

University of Arkansas

College of Engineering

Department of Electrical Engineering and Computer Science

ELEG 29004, Lab Section: 002

Implementing a Greatest Common Divisor in VHDL

Submitted by

Juan Campos

ID: 100470890

Date Submitted:	04/30/2026
Date Performed:	04/29/2026
Lab Section:	L002
Course Instructor:	Chad Workman
Lab TA:	Robert Russell

Implementing a Greatest Common Divisor in VHDL

ELEG 29004

by

Juan Campos

Abstract

In this lab, a digital circuit was designed and implemented to calculate the greatest common divisor (GCD) of two 8-bit binary numbers using Euclid's subtraction algorithm. The digital system was divided into two parts: the data path and the finite state machine (FSM). The data path managed the operations of storing, comparing, and subtracting the values. And the FSM performed the necessary sequence of actions needed to apply Euclid's algorithm correctly. The design was implemented using VHDL, and the correct functionality of the circuit was first tested using a simulation testbench. Once the results of the simulation testbench were correct, the implementation was tested on a Boolean Board. On the Boolean Board, the numbers 160 and 128 were introduced, and the Boolean Board correctly displayed the result of 32. This lab demonstrated how combinational and sequential logical elements work together and demonstrated the practical application of FSMs.

Introduction

Digital systems are built not only using combinational logic elements, where the input determines the output. Digital systems are also built using sequential logic elements, where the output depends on the inputs and past outputs. Flip-flops, registers, and finite state machines (FSM) are examples of sequential logical elements. For example, in the design of a calculator. A calculator may need to perform calculations, make decisions, and store values. This can be accomplished using sequential logical elements. In this lab, the two types of logic were combined to make a hardware system that calculates the greatest common divisor, or GCD, of two 8-bit binary numbers.

“The greatest common divisor (GCD) of integers a and b , at least one of which is nonzero, is the greatest positive integer d such that d is a divisor of both a and b [1].” There are different ways to find this value; in this lab, the subtraction-based version of Euclid’s algorithm was implemented. This algorithm starts with two values, A and B . If A and B are not equal, the smaller value is subtracted from the larger one. Then the larger value is replaced by the result of the subtraction. This process continues until A and B become equal [2]. Once both values are equal, the value of GCD is found, and it can be displayed.

The design of the GCD was implemented using a data path and a finite state machine in VHSIC Hardware Description Language, or more commonly named VHDL. The data path detailed the different components needed to perform the calculations. Some of the components are registers to store the values of A and B , muxes to choose between values, a comparator to determine which value is larger, a ripple carry adder to perform the calculations, and a system to display the data on the displays.

The FSM controlled the operations of the data path by activating the correct control signals at the determined state of the machine. For example, in state zero, the FSM begins by waiting for the start signal. Once the signal is received, the data is loaded into the registers.

The concepts used in this lab are important in the industry. Many digital systems are designed using the same approach, dividing the data path and the control logic. A finite state machine can be used to control what happens at different moments of the circuit or at different times or clock cycles. This functionality is similar to how calculators, traffic lights, and vending machines work. While the data path controls when the operations happen, the data path is the one doing the operations or the calculations.

The purpose of this lab was to design, implement, and test a GCD calculator. The report describes the data path components, the FSM states, the simulation results, and the hardware verification in the Boolean Board. The expected results were that the digital system would apply Euclid's algorithm and display the right GCD on the seven-segment display.

Experimental Procedure

The greatest common divisor design consists of two parts: the first one is the data path, and the second one is the finite state machine. The data path describes how the data is stored, what data to compare, what data to subtract, and how to display the data on the displays. The finite state machine controlled when each operation occurred based on the input data and past stored data.

The first step was to review the provided top-level GCD VHDL module. The provided module defined the main interface of the design. It included the inputs A(7:0), B(7:0), start, clk, and reset. And the outputs Dis(6:0) and DisplayEN(3:0). Those outputs will be used for the seven-segment display.

