Bilkent University

Electrical and Electronics Department

EE102-01 Lab 6 Report:

“Greatest Common Divisor”

13/11/2023

Fatih Mehmet Çetin - 22201689

Purpose:

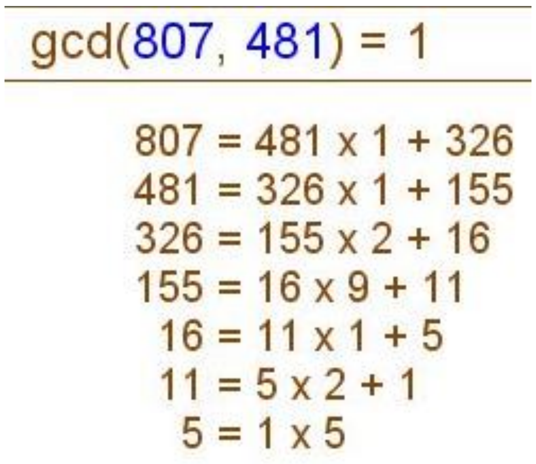
The main purpose of the experiment was to design a circuit that was capable of calculating the greatest common divisor of two 8-bit unsigned binary numbers. The main focus of this lab was to understand how registers work and how to implement them on VHDL.

Methodology:

Question 1: What was your algorithm to calculate the GCD?

There are many algorithms for finding the greatest common divisor of two numbers such as the Euclidean Algorithm, Extended Euclidean Algorithm, Binary GCD Algorithm, Lehmer’s Algorithm etc. In this specific project, the Euclidean Algorithm will be used in order to calculate the GCD.

Here is how it works: First, you compare the two numbers; if they are equal, the GCD is simply the input numbers. If they are not equal, find the larger number and keep subtracting the smaller number from the larger one and keep updating the larger number until the result becomes smaller from the the initial smaller number. Then, keep on this procedure until the two updated numbers are equal to each other. Once they are equal to each other, they become the GCD. Here is an example of the Euclidean Algorithm **(Figure 1)**:



$$\gcd(807, 481) = 1$$

$$\begin{aligned}
807 &= 481 \times 1 + 326 \\
481 &= 326 \times 1 + 155 \\
326 &= 155 \times 2 + 16 \\
155 &= 16 \times 9 + 11 \\
16 &= 11 \times 1 + 5 \\
11 &= 5 \times 2 + 1 \\
5 &= 1 \times 5
\end{aligned}$$

Figure 1: The Euclidean Procedure to find the GCD of 807 and 481

Question 2.1: Is your module a combinational circuit or an FSM?

A combinatorial circuit is basically the circuit where the output only depends on the current input. A combinatorial circuit has no memory element and is usually faster. An FSM (Finite State Machine) on the other hand, is the circuit where the output depends on both the inputs and the current state of the system. An FSM contains memory elements and due to the more complex operations such as the transitions between states it contains, the delay is a significant drawback compared to combinatorial circuits.

The module I designed in this project is an FSM since it contains a bunch of different states and memory elements.

Question 2.2: If the latter, is it a Moore machine or a Mealy machine?

A Moore machine is the type of FSM machine where the outputs are solely determined by the current state. It does not depend directly on the input. This means that the output only changes when the machine transitions to a different state. A Mealy machine, on the other hand, is another type of FSM where the output is determined by both the current state and the current input. There is a more immediate response to the inputs in Mealy Machines, since the output can change without any change in the current state.

The module I designed contains both the elements of a Moore machine and a Mealy Machine. In the design, changing the number inputs after the “GO” button has been pressed does not change the 8-bit binary output. The displayed GCD is the GCD of the initial two 8-bit numbers. Whereas, pressing the asynchronous reset input change the output of the system to zero independent of the current state. Therefore, the module I designed is a combination of them both.

Question 2.3: Would it be cheaper to implement GCD as a combinational circuit or as an FSM? Would it be faster? What are the drawbacks in each case?

The FSM design is definitely cheaper than a combinatorial model since the combinatorial model would need many numbers of logic gates. Nevertheless, since it is dependent on the clock, an FSM model is slower than a combinatorial model. There are advantages and disadvantages to them both.

