**ECS787P - Integrated Circuit Design: Lab Project (VHDL)**

# Aims and Objectives

In this lab coursework, the aim is to develop an HDL program for solving a practical problem. The objectives are:

* to design a finite state machine (FSM) for solving a given coding problem;
* to implement the FSM in VHDL software;
* to validate the VHDL source code through simulation;
* to develop a test bench for testing the code for a variety of different user-defined inputs and observed outputs;
* to synthesise the design based on different constraints to explore the resources used in different instances. In addition, the development kit will be used
* to evaluate your design in a practical environment based on the software development kit. You are expected to prepare your source code **before this lab**.

# Design Problem: Prime Number Checker (PNC) Finite State Machine

The task is to build a machine that can check whether or not a given decimal natural number between 0 and 63 is a prime number. Knowing whether or not a number is prime is important in certain digital information technology applications, e.g., cryptography and cyber security.

Recall that a prime number is defined as any number that is divisible by 1 and by itself only, whereby both these divisors must be different. (The number 1 is thus not considered to be a prime number. A number *N* is said to be divisible by a divisor *n* of the remainder of the division *N*/*n* is zero.)

One simple algorithm that is able to determine whether a given number *N* is prime can be based on checking whether there are no prime divisors, starting from 2 up to and including *N.* Example: 61 is a prime because it is not divisible by the prime numbers 2, 3, 5, 7, while 7 < 61 < 11.

VHDL in the Xilinx Vivado supports the use of *modulo* operation via the MOD(,) command: *r*=MOD(*a,b*) returns the remainder *r* after integer division of *a* by *b*. Examples: MOD(5,3)=2, MOD(18,5)=3.

In several VHDL implementations other than Vivado, the MOD(,) operation is not available. Divisibility rules exist to check remainders for prime divisors, but are cumbersome to implement for binary numbers. As an alternative, a MOD(,) function (subroutine) can be programmed and then instantiated in VHDL code.

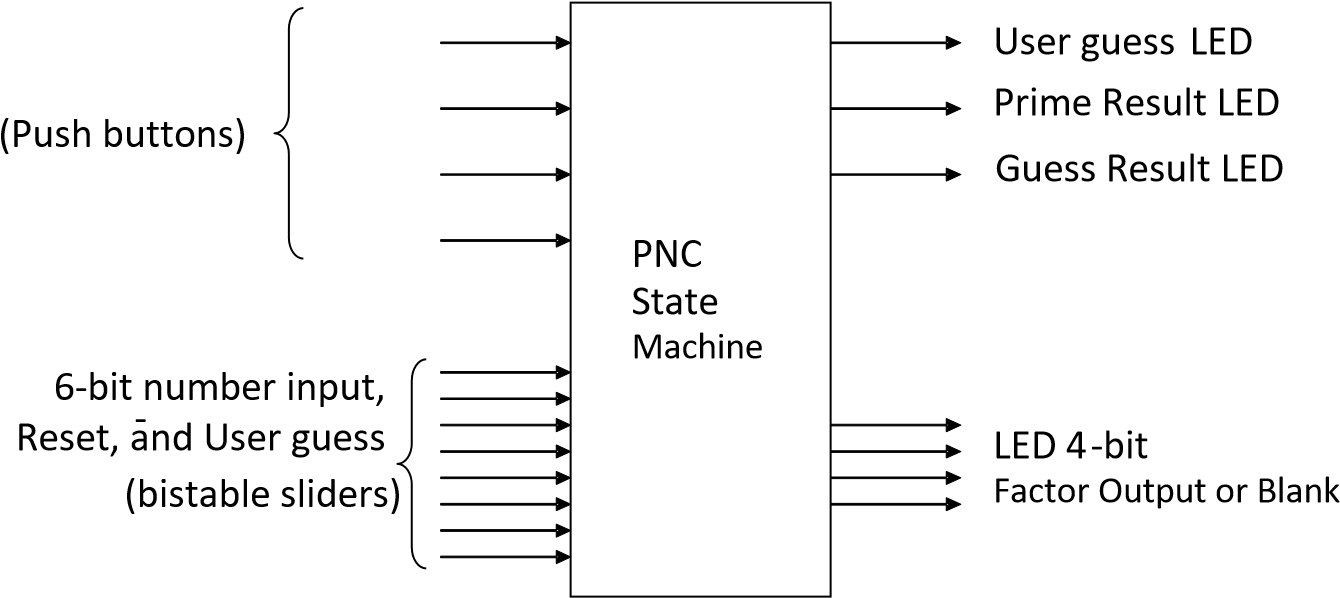


Figure 1: Prime number checker FSM.

