Traffic Intersection VHDL project

EEE20001 Digital Electronics Design

Lab Day (Wed etc.)	Tuesday	**Lab Time e.g. 12:30 pm**	4.30pm
Name of Lab supervisors		**???**	

Student information (please print)		
Family Name	**Given Name**	**Student ID**
Lillington-Moore	Joshua	103666887
Cos	Julio	102675847

Design Summary (from ISE Summary report)

Number of Latches:	______	
Macrocells Used:	28/64	(44%)
Pterms Used:	61/224	(28%)
Registers Used	19/64	(30%)

Contents

Introduction :

The following report is about the design, creation and finally implementation of a Traffic Light system, using VHDL and the in-built traffic system on the CPLD board. First we will state the requirements of the board, then we will move onto our design solution, including the approach we took, as well as the required tables and diagrams needed to understand it, finally the VHDL code is displayed with a number of test cases shown at the bottom and a short reflection.

Design Requirements :

To start with here is the visualisation given for the traffic intersection shown in figure 1.

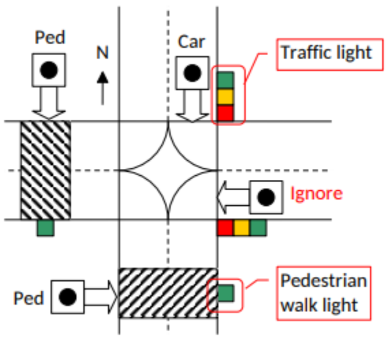

Figure 1 : Traffic Intersection

Below is a list of the main requirements given :

1. The East West road is priority, this means it will be the starting state, and no matter what is pressed it will always go back to it
2. If a North South car was detected the lights will cycle to green in that direction, the walk light will only go on if a pedestrian is detected before the lights cycle to green
3. If the North South pedestrian detected than the lights will cycle similarly
4. If East West pedestrian is detected than the appropriate walk light will be displayed for a short time

There are many other constraints and requirements in the actual top level design but these are shown later. It is made with the intent that the intersection would accept all inputs no matter the situation, eventually cycling back to the East West road.

Design Solution :

To begin we will describe our completed solution, for steps on how we got to this point look down at the testing. When considering the design we opted to use the skeleton given, the design is split into 2 modules. One is the state machine, and takes into account the main process of traffic intersection while the other is the counter. Firstly we can talk about the Traffic module, obviously it starts with all the ports in the entity, with all of the required inputs and outputs. Looking at the architecture, it starts with encoding the lights, these will be used throughout, as well as creating signals for both the time delays for the lights/buttons and when the memory of the pedestrian button being pushed. It lists all of the states, there are 6 in total (EWgreen, EWamber, EWwalk, NSgreen, NSamber, NSwalk).

The next part is the synchronous process, handling the sequential behavior of the system; it uses the reset and clock signals to control the state transitions. When the reset button is pressed it goes back to the desired starting state and resets the variables for whether the buttons are pressed. Otherwise it follows the rising edge of the clock, going from one state to the next each time, also looking out for if either Pedestrian button is pushed or cleared, saving it in its memory.

Following is the combinational process, it is sensitive to the state, delays and the memory for the button being pressed. It sets initial conditions for buttons, and lights before starting the state machine; to design the state machine it is easiest to create a state transition diagram which is shown below in the top level structure. Before going into this process it is important to discuss the counter.

The other module created is the counter, it has its own entity and architecture. For its port map it has inputs for the clock, and clear, where clear is used to set the counter to 0, as well as the delays. Next is the counters architecture, it sets initial conditions for reasonable time delays for certain conditions, the normal (meaning when it is green) is set to the longest at 2 seconds, with the amber being 1 and the pedestrian light being 1.5 seconds, these are constant integers as they do not change. A signal is then used for the count, setting its range from 0 to the normal time.

When looking at the process, it is sensitive only to the reset and the clock. When the reset button is pushed, it sets the counter back to 0 as well as the states for the delays, for the rising edge of the clock, it has a set of conditions, if clear is equal to 1, it will reset the counter and all of the delays, if clear is not set to one, than the counter will count up by 1, if the count is equal to the amberTime than delayAmber will equal to 1 and so on for the other delays. This counter is essential to the functioning of the state machine, it all runs synchronously with the clock.