Question 3: How many clock cycles did it take in your simulation to calculate the GCD? Do you think this can be optimized? How so?

The number of clock cycles for the design to find the GCD depends on the number of transitions between states and therefore, the input numbers. When the numbers were 140 and 12, it took 42 clock cycles for the design to find the GCD. This number can be optimized if we were able to decrease the number of states the design goes through in each procedure.

Design Specifications:

In the design, there are 5 inputs and a single output. The inputs are the "GO" input, "RESET" input, the clock input and the two 8-bit number inputs. The output is basically the GCD output. In the design we have a multiplexer and a register sub-module. Also, a states sub-module and a datapath component is used in the project.

The mux is used to assign and change the register inputs according to the select input it was given. 2 multiplexers have been used for the two numbers in the design.

The register is also another key component of the design. There are 2 registers for two number inputs and another one for the GCD output which makes the total number of registers 3. The registers change their outputs according to the state they are currently in.

There are seven states in the states sub-module. The start state is the initial state where the "GO" input is waited for the whole process to start. The input state's only job is to create an enough delay time for registers and multiplexers. The test and update states are the states where we implement the Euclidean algorithm to the design. The algorithm is completed when the two updated numbers are equal to each other.

Here you can see the the RTL Schematics of the design **(Figures 2.1, 2.2, 2.3)**:

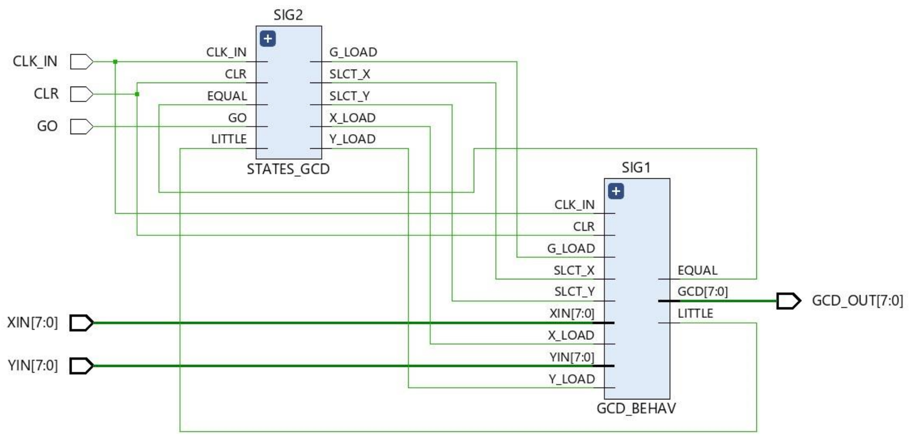


Figure 2.1: The RTL Schematic of the Design

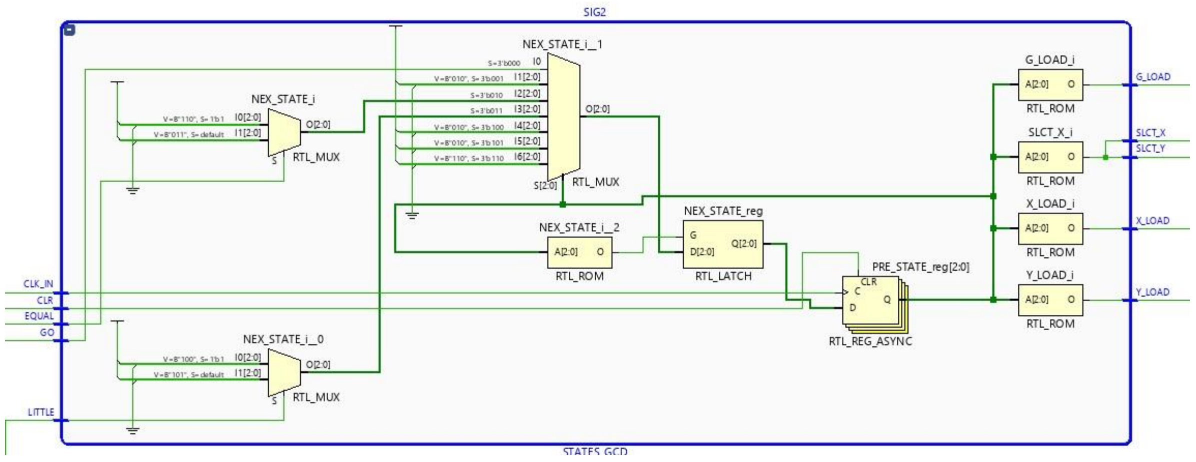


Figure 2.2: The RTL Schematic of "STATES_GCD.vhd"