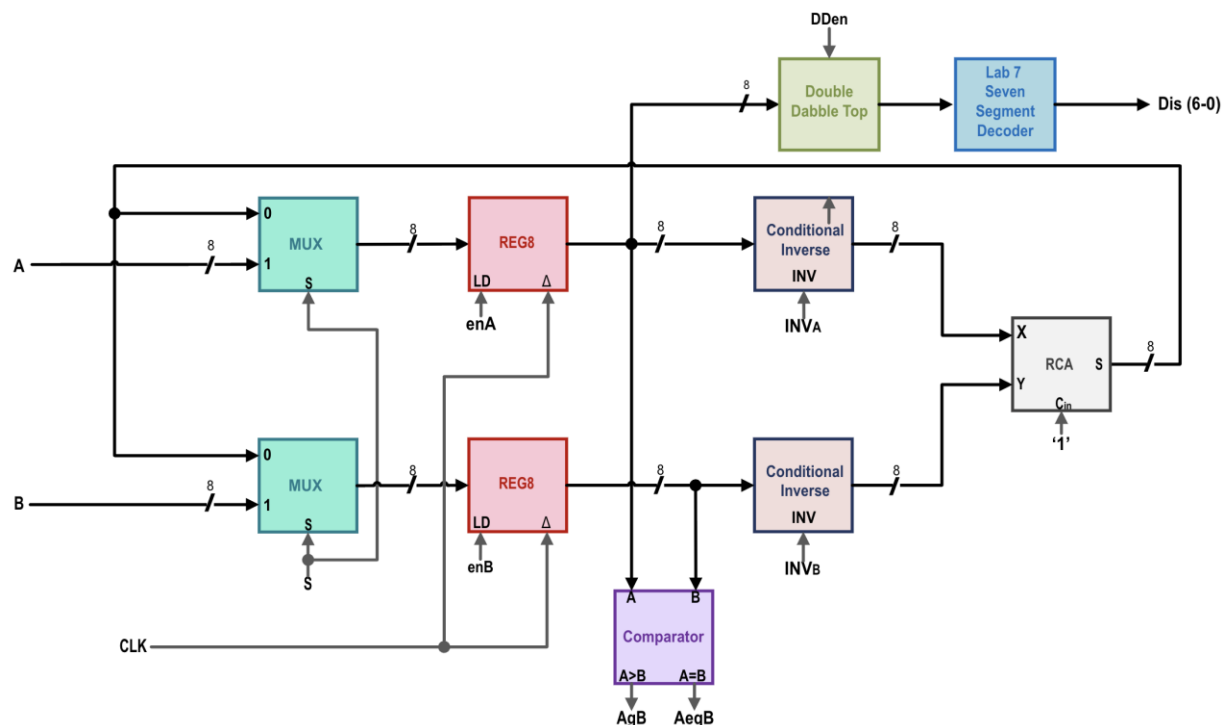


Figure 1: Data path of the circuit.

The data path shown above in Figure 1 was also provided, as well as the Double Dabble Top component. The next step was to implement the data path and create or export the rest of the components.

The registers were used to store the values of A and B when the enable signal (enA and enB) was received. The comparator was used to determine if A and B were equal or not. If A was greater than B it returned the signal AgB. If A were equal to B it returned the AeqB signal. The conditional inverse component inverted the bits received if the signal INVa was high. The ripple carry adder, or RCA, was used to perform the subtraction. Because A and B are 8-bit binary numbers, to perform the subtraction, two's complements was used. For two's complement, the bits of one of the numbers were inverted. This was done by setting INVa or INVb signal high and receiving that signal in the conditional inverse component. The next component was the mux. Muxes were used to select whether the register stored the original values or the values from the RCA, the signal to control this was the Sel. The double dabble top component converted the unsigned 8-bit binary number to an appropriate binary combination to display in the seven-segment display. Finally, the seven-segment decoder was in charge of turning the defined LEDs inside the seven-segment display.

Once all the components were declared, the connections between them were made using port maps. This was made by declaring new signals. Those signals worked as wires, connecting every component as shown in Figure 1. The signals were declared appropriately for the kind of signal they were going to carry. For example, the signal connecting the muxes and the registers was an 8-bit, in contrast, the signals anA and INa were 1-bit signals.

The next step was to implement the Finite State Machine shown in Figure 2.

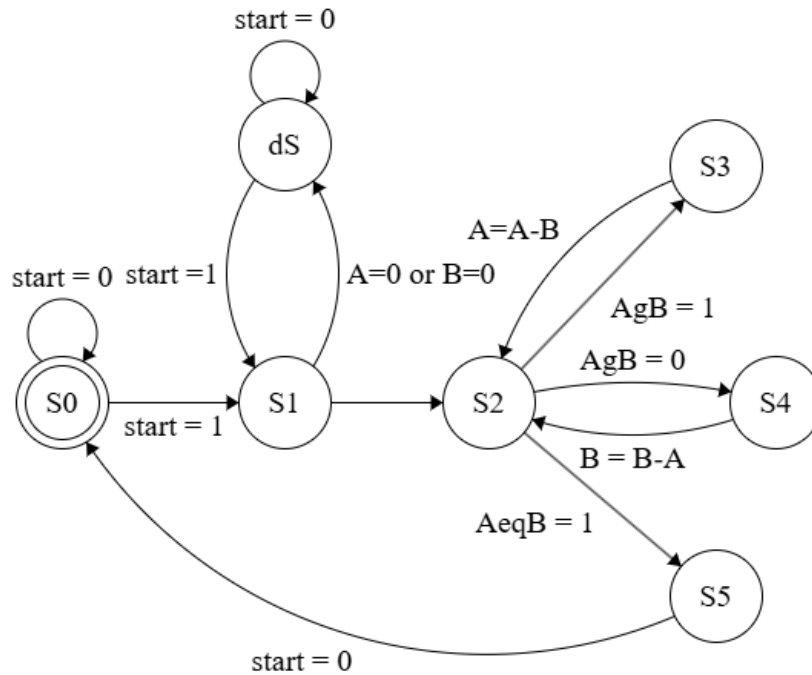


Figure 2: Moore Finite State Machine

The GCD controller was implemented using a Moore finite state machine because the control output depends only on the current state.

