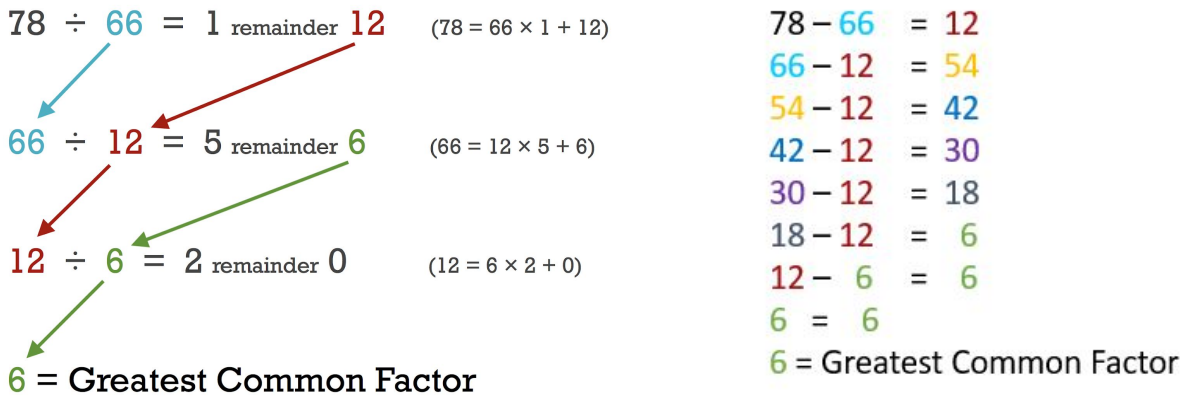
Bruno E. Gracia Villalobos & Bobby Reyes
EE 5193
Professor Eratne
Implementation of Euclid's Algorithm on Nexys 4 DDR FPGA
July 29, 2019

ABSTRACT
This project report explores the design of a processor to implement Euclid's algorithm on a Nexys-4 DDR FPGA using the Verilog HDL. The processor calculates the greatest common divisor (GCD) of two 8-bit integers selected using the onboard switches of the FPGA board and displays it in hexadecimal on the 7-segment displays.

PROBLEM
Euclid's algorithm is used to calculate the GCD of two integers using recursion. It is accomplished in different ways but common methods involve taking the remainder or the difference of both integers. Examples 1 & 2 show the two different methods using the same integer values. The processor for this project will implement the algorithm using one of the two methods and produce a solution that ranges from two 8-bit integer values. In implementation of this problem, the efficiency of power and hardware utilization should be considered in the design.



Examples 1 & 2 - Euclid's Algorithm using remainder (LEFT), and using subtraction (RIGHT)

SOLUTION

Step 1: Design the ASMD chart

Due to the circuit complexity of implementing division in hardware, the processor implements a simplified version of Euclid's algorithm using subtraction (1). To verify the accuracy of the simplified algorithm, Figure 1 showcases a MATLAB script used to calculate the GCD. Once multiple numbers were accurately tested for this algorithm, a Moore machine was designed with an ASMD chart for the processor.

```
1  A = 32;
2  B = 12;
3
4  while(~(A==B))
5
6      if(A>B)
7          A = A-B;
8      else
9          B = B-A;
10     end
11
12 end
13
14 disp(B);
```

State	Function
SET	Serves as a reset state and awaits user input to begin the GCD calculation
START	A "binning" state for determining the next step of Euclid's algorithm
S1	Subtracts the lowest number from the highest number with respect to A
S2	Subtracts the lowest number from the highest number with respect to B
S3	Since A==B at this state, GCD is transferred the data from register A for display to the 7-segments.

Figure 1 & Table 1 - MATLAB tests (LEFT), Processor state behavior (RIGHT)

To simplify the behavior of the ASMD chart, Table 1 showcases how the processor operates between five distinct states. Essentially, the first state, SET, takes in user input for the two 8-bit integers to use for the GCD calculation, and the last state, S3, displays the answer on the 7-segment displays. The processor will hold at state SET until the user is ready to begin the calculation.