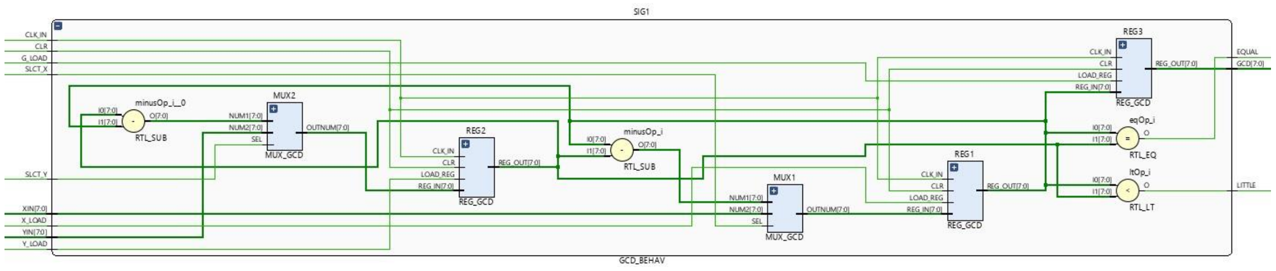


Figure 2.3: The RTL Schematic of "GCD_BEHAV.vhd"

Results:

After completing the design step, a testbench code was written in order to simulate our findings. The first number was set to 140 and the second one was set to 12. It was observed that the Euclidean algorithm was coded and implemented correctly, and the results were consistent. Here is the waveform simulation of the design **(Figure 3)**:

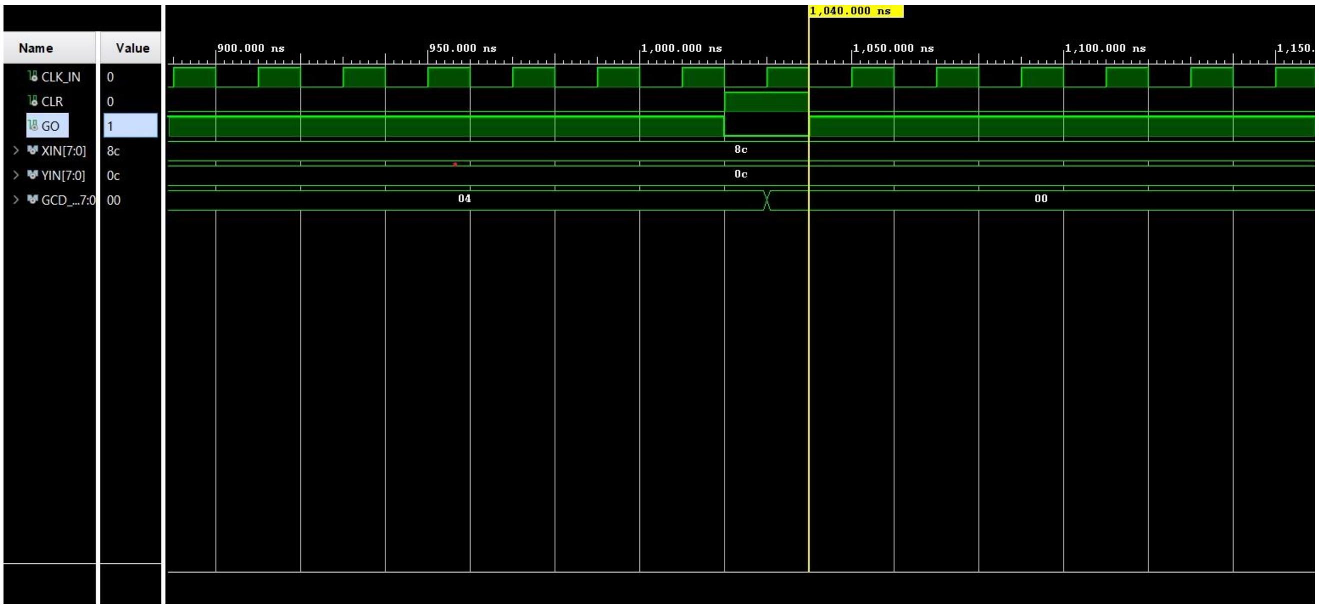


Figure 3: The Waveform Simulation of ”GCD_sim.vhd”

Then, a constraints file was written and a bitstream code was generated and the design was implemented on BASYS3. Here you can see the example input&output combinations **(Figures 4.1, 4.2, 4.3, 4.4)**:

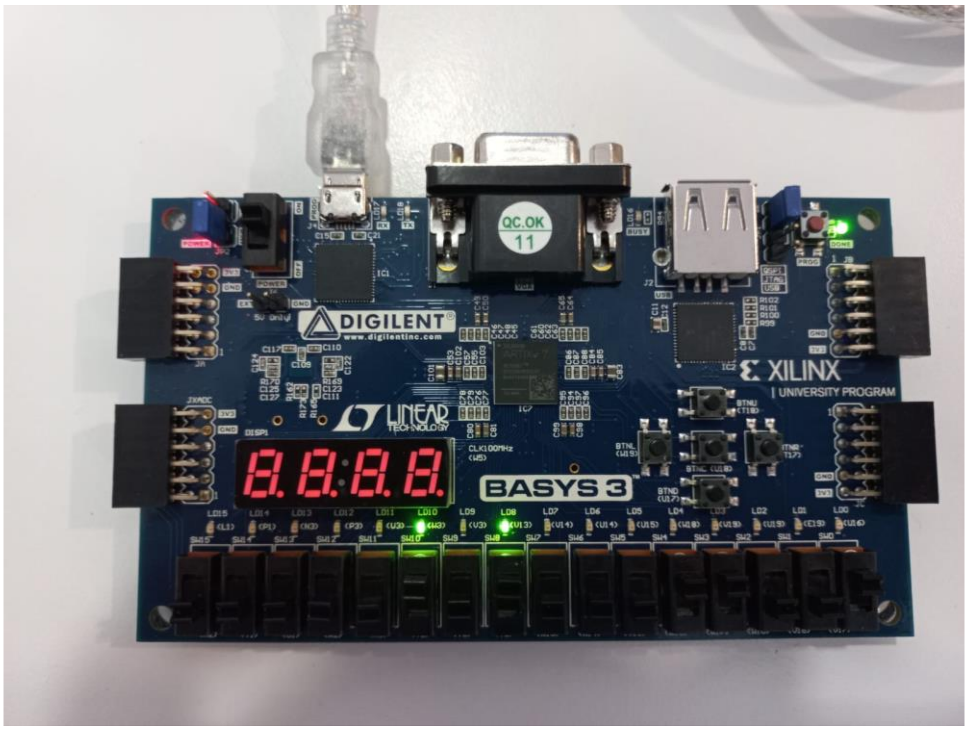


Figure 4.1: The Output LED's when right after "GO" button has been pressed when num1=5, num2=25