Now going back to the state machine, for the EWgreen state, this is the priority road, so the counter is constantly cleared so it cannot move to the next state unless the NS pedestrian or NS car button is pushed. If either is pushed, it will start counting until it reaches the delayNormal time, where it will then go to EWamber state and clear the counter. When in the EWamber state, it sets the EW lights to amber, waits for the counter to reach the delayAmber time, when it does, it checks if the NS pedestrian button was pressed, if it was, than go to NSwalk otherwise go to NSgreen, it than clears the counter again. The rest of the cases are all quite similar, it should be noted that when implementing the pedestrian buttons memory, issues arose with the if loops, that we didn't have time to debug, this will be shown in the testing section. Also it initializes the counter inside of the traffic modules, mapping the variables equal to each other, so that the traffic module can recognize the counter.

Top Level Structure :

The following figures include the block diagram for the top-level as well as the state transition diagram.

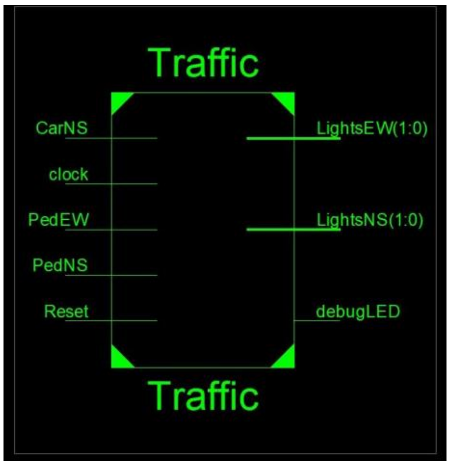

Figure 2 : Top Level State Machine

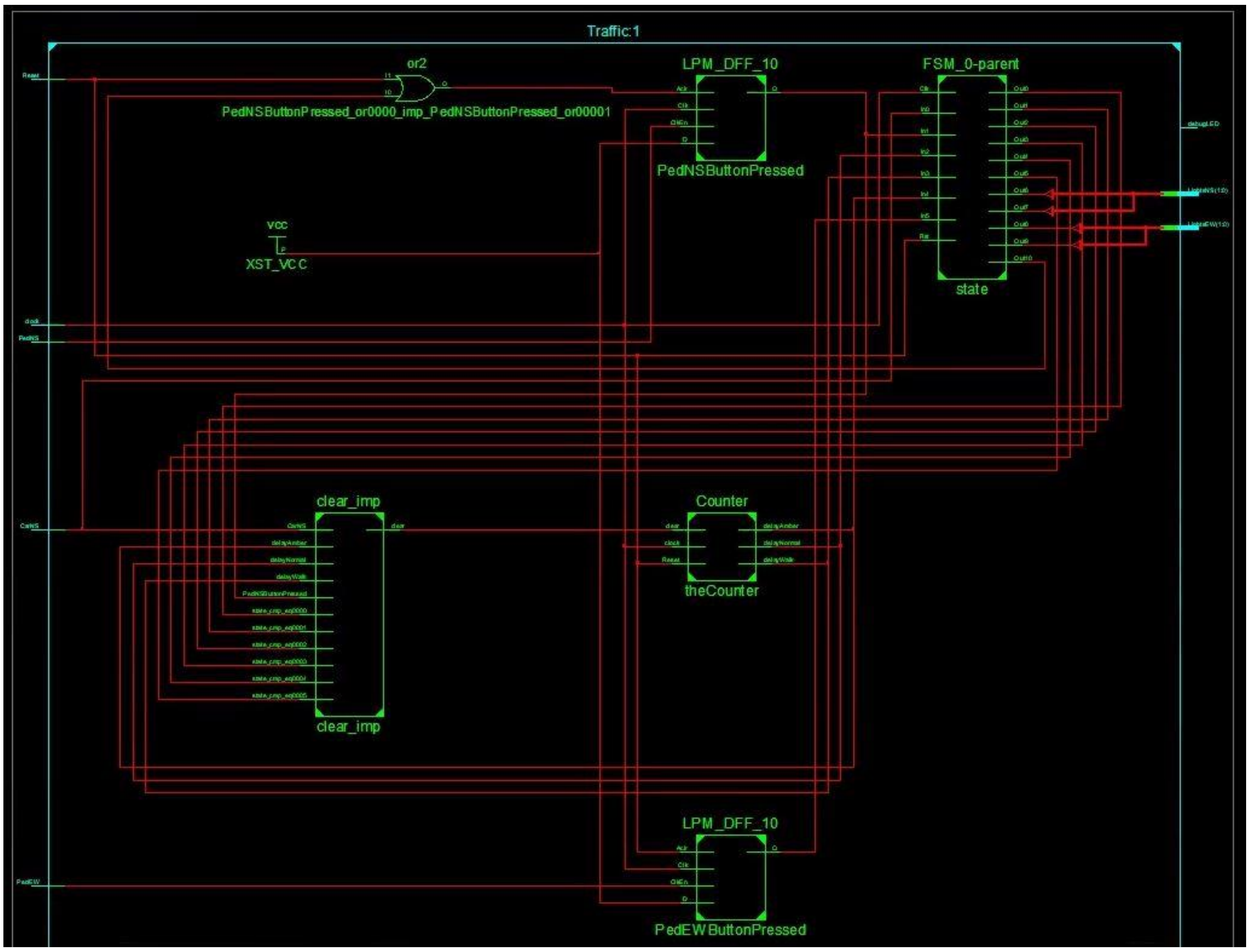

Figure 3 : Detailed Top Level Diagram

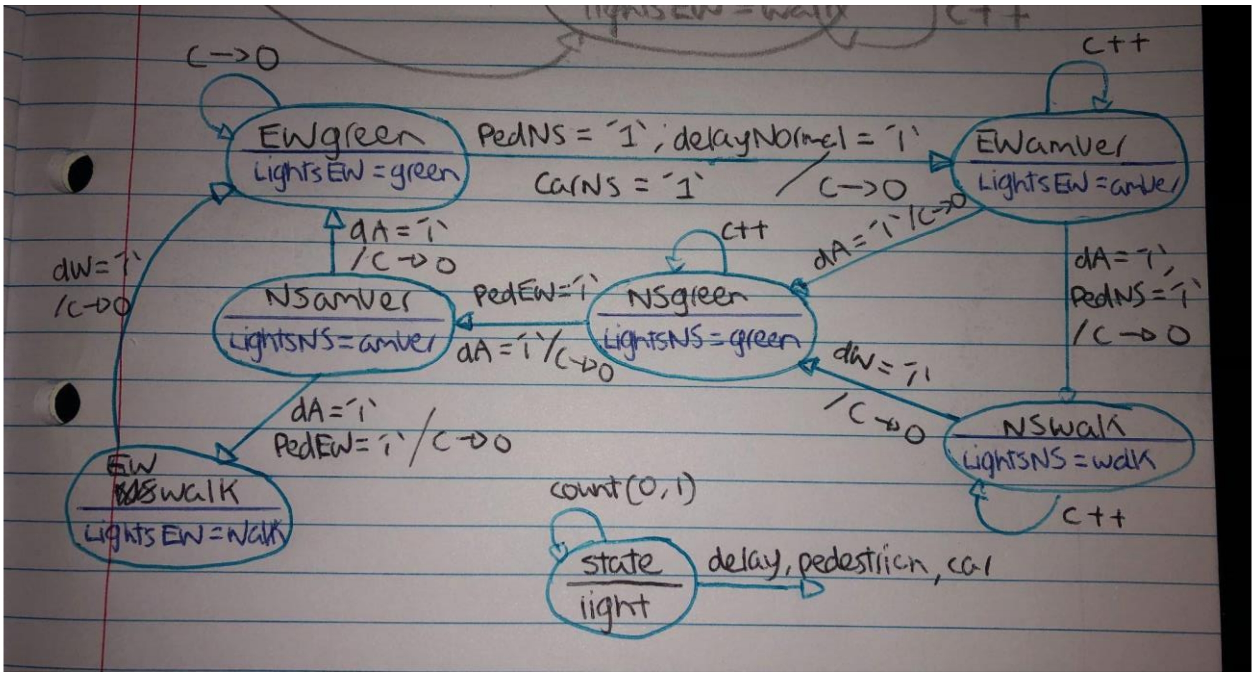

Figure 4 : State Transition Diagram

Testing :