**Requirements :**

At any time the asynchronous reset signal can be asserted (active high) to force the state machine back into its default state awaiting input. In the default state, the Prime and Non-Prime indicator Light Emitting Diodes (LEDs) are both asserted (illuminating).

The architecture employs a user input in the form of an unsigned 6-bit binary number (range = 20-1…26-1, i.e. decimal 0…63). A further 7th bit does not form part of the input number, but it is used to set the user’s guess on the primality of the input number, with ‘1’ (high) representing a guess that the number is prime, and ‘0’ a guess that it is not prime. As soon as the Reset slide is activated, both the Prime Result and Guess Result indicator LEDs are asserted and the FSM is ready to accept a number and the User guess as inputs.

The FSM starts by handling the cases that cannot be checked by the above algorithm, i.e., if the numbers 0 or 1 (non-prime) are provided as input. Next, the input number is checked for prime divisors, starting from 2. As soon as a the number is found to have a divisor, the output shows this *lowest* divisor as a binary 4-bit number output. If no prime divisor up to and including 7 is found, the Prime LED is asserted. The Guess Result indicator LED is either asserted (if user’s guess was correct) or remains OFF (if guess was incorrect) and the algorithm terminates.

Task is to design and build a digital circuit that controls this PNC machine in two ways:

1. Exploiting the MOD operation provided in Vivado;
2. Not using the built-in MOD() function for use with more restricted implementations of VHDL, but by implementing and instantiating your own MOD() function.

The controller takes the form of a state machine with 8 inputs and 7 outputs, as follows:

Inputs:

Reset (slider “7”) – A switch-on of Reset (UP) causes the controller: (1) to enter the default state that awaits input of the prime/nonprime number by the user, (2) to turn off the output Number LEDs, and (3) to assert both the Prime and Guess Result LEDs, as a check that these LEDs are functioning. Switching off of Reset (DOWN) starts the evaluation of the inputs (number and guess).

User Guess (slider “6”) – User’s guess on primality: HIGH = prime, LOW = not prime.

|  |  |
| --- | --- |
| Sliders “0”...”5”    Outputs: | – Inputs that are sampled on the rising edge of the clock signal immediately after a Reset event. |
| Number LEDs | – 4-bit output that represents the lowest prime divisor (if input number is non-prime) or no illumination if input number is found to be prime. |
| User guess LED | – Output is HIGH when the user believes the input number is prime. The user should input his guess before the algorithm starts checking, i.e., before Reset is switched back to the OFF state. |
| Prime Result LED | – Output is HIGH (ON) only when the input number is prime, or LOW (OFF) when not prime. |
| Guess Result LED | – Output is HIGH only when the user’s guess is correct. |

# Lab – Part 1: Compiling and Simulating your VHDL Design

In this first part of the lab, you need to import your VHDL code into the Vivado environment and then compile it. The code will then be associated with a testbench and simulated. You should observe the signal waveforms to confirm the correct functioning of the state machine. This includes the normal cycling between states driven by external signal sources and the behaviour of the system when reset.

The default is to develop and modify your VHDL code within the Vivado environment (see Vivado lab guide and demonstration session), which will get the .vhd extension. Alternatively, you can write your VHDL code in plain text (.txt extension), which you can import in Vivado as such and then convert within Vivado to a .vhd-type file. The same applies to test bench code. Your project files should include the correct constraints file (.xdc), to run on a DSDB board in the lab, in-session or remotely by operating the virtual switches as demonstrated during the lab info session.