Figure 2 displays the ASMD chart for the processor. Along the datapath are Verilog statements that showcase the transfer of information between registers. Two important remarks can be made about the datapath:

1) Nextstate register transfers are *calculated* upon change of state or user input
2) GCD calculation register transfers are *checked* upon the posedge of the onboard 100kHz clock, but *calculated* upon change of state (explanation is in the following steps)

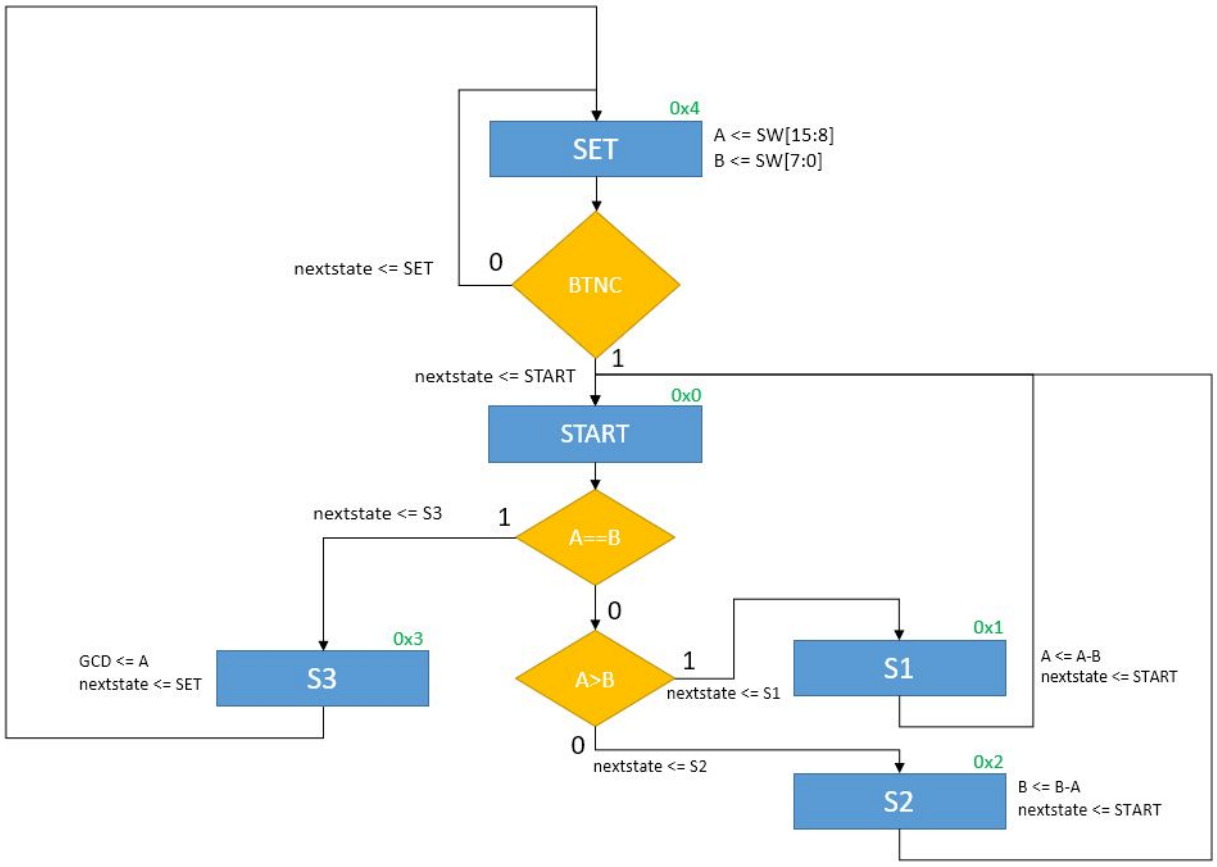


Figure 2 ASMD Chart for Euclid's Algorithm Processor

The blue boxes represent each state, and the yellow rhombuses represent decision boxes. On the top right corner of each state box is its binary assignment as will be discussed in ***Step 3***.

Step 2: Behavioral Model in Verilog

Three procedural blocks are used to implement the ASMD chart for the processor for both clarity and desired operation. In Figure 3, the operation of the Top Module is showcased. The purpose of each procedural block can be broken down given its function as a control signal, decision block evaluation, or state register transfers. Each block is triggered depending on the clock, change of state, or user input. This configuration helped to visualize the implementation of the ASMD chart in the behavioral model of the processor.