The FSM began in state S0 or idle. In this state, it was waiting for the start input signal. Once the start signal was high, the next state was S1 or load.

In state S1, the muxes were set to pass the original values of A and B. This was done by setting the signal Sel high. The registers were enabled to store the data coming from the muxes. After that, the next state was S2, or the compare state.

In state S2, the values of A and B were compared. If A was greater than B, the FSM moved to state S3, which enabled the calculation of $A - B$ and then stored the value in register A. To do this, inverter B signal, INVb, and register A signal, enA, were set to high. The mux was switched to let the data coming from the RCA pass to register A.

If B was greater than A, the FSM moved to the state S4 to calculate $B - A$ and then stored the result in register B. To do this, the inverter A signal, INVa, and register B signal, enB, were set to high. The mux was switched to let the data coming from the RCA pass to register B. The FSM repeated this cycle to compare and subtract until both A and B were equal.

If $A = B$, then the FSM moved to state S5. In this state, the display was enabled, which was done by setting DDen signal to high, which allowed the data from register A that passed to the double dabble top component to be displayed on the seven-segment displays.

Bonus Section

The Euclid's algorithm implemented in this lab used subtraction. The smaller number is subtracted from the greater one. The implementation described above assumes that both input values are greater than zero. But if A or B is zero, this becomes an infinite loop. To fix this problem, the data path and the FSM were modified. A new state was added, one signal was modified, and the seven-segment decoder was also modified to show an error message on the displays.

The first thing modified was state S1. Before doing any calculation, an if statement was implemented. If either input A or B was a zero, the next state was dS. In this new state, dS, the FSM just stayed in the same state until the start signal was set to high again.

In the data path, the signal that connects the output of the double dabble top and the seven-segment decoder was modified. In this signal and if statement was added. If the current state is dS then a 16 bit signal was sent to the decoder. In any other state, the signal remained the same, just connecting the two components.

In the decoder, two new signal cases were added. Table A1 shows the inputs and outputs of the decoder. Those new signals made a sad face, meaning something was wrong in the inputs, as shown in Figure 3.

Figure 3 shows the error message when one of the inputs is zero. The error message consists of a zero initially, meaning that a zero was introduced, followed by a sad face, meaning that it is an error.