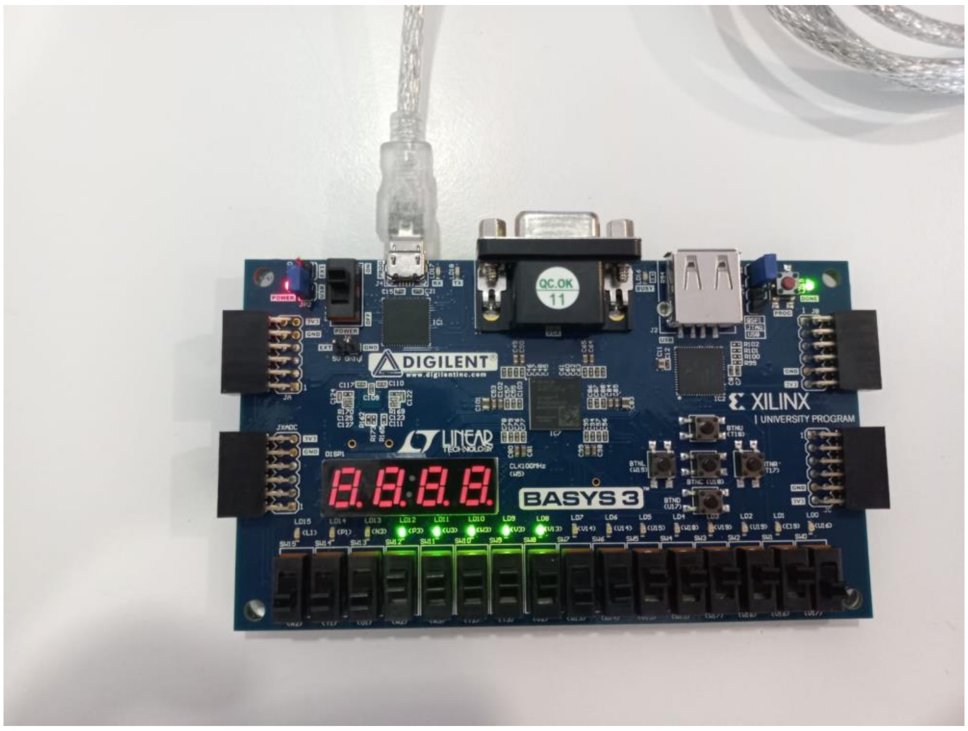


Figure 4.2: The Output LED's when right after "GO" button has been pressed when num1=31, num2=62

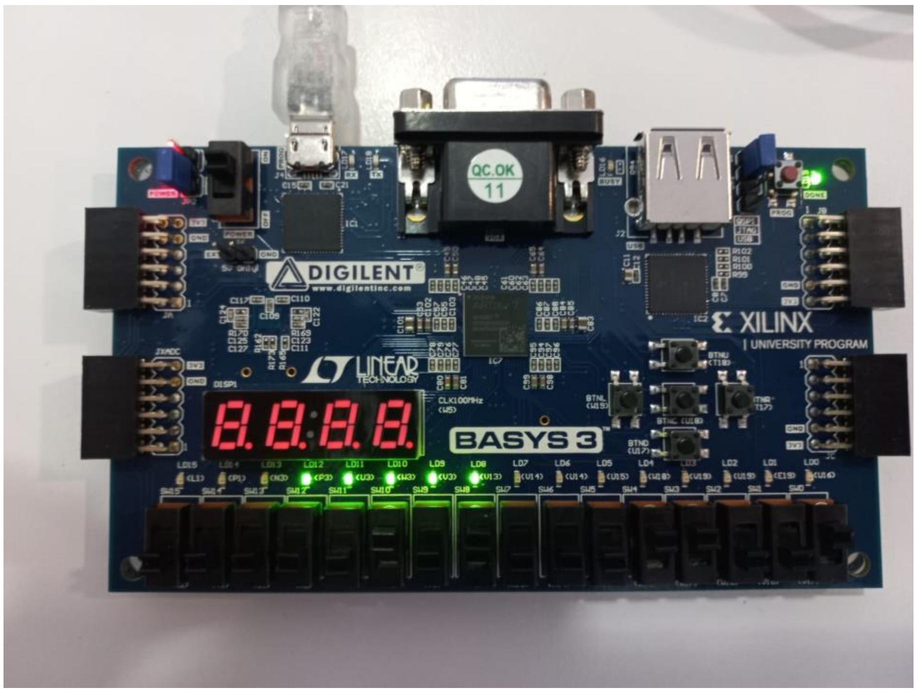


Figure 4.2: The Output LED's when the number inputs have been changed to 5 and 25 but no "RESET" or "GO" button have been pressed since the last photo