Test	Expected Outcome	Results
Regular Lights	The lights will transition from one state to the next another	This functioned perfectly, the lights would start at EWgreen and cycle through.
Counter	Have it counting the green and amber lights, so it will wait a certain time before going to the next state.	Ran into issue as the Traffic module was not recognizing the counter.
Counter 2.0	Now that we have implemented the counter into the traffic module it should work.	It cycles through the states, but slowly with reasonable time to wait in between.
Car Button	When the NScar button was pressed it should cycle to green then continue through the states.	Functioned successfully.
Making it Revert back to original State	Make it so it will cycle through everything and then stop once it gets back to the Ewgreen state.	We misinterpreted the question. It didn't work but it also wasn't required. This ended up wasting a bit of time.
2nd Attempt	Now it should stay on the EWgreen unless the CarNS button is pressed.	This worked, by constantly clearing the counter.
Pedestrian Button EW	This required 2 new states, EWwalk and NSwalk, first we tested so if the user pressed NSped it should go to the EWwalk state .	It did this successfully.
Pedestrian Counter	This counter is similar to the delayAmber and delayNormal. Just adding a timer for the pedestrian light.	After a suitable time is met it goes back to EWgreen state.
NSped Button Test	When the NScar is pressed, if EWped is pressed before it goes back to red it should go to EWwalk state.	This was successful.
Testing PedNS again	Now we had added memory to the button, so it could keep track of whether it had been pressed or not. This time if the PedNS is pressed when its state is EWgreen it should cycle to it.	This ended up being successful. But the PedEW and CarNS states were now not working in the same loop.
Testing PedNS with CarNS	With two different if statements, when either is pressed it should do its required states.	This was now successful.
Testing PedEW when in NSgreen	Now if the state is NSgreen and the PedEW is pressed, eventually it should	This was successful

	cycle back to it.	
Testing PedEW when in EWgreen	Should go to state EWwalk when PedEW is pressed for the suitable time.	Unfortunately this didn't work when I added another if loop, nor when I tried to nest it.

Reflection

When looking back on the project, we are pretty content with our results. Obviously we were unable to get a register for the pedestrians, we definitely left it too late to fix but the traffic lights work to a fair extent. When adding the pedestrian buttons memory we ran out of time to get them fully functioning, we ran into a lot of issues with the nested if loops, as well as when separating them due to it prioritizing the ones at the top. In the end we can be satisfied with what we have done, but always wish we spent more time on it to get a better result and make it fully functioning.