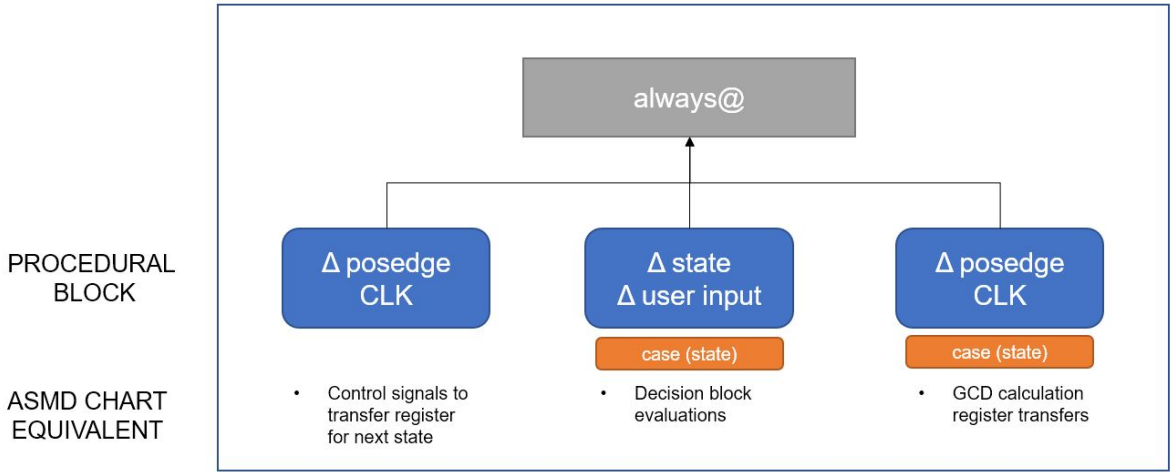


Figure 3 Top Module Procedural Blocks

To implement Euclid's Algorithm in this manner, a state machine was created which consists of the three procedural blocks in Figure 3. The duties of each block and the corresponding Verilog code are presented below:

Block 1. Update the state accordingly to the ASMD chart.

```
always @(state or BTNC) begin
    case(state)
          SET: begin
            if(BTNC)
                nextstate <= START;
            else
                nextstate <= SET;
            end
        START: begin
            if(A == B) nextstate <= S3;
            else if(A > B) nextstate <= S1;
            else nextstate <= S2;
        end
        S1: nextstate <= START;
        S2: nextstate <= START;
        S3: nextstate <= SET;
    endcase
end
```

Block 2. Calculate the output at each state.

```
always @(posedge CLK100MHZ) begin
    case(state)
       SET: begin
                A <= SW[15:8];
                B <= SW[7:0];
            end
        S1: A <= A - B;
        S2: B <= B - A;
        S3: GCD <= A;
    endcase
end
endmodule
```

Block 3. Advance the state to the next state of the machine.

```
always @(posedge CLK100MHZ) begin
    state <= nextstate;
end
```

It is important to note that procedural blocks 1 & 2 were initially combined and behavioural simulations of this method resulted in the desired outcome of calculating the GCD, however during post implementation functional testing the result was incorrect and unexpected. This may have been a consequence of issues with timing in hardware, that the behavioural simulation does not take into account. It was because of these issues that the output of each state and the assignment of the next state were separated. Doing this resolved all issues that resulted in unexpected behaviour of the processor.

Step 3: Behavioral Simulation

With the behavioral model in place as the Top Module, a testbench is next presented to test the desired operation of the processor. To better understand the simulation results, Figure 4 showcases the binary state assignment for each of the five states in the ASMD chart. Even though the ASMD chart signifies state SET at the top, state START has binary assignment 000 and state SET has binary assignment 100 to signal the “first” step in executing the GCD algorithm.

```
parameter START = 3'b000,
          S1    = 3'b001,
          S2    = 3'b010,
          S3    = 3'b011,
          SET   = 3'b100;
```

Figure 4 Moore Machine Binary State Assignment

Running a behavioral simulation in Vivado verified the efficacy of the processor. For example, using the following test scenario, the processor accurately calculated the GCD in 17 clock cycles.

A = 0x4e = 78 dec
B = 0x42 = 66 dec
GCD = 0x06

In Figure 5, the left side bar shows the nets and registers communicating with the processor. To prepare for implementation on the board, registers SW[15:0], clock CLK100MHZ, and push button BTNC are used as specified in the constraint file. Nets GCD[7:0] and state[2:0] represent the answer of the algorithm and current state respectively. Both SW[15:0] and BTNC are specified as follows to simulate the user input:

SW[15:0] = 0x4e42 (upper 8bits is A, lower 8bits is B)
BTNC = 1

The arrows shown below signify the execution of its respective state, and counting these from left to right minus 1 will give the running time of the algorithm in clock cycles.