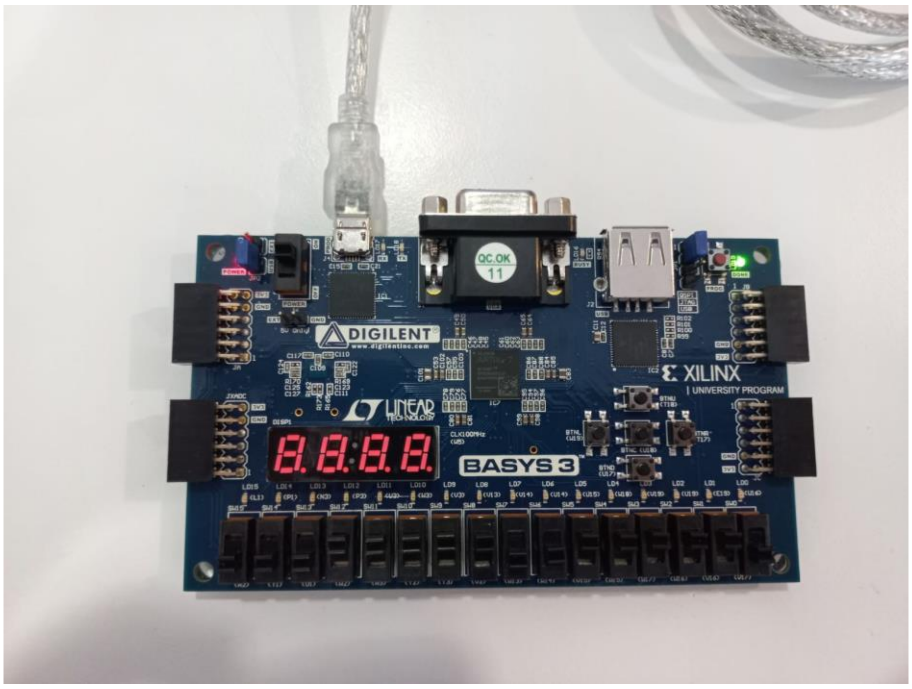


Figure 4.2: The Output LED's when right after "RESET" button has been pressed when num1=31, num2=62

Conclusion:

The purpose of this lab was to design a circuit that finds the GCD of two 8-bit numbers. Initially, I struggled with coming up with an algorithm for the calculation of GCD and then looked up to several resources on the internet. Then I decided to use the Euclidean algorithm.

This lab was pretty significant in order to understand how to use the registers and memory elements in VHDL projects in general. I believe this lab will help me in my term project a lot.

Appendices:

MUX_GCD.vhd:

```
 library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;

entity MUX_GCD is
port(
   NUM1: in std_logic_vector(7 downto 0);
   NUM2:in std_logic_vector(7 downto 0);
   SEL: in std_logic;
   OUTNUM:out std_logic_vector(7 downto 0));
end MUX_GCD;

architecture MUX_GCD of MUX_GCD is

begin
```

```
with SEL select OUTNUM<=
   NUM1 when '0',
   NUM2 when others;
end MUX_GCD;
```

REG_GCD.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity REG_GCD is
port(
   LOAD_REG: in std_logic;
   REG_IN: in std_logic_vector(7 downto 0);
   CLOCK:in std_logic;
   RESET: in std_logic;
   REG_OUT:out std_logic_vector(7 downto 0));
end REG_GCD;

architecture REG_GCD of REG_GCD is

begin

process(CLOCK)
begin
   if rising_edge(CLOCK) then
      if RESET = '1' then
          REG_OUT<=(others=>'0');
      else
        if LOAD_REG='1' then
            REG_OUT <= REG_IN;
        end if;
      end if;
```