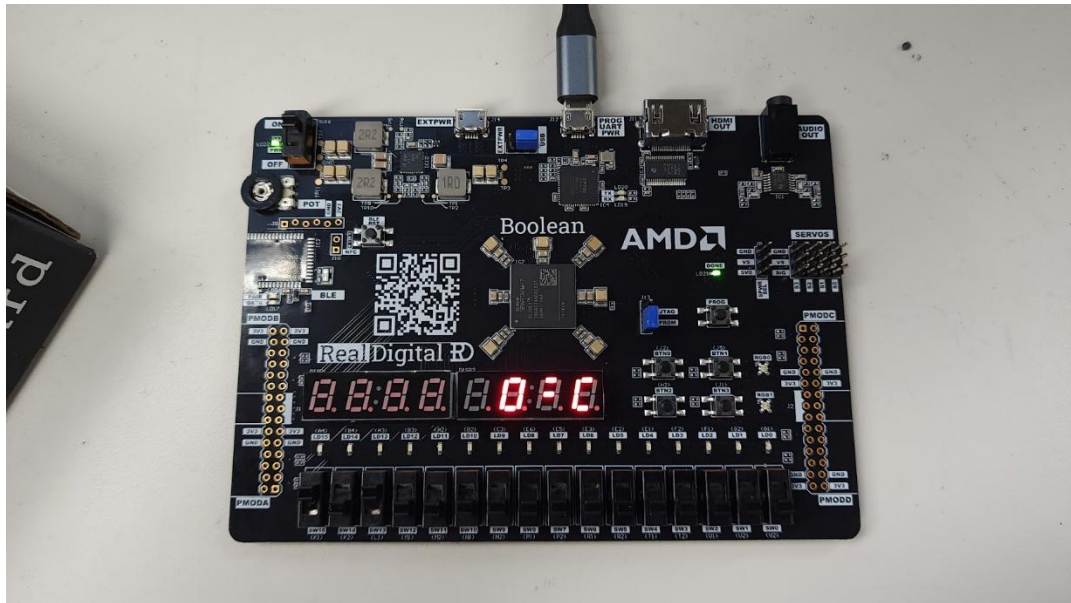


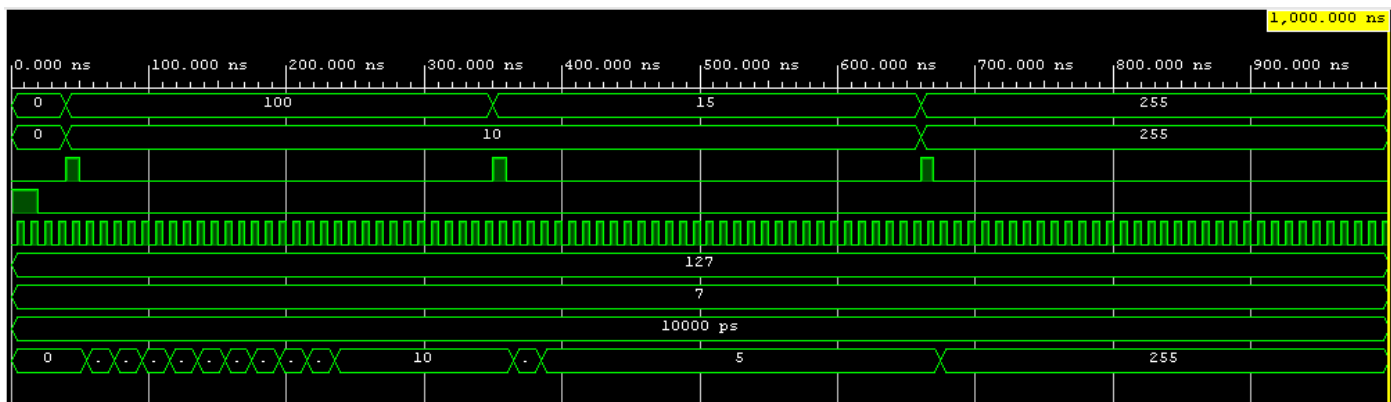
Figure 3: Output error when A or B was zero. The first digit “0” means a zero was introduced. The next two digits make a sad face “:(”.

Results & Discussion

Before testing the VHDL code on the Boolean Board, the circuit was first tested using the provided simulation testbench. The purpose of the testbench was to verify that the data path and the components, as well as the finite state machine, worked correctly before using the Boolean Board. The test bench had the values of A and B shown in Table 1.

Table 1: Values of A and B in binary and decimal.

Value of A		Value of B		GCD
Binary	Decimal	Binary	Decimal	Decimal
00000000	0	00000000	0	
01100100	100	00001010	10	10
00001111	15	00001010	10	5
11111111	255	11111111	255	255



The testbench provided was set to show the values of A, B, start, rst, clk, Dis, and Disp_en. As shown in Figures 4, all the results were exactly expected. However, the waveform of Dis, which is the waveform of the data that will be displayed in the seven-segment displays, is a continuous waveform. This was because of the way the seven-segment display works. The work involves switching on and off the LEDs inside the segments very fast. In addition to that, the resulting waveform doesn't actually mean a number; it's just a sequence of binary to turn on and off the LEDs. To actually see the data that is being output, the output waveform of register A was added. This last waveform showed the GCD of the two input numbers, as shown in Figure 4.

The results of the simulation matched the expected sequence. After the inputs were loaded into the registers, the comparator detected which number was greater or if the numbers were equal, and then decided whether to show the value of GCD or to continue the arithmetic operations. This is shown in Figure 5, which is a zoomed-in picture of the waveform shown in Figure 4. In this picture, the output signal of register A is shown, and according to the data path, this output signal is the one displayed when both A and B are equal. Therefore, that will be the number displayed on the seven-segment displays.

