LAB #5

Display the output of sequential circuit using VHDL programming techniques

Objective

- To learn how to make procedures and functions in VHDL
- To learn to behavioural modelling of sequential circuits in VHDL
- To learn how to design finite state machines in VHDL

Pre-Lab

Lab 5 is divided into two parts, in part one we will learn about two new constructs in VHDL programming, procedures and functions. In second part we will learn about how state machines are implemented in VHDL.

Part-1 Procedures and Functions

FUNCTION and PROCEDURE (called subprograms) are very similar to PROCESS, in the sense that these three are the only sections of VHDL code that are interpreted sequentially. Consequently, only sequential statements (IF, WAIT, LOOP, CASE) are allowed (plus operators, of course, because these can be used in any kind of code).

On the other hand, contrary to PROCESS, which is intended for the ARCHITECTURE body (regular codes), subprograms can be constructed in a PACKAGE, ENTITY, ARCHITECTURE, or PROCESS. Because PACKAGE is the most common location, with which the VHDL libraries are built, in our context subprograms are considered system-level units (along with PACKAGE and COMPONENT).

FUNCTION

FUNCTION is a section of sequential VHDL code whose main purpose is to allow the creation and storage in libraries of solutions for commonly encountered problems, like data-type conversions, logical and arithmetic operations, etc.

FUNCTION is similar to PROCESS in the sense that it too is sequential and therefore can only use the same statements (IF, WAIT, LOOP, and CASE). A simplified syntax for the construction of functions is shown below.

```
[PURE | IMPURE] FUNCTION function_name [(input_list)]
RETURN return_value_type IS
   [declarative_part]
BEGIN
   statement_part
  [label:] RETURN expression;
END [FUNCTION] [function_name];
```

Let's explain each thing separately,

PURE: A function is said to be pure when it can only modify its own variables. If left unspecified, the default value (PURE) is assumed.

IMPURE: An impure function, on the other hand, may also modify signals or variables from the architecture, process, or subprogram where it is declared.

The input list can contain any number of parameters (including zero), which are all of mode IN (that is, all parameters are inputs to the function). The list can only contain the objects CONSTANT (default), SIGNAL, and FILE (VARIABLE is not allowed).

Example The function below, named ***positive_edge***, receives a signal called s, returning TRUE when a positive transition occurs on s.
A function can be declared inside an ARCHITECTURE, but we will always be declaring FUNCTION and PROCEDURE in a PACKAGE like the example shown below

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

PACKAGE my_components is
    FUNCTION positive_edge (SIGNAL s: STD_LOGIC) RETURN BOOLEAN;
END PACKAGE;

PACKAGE BODY my_components IS

    FUNCTION positive_edge (SIGNAL s: STD_LOGIC) RETURN BOOLEAN IS
        BEGIN
            RETURN (s'EVENT AND s='1');
    END FUNCTION positive_edge;

END PACKAGE BODY;
```

Once the function is declared then the other point is how to **CALL** the function in our ARCHITECURE block.
Three equivalent function calls are shown below.

```
-----Function declaration:---------------------------
FUNCTION my_function (SIGNAL a, b: BIT) RETURN BIT;
-----Equivalent function calls:----------------------
y <= my_function (x1, x2); --positional mapping
y <= my_function (a=>x1, b=>x2); --nominal mapping
y <= my_function (b=>x2, a=>x1); --nominal mapping
-----------------------------------------------------
```

PROCEDURE

The purpose, construction, and usage of PROCEDURE are similar to those of FUNCTION. Their main difference is that a PROCEDURE can return more than one value. A syntax for the construction of procedures is shown below.

```
PROCEDURE procedure_name (input_output_list) IS
   [declarative_part]
BEGIN
   statement_part
END [PROCEDURE] [procedure_name]
```

The **input-output list** can contain CONSTANT, SIGNAL, and VARIABLE. Their mode can be IN, OUT, or INOUT; if it is IN, then CONSTANT is the default object, while for OUT and INOUT the default is VARIABLE. Their declarations are as follows:

```
CONSTANT constant_name: mode constant_type;
SIGNAL signal_name: mode signal_type;
VARIABLE variable_name: mode variable_type;
```

Both FUNCTION and PROCEDURE are sequential codes, so only sequential statements are allowed. PROCEDURE can be constructed and used in the same way, with PACKAGE (plus the corresponding PACKAGE BODY) as the most common location (for libraries).

Like function calls, procedure calls can be made basically anywhere (in sequential as well concurrent code, in subprograms, etc.). However, the former is called as part of an expression, while the latter is a statement on its own. Examples of procedure **CALLS** are shown below.

