PWM Custom Slave: Lab2

CS 473 Embedded Systems

By: Abhi Kamboj

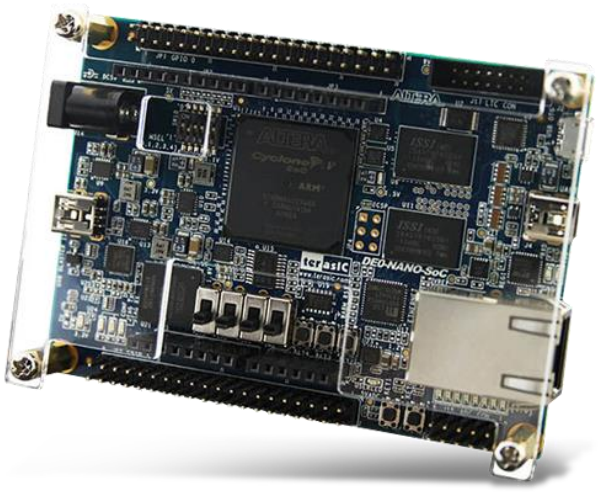

Table of Contents

Introduction:

In this lab, a Pulse Width Modulator component was created as an Avalon bus slave programmable interface using the DE0-Nano-Soc. The interface supports a programmable period, duty cycle and polarity and outputs the Pulse Width Modulation (PWM) to a General Purpose Input Output (GPIO) pin. It was created using the VHSIC Hardware Description Language (VHDL) using Quartus 18.0 and tested with ModelSim. Lastly, a small C program was written in the NIOS II Eclipse studio to use the PWM peripheral to control a Servo Motor. The logic analyzer was also used to test the output signal.

Design:

Block Diagram:

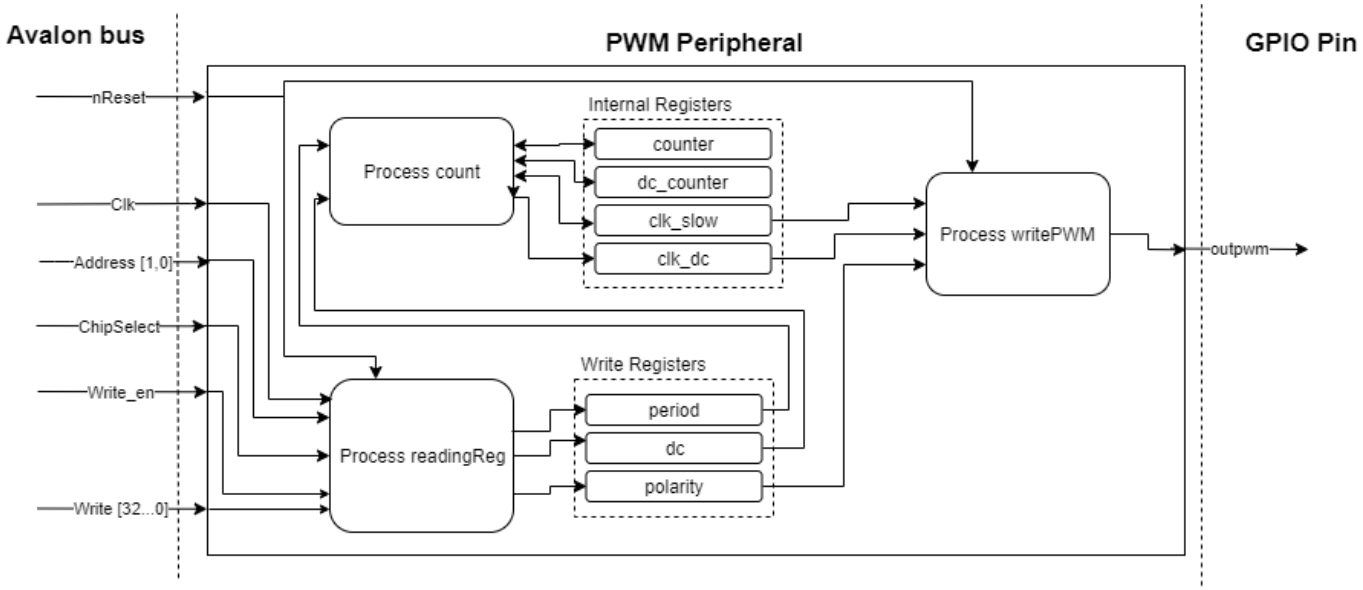

Figure 1 Block Diagram of PWM System

Before beginning to code, the system was designed with this high-level block diagram. As shown in the diagram, there are 3 registers that the master can write to: period, dc and polarity. The system uses these 3 parameters to change the period, duty cycle and polarity respectively. There are also 4 internal registers that were used to communicate amongst the processes. The source code can be found in Appendix 1.

FSM:

The FSM diagram below helps better understand how the system works. Notice that the 'clk_slow' and 'clk_dc' signals are always held at 0 and are only used to indicate transitions amongst the processes. Also, the diagram shown applies only when 'polarity' is set to '1', but the diagram for a polarity of '0' would be identical except the outputpmw=1 state would be switched with the outputpwm=0 state.

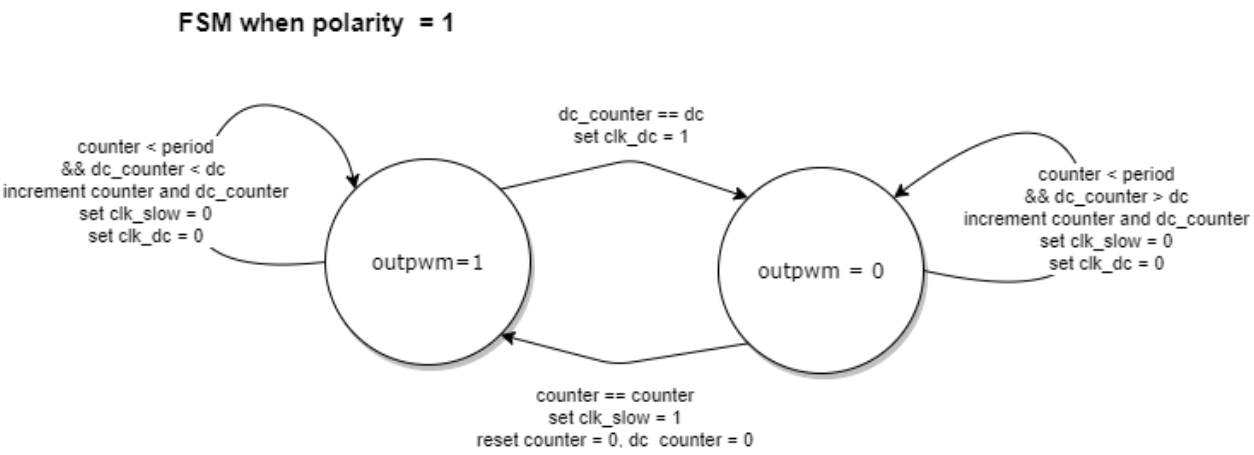

Figure 2 FSM for PWM System when polarity = 1

Process readingReg:

This reads the values from the Avalon bus and puts it into the correct registers depending on the 'Address' signal. It checks 'ChipSelect' and 'Write_en' signals to make sure that the write is supposed to happen, and then writes synchronously with the clock. It also performs an asynchronous reset if 'nReset' is '1', setting the period to 1 ms, duty cycle to 50% and the polarity to 1. This process is sensitive to the 'Clk' and the 'nReset' signals.

Process count:

This process simulates two slower clock by incrementing the internal registers 'counter' and 'dc_counter'. Once 'counter' has reached the value in register 'period' the process sets the 'clk_slow' signal to '1', and similarly once 'dc_counter' has reached 'dc' it sets the 'clk_dc' signal to '1'. Both the counters are reset to '0' only after 'clk_slow' has been set to '1', and both of the simulated clock signals ('clk_slow' and 'clk_dc') are held at '0' otherwise. Both the counters count synchronously to the 'Clk', so this process is only sensitive to 'Clk'.

