Abstract

In this lab, we set out to create a version of the first video game: Pong. This required the use of a seven-segment display, 8x16 dot matrix, 3 input buttons, and a FPGA programed using a Xilinx ISE. The 8x16 dot matrix comprised of LEDs that are controlled using multiplexing. The game functioned similarly to Pong, except the ball only moved in one dimension and the paddles were stationary. The game winner would be whoever gets to 9 points first. In the end, the system successfully showed a functioning game, albeit a bit barebones. There are bugs where when the Reset button is pressed during a round, the ball moves to the middle left of the screen.

Table of Contents

The Process

This lab was the most involved of the previous labs as it required multiple finite state machines. The overall game flow was relatively simple (Figure 1). On startup, the game is initialized by pressing the reset button (sw11). Then the ball will appear on the dot matrix on the far right side, (address 15). The game begins when player one presses their paddle button (sw12). The ball will begin to move from right to left at a speed of 2 Hz, or 2 pixels per second. When the ball reaches the far left (address 0), player two must press their paddle button (sw10) the moment the ball reaches that point. If the player holds the button before or doesn't press it in time, player one will gain a point. If player two successfully hits the ball, the ball speed increases to 3Hz towards player one and the game continues until one player gets nine points. The speed of the ball maxes out at 9Hz. When the game ends, the reset button can be pressed to start a new game. It can also be pressed at any point during the game to restart the game. This required the use of 2 random state machines

The first of the two random state machines the main game control finite state machine (Figure 2). This machine has eleven states. The left and right shift states determine the ball movement, the left and right hit zone states determine if the ball was successfully hit at the right time, the left and right sHit states send out the sHit signal if the ball successfully hit, the left and right point state sends out a signal to the score counter when a player gains a point to update the seven segment display, and lastly the left and right win state ends the game when a player gets nine points. When the game control FSM is the left or right shift state, it sends the shift signal to the second random state machine: the ball movement FSM (Figure 3). The ball movement FSM is far simpler than the game control state machine. It only has five states, the first two are a part of the initialization process. When the reset button is pressed, it resets the address and column. The rest of the states dictate how the ball moves. HoldBall keeps the ball in its current place between pulses from a variable clock. RemoveBall removes the ball from its current address then sends a signal to move to the next address. AddBall sets the new address to be the current ball position to the new address. The signal game_pulse controls the speed of the ball. Its dictated by the variable clock generator that will output a signal between 2 and 9 Hz depending on how far into the round the game is.

The other important blocks of the design are simple. The first is the score counter block. It sends out the score of the players to the seven-segment display driver to update the score on the seven-segment display. If the score somehow goes above 9, the display will only show an 'E' for error. The next block is the variable 'm' generator. 'm' is the number saved in RAM that is used to calculate the frequency of the game_pulse signal. It increases the speed after four successful hits. The block game_f_gen uses the 'm' signal to output the game pulse. The address incrementor block is what determines the address the ball is currently at depending on the signal from the ball movement FSM. The last two blocks are drivers for the seven segment display and the dot matrix display provided by Dr. Larry Aamodt.

Conclusion

The overall game works, albeit barebones. The ball moves across the dot matrix display at the desired speed with no lag, the hit registration works consistently, and the score will increment with every point gained. The seven-segment display will show the correct score and properly display the colon. We have encountered some bugs, however.

One of the bugs that we've encountered we believe are due to the derived clock speed. There was an issue we encountered where the score would stop at ten instead of nine. We believe this was because the derived clock was faster than the system could check if a player had reached nine points. This allowed the score to go up to ten before the game ended. We did a quick fix by having the system check if the score was at eight instead of nine, thus ending the game at nine. Another bug we encountered involved the reset function. When the reset button is pressed when the ball is in movement or during a round, the ball will reset its position to a little left of the middle of the screen. Pressing the reset button again will reset the ball position to the correct spot on the rightmost pixel of the dot matrix display. The cause of this bug is unknown as of the writing of this report, but it could be due to code in init_b1 or init_b2 where the value of DATA is set or with the reset_addr signal.

There are improvements that could be made to the design. The first, most basic, is the inclusion of visual paddles. The next is resetting the ball after a successful point. We wanted the round to reset after a point is scored the leftmost or rightmost pixel on the dot matrix display depending on who scored the point. If player 2 scored the point, player 1 would have the ball. If player 1 scored the point, player 2 would have the ball. The last would be to increase the maximum speed of the ball. It currently maxes out at 9 Hz, which could be hard for some players, but most people would only find it moderately difficult. Having a much higher maximum speed would provide all players some difficulty as the narrow hit window already provides a decent amount of difficulty. We didn't include these so that we could make sure that the basic game functions worked, but ran out of time to add them in. With better time management and planning of the project, we could have added these features and possibly other features like having moving paddles and/or having the ball move diagonally.