VHDL Modules

```
----------------------------------------------------------------------------------
--  Traffic.vhd
--
-- Traffic light system to control an intersection
--
-- Accepts inputs from two car sensors and two pedestrian call buttons
-- Controls two sets of lights consisting of Red, Amber and Green traffic lights and
-- a pedestrian walk light.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Traffic is
    Port ( Reset      : in   STD_LOGIC;
           clock      : in   STD_LOGIC;

           -- for debug
           debugLED   : out  std_logic;
--         LEDs       : out  std_logic_vector(2 downto 0);

           -- Car and pedestrian buttons
--          CarEW      : in   STD_LOGIC; -- Car on EW road
           CarNS      : in   STD_LOGIC; -- Car on NS road
           PedEW      : in   STD_LOGIC; -- Pedestrian moving EW (crossing NS road)
           PedNS      : in   STD_LOGIC; -- Pedestrian moving NS (crossing EW road)

           -- Light control
           LightsEW   : out STD_LOGIC_VECTOR (1 downto 0); -- controls EW lights
           LightsNS   : out STD_LOGIC_VECTOR (1 downto 0)  -- controls NS lights
           );
end Traffic;

architecture Behavioral of Traffic is

-- Encoding for lights
constant RED   : std_logic_vector(1 downto 0)  := "00";
constant AMBER : std_logic_vector(1 downto 0)  := "01";
constant GREEN : std_logic_vector(1 downto 0)  := "10";
constant WALK  : std_logic_vector(1 downto 0)  := "11";

signal clear : std_logic;
signal delayAmber : std_logic;
signal delayNormal : std_logic;
signal delayWalk : std_logic;

signal PedNSButtonPressed, PedEWButtonPressed : std_logic;
signal ClearPedNSButtonPressed, ClearPedEWButtonPressed : std_logic;
```

```
type StateType is (EWgreen, EWamber, EWwalk, NSgreen, NSamber, NSwalk);
signal state, nextState : StateType;

begin
--Handles state transisition and button pressed signals
SynchronousProcess:
process(Reset, clock, state, ClearPedNSButtonPressed, ClearPedEWButtonPressed)
begin
            if (Reset = '1') then
               state <= EWgreen;
               PedNSButtonPressed <= '0';
               PedEWButtonPressed <= '0';
            elsif rising_edge(clock) then
               --updates the button pressed signals based on there corresponding inputs
               if PedEW = '1' then
                       PedEWButtonPressed <= '1';
               end if;
               if PedNS = '1' then
                       PedNSButtonPressed <= '1';
               end if;
                       state <= nextState;
            end if;
            if ClearPedNSButtonPressed = '1' then
               PedNSButtonPressed <= '0';
            end if;
            if ClearPedEWButtonPressed = '1' then
               PedEWButtonPressed <= '0';
            end if;
end Process SynchronousProcess;
--handles the control of the traffic lights based on the current state and input signals
CombinationalProcess:
process(state, CarNS, delayAmber, delayNormal, delayWalk, PedNSButtonPressed,
PedEWButtonPressed)

begin
LightsEW <= RED;
LightsNS <= RED;
nextState <= state;
ClearPedNSButtonPressed <= '0';
ClearPedNSButtonPressed <= '0';
clear <= '0';
case state is
            when EWgreen =>
               clear <='1';
               LightsEW <= GREEN;
               if PedNSButtonPressed = '1' then
                               clear <= '0';
                       if delayNormal = '1' then
                               nextState <= EWAmber;
                               clear <= '1';
                       end if;
               end if;
               if CarNS = '1' then
                               nextState <= EWAmber;
                               clear <= '1';
               end if;
            when EWwalk =>
               LightsEW <= WALK;
               ClearPedNSButtonPressed <= '1';
               clear <= '0';
               if (delayWalk = '1') then
                       nextState <= EWgreen;
                       clear <='1';
               end if;
            when EWamber =>
               LightsEW <= AMBER;
               if (delayAmber = '1') then
                       if PedNSButtonPressed = '1' then
                               nextState <= NSwalk;
                       else
                               nextState <= NSgreen;
                       end if;
                               clear <='1';
               end if;
            when NSgreen =>
```

```
               LightsNS <= GREEN;
               if (PedEWButtonPressed = '1') then
                      clear <= '0';
                      if (delayNormal = '1') then
                             nextState <= NSamber;
                             clear <= '1';
                      end if;
               end if;
               if (delayNormal = '1') then
                      nextState <= NSamber;
                      clear <= '1';
               end if;
            when NSwalk =>
               LightsNS <= WALK;
               ClearPedNSButtonPressed <= '1';
               if (delayWalk ='1') then
                      nextState <= NSgreen;
                      clear <= '1';
               end if;

            when NSamber =>
               LightsNS <= AMBER;
               if (delayAmber = '1') then
                      if PedEWbuttonPressed = '1' then
                             nextState <= EWwalk;
                      else
                             nextState <= EWgreen;
                      end if;
                             clear <='1';
               end if;
            end case;
end process CombinationalProcess;

theCounter :
Entity work.Counter
            port map (
            Reset => Reset,
            clock => clock,
            clear => clear,
            delayAmber => delayAmber,
            delayNormal => delayNormal,
            delayWalk => delayWalk
            );

end architecture Behavioral;
```

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    14:39:22 05/25/2023
-- Design Name:
-- Module Name:    Counter - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
```

```
--use UNISIM.VComponents.all;

entity Counter is
    Port ( Reset : in  STD_LOGIC;
           clock : in  STD_LOGIC;

           clear : in  STD_LOGIC;

           delayAmber : out  STD_LOGIC;
           delayNormal : out  STD_LOGIC;
          delayWalk : out STD_LOGIC);
end Counter;

architecture Behavioral of Counter is
constant amberTime : integer := 100;
constant normalTime : integer := 200;
constant walkTime : integer := 150;

signal count : natural range 0 to normalTime;

begin

       process (reset, clock)
       begin
              if (reset = '1') then
                     count <= 0;
                     delayAmber <= '0';
                     delayNormal <= '0';
                     delayWalk <= '0';
              elsif rising_edge(clock) then
                     if (clear = '1') then
                            count <= 0;
                            delayAmber <= '0';
                            delayNormal <= '0';
                            delayWalk <= '0';
                     else
                            count <= count + 1;
                            if (count = amberTime) then
                                   delayAmber <= '1';
                            end if;
                            if (count = normalTime) then
                                   delayNormal <= '1';
                            end if;
                            if (count = walkTime) then
                                   delayWalk <= '1';
                     end if;
              end if;
       end if;
       end process;
end Behavioral;
```