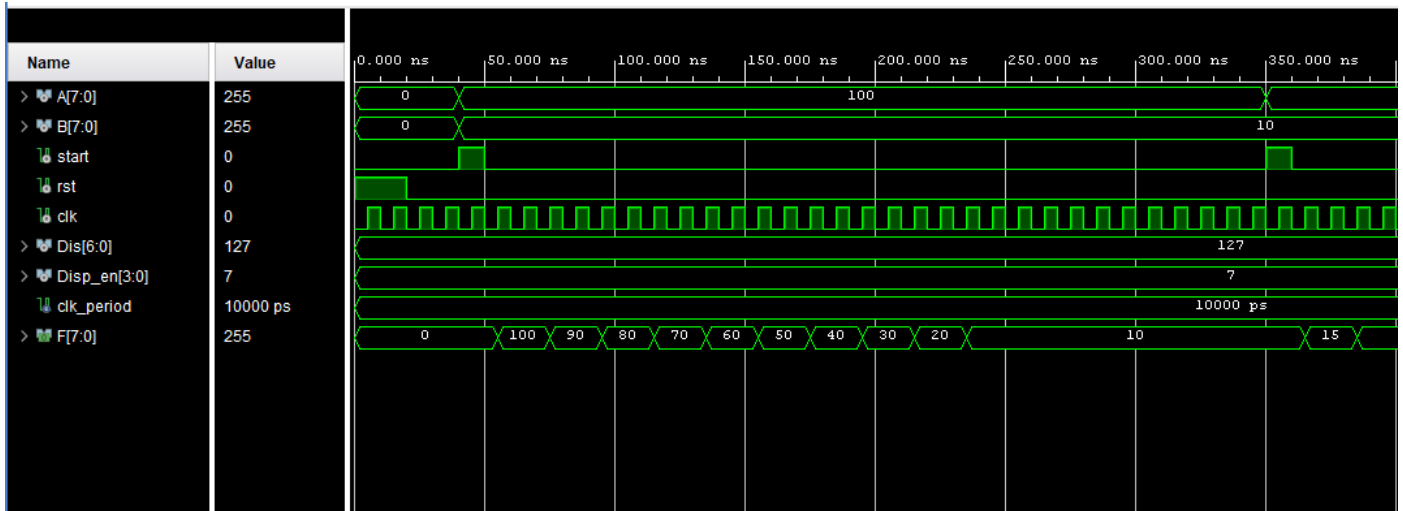


Figure 5: Waveform of the output of register A.

After the testbench results were as expected, the next step was to load the code into actual hardware. The hardware used was a Boolean Board. The board was programmed with the generated bitstream, and using the input switches, the values of $A = 160$ and $B = 128$ were introduced, as shown in Figure XX. This was done in binary. The switches represented binary, where from 0 to 7 was input A and from 8 to 15 was input B. After pressing the start button, the Boolean Board calculated the GCD and displayed the result on the seven-segment display. The expected result was 32, and the result obtained was 32, confirming that the hardware implementation was right.

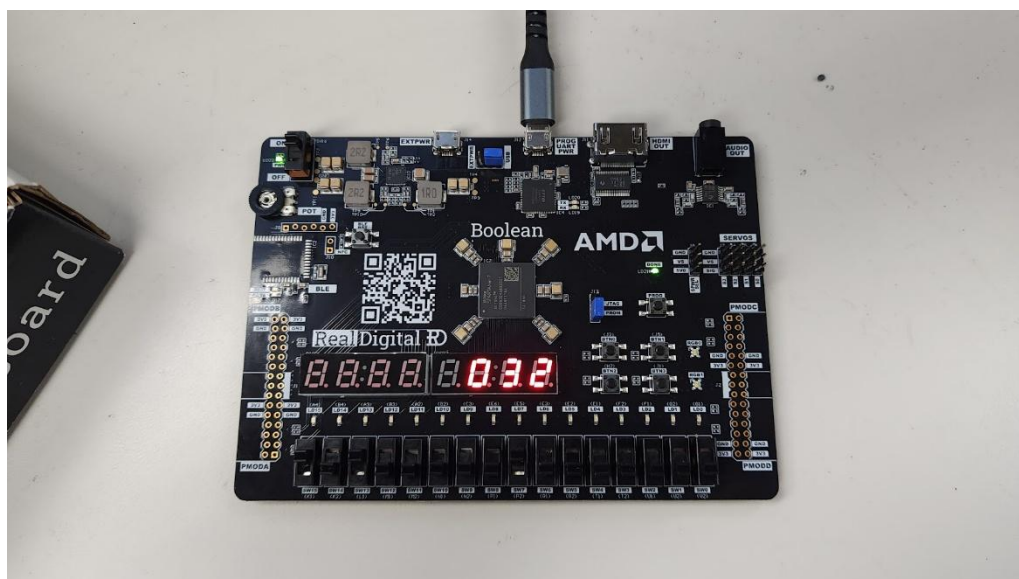


Figure 6: Boolean Board with the values of 160 and 128.

Overall, the simulation and the Boolean Board results showed that the GCD circuit worked as expected. The FSM correctly controlled the data path by turning on and off signals. And the data path correctly did the calculations and the storage of the data.

Conclusions

The purpose of this lab was to implement and test Euclid's algorithm to perform the necessary calculations to get the greatest common divisor of two 8-bit binary numbers. This was completed using a data path controlled by a Moore finite state machine. The data path managed storage, comparison, subtraction of two input values, and display of the greatest common divisor. The finite state machine performed the necessary sequence of actions needed to apply Euclid's algorithm correctly.