```
-------------------------------------------------
sort (a1, a2, a3, b1, b2, b3);
-------------------------------------------------
divide (dividend, divisor, quotient, remainder);
-------------------------------------------------
IF (x>y) THEN get_max (x1, x2, x3, x4, y1, y2);
-------------------------------------------------
```

Here, the procedure names are sort, divide and get_max. Another important reason for using PROCEDURE and FUNCTION over PORT MAP is that the formers can be used inside a sequential code block(process).

Example: The function, named ***positive_edge***, can be declared using procedures as

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

PACKAGE my_components is
    FUNCTION positive_edge (SIGNAL s: STD_LOGIC) RETURN BOOLEAN;
    PROCEDURE pos_edge (SIGNAL s: in std_logic; SIGNAL sout : out std_logic);
END PACKAGE;

PACKAGE BODY my_components IS

    FUNCTION positive_edge (SIGNAL s: STD_LOGIC) RETURN BOOLEAN IS
        BEGIN
            RETURN (s'EVENT AND s='1');
    END FUNCTION positive_edge;

    PROCEDURE pos_edge (SIGNAL s: in std_logic; SIGNAL sout : out std_logic) IS
    BEGIN
        if (s'event and s='1') then
            sout <= '1';
        else
            sout <='0';
        end if;
    END PROCEDURE pos_edge;
END PACKAGE BODY;
END my_components;
```

The code for our package (my_components) given above has a function named positive_edge and a procedure named pos_edge. As a beginner student, you should pay close attention to how both functions and procedures are declared and later **CALLed** in an architecture.

```
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.my_package.all;
entity fastcounter is port(
    clk : in std_logic;
    binout : out std_logic_vector(3 downto 0)); end fastcounter;

ARCHITECTURE behav of fastcounter is
signal cout: std_logic_vector(3 downto 0):="0000";
SIGNAL sclk: std_logic;
BEGIN
    binout <= cout;
    PROCESS(clk)
    begin
        --if clk='1' then
    ------------ Using FUNCTION------------
        if (positive_edge(clk)) then

            cout <= cout + '1';
        end if;

    ------------ Using PROCEDURE------------
        pos_edge( clk, sclk);
        if (sclk ='1') then
            cout <= cout + '1';
        end if;
    end process;
end behav;
```

While testing the above code try to comment either FUNCTION or PROCEDURE part of the PROCESS. As both are doing the same thing.

Pre-Lab Tasks

Implement 4-bit Binary counter implemented in Lab 4 using procedures and functions

You are to design the 4-bit binary counter and this time you have to show the output on 7-segment display. In order to do that work, you need to make a function/procedure for 7-segment display. You can use the code given below for counter

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity fastcounter is port(
    clk : in std_logic;
    binout : out std_logic_vector(3 downto 0)


); end fastcounter;

architecture behav of fastcounter is
signal cout: std_logic_vector(3 downto 0):="0000";
begin

    binout <= cout;
    process(clk)
    begin
        if clk='1' then
            cout <= cout + '1';
        end if;