In conclusion, the final design is a functional game that works as intended, with some minor bugs. The bugs are most likely simple fixes, but we ran out of time to really look into the issue as we discovered it on the last day. Some of the improvements should theoretically not take much effort or coding to implement, like the increased ball speed or the visual paddles. The game is still fun to play regardless of the bugs or missing features. And that, to me, feels like a success.

Appendix I – Figures

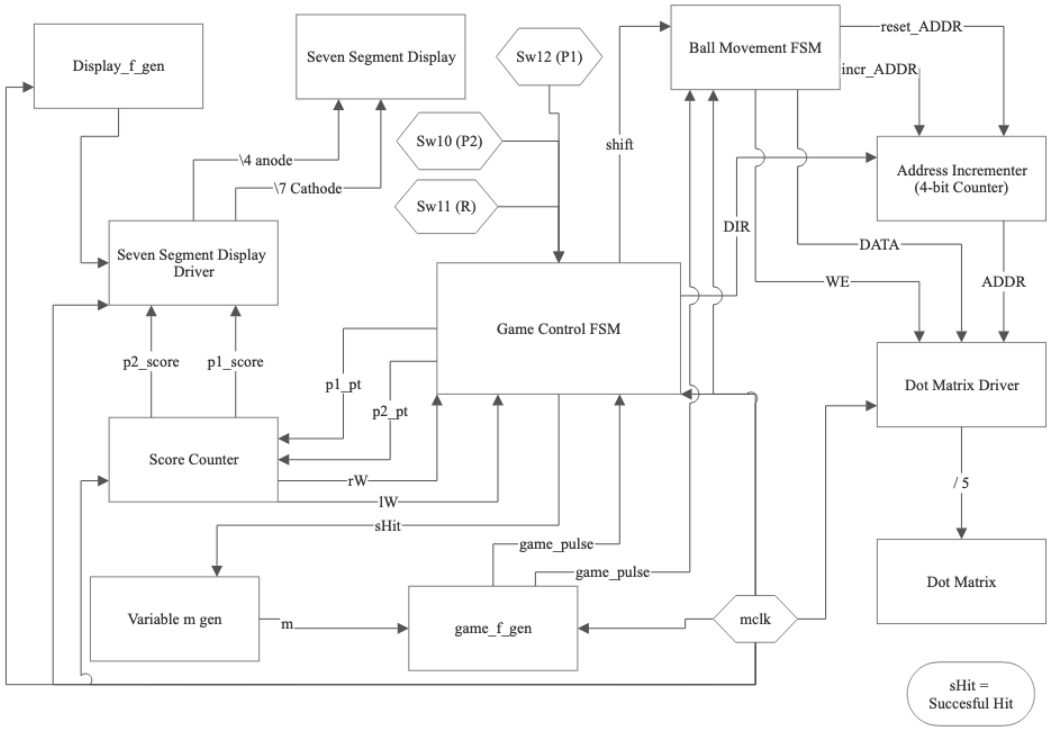

Figure 1 The Block Diagram

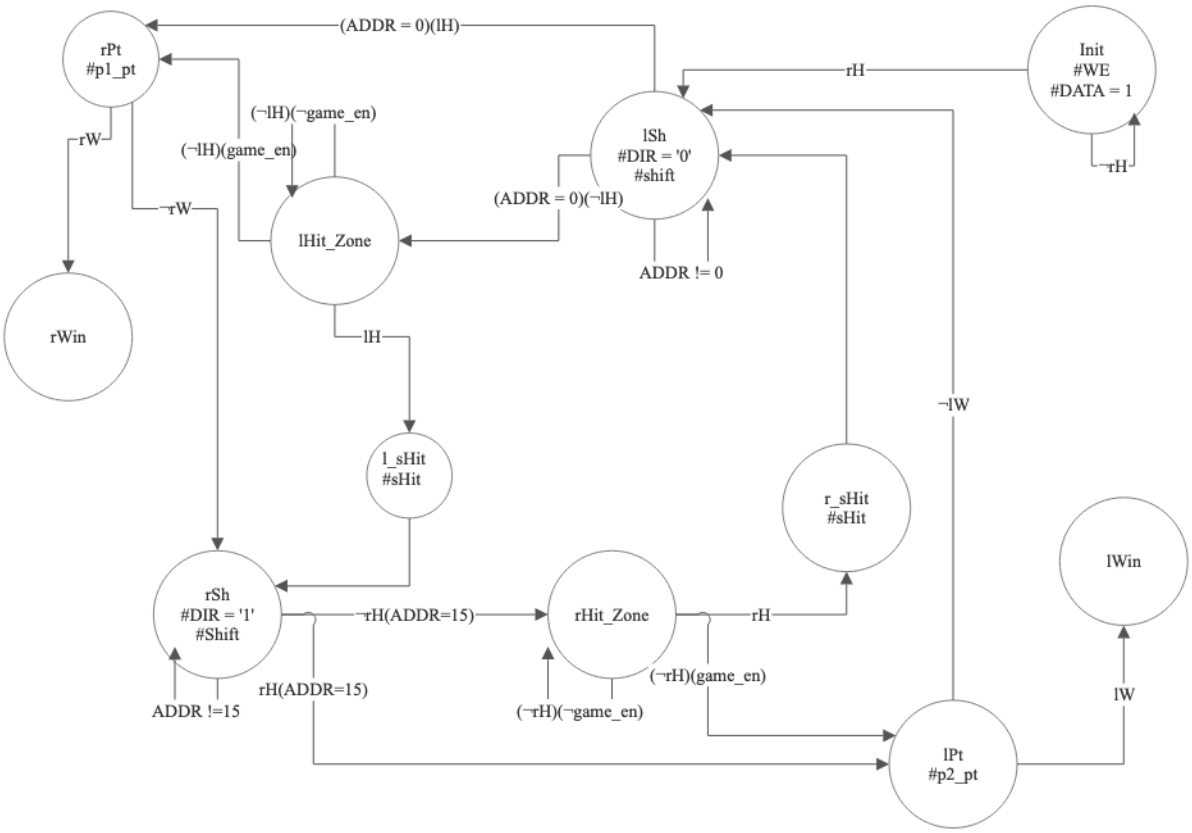

Figure 2 The Game Control FSM

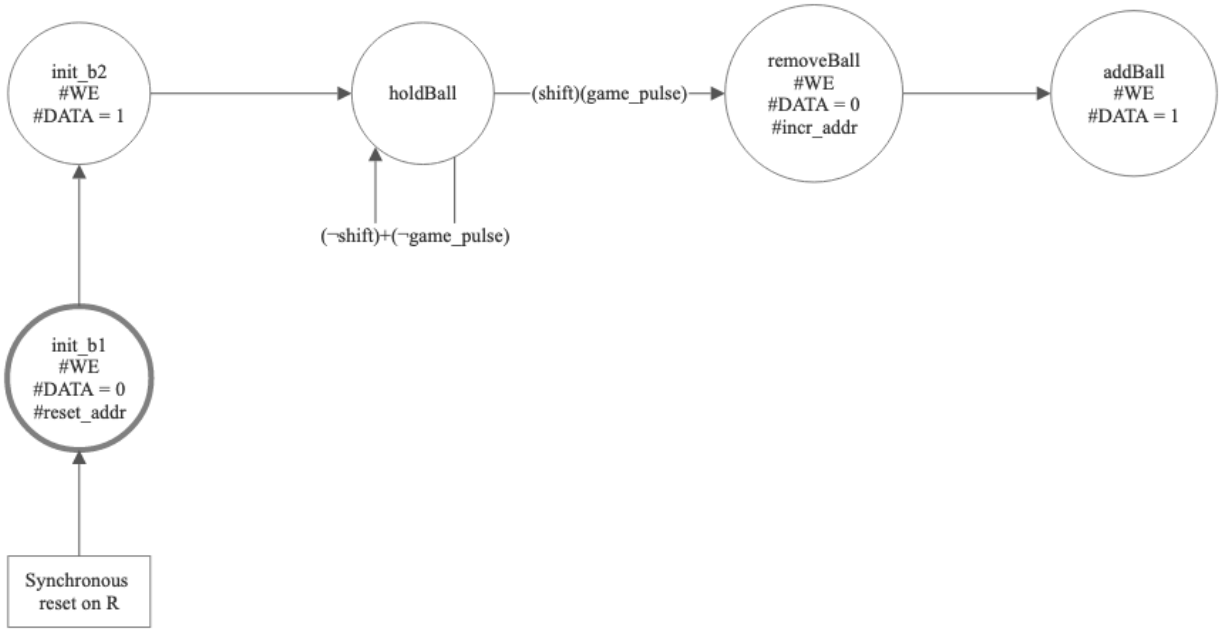

Figure 3 The Ball Movement FSM

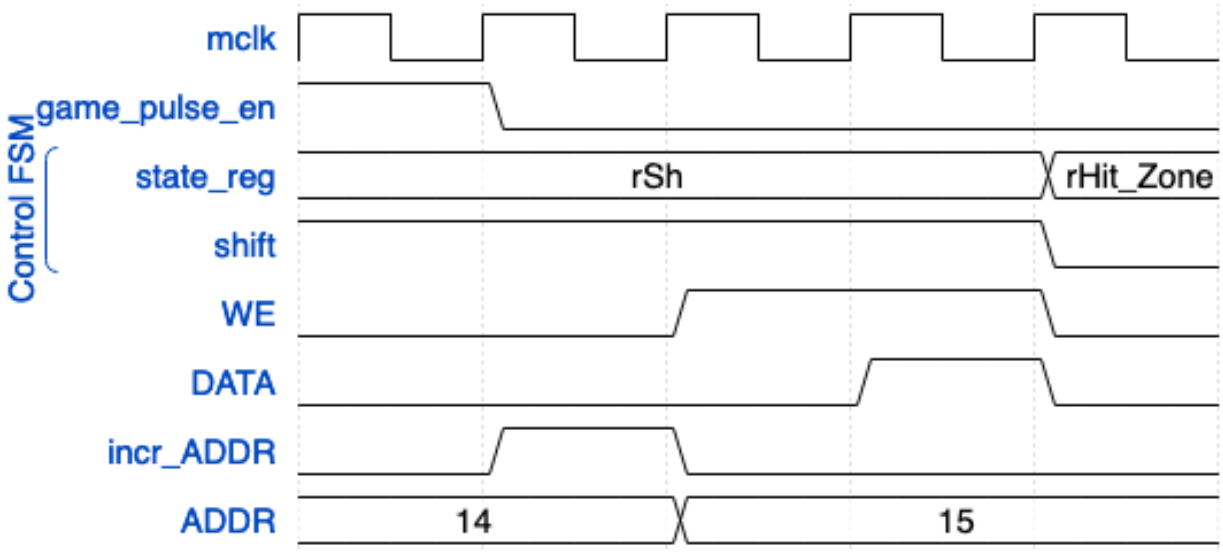

Figure 4 The Timing Diagram

Appendix II – Device Summary

Description	Stat
Slice F/F	86/126800 + 4 latches
Occupied Slices	51/15850
Slices Unrelated Logic	N/A
Bonded IOBs	49/300
Number of BUFGs	2/32
Avg Fanout	3.33