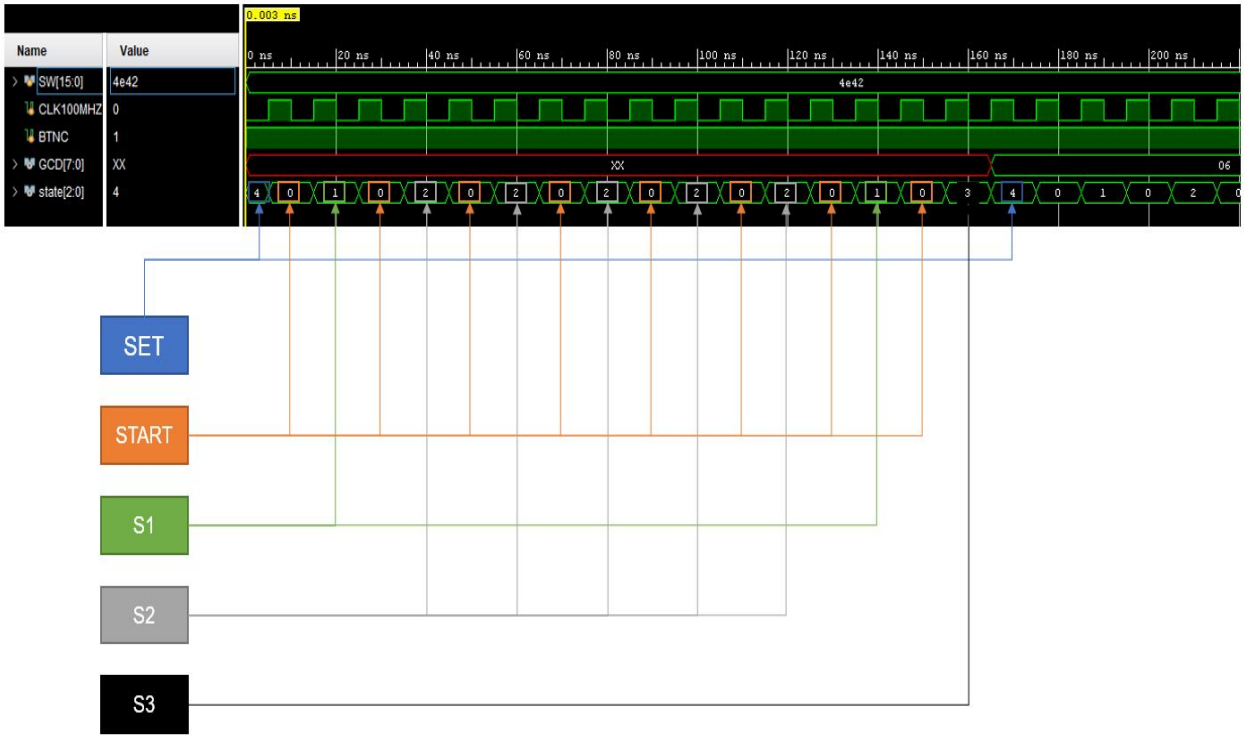


Figure 5 Behavioral Simulation in Xilinx Vivado

Step 4: 7-Segment Display Integration

With the primary operation of the processor verified, the next step included displaying the answer on the 7-segment display. Given that the input values into the processor are each 8 bits in size, the processor would produce an answer that itself would not exceed 8 bits. Therefore, two hex digits would be required to display the largest possible outcome of the GCD. This in-turn means that two of the eight, 7-segment display anodes would be required to display each hex value. In order to produce this display we were able to reuse 2 modules from assignment 3; the *SLOW_CLK* module, and the *display* module.

The *SLOW_CLK* module is utilized to slow down the onboard 100Mhz clock which is too fast to multiplex between the anode of the 7-segment displays. This module slows that clock down to 1KHz which is needed to rapidly switch on and off individual displays to have the “constantly on” effect of the 2 anodes being used.