The waveform from the testbench and the Boolean Board successfully verified the correct implementation of the subtraction-based GCD process. It started loading the input values into the registers, then comparing the values, and after that, subtracting the corresponding value until both A and B were equal.

In summary, the resulting digital system matched the expected behavior of Euclid's algorithm. And that was made possible by using combinational logic components, such as comparators, muxes, ripple carry adders, and sequential logic elements, such as registers.

References

- [1] Wikipedia contributors, "Greatest common divisor," Wikipedia, The Free Encyclopedia, [Online]. Available: https://en.wikipedia.org/wiki/Greatest_common_divisor.
- [2] Wikipedia contributors, "Euclidean algorithm," Wikipedia, The Free Encyclopedia, [Online]. Available: https://en.wikipedia.org/wiki/Euclidean_algorithm.

Appendix A

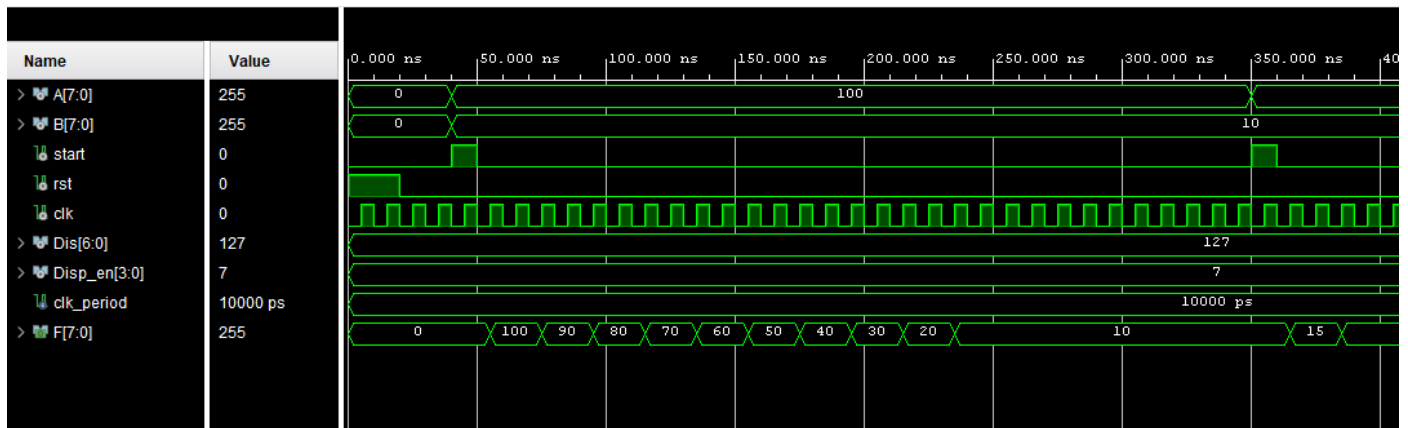


Figure A 1: Waveform of testbench with inputs A = 100 and B = 10.

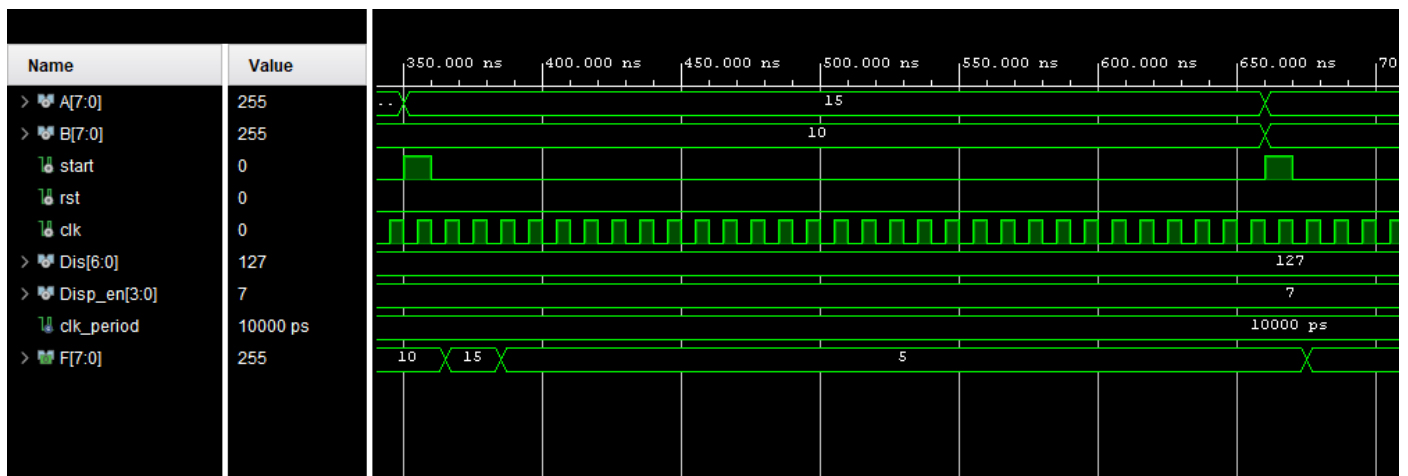


Figure A 2: Waveform of testbench with inputs A = 15 and B = 10.