Appendix IIIa – The VHDL Code – Lab8_top_sch.vhd

```
----------------------------------------------------------------------------------
-- Company: Walla Walla University
-- Engineer: Eric Walsh and Nicholas Zimmerman
--
-- Create Date:    15:50:34 11/16/2021
-- Design Name:
-- Module Name:    Lab8_top_sch - Behavioral
-- Project Name:         Pong_1d
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity Lab8_top_sch is
        port(    mclk : in std_logic;
                        btn1, btn2, btn3 : in std_logic;
                        led : out std_logic_vector(15 downto 0);
                                --7-segment display
                                cath : out std_logic_vector(7 downto 0);
                                anode : out std_logic_vector(5 downto 1);     --We must include
unused anodes and drive them to '1'
                        extout : out std_logic_vector(8 downto 0);
                        tek4 : out std_logic_vector(1 downto 0);
                        tek1 : out std_logic_vector(7 downto 0);
                        tek5 : out std_logic);
end Lab8_top_sch;

architecture SingleFSM of Lab8_top_sch is
        ----signals should be grouped by block of signal origin----
                signal R : std_logic;    --An alias for btn2
        --next_game_f_gen
                        --signal c : unsigned(3 downto 0);      --Allows 15 frequencies, 1 to 15
```

```
		signal sHit_cnt_next, sHit_cnt_reg, sHit_cnt_next_en : unsigned(1 downto 0);
	--2 bit counter of successful hits, should be 1 bit TFF
		signal select_f_next, select_f_reg, select_f_notMax : unsigned(2 downto 0); --:=
(others=>'0');  --attempting RAM of 8 predetermined frequencies
		signal m : unsigned(25 downto 0); --:= to_unsigned(50000000,26); --26 bit
unsigned value,
		signal select_f_forOutput_next, select_f_forOutput_reg : std_logic_vector(7
downto 0); --:= (others=>'0');
	--game_f_gen  --generates constant sys_clk and variable game_pulse which counts in
units of sys_clk
		signal clk_reg, clk_next, clk_next_check : unsigned(25 downto 0); --2^26 =
67,108,864 > 50E6; 32 bits to compare with m
		--signal clk_overflow : std_logic;
		--signal t_reg, t_next: std_logic;
		--signal game_clk : std_logic;
		signal game_pulse : std_logic;
	--control_FSM
		type control_state_type is (init_c,	lSh,lHit_zone,rPt,rWin,l_sHit,
	--Seems to initialize to 1st state in list

	rSh,rHit_zone,lPt,lWin,r_sHit);
		signal control_state_reg, control_state_next : control_state_type;
		signal rHit,lHit : std_logic;	--Logical high of btn3,btn1 respectively
		signal sHit : std_logic;		--Successful hit by either player
		signal p1_pt, p2_pt : std_logic; --Increment score counter
		signal dir,shift : std_logic;	--dir: 0 shifts left, 1 shifts right
		signal DATA : std_logic_vector(7 downto 0);
		signal WE : std_logic;
	--score_counter
		signal p1_score_reg, p1_score_next, p2_score_reg, p2_score_next: unsigned(3
downto 0);	--Seems to initialize to 0
		signal rW, lW : std_logic;	--Win condition met
	--ball_movement_FSM
		type ball_state_type is (init_b1, init_b2, holdBall, removeBall, addBall);
		signal ball_state_reg, ball_state_next : ball_state_type;
		signal incr_addr : std_logic;
		signal reset_addr : std_logic;
	--address_incrementer
		signal addr_reg : unsigned(3 downto 0); --:= to_unsigned(15,4);	--Point to the
rightmost part of the screen
		signal addr_next, enabled_addr_next, overwritten_addr_next : unsigned(3 downto
0);
		signal ADDR : std_logic_vector(3 downto 0);		--std_logic_vector data type
	---Components---
		--seg7Driver
		component Seg7Driver
```

```
                port(   mclk : in std_logic;
                        p1_score, p2_score : in unsigned(3 downto 0);
                        to_anode2, to_anode4 : out std_logic;
                        to_cathode : out std_logic_vector(7 downto 0)       );
        end component;

        --dot_matrix_driver
        component led_8x16_driver
                port(   mclk:                           in  STD_LOGIC;    -- 50 Mhz
clock
                        sys_clk:                        in  STD_LOGIC;    --
clock from game
                        we:                             in  STD_LOGIC;    --
write enable
                        data_in:                        in
STD_LOGIC_VECTOR(7 downto 0);
                        wrt_addr:                   in
STD_LOGIC_VECTOR(3 downto 0);
                        out_to_display:         out STD_LOGIC_VECTOR(7
downto 0) );
        end component;
begin
        ----------------------------------------------------------------------------
        --Next_game_f_gen   (50 MHz to 1 Hz) (not yet resettable)
        ----------------------------------------------------------------------------
        --Successful Hit counter
                process(mclk)
                begin
                        if (mclk'event and mclk='1') then
                                if(R='1') then
                                        select_f_reg <= (others=>'0');
                                        select_f_forOutput_reg <= (others=>'0');
                                else
--                                      sHit_cnt_reg <= sHit_cnt_next;        --For counting
number of successful hits
                                        select_f_reg <= select_f_next;
                                        select_f_forOutput_reg <= select_f_forOutput_next;
                                end if;
                        end if;
                end process;

                --NS Logic
--              sHit_cnt_next <= sHit_cnt_next_en when (sHit='1') else sHit_cnt_reg;      --
enable
--              sHit_cnt_next_en <= (others => '0') when (sHit_cnt_reg=3) else
(sHit_cnt_reg+1);