Process writePWM:

This process outputs the PWM signals according to the simulated clock signals 'clk_slow', 'clk_dc' and the signal 'polarity'. If the polarity is '1', then after a full period, indicated by the rising edge of 'clk_slow', the output is going to be set to '1', then on the rising edge of 'clk_dc' it will be reset to '0'. If polarity is '0', then after the full period the output is set to low, '0', and after the duty cycle clock is set the output is set to high '1'. This process also resets the output to '0' if nReset is '1', so the process is sensitive to 'clk_slow', 'clk_dc', 'polarity', and 'nReset'.

Testing:

The design was tested by simulating the hardware on ModelSim and writing an application on Eclipse.

ModelSim:

In order to use ModelSim a testbench had to be created implementing the Quartus component for the PWM peripheral. This test bench simulated all the Avalon bus signals and set the signals accordingly as if the master processor was writing to the Avalon bus. The full test bench code can be seen in Appendix 2.

After, testing various periods, duty cycles and polarities it was confirmed that the peripheral was working correctly. Refer to Figure 3 and Figure 4 below to see example ModelSim outputs. Notice, how the first couple clock cycles (about 100 ns) are used to read data from the Avalon bus, and how the read values effect the PWM of the 'outputpwm' signal for the remained of the simulation. These figures specifically show the difference that occurs when changing the polarity.