The *display* module produces the cathode bit pattern used to activate the 7-segments themselves. This module outputs an 8-bit pattern based on a 4-bit input, which is used to represent the corresponding hex value for the input. This module is instantiated twice

using the upper and lower 4-bits of the GCD result and producing an 8-bit pattern for each half as seen in Figure 6.

```
display  d1(GCD[7:4],cath[1]);
display  d2(GCD[3:0],cath[0]);
```

Figure 6 display module instantiations

Finally the procedural block in Figure 7, which is triggered on the posedge of the *SLOW_CLK* module is used to multiplex between the anode and cathode bit pattern being used to display. Using a counter the block switches between (regAN[0], cath[0]), and (regAN[1], cath[1]) which represent both the first and second anodes, and the most significant and least significant 4-bits of the resultant GCD, respectively.

```
always @(posedge SLOWCLK) begin
    val = val + 1'b1;
    regAN[1:0] = {regAN[0], regAN[1]};
    regCA = cath[val];
end
```

Figure 7 Procedural block to multiplex between active anode and cathode of display

Step 5: Results

With all of the modules designed and working properly the verilog code was Synthesized and programmed into the FPGA. The result was that the user was able to calculate the GCD of two 8-bit binary numbers by setting SW[15:8] for input A, and SW[7:0] for input B. Once the switches were set, pressing the push button *BTNC* on the FPGA would trigger the state machine within the hardware and the result was displayed in hex on two of the eight seven segment displays. This worked for all number combinations up to and including the largest possible GCD, 255 which would result if all 16 switches were asserted, and inputs A and B were both 255.

CONCLUSION

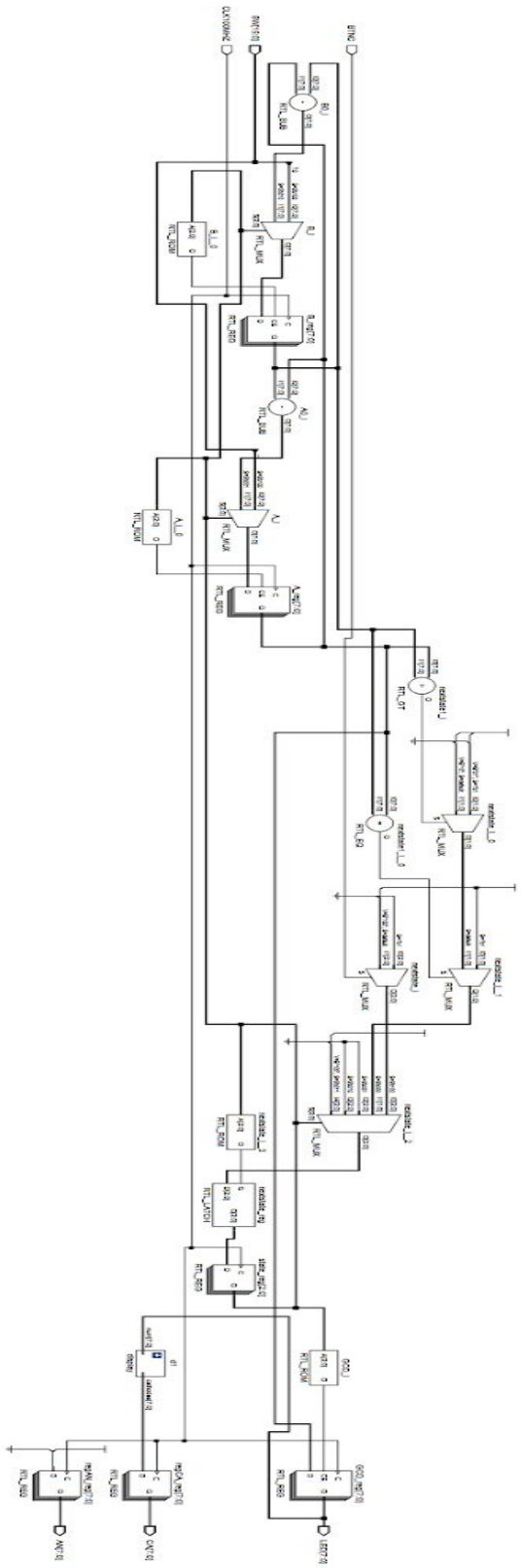
Euclid's Algorithm was a great opportunity to implement a FSM in hardware. Without having to implement the design in hardware the problem of finding the GCD is not very hard. The modulus operator (%) can be used within a simple block to obtain the result easily. However, using the modulus operator would add to the complexity of the hardware design and therefore would be a non-ideal approach to the problem. Thus, using the simplified version of Euclid's algorithm was the best approach.