```

```
                --Update the frequency every 4 non-consecutive successful hits
--              select_f_next <= (select_f_reg+1) when (sHit='1' and sHit_cnt_reg=3) else
select_f_reg;
        select_f_next <= select_f_reg when (select_f_reg=7) else select_f_notMax;
    --Hold value instead of cycling back to 0
        select_f_notMax <= (select_f_reg+1) when (sHit='1') else select_f_reg;

        --Output Logic

            --RAM for m                 --tried a case statement earlier
                with select_f_reg select
                    m <=    to_unsigned(25000000,26) when "000",
        --2Hz
                                        to_unsigned(16666666,26) when "001",
            --3Hz
                                        to_unsigned(12500000,26) when "010",
            --4Hz
                                        to_unsigned(10000000,26) when "011",
            --5Hz
                                        to_unsigned(8333333,26) when "100",
                --6Hz
                                        to_unsigned(7142857,26) when "101",
                --7Hz
                                        to_unsigned(6250000,26) when "110",
                --8Hz
                                        to_unsigned(5555555,26) when others; --
111         --9Hz

            --c generation  --unused
                --c <= to_unsigned(1,4);       --Magic number for now

        select_f_forOutput_next <= ('1' & select_f_forOutput_reg(7 downto 1)) when
(sHit='1') else select_f_forOutput_reg;
    ---------------------------------------------------------------------------
    --Game_f_gen (50 MHz to 1 Hz) (not yet resettable)
    ---------------------------------------------------------------------------
    process(mclk)
    begin
        if (mclk'event and mclk='1') then
            clk_reg <= clk_next;
        end if;
    end process;

    --NS Logic
        --clk_overflow <= '1' when (clk_reg >= 24999999) else '0';
```

```
		--clk_next <= (others=>'0') when (clk_overflow='1') else (clk_reg + c);	--
Possible timing hazard with c on overflow, so we won't use it
		--t_reg <= (not t_reg) when (clk_overflow='1') else t_reg;

	clk_next_check <= (others=>'0') when (clk_reg = 49999999) else clk_next;	--make
sure clk_reg is reset when m is changed
	clk_next <= (others=>'0') when (clk_reg = (m-1)) else (clk_reg + 1);

	--Output Logic
	--Clk_Buffer: BUFG  -- Put t_reg on a buffered clock line
	--		port map ( I => t_reg, O => game_clk);

	--game_pulse <= '1' when (clk_overflow='1' and t_reg='1') else '0';
	game_pulse <= '1' when (clk_reg = (m-1)) else '0';		--comparison of 26 bit
unsigned values
	--It would be a good idea to buffer the above signal signal from glitches

	--Note: There will need a comparator for variable c but not for variable m
	----------------------------------------------------------------------------
	--Score_counter
	----------------------------------------------------------------------------
	process(mclk)
	begin
		if (mclk'event and mclk='1') then
			if(R='1') then
				p1_score_reg <= (others=>'0');
				p2_score_reg <= (others=>'0');
			else
				p1_score_reg <= p1_score_next;
				p2_score_reg <= p2_score_next;
			end if;
		end if;
	end process;

	--NS Logic
	p1_score_next <= p1_score_reg+1 when (p1_pt='1') else p1_score_reg;
	p2_score_next <= p2_score_reg+1 when (p2_pt='1') else p2_score_reg;

	--Out Logic
	rW <= '1' when (p1_score_reg=8) else '0';	--right player wins, should win on 9... but
timing
	lW <= '1' when (p2_score_reg=8) else '0';	--left player wins, should win on 9... but
timing

	----------------------------------------------------------------------------
	--Control_FSM
```

```
----------------------------------------------------------------------------
process(mclk)
begin
	--if (R='1') then			--asynchronous reset?
		--control_state_reg <= init;
	--end if?? else?
		if (mclk'event and mclk='1') then
			if (R='1') then		--synchronous reset
				control_state_reg <= init_c;
			else
				control_state_reg <= control_state_next;
			end if;
		end if;
	--end if??
end process;

--NS Logic
process(control_state_reg)
begin
	case control_state_reg is
		when init_c =>
			if (rHit='1') then
				control_state_next <= lSh;
			else
				control_state_next <= init_c;
			end if;
		when lSh =>
			if (addr_reg /= 0) then
				control_state_next <= lSh;
			else	--addr_reg=0
				if (lHit='1') then
					control_state_next <= rPt;
				else
					control_state_next <= lHit_zone;
				end if;
			end if;
		when lHit_zone =>
			if (lHit='1') then
				control_state_next <= l_sHit;
			else	--lHit='0'
				if (game_pulse='1') then
					control_state_next <= rPt;
				else
					control_state_next <= lHit_zone;
				end if;
			end if;
```

```
                when rPt =>
                        if (rW='1') then
                                control_state_next <= rWin;
                        else
                                control_state_next <= rSh;
                        end if;
                when rWin =>
                when l_sHit =>
                        control_state_next <= rSh;
                when rSh =>
                        if (addr_reg /= 15) then
                                control_state_next <= rSh;
                        else    --addr_reg=15
                                if (rHit='1') then
                                        control_state_next <= lPt;
                                else
                                        control_state_next <= rHit_zone;
                                end if;
                        end if;
                when rHit_zone =>
                        if (rHit='1') then
                                control_state_next <= r_sHit;
