule

//----------------------------------------------------------------------------
module Comparator(
    input  [15:0] I0_wire,
    input  [15:0] I1_wire,
```

```
    output eq,

    output gth

    );



    assign eq  = (I0_wire == I1_wire) ? 1 : 0;

    assign gth = (I0_wire > I1_wire) ? 1 : 0;



endmodule
```

Simulation Waveform:

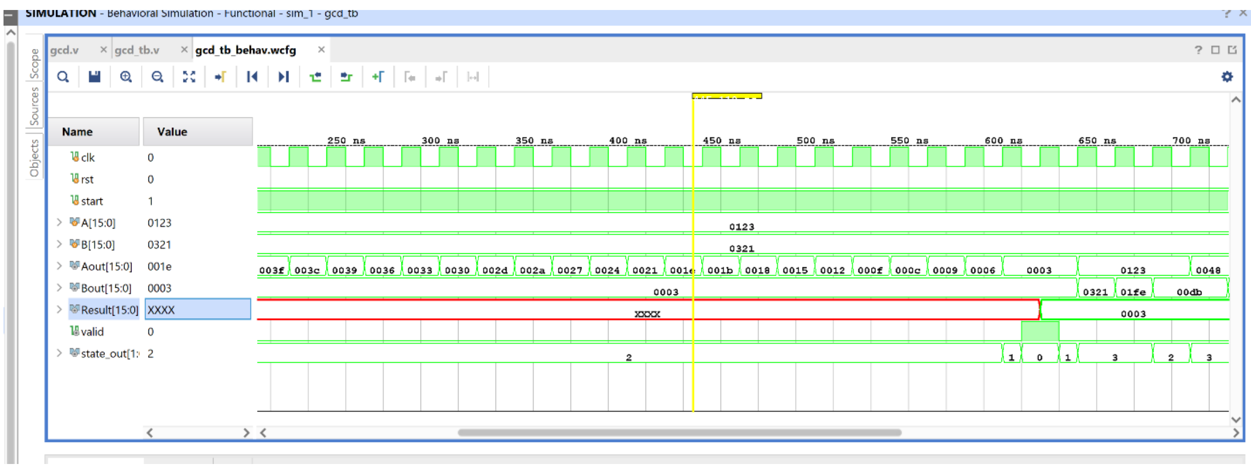


Figure 1: GCD sim