This project gave more opportunity to learn from mistakes, and problems that came with designing the processor and implementing it into hardware. The issue of the *behavioural simulation* not matching the behavior of the FPGA post implementation, when the outputs and state assignments were placed in the same procedural block was the very first problem that was encountered. Because, the behavioural simulation showed the expected result it was hard to understand why the hardware did not behave properly, but by looking at the schematic generated by Vivado it was clear that by nesting the outputs of the states, which were performing subtraction into the same block that assigned the next state, there was a huge tangling of nets and wires and hardware that created problems with timing of the components and signals. By separating these two operations with three procedural blocks as defined in 2, the schematic was greatly simplified and the problem rectified.

The next issue came from displaying the result in the 7-segment displays. In this particular problem only 2 of the 8 seven segment displays were needed. Setting up the anodes so that only 2 of the displays would be on was initially tricky, but upon realizing how to properly create the output and register vectors (assign AN[1:0] = regAN) to achieve this we have learned how to activate any of the 7-segment displays we need for any future projects.

This project succeeded in strengthening this groups understanding of ASMD's, 7-segment displays, clocks, and hardware optimization. Given further time to improve, we believe we could solve this problem more efficiently using other methods to calculate the GCD using other methods of Euclid's algorithm.

SCHEMATIC



BLOCK DIAGRAM

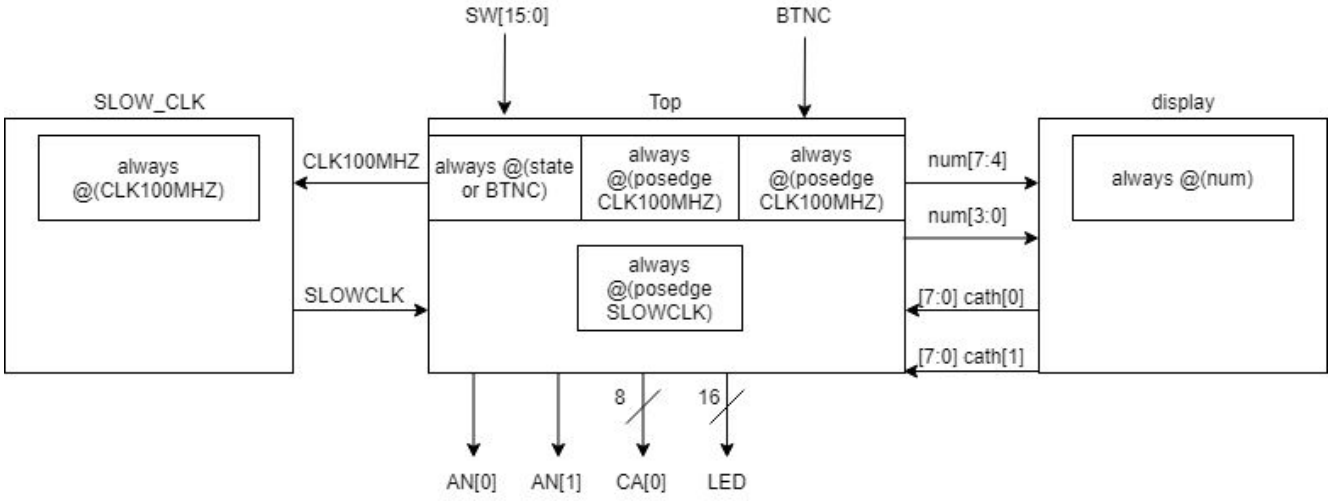


Figure 8 Internal Block Diagram

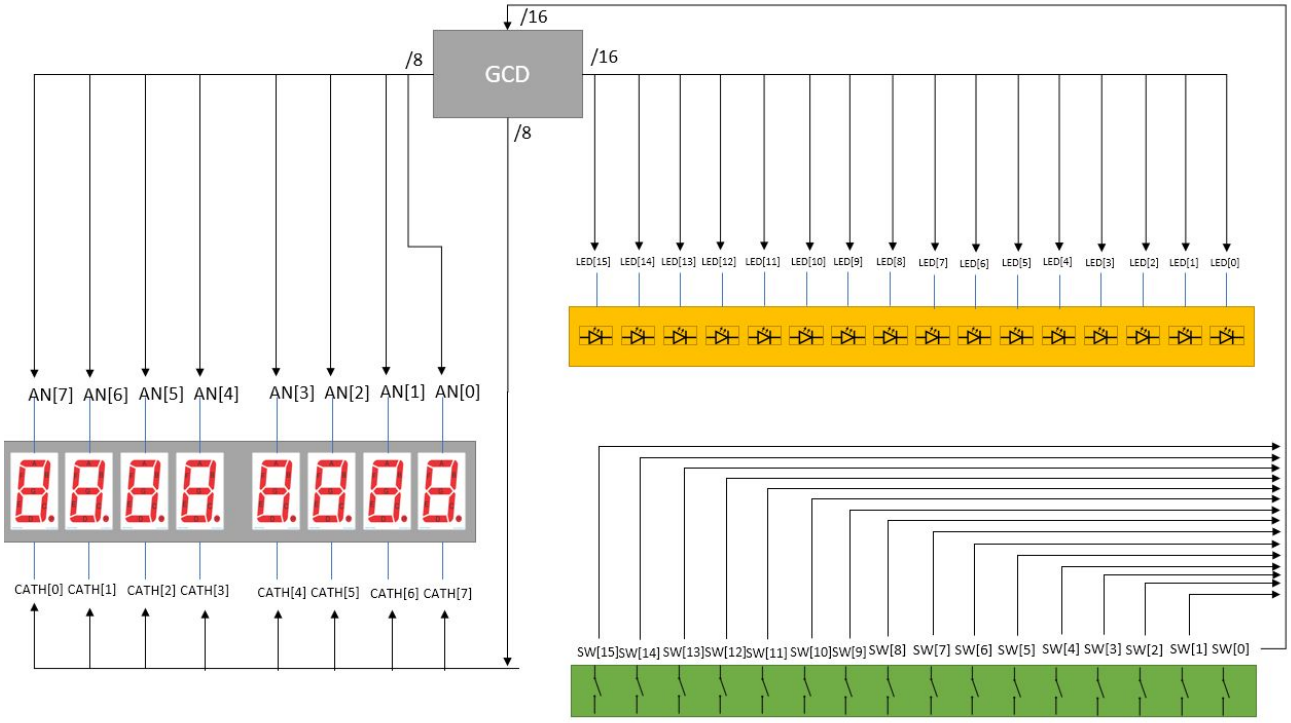


Figure 9 External Block Diagram

VERILOG CODE

```
module GCD(
input [15:0] SW,
input CLK100MHZ,
input BTNC,
output [7:0] CA,
output [7:0] AN
);

parameter START = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011, SET = 3'b100;
reg [2:0] state, nextstate;
reg [7:0] GCD, A, B, regCA,regAN;
reg val;
wire [7:0] cath[7:0];
wire SLOWCLK;

SLOW_CLK      s1(CLK100MHZ,SLOWCLK);
display       d1(GCD[7:4],cath[1]);