                        else    --lHit='0'
                                if (game_pulse='1') then
                                        control_state_next <= lPt;
                                else
                                        control_state_next <= rHit_zone;
                                end if;
                        end if;
                when lPt =>
                        if (lW='1') then
                                control_state_next <= lWin;
                        else
                                control_state_next <= lSh;
                        end if;
                when lWin =>
                when r_sHit =>
                        control_state_next <= lSh;
        end case;
end process;

--Out Logic
process(control_state_reg)
begin
        shift   <= '0';
        dir     <= '0';
```

```
		p1_pt	<= '0';
		p2_pt	<= '0';
		sHit <= '0';		--Leaving this out was a beginner's mistake! It broke the clock.
		case control_state_reg is
			when init_c =>
			when lSh =>
				shift <= '1';
				--dir <= '0';
			when lHit_zone =>
			when rPt =>
				p1_pt <= '1';
			when rWin =>
			when l_sHit =>
				sHit <= '1';
			when rSh =>
				shift <= '1';
				dir <= '1';
			when rHit_zone =>
			when lPt =>
				p2_pt	<= '1';
			when lWin =>
			when r_sHit =>
				sHit <= '1';
		end case;
	end process;
	--------------------------------------------------------------------------------
	--Ball_Movement_FSM
	--------------------------------------------------------------------------------
	process(mclk)
	begin
		if (mclk'event and mclk='1') then
				if (R='1') then		--synchronous reset
					ball_state_reg <= init_b1;
				else
					ball_state_reg <= ball_state_next;
				end if;
		end if;
	end process;

	--NS Logic
	process(ball_state_reg)		--We're choosing not to have game_pulse in the sensitivity list because glitches
	begin
		case ball_state_reg is
			when init_b1 =>
```

```
                        ball_state_next <= init_b2;
                when init_b2 =>
                        ball_state_next <= holdBall;
                when holdBall =>
                        if (shift='1' and game_pulse='1') then
                                ball_state_next <= removeBall;
                        else
                                ball_state_next <= holdBall;
                        end if;
                when removeBall =>
                        ball_state_next <= addBall;
                when addBall =>
                        ball_state_next <= holdBall;
        end case;
end process;

--Out Logic
process(ball_state_reg)
begin
        incr_addr <= '0';
        DATA <= "10101010";
        WE <= '0';
        reset_addr <= '0';
        case ball_state_reg is
                when init_b1 =>
                        --Change the next address
                        reset_addr <= '1';      --Change the next address
                        --Write in the the current addressed location
                        DATA <= (others=>'0');      --Set the currently addressed column
                        WE <= '1';
                when init_b2 =>
                        DATA <= "00000001";
                        WE <= '1';
                when holdBall =>
                when removeBall =>
                        incr_addr <= '1';
                        DATA <= (others=>'0');      --Redundant statement?
                        WE <= '1';
                        --Wait longer to write value? (use a counter)
                when addBall =>
                        DATA <= "00000001";
                        WE <= '1';
                        --Wait longer to write value? (use a counter)
        end case;
end process;
```

```
-------------------------------------------------------------------------------
	--Address_incrementer
	-------------------------------------------------------------------------------
	process(mclk)
	begin
			if (mclk'event and mclk='1') then
					addr_reg <= addr_next;
			end if;
	end process;

	--NS Logic
		overwritten_addr_next <= to_unsigned(15,4) when (reset_addr ='1') else
addr_next;	--override
		addr_next <= (enabled_addr_next) when (incr_addr='1') else addr_reg;
	--enable
		enabled_addr_next <= (addr_reg+1) when (dir='1') else (addr_reg-1);

	--Output Logic
	ADDR <= std_logic_vector(addr_reg);

	-------------------------------------------------------------------------------
	--Top level code
	-------------------------------------------------------------------------------
	rHit <= not btn3;	--player 1
	lHit <= not btn1;	--player 2
	R <= not btn2;--Initialize/Reset

	Seg7Driver_1: Seg7Driver port map( mclk => mclk,

	p1_score => p1_score_reg,

	p2_score => p2_score_reg,

	to_anode2 => anode(2),

	to_anode4 => anode(4),

	to_cathode => cath		);
	anode(1) <= '1';	--unused
	anode(3) <= '1';	--unused
	anode(5) <= '1';	--unused

	dotMatrixDriver : led_8x16_driver
		port map (	mclk => mclk,
					sys_clk => mclk,
					we => WE,
```

```
                                        data_in => DATA,
                                        wrt_addr => ADDR,
                                        out_to_display => extout(7 downto 0) );

            --Score
            led(7 downto 0) <= std_logic_vector(p2_score_reg) &
std_logic_vector(p1_score_reg);
            --led(15 downto 8) <= std_logic_vector(m(15 downto 8));
            led(15 downto 8) <= lW & "000000" & rW;
            --led(15 downto 8) <= select_f_forOutput_reg;

        tek4(0) <= WE;
        tek1(0) <= WE;
        tek4(1) <= game_pulse;      --was DATA(0)
        tek1(1) <= game_pulse;      --was DATA(0)
        tek1(2) <= game_pulse;
        extout(8) <= mclk;
        tek1(7) <= mclk;
        tek5 <= mclk;

end SingleFSM;
```