Table A 1: Encoder values

Input A	Output F	Displayed value
0000	1000000	0
0001	1111001	1
0010	0100100	2
0011	0110000	3
0100	0011001	4
0101	0010010	5
0110	0000010	6
0111	1111000	7
1000	0000000	8
1001	0011000	9
1010	0111110	-
1011	1000110	(
others	1111111	Off

Name	Value
> A[7:0]	255
> B[7:0]	255
start	0
rst	0
clk	0
> Dis[6:0]	127
> Disp_en[3:0]	7
clk_period	10000 ps
> F[7:0]	255

Figure A 3: Signals analyzed and shown in the waveform in Figure 5.

Table A 2: States in VHDL and what they mean.

State	Description
S0	Idle
S1	Load
S2	Compare
S3	A greater than B
S4	B grater than A
S5	Display GCD
dS	Dummy State for error handling

GCD_top VHDL code:

```

-----
--Juan Campos
--Kevin Durden
--DIGITAL DESIGN
--2026
-----

Library IEEE;
use IEEE.std_logic_1164.all;

entity GCD_top is
    port(
        A,B: in std_logic_vector(7 downto 0);
        start, rst, clk: in STD_LOGIC; --RESET is active high
        Dis: out std_logic_vector(6 downto 0);
        Disp_en : out STD_LOGIC_VECTOR (3 downto 0)
    );
end;

architecture BEHAVIOR of GCD_top is
--Component Declarations

    --Add all of the components you will need for the GCD

    component DoubleDabbleTop is --added Double Dabble already for your convience
    Port (
        A : in std_logic_vector(7 downto 0);
        clk : in std_logic;
        start : in std_logic;
        rst : in std_logic;
        BCDDispData : out std_logic_vector(9 downto 0)
    );
    end component;

```

component Comparitor is

```
Port (  
  A: in std_logic_vector(7 downto 0);  
  B: in std_logic_vector(7 downto 0);  
  AeqB: out std_logic;  
  AgB: out std_logic  
);  
end component;
```

component Mux_2to1 is

```
port(  
  A, B: in STD_LOGIC_VECTOR(7 downto 0);  
  S: in STD_LOGIC;  
  F: out STD_LOGIC_VECTOR(7 downto 0));  
end component;
```

component Reg8 is

```
Port ( A: in std_logic_vector(7 downto 0);  
  en: in std_logic;  
  clk: in std_logic;  
  rst: in std_logic;  
  F : out std_logic_vector(7 downto 0));  
end component;
```

component ConInv is

```
Port (  
  A: in STD_LOGIC_VECTOR(7 downto 0);  
  Inv: in STD_LOGIC;  
  F: out STD_LOGIC_VECTOR(7 downto 0));  
end component;
```

component RCA8 is

```
Port(  
  A : in STD_LOGIC_VECTOR(7 downto 0);  
  B : in STD_LOGIC_VECTOR(7 downto 0);  
  Sub : in STD_LOGIC;  
  S : out STD_LOGIC_VECTOR(7 downto 0));
```

end component;

component Seven_Segment_Top is

```
Port (  
  Data : in STD_LOGIC_VECTOR (15 downto 0);  
  clk : in STD_LOGIC;  
  rst : in STD_LOGIC;  
  Disp_en : out STD_LOGIC_VECTOR (3 downto 0);
```

```

    Seven_Segment_out : out STD_LOGIC_VECTOR (6 downto 0));
end component;

```

```

-- FSM Signal Declarations

```

```

    type state_type is (S0, S1, S2, S3, S4 , S5, dS); --Add more states if needed.

```

```

    -- signal for states

```

```

    signal CS, NS: state_type;

```

```

--Control signals used in the FSM to interact with the datapath

```

```

    signal sel, enA, enB, AgB, AeqB, invA, invB, DDen: std_logic;

```

```

--Signal Declarations for port mapping internal signals in your datapath

```

```

    -- Write them yourself!