**Create a new project:**

* Login to the PC and place your code in the H: drive either by copying it from a USB memory stick or by downloading it through a browser connected to your email client.
* Start the Vivado software by clicking on the appropriate desktop icon ▪ Select a “New Project”, give it a name and select the location on drive H.
* Select the “Project Type” to be “RTL Project”. Then click “Next” and add your file with target language set to VHDL. There is no need to add constraints at this time.
* Select “XC7Z020CLG484-1” as the target device.
* Click “Finish” on the Summary

**Creating a VHDL testbench:**

* Go to Project Manager and select “Add Source”
* Add or create simulation sources (followed by Next) ▪ Add your testbench VHDL file ▪ Run the simulation.
* If you need to modify your source code, relaunch the simulation.

Configure the testbench so that the guessing game controller is tested with valid and invalid combinations and that the Reset line is asserted at some point (you might also try different assertion durations). Capture the graphical output so that you can confirm that the VHDL code performs as intended. You can add part of it to your report along with comments (after the lab) to provide evidence that your code adheres to the specified requirements.

# Source code:

## With MOD Function:

SOURCE CODE:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity PrimeNumberFSM is

Port (

clk, reset : in STD\_LOGIC;

user\_guess : in STD\_LOGIC;

input\_number : in STD\_LOGIC\_VECTOR(5 downto 0);

prime\_result\_led, guess\_result\_led : out STD\_LOGIC;

number\_leds : out STD\_LOGIC\_VECTOR(3 downto 0);

user\_guess\_led : out STD\_LOGIC

);

end PrimeNumberFSM;

architecture Behavioral of PrimeNumberFSM is

-- Define states

type state\_type is (IDLE, WAIT\_INPUT, CHECK\_PRIME, CHECK\_DIVISORS, OUTPUT\_RESULT);

signal current\_state, next\_state : state\_type;

-- Signals for state machine operation

signal divisor : STD\_LOGIC\_VECTOR(5 downto 0);

signal is\_prime, guess\_correct : STD\_LOGIC;

signal lowest\_divisor : STD\_LOGIC\_VECTOR(3 downto 0);

signal num, sqrt\_num, i : integer range 0 to 63;

signal is\_divisible : boolean;

constant a : integer := 7;

-- Custom to\_integer function

function to\_integer(input\_vector: STD\_LOGIC\_VECTOR) return integer is

variable result : integer := 0;

begin

for i in input\_vector'range loop

if input\_vector(i) = '1' then

result := result \* 2 + 1;

else

result := result \* 2

end if;

end loop;

return result;

end function;

begin

process (clk, reset)

begin

if reset = '1' then

current\_state <= IDLE;

divisor <= (others => '0');

is\_prime <= '0';

guess\_correct <= '0';

lowest\_divisor <= (others => '0');

prime\_result\_led <= '1'; -- Prime result LED initially off

guess\_result\_led <= '1'; -- Guess result LED initially off

user\_guess\_led <= '1'; -- User guess LED initially off

number\_leds <= (others => '0'); -- Number LEDs initially off

elsif rising\_edge(clk) then

current\_state <= next\_state;

end if;

end process;

process (current\_state, input\_number, divisor, user\_guess)

begin

num <= to\_integer(unsigned(input\_number));

case current\_state is

when IDLE =>

if reset = '0' then

next\_state <= WAIT\_INPUT;

else

next\_state <= IDLE;

end if;

when WAIT\_INPUT =>

if reset = '0' then

next\_state <= CHECK\_PRIME;

else

next\_state <= WAIT\_INPUT;

end if;

when CHECK\_PRIME =>

if num = 0 or num = 1 then

is\_prime <= '0';

next\_state <= OUTPUT\_RESULT;

else

is\_prime <= '1';

divisor <= "000010"; -- Start checking with 2

i <= 2;

next\_state <= CHECK\_DIVISORS;

end if;

when CHECK\_DIVISORS =>

is\_divisible <= false;

while i <= a loop

if num mod i = 0 then

is\_divisible <= true;

lowest\_divisor <= std\_logic\_vector(to\_unsigned(i, 4));

next\_state <= OUTPUT\_RESULT;

exit; -- Exit loop if divisor found

end if;

i <= i + 1;

end loop;

if not is\_divisible or i > a then

next\_state <= OUTPUT\_RESULT;

end if;

when OUTPUT\_RESULT =>

if user\_guess = is\_prime then

guess\_correct <= '1'; -- User's guess is correct

else

guess\_correct <= '0'; -- User's guess is incorrect

end if;

prime\_result\_led <= is\_prime;

guess\_result\_led <= guess\_correct;

user\_guess\_led <= user\_guess;

number\_leds <= lowest\_divisor;

next\_state <= WAIT\_INPUT;

when others =>

next\_state <= IDLE;

end case;

end process;

end Behavioral;