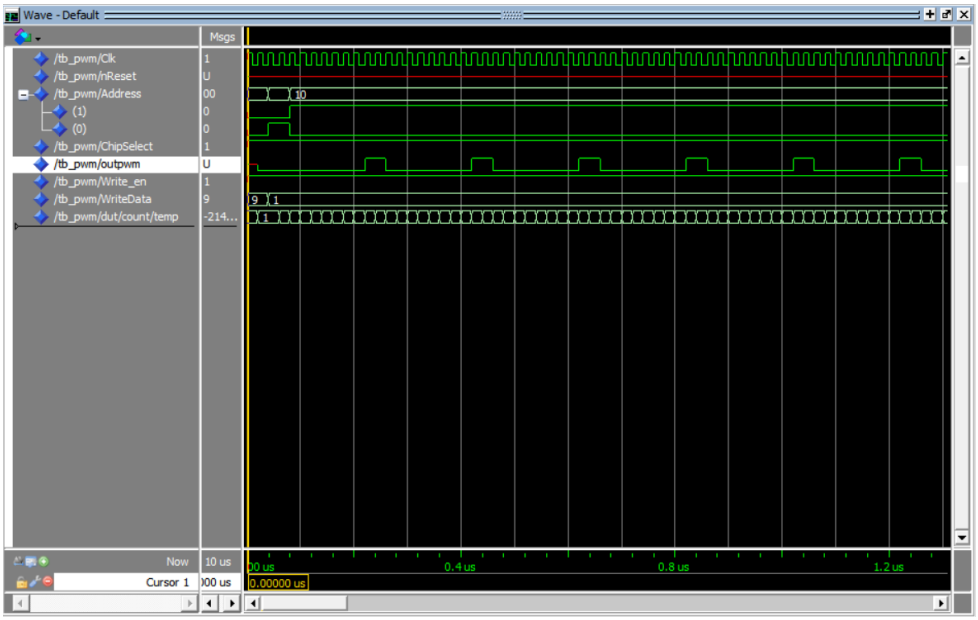

Figure 3 ModelSim output from testing 200ns period, 20% DC, Polarity 1

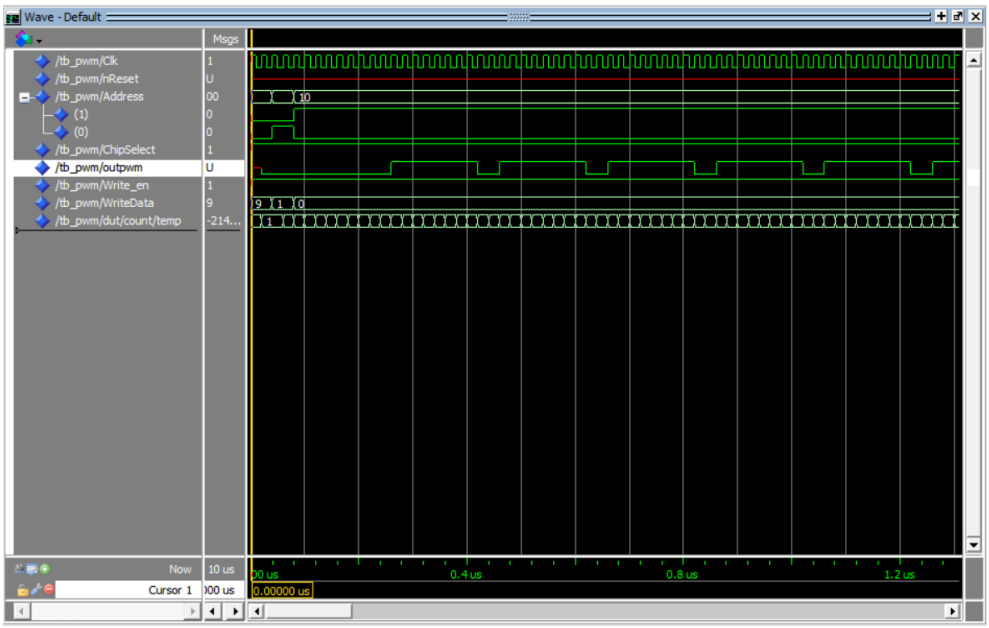

Figure 4 ModelSim output from testing 200ns period, 20% DC, Polarity 0

Nios II Eclipse:

Finally, the component was tested with C code that programs the NIOS II processor on the board to be the master interacting with the PWM peripheral slave on the Avalon Bus. First a function was written to interface the software with the hardware. This function was intended to input a period, duty cycle and polarity and write the correct values to the Avalon bus registers. The function was then going to be used to drive a servo arm back and forth repeatedly. Unfortunately, errors when running the code on the board prevented the fully functionality of this servo motor system from being tested. Nevertheless, the code can be found in Appendix 3.

Appendix 1: VHDL PWM Peripheral

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

Entity PWM_AvalonSlave is
	Port(
		Clk			: IN	std_logic;
		nReset		: IN	std_logic;

		Address		: IN	std_logic_vector (1 DOWNTO 0);
		ChipSelect	: IN	std_logic;

		Write_en	: IN	std_logic;
		WriteData	: IN	integer;

		outpwm		: OUT	std_logic
	);
end PWM_AvalonSlave;


architecture rtl of PWM_AvalonSlave is
	--these are registers the user is going to write to (they have addresses)
	signal period: integer		:= 0;
	signal dc	: integer		:= 0;
	signal polarity	: integer		:=0;

	--these are used internally
	signal counter : integer	:=	0;
	signal clk_slow : std_logic	:=	'0';
	signal clk_dc: std_logic	:=	'0';
	signal dc_counter: integer	:=	0;
```

```
begin

--this process reads what the master has written
	readingReg : Process(Clk, nReset)
	begin
		if nReset = '0' then
			period <= 49999;	--resets period to 1ms
			dc <= 24999;		--duty cycle to 50%
			polarity <=1;
		else
			if rising_edge(Clk) then
				if ChipSelect = '1' and Write_en='1' then
					case Address(1 DOWNTO 0) is
						when "00"=> period <= WriteData;
						when "01"=> dc <= WriteData;
						when "10"=> polarity <= WriteData;
						when others=> null;
					end case;
				end if;
			end if;
	end process readingReg;

--a process that counts to simiulate period and duty cycle
	count : Process(Clk)
	variable temp : integer;
	begin
		if rising_edge(Clk) then
			temp := counter+1;
			counter <= temp;
```

```
                    dc_counter <= temp;
                    --this construct assumes dc < period
                    --NOTE: After testing in modelSim. I found out period and dc is
always 1 greater than you think
                    --so Clk is 20 ms, but if you make period 6 and dc 1, the output
pwm period will be 7*20=140 ms, and 40 ms high
                    if counter = period then --period is how many 50 mhz clk ticks
                          counter <= 0;
                          dc_counter <=0; --need to restart the counters
                          clk_slow <= '1';
                    else
                          if dc_counter = dc then
                                clk_dc <= '1'; -- don't reset the fake clks
                          else
                                clk_dc <= '0';
                                clk_slow <= '0';
                          end if;
                    end if;
             end if;
      end process count;