```

```

    signal inputA : std_logic_vector(7 downto 0);

```

```

    signal inputB : std_logic_vector(7 downto 0);

```

```

    signal RCAtoMux : std_logic_vector(7 downto 0);

```

```

    signal MuxAtoReg8A : std_logic_vector(7 downto 0);

```

```

    signal MuxBtoReg8B : std_logic_vector(7 downto 0);

```

```

    signal Reg8AtoCondInvA : std_logic_vector(7 downto 0);

```

```

    signal Reg8BtoCondInvB : std_logic_vector(7 downto 0);

```

```

    signal InvAtoRCA : std_logic_vector(7 downto 0);

```

```

    signal InvBtoRCA : std_logic_vector(7 downto 0);

```

```

    signal DDTtoDecoder : std_logic_vector(9 downto 0);

```

```

    signal tempsig : std_logic_vector(15 downto 0);

```

```

    signal error : std_logic;

```

```

begin

```

```

-----
--MUXs
-----

```

```

    MUX1: Mux_2to1

```

```

        port map(

```

```

        A => RCAtoMux,

```

```

        B => A,

```

```

        S => sel,

```

```

        F => MuxAtoReg8A

```

```

        );

```

```

    MUX2: Mux_2to1

```

```

        port map(

```

```

        A => RCAtoMux,

```

```

        B => B,

```

```

        S => sel,

```

```

        F => MuxBtoReg8B
    );
-----
-- Registers
-----
REG81: Reg8
    port map(
        A => MuxAtoReg8A,
        en => enA,
        clk => clk,
        rst => rst,
        F => Reg8AtoCondInvA
    );

REG82: Reg8
    port map(
        A => MuxBtoReg8B,
        en => enB,
        clk => clk,
        rst => rst,
        F => Reg8BtoCondInvB
    );
-----
-- Comparator
-----
    Comparator: Comparitor
        port map(
            A => Reg8AtoCondInvA,
            B => Reg8BtoCondInvB,
            AeqB => AeqB,
            AgB => AgB
        );
-----
-- Conditional Inverse
-----
    InverterA: ConInv
        port map(
            A => Reg8AtoCondInvA,
            Inv => INVa,
            F => InvAtoRCA
        );

    InverterB: ConInv
        port map(
            A => Reg8BtoCondInvB,

```

```

        Inv => INVb,
        F => InvBtoRCA
    );
-----
-- Ripple carry adder
-----

    RippleCA: RCA8
    port map(
        A => InvAtoRCA,
        B => InvBtoRCA,
        Sub => '1',
        S => RCAtoMux
    );
-----
-- Double Dabble
-----
    DD : DoubleDabbleTop --Example from class. Shows how to structure FSM with datapath. Feel
free to modify!
    port map(
        A => Reg8AtoCondInvA, --output from one of the registers
        clk => clk,
        start => DDen,
        rst => rst,
        BCDDispData => DDTtoDecoder --output from Double Dabble. Should connect to the 7
segment display decode driver from lab 7.
    );
-----
-- Double Dabble
-----

tempsig <= "1111000010101011" when CS = dS else "111100" & DDTtoDecoder;

display : Seven_Segment_Top
    port map(
        Data => tempsig,
        clk=> clk,
        rst=> rst,
        Disp_en => Disp_en,
        Seven_Segment_out => Dis
    );
-----
-- FSM Control
-----

    comb: process(CS, start, AeqB, AgB)

```

```

begin
    sel <= '0';
    enA <= '0';
    enB <= '0';
    invA <= '0';
    invB <= '0';
    DDen <= '0';
    NS <= CS;
    error <= '0';

    case CS is
        --idle
        when S0 =>
            if start = '1' then
                NS <= S1;
            else
                NS <= S0;
            end if;

        --load
        when S1 =>
            sel <= '1'; -- load external A and B
            enA <= '1';
            enB <= '1';
            if A = "00000000" then
                NS <= dS;

            elsif B = "00000000" then
                NS <= dS;
            else

                NS <= S2;
            end if;

        --compare
        when S2 =>

            if AeqB = '1' then
                NS <= S5;
            elsif AgB = '1' then
                NS <= S3;
            else
                NS <= S4;
            end if;

        --A greater than B
        when S3 =>

```

```

    invB <= '1';
    enA <= '1';
    sel <= '0'; -- load RCA result into A
    NS <= S2;

--B greater than A
when S4 =>
    invA <= '1';
    enB <= '1';
    sel <= '0'; -- load RCA result into B
    NS <= S2;

--A equal B
when S5 =>
    DDen <= '1';
    if start = '0' then
        NS <= S0;
    else
        NS <= S5;
    end if;

--Inputs zero
when dS =>
    --DDTtoDecoder <= "0010101011";
    if start = '1' then
        NS <= S1;
    else
        NS <= dS;
    end if;

when others =>
    NS <= S0;
end case;
end process;

-- Update Current State based on Next State Logic
sync: process(clk, rst)
begin
    if rst = '1' then
        CS <= S0;
    elsif rising_edge(clk) then
        CS <= NS;
    end if;
end process;
end;

```