```
    end if;
end process;
end REG_GCD;
```

STATES_GCD.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;

entity STATES_GCD is
Port (
    CLOCK: in std_logic;
    RESET: in std_logic;
    GO:in std_logic;
    EQUAL: in std_logic;
    LITTLE: in std_logic;
    SLCT_X: out std_logic;
    SLCT_Y: out std_logic;
    X_LOAD: out std_logic;
    Y_LOAD: out std_logic;
    G_LOAD: out std_logic);
end STATES_GCD;

architecture STATES_GCD of STATES_GCD is

type state_type is(START, INPUT, TEST1,TEST2,UPDATE1,UPDATE2,DONE);

signal PRE_STATE, NEX_STATE: state_type;

begin
SREG: process(CLOCK, RESET)

begin
```

```
    if RESET = '1' then
        PRE_STATE<=START;
    elsif RISING_EDGE(CLOCK)then
        PRE_STATE<=NEX_STATE;
    end if;
end process;



C1: process(PRE_STATE,GO,EQUAL,LITTLE)
begin

case PRE_STATE   is

when START=>
    if GO='1' then
    NEX_STATE<=INPUT;
    else
    NEX_STATE<=START;
    end if;

when INPUT=>
    NEX_STATE<=TEST1;

when TEST1=>
    if EQUAL = '1' then
    NEX_STATE<=DONE;
    else
    NEX_STATE<=TEST2;
    end if;

when TEST2=>
    if LITTLE='1' then
    NEX_STATE<= UPDATE1;
    else
    NEX_STATE<=UPDATE2;
    end if;

when UPDATE1=>
    NEX_STATE<=TEST1;

when UPDATE2=>
    NEX_STATE<=TEST1;

when DONE=>
    NEX_STATE<=DONE;

When others =>
    NULL;

end case;
end process;



C2: process(PRE_STATE)
begin
X_LOAD<='0';Y_LOAD<='0';G_LOAD<='0';SLCT_X<='0';SLCT_Y<='0';

case PRE_STATE is

when INPUT=>
X_LOAD<='1';Y_LOAD<='1';
SLCT_X<='1';SLCT_Y<='1';

when UPDATE1=>
Y_LOAD<='1';
```

```
when UPDATE2=>
X_LOAD<='1';

when DONE=>
G_LOAD<='1';

when others =>
NULL;
end case;
end process;
end STATES_GCD;
```

GCD_BEHAV.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;

entity GCD_BEHAV is
Port (
CLOCK:in std_logic;
RESET: in std_logic;
SLCT_X: in std_logic;
SLCT_Y: in std_logic;
X_LOAD: in std_logic;
Y_LOAD: in std_logic;
G_LOAD: in std_logic;
NUM1: in std_logic_vector(7 downto 0);
NUM2: in std_logic_vector(7 downto 0);
GCD: out std_logic_vector(7 downto 0);
EQUAL: out std_logic;
LITTLE: out std_logic);
```

```
end GCD_BEHAV;



architecture Behavioral of GCD_BEHAV is
component MUX_GCD
port(
NUM1: in std_logic_vector(7 downto 0);
NUM2:in std_logic_vector(7 downto 0);
SEL: in std_logic;
OUTNUM:out std_logic_vector(7 downto 0));
end component;
component REG_GCD
port(
LOAD_REG: in std_logic;
REG_IN: in std_logic_vector(7 downto 0);
CLOCK:in std_logic;
RESET: in std_logic;
REG_OUT:out std_logic_vector(7 downto 0));
end component;

signal X,OUTNUM,F_X,F_Y,X_MINUS_Y,Y_MINUS_X: std_logic_vector(7 downto 0);
begin
X_MINUS_Y<=X-OUTNUM;
Y_MINUS_X<=OUTNUM-X;
EQ: process(X,OUTNUM)
begin
if X=OUTNUM then
EQUAL<= '1';
else
EQUAL<='0';
end if;
end process;
```

```
LT: process(X,OUTNUM)

begin
if X<OUTNUM then
LITTLE<='1';
else
LITTLE<='0';
end if;
end process;
MUX1: MUX_GCD
port map(NUM1=>X_MINUS_Y, NUM2=>NUM1,SEL=>SLCT_X, OUTNUM=>F_X);
MUX2: MUX_GCD
port map(NUM1=>Y_MINUS_X, NUM2=>NUM2, SEL=>SLCT_Y, OUTNUM=>F_Y);
REG1: REG_GCD
port map(LOAD_REG=>X_LOAD, REG_IN=>F_X, CLOCK=>CLOCK,
RESET=>RESET,
REG_OUT=>X);