## Without MOD Function:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity PrimeNumberFSM is

Port (

clk, reset : in STD\_LOGIC;

user\_guess : in STD\_LOGIC;

input\_number : in STD\_LOGIC\_VECTOR(5 downto 0);

prime\_result\_led, guess\_result\_led : out STD\_LOGIC;

number\_leds : out STD\_LOGIC\_VECTOR(3 downto 0);

user\_guess\_led : out STD\_LOGIC

);

end PrimeNumberFSM;

architecture Behavioral of PrimeNumberFSM is

-- Define states

type state\_type is (IDLE, WAIT\_INPUT, CHECK\_PRIME, CHECK\_DIVISORS, OUTPUT\_RESULT);

signal current\_state, next\_state : state\_type;

-- Signals for state machine operation

signal divisor : STD\_LOGIC\_VECTOR(5 downto 0);

signal is\_prime, guess\_correct : STD\_LOGIC;

signal lowest\_divisor : STD\_LOGIC\_VECTOR(3 downto 0);

signal num, sqrt\_num, i : integer range 0 to 63;

signal is\_divisible : boolean;

constant a : integer := 7;

-- Custom to\_integer function

function to\_integer(input\_vector: STD\_LOGIC\_VECTOR) return integer is

variable result : integer := 0;

begin

for i in input\_vector'range loop

if input\_vector(i) = '1' then

result := result \* 2 + 1;

else

result := result \* 2;

end if;

end loop;

return result;

end function;

-- Custom mod function

function custom\_mod(dividend : integer; divisor : integer) return integer is

begin

return dividend - ((dividend / divisor) \* divisor);

end function;

begin

process (clk, reset)

begin

if reset = '1' then

current\_state <= IDLE;

divisor <= (others => '0');

is\_prime <= '0';

guess\_correct <= '0';

lowest\_divisor <= (others => '0');

prime\_result\_led <= '1'; -- Prime result LED initially off

guess\_result\_led <= '1'; -- Guess result LED initially off

user\_guess\_led <= '1'; -- User guess LED initially off

number\_leds <= (others => '0'); -- Number LEDs initially off

elsif rising\_edge(clk) then

current\_state <= next\_state;

end if;

end process;

process (current\_state, input\_number, divisor, user\_guess)

begin

num <= to\_integer(unsigned(input\_number));

case current\_state is

when IDLE =>

if reset = '0' then

next\_state <= WAIT\_INPUT;

else

next\_state <= IDLE;

end if;

when WAIT\_INPUT =>

if reset = '0' then

next\_state <= CHECK\_PRIME;

else

next\_state <= WAIT\_INPUT;

end if;

when CHECK\_PRIME =>

if num = 0 or num = 1 then

is\_prime <= '0';

next\_state <= OUTPUT\_RESULT;

else

is\_prime <= '1';

divisor <= "000010"; -- Start checking with 2

i <= 2;

next\_state <= CHECK\_DIVISORS;

end if;

when CHECK\_DIVISORS =>

is\_divisible <= false;

while i <= a loop

if custom\_mod(num, i) = 0 then

is\_divisible <= true;

lowest\_divisor <= std\_logic\_vector(to\_unsigned(i, 4));

next\_state <= OUTPUT\_RESULT;

exit; -- Exit loop if divisor found

end if;

i <= i + 1;

end loop;

if not is\_divisible or i > a then

next\_state <= OUTPUT\_RESULT;

end if;

when OUTPUT\_RESULT =>

if user\_guess = is\_prime then

guess\_correct <= '1'; -- User's guess is correct

else

guess\_correct <= '0'; -- User's guess is incorrect

end if;

prime\_result\_led <= is\_prime;

guess\_result\_led <= guess\_correct;

user\_guess\_led <= user\_guess;

number\_leds <= lowest\_divisor;

next\_state <= WAIT\_INPUT;

when others =>

next\_state <= IDLE;

end case;

end process;

end Behavioral

## Test Bench:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity PrimeNumberFSM\_TB is

end PrimeNumberFSM\_TB;

architecture Behavioral of PrimeNumberFSM\_TB is

-- Constants

constant CLK\_PERIOD : time := 10 ns;