display       d2(GCD[3:0],cath[0]);

assign AN[1:0] = regAN;
assign CA = regCA;

initial val = 1'b0;
initial regAN[1:0] = 2'b10;

always @(posedge SLOWCLK) begin
   val = val + 1'b1;
   regAN[1:0] = {regAN[0], regAN[1]};
   regCA = cath[val];
end

always @(posedge CLK100MHZ) begin
   state <= nextstate;
end

initial state = SET;

always @(state or BTNC) begin
   case(state)
        SET: begin
          if(BTNC)
             nextstate <= START;
          else
             nextstate <= SET;
          end
       START: begin
          if(A == B) nextstate <= S3;
          else if(A > B) nextstate <= S1;
          else nextstate <= S2;
       end
       S1: nextstate <= START;
       S2: nextstate <= START;
```

```
        S3: nextstate <= SET;
    endcase
end

always @(posedge CLK100MHZ) begin
    case(state)
       SET: begin
                A <= SW[15:8];
                B <= SW[7:0];
            end
        S1: A <= A - B;
        S2: B <= B - A;
        S3: GCD <= A;
    endcase
end
endmodule

module display(
input      [7:0] num,
output reg [7:0] cathodes
);

always @(num) begin
    case(num)
        8'h0:   cathodes = 7'b1111110;
        8'h1:   cathodes = 7'b0110000;
        8'h2:   cathodes = 7'b1101101;
        8'h3:   cathodes = 7'b1111001;

        8'h4:   cathodes = 7'b0110011;
        8'h5:   cathodes = 7'b1011011;
        8'h6:   cathodes = 7'b1011111;
        8'h7:   cathodes = 7'b1110000;

        8'h8:   cathodes = 7'b1111111;
        8'h9:   cathodes = 7'b1111011;
        8'hA:   cathodes = 7'b1110111;
        8'hB:   cathodes = 7'b0011111;

        8'hC:   cathodes = 7'b1001110;
        8'hD:   cathodes = 7'b0111101;
        8'hE:   cathodes = 7'b1001111;
        8'hF:   cathodes = 7'b1000111;
     default:   cathodes = 7'b1111110;
    endcase

    cathodes = {cathodes, 1'b0}; //for decimal point
    cathodes = ~cathodes; //for active low
end
endmodule

module SLOW_CLK(
            input FASTCLK,
            output reg SLOWCLK
```

```
);

reg [31:0] ctr;
initial ctr = 32'h0000_0000;
initial SLOWCLK = 0;

always@(posedge FASTCLK) begin
        ctr = ctr + 32'h0000_0001;
        if(ctr >= 32'h0001_86A0 ) begin
        ctr = 32'h0000_0000;
        SLOWCLK = ~SLOWCLK;
        end
end
endmodule
```