Appendix IIb – VHDL Code – bcd_to_7seg.vhd

```
----------------------------------------------------------------------------------
-- Company:       Walla Walla University
-- Engineer:      L.Aamodt
--
-- Create Date:   16:42:06 10/13/2020
-- Design Name:   Binary to 7-segment decoder
-- Module Name:   bcd2_7seg - Behavioral
-- Project Name:
--
-- Description:   Converts 4-bit binary representing one BCD digit
--                to 7-segment display encoding
-- Dependencies:  WWU FPGA3 board
--
-- Revision:
-- Revision 1.0 - File Created
-- Additional Comments:
--   Cathode signals asserted low to turn on the segment.
--   The cathode that controls the decimal point is always turned off.
--   Segment a of the display is cath_out(0) and segment g is cath_out(6).
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity bcd2_7seg is
    port ( data : in  STD_LOGIC_VECTOR (3 downto 0);
           cath_out : out  STD_LOGIC_VECTOR (7 downto 0));
end bcd2_7seg;

architecture Behavioral of bcd2_7seg is

begin
    process(data)
    begin
        case data is
            when "0000" => cath_out <= "11000000";
            when "0001" => cath_out <= "11111001";
            when "0010" => cath_out <= "10100100";
            when "0011" => cath_out <= "10110000";
            when "0100" => cath_out <= "10011001";
            when "0101" => cath_out <= "10010010";
            when "0110" => cath_out <= "10000010";
            when "0111" => cath_out <= "11111000";
            when "1000" => cath_out <= "10000000";
            when "1001" => cath_out <= "10011000";
            when others => cath_out <= "10000110";
```

```
        end case;
    end process;
end Behavioral;
```