    end process;
end behav;
```

Figure 1 Up counter using behavioral modelling in VHDL

Part-2 Implementing State Machines in VHDL

A Finite State Machine (FSM) is a circuit/machine/system that goes through a fixed number of states and has fixed numbers of input/output combinations. FSMs are typically used to control, monitor, calculate a circuit or to implement a communication protocol etc.
To implement FSM in VHDL, we need to know about three important things

i. **Next State Logic (NSL)**
   It will help us decide which state we will go under what condition
ii. **Output Function Logic (OFL)**
   What to do in what state under what condition
iii. **State Memory (SM)**
   Holds the current state

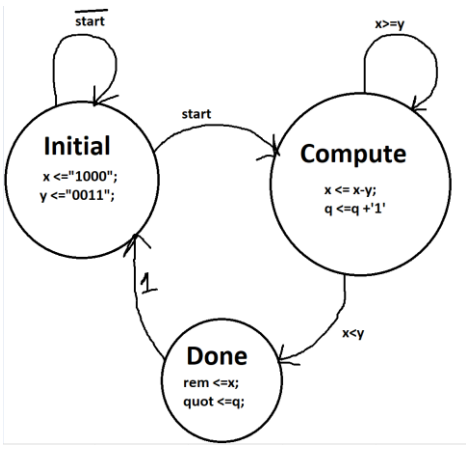
Writing an FSM

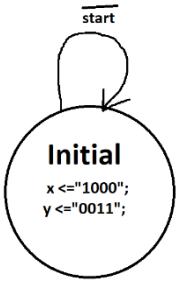
- We will model NSL using if-else statement
- Each state will be modeled using case statement
- State variables will be defined as constants (using one-hot encoding)
- OFL will be modeled using conditional assignments

In-Lab Tasks

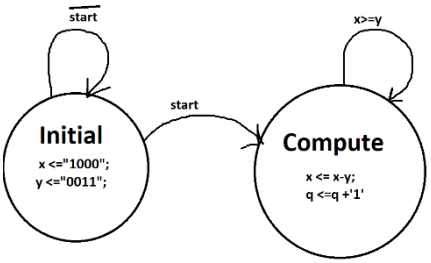
Lab Task 1: Implement the FSM on FPGA

In Lab Task 1, we will learn how to implement a divider circuit using FSM. We will be implementing division using subtraction. The state diagram shown has three states namely, initial, compute and done.

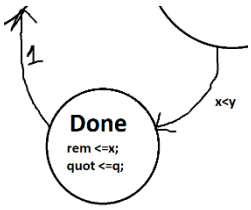




In state **initial**, we will initialize the dividend (x as 1000_2) and divisor (y as 0011_2), this will act as our OFL. Our state machine will stay in initial state until a start button is turned ON, so the NSL will be, go to state compute if start is high otherwise stay in state initial.



In state **Compute**, we will start the actual division process. At every clock positive edge, we will compute x <= x-y; and increment "q", this will act as our OFL. For NSL, our state will change to state done when x goes less than y.



In state **done**, we will simply assign x to our remainder and q to our quotient and unconditionally move to state initial.

Now your task is to test the functionality of the code on FPGA.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity FSMdiv is port(
    clk : in std_logic;
    letstart : in std_logic;
    remaind : out std_logic_vector(3 downto 0);
    quot    : out std_logic_vector(3 downto 0)
);end FSMdiv;

architecture behav of FSMdiv is
signal x: std_logic_vector(3 downto 0) :="1000";  -- Dividend is 8
signal y: std_logic_vector(3 downto 0) := "0011";  -- Divisor is 3
signal q: std_logic_vector(3 downto 0) := "0000"; -- For Quotient

-- defining state using one-hot coding
constant initial : std_logic_vector(2 downto 0) :="001";
constant compute : std_logic_vector(2 downto 0) :="010";
constant done : std_logic_vector(2 downto 0)    :="100";
-- This is our State Memory SM
signal mystate : std_logic_vector(2 downto 0) := "001";
begin
    process(clk)
```

```
begin
    if (clk'event and clk='1') then

        case mystate is
            when initial =>
                -- OFL
                x <= "1000";  -- 8
                y <= "0011";  -- 3

        -- NSL
                if (letstart ='1') then
                    mystate <= compute;
                else
                    mystate <= initial;
                end if;

            when compute =>
             -- OFL
             if (x>=y) then
                x <= x-y;
                q <= q+1;

             end if;

             --NSL

                if (x>=y) then
                    mystate <= compute;
                else
                    mystate <= done;

                end if;

            when done =>
            --OFL binding input signals with top-level entity signals
                remaind <= x;
                quot <= q;
            --NSL
                mystate <=initial;
      when others =>
          mystate <=initial;

      end case;
      end if;  -- end if of clk'event

   end process;
end behav;
```

Lab Task 2: Implement a 4-bit sequence detector

In Lab Task 2, you are to design a FSM that detects the input pattern of 1011. Create its state diagram and implement it using the technique learnt in previous lab task.

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ Date:_____