CONSTRAINTS

```
## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];

##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { SW[0] }];
#IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { SW[1] }];
#IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { SW[2] }];
#IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }];
#IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { SW[4] }];
#IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { SW[5] }];
#IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { SW[6] }];
#IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { SW[7] }];
#IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { SW[8] }];
#IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { SW[9] }]; #IO_25_34
Sch=sw[9]
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { SW[10] }];
#IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { SW[11] }];
#IO_L23P_T3_A03_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { SW[12] }];
#IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { SW[13] }];
#IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { SW[14] }];
#IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { SW[15] }];
#IO_L21P_T3_DQS_14 Sch=sw[15]

#7 segment display

set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { CA[7] }];
#IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { CA[6] }]; #IO_25_14
Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { CA[5] }]; #IO_25_15
Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { CA[4] }];
#IO_L17P_T2_A26_15 Sch=cd
```

```
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { CA[3] }];
#IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { CA[2] }];
#IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { CA[1] }];
#IO_L4P_T0_D04_14 Sch=cg

set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { CA[0] }];
#IO_L19N_T3_A21_VREF_15 Sch=dp

set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { AN[0] }];
#IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { AN[1] }];
#IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { AN[2] }];
#IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { AN[3] }];
#IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { AN[4] }];
#IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { AN[5] }];
#IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { AN[6] }];
#IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { AN[7] }];
#IO_L23N_T3_A02_D18_14 Sch=an[7]


##Buttons

#set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports { CPU_RESETN }];
#IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn

set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { BTNC }];
#IO_L9P_T1_DQS_14 Sch=btnc
```

TESTBENCH

```
`timescale 1ns / 1ps

module GCD_tb();
reg [15:0] SW;
reg CLK100MHZ;
reg BTNC;
wire [7:0] CA,AN;
wire [7:0] LED;
wire [7:0] GCD;
wire [2:0] state;

	//GCD uut(SW,CLK100MHZ, BTNC, LED, CA, AN, GCD, state);
	GCD uut(.SW(SW), .CLK100MHZ(CLK100MHZ), .BTNC(BTNC), .GCD(GCD), .state(state));

	initial CLK100MHZ = 0;

	initial begin

	SW[15:0] = 16'h4e42;
	BTNC = 1;

	end

	always
	#5 CLK100MHZ = ~CLK100MHZ;
endmodule
```

REFERENCES

(1) *Ibrahim Hazmi "*Design and Implementation of the Euclidean Algorithm for Computing the Greatest Common Divisor using Xilinx Spartan6 FPGA" https://www.researchgate.net/publication/283142320_Design_and_Implementation_of_the_Euclidean_Algorithm_for_Computing_the_Greatest_Common_Divisor_using_Xilinx_Spartan6_FPGA

TERMINATING SHEET

Bobby Work	**Bruno Work**
State assignment block	State assignment block
State outputs block	State update Block
State update block	SLOW_CLK module
7-segment display block	display module
Problem, conclusion section report	Abstract,Problem section report
Solution step3,step4,step5	Solution step1,step2,step3 section report
Block Diagram Internal	Block Diagram External

Hours Spent on Project	12
Coding	4
Troubleshooting	1
Report	7