REG2: REG_GCD
port map(LOAD_REG=>Y_LOAD, REG_IN=>F_Y, CLOCK=>CLOCK,
RESET=>RESET,
REG_OUT=>OUTNUM);
REG3: REG_GCD
port map(LOAD_REG=>G_LOAD, REG_IN=>X, CLOCK=>CLOCK, RESET=>RESET,
REG_OUT=>GCD);
end Behavioral;
```

MOD_GCD.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MOD_GCD is
```

```
Port (
GO:in std_logic;
NUM1:in std_logic_vector(7 downto 0);
NUM2: in std_logic_vector(7 downto 0);
CLOCK: in std_logic;
RESET: in std_logic;
GCD_OUT: out std_logic_vector(7 downto 0));
end MOD_GCD;

architecture Behavioral of MOD_GCD is
component GCD_BEHAV
Port (
CLOCK:in std_logic;
RESET: in std_logic;
SLCT_X: in std_logic;
SLCT_Y: in std_logic;
X_LOAD: in std_logic;
Y_LOAD: in std_logic;
G_LOAD: in std_logic;
NUM1: in std_logic_vector(7 downto 0);
NUM2: in std_logic_vector(7 downto 0);
GCD: out std_logic_vector(7 downto 0);
EQUAL: out std_logic;
LITTLE: out std_logic);
end component;

component STATES_GCD
Port(
CLOCK: in std_logic;
RESET: in std_logic;
GO:in std_logic;
EQUAL: in std_logic;
LITTLE: in std_logic;
SLCT_X: out std_logic;
SLCT_Y: out std_logic;
X_LOAD: out std_logic;
Y_LOAD: out std_logic;
G_LOAD: out std_logic);
end component;
signal EQUAL, LITTLE, SLCT_X,SLCT_Y: std_logic;
signal X_LOAD,Y_LOAD,G_LOAD: std_logic;
begin

SIG1:GCD_BEHAV port

map(CLOCK=>CLOCK,RESET=>RESET,SLCT_X=>SLCT_X,SLCT_Y=>SLCT_Y,X_LOAD=>X_LOAD
,Y_LOAD=>Y_LOAD,
G_LOAD=>G_LOAD,NUM1=>NUM1,NUM2=>NUM2,GCD=>GCD_OUT,EQUAL=>EQUAL,LITTLE=>LITTLE);

SIG2: STATES_GCD

port map(CLOCK=>CLOCK,RESET=>RESET,GO=>GO,EQUAL=>EQUAL,LITTLE=>LITTLE,
SLCT_X=>SLCT_X,SLCT_Y=>SLCT_Y,X_LOAD=>X_LOAD,Y_LOAD=>Y_LOAD,G_LOAD=>G_LOAD);
end Behavioral;
```

GCD_SIM.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity GCD_SIM is

end GCD_SIM;
```

```
architecture Behavioral of GCD_SIM is

COMPONENT MOD_GCD is
PORT(
CLOCK: in std_logic;
RESET: in std_logic;
GO:in std_logic;
NUM1:in std_logic_vector(7 downto 0);
NUM2: in std_logic_vector(7 downto 0);
GCD_OUT: out std_logic_vector(7 downto 0));
END COMPONENT;

SIGNAL CLOCK,RESET,GO : STD_LOGIC;
SIGNAL NUM1, NUM2, GCD_OUT: STD_LOGIC_VECTOR(7 DOWNTO 0);

begin
SIM_GCD : MOD_GCD PORT MAP(
CLOCK => CLOCK,
RESET => RESET,
GO => GO,
NUM1 => NUM1,
NUM2 => NUM2,
GCD_OUT => GCD_OUT
);

CLOCK : PROCESS
BEGIN
WAIT FOR 10NS;
CLOCK <= '1';
WAIT FOR 10NS;
CLOCK <= '0';
END PROCESS;
```

```
TESTBENCH1 : PROCESS
BEGIN
RESET <= '1';
WAIT FOR 20NS;
RESET <= '0';

NUM1 <= "10001100";
NUM2 <= "00001100";
WAIT FOR 1000NS;
END PROCESS;

TESTBENCH2 : PROCESS
BEGIN
GO <= '0';
WAIT FOR 20NS;
GO <= '1';
WAIT FOR 1000NS;
END PROCESS;

end Behavioral;
```