-- Signals

signal clk, reset, user\_guess : std\_logic := '0';

signal input\_number : std\_logic\_vector(5 downto 0) := "000000";

signal prime\_result\_led, guess\_result\_led, user\_guess\_led : std\_logic;

signal number\_leds : std\_logic\_vector(3 downto 0);

-- Component instantiation

component PrimeNumberFSM

Port (

clk, reset : in std\_logic;

user\_guess : in std\_logic;

input\_number : in std\_logic\_vector(5 downto 0);

prime\_result\_led, guess\_result\_led : out std\_logic;

number\_leds : out std\_logic\_vector(3 downto 0);

user\_guess\_led : out std\_logic

);

end component;

begin

-- Instantiate DUT

DUT: PrimeNumberFSM

port map (

clk => clk,

reset => reset,

user\_guess => user\_guess,

input\_number => input\_number,

prime\_result\_led => prime\_result\_led,

guess\_result\_led => guess\_result\_led,

number\_leds => number\_leds,

user\_guess\_led => user\_guess\_led

);

-- Clock process

clk\_process: process

begin

clk <= '0';

wait for CLK\_PERIOD / 2;

clk <= '1';

wait for CLK\_PERIOD / 2;

end process;

-- Stimulus process

stimulus\_process: process

begin

wait for 10 ns; -- Wait for initial signals to stabilize

reset <= '1';

wait for CLK\_PERIOD \* 2;

reset <= '0';

-- Test case 1: Non-prime number (e.g., 10)

input\_number <= "010010"; -- 10 in binary

user\_guess <= '0'; -- Non-prime guess

wait for CLK\_PERIOD \* 10; -- Allow some time for the state machine to process

-- Test case 2: Prime number (e.g., 13)

input\_number <= "001101"; -- 13 in binary

user\_guess <= '1'; -- Prime guess

wait for CLK\_PERIOD \* 10; -- Allow some time for the state machine to process

-- Add more test cases as needed

wait;

end process;

end Behavioral

# Lab – Part 2: Synthesis of your VHDL Design

In this second part of the lab, the purpose is to synthesize your design in two ways as described in section 2, and to observe the resulting schematics and resource usage in the Design Runs table.

You should attempt two state-machine encodings, targeting your synthesis at an FPGA device. In both cases, examine the Register Transfer Level (RTL) schematic and the implementation-specific schematics, if available. Note that optimization cannot be disabled, and the resulting low-level designs may differ from your expectation.

In your final report, compare the two methods and discuss in terms of resources used and time needed.

# Lab – Part 3: Testing your Design on the Development Kit

In the final part of the lab, you should download your design to the Xilinx FPGA on the DSDB development board and confirm that its operation is satisfactory, by checking examples from your test bench and observing the output LEDs.

**Programming the Target FPGA (based on live in-lab operation; requires appropriate changes for remote operation and demonstration):**

* Connect the development kit to the ELVIS base unit.
* Turn on the development board.
* In Vivado, go to Hardware Manager in the Program and Debug menu.
* Select open target and select auto-connect to detect the board.
* Open elaborate design in RTL analysis.
* Go to the I/O ports tab and assign pins to the various inputs and outputs of your design. Select the voltage to be LVCMOS33. The name of the pins on the development board are written on the silkscreen mask (between “<” and “>” symbols). These match the I/O pin names in the Vivado software ▪ Save the project.
* If asked for a constraint file, select “add new file” to the project and manually add the constraint: set\_property CLOCK\_DEDICATED\_ROUTE FALSE [get\_nets Clock]

The name of the clock signal should match the name you have given to the clock input in your code.

* Then select “Run implementation” followed by “Generate bitstream”. In hardware manager select program device and select the FPGA board. Select the bitstream file you created and press

“Program”.

* Test your PNC controller on the DSDB development board to confirm satisfactory operation.