-- writePWM creates pwm period
      writePWM : Process(clk_slow, clk_dc, polarity, nReset)
      begin
             if nReset = '0' then --Asynchronous reset
                    outpwm <= '0';            --it looks like in the parallel port
they do this
             else
                    --if rising_edge(clk_slow) OR rising_edge(clk_dc) then
                    --can't use two rising edges at once ^^
                    --the solution below should work as long as the dc and period are
greater than one Clk cycle.
```

```
                    if clk_slow = '1' then --this assumes clk_dc and clk_slow are at
least one Clk tick apart
                          if polarity = 0 then
                                outpwm <='0';
                          else --polarity is 1 (should be default)
                                outpwm <='1';
                          end if;
                    else
                          if clk_dc = '1' then
                                if polarity = 0 then
                                      outpwm <='1';
                                else
                                      outpwm <='0';
                                end if;
                          end if;
                    end if;
             end if;
      end process writePWM;


end rtl;
```

Appendix 2: VHDL Test Bench:

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;


entity tb_PWM is

end tb_PWM;


--to test use 'restart -f; run 10us;' command in the transcript cmd prompt


architecture test of tb_PWM is

        constant CLK_PERIOD : time := 20 ns; --this is 50MHz clk


        signal Clk              :       std_logic;

        signal nReset           :       std_logic;

        signal Address          :       std_logic_vector (1 DOWNTO 0);

        signal ChipSelect       :       std_logic;

        signal Write_en         :       std_logic;

        signal WriteData        :       integer; --std_logic_vector (7 DOWNTO 0);

        signal outpwm           :       std_logic;

begin

        dut : entity work.PWM_AvalonSlave

        port map(Clk => Clk,

                nReset => nReset,

                Address => Address,

                ChipSelect => ChipSelect,

                Write_en =>Write_en,

                WriteData => Writedata,

                outpwm => outpwm
```

```
        );

clk_gen : process
begin
        Clk <= '1';
        wait for CLK_PERIOD/2;
        Clk <= '0';
        wait for CLK_PERIOD/2;
end process clk_gen;

simulation : process
begin
        -- write period
        Address <= "00";
        ChipSelect <= '1';
        Write_en <= '1';
        WriteData <=  9; -- 100ns period
        wait for CLK_PERIOD*2;
        -- write dc
        Address <= "01";
        ChipSelect <= '1';
        Write_en <= '1';
        WriteData <= 1;
        wait for CLK_PERIOD*2;
        --write polarity
        Address <= "10";
        ChipSelect <= '1';
        Write_en <= '1';
        WriteData <= 1;
```

```
            wait for CLK_PERIOD*2;


            wait for 10 us;
        end process simulation;

end architecture test;
```

Appendix 3: C Code using PWM

```
/*
 * pwm_servo.c
 *
 *  Created on: Nov 17, 2019
 *      Author: Abhi Kamboj
 */
#include <stdio.h>
#include <inttypes.h>
#include "system.h"
#include "io.h"

/* generate_pwm:
* inputs: period in milliseconds, percent duty cycle, and polarity (either 0
or 1).
* sets the Avalon bus registers to output a pwm with these characteristics.
*/
void generate_pwm(int period, double dc, int polarity){
      uint32_t set_period = period*1000000/20; //convert ms to ns and clk
period is 20 ns so divide by that
      uint32_t set_dc = set_period*dc;
      uint32_t set_pol = polarity;

      IOWR_32DIRECT(PWM_0_BASE, 0, set_period);
      IOWR_32DIRECT(PWM_0_BASE, 1, set_dc);
      IOWR_32DIRECT(PWM_0_BASE, 2, set_pol);
}

int main()
{
      //this code should repeatedly move the arm of a servo motor back and
fourth with a short pause in between
      //the servo should be connected to GPIO_0 pin 1 header of the board and
gnd which is GPIO_0 pin 12
      int i =0;
      while(1){
            generate_pwm(20, .05, 1);
            for (i=0; i<1000000; i++);
            generate_pwm(20, .1, 1);
            for(i=0;i<1000000; i++);
      }
      return 0;
}
```