Appendix IIIc – VHDL Code – led_8x16_driver.vhd

```
----------------------------------------------------------------------------------
-- Company:        Walla Walla University
-- Engineer:       L.Aamodt
--
-- Create Date:    21:14:50 12/06/2015
-- Design Name:      led_8x16_driver
-- Module Name:    led_8x16_driver - Behavioral
-- Project Name:     led_8x16_driver
-- Target Devices: Spartan 3e
-- Tool versions:    ISE 14.7
-- Description:      Driver to send data to a 8x16 LED array arranged as
--                   8 rows of 16 bits (i.e. 16 columns).  Data is read
--                   asynchronously as bytes from a 16 byte RAM, bits in
--                   the byte are selected sequentially with a multiplexer,
--                   and together with an address are used to select one
--                   LED at a time for potential turn on.  All 128 bits are
--                   scanned about 79 times per second.
--
-- Dependencies:
--
-- Revision:
-- Revision 2.0     12/7/16
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity led_8x16_driver is
   Port (mclk:          in  STD_LOGIC;    -- 50 Mhz clock
         sys_clk:       in  STD_LOGIC;    -- clock from game
         we:            in  STD_LOGIC;    -- write enable
         data_in:       in  STD_LOGIC_VECTOR(7 downto 0);
         wrt_addr:      in  STD_LOGIC_VECTOR(3 downto 0);
         out_to_display: out STD_LOGIC_VECTOR(7 downto 0));
end led_8x16_driver;

architecture Behavioral of led_8x16_driver is
   signal r_reg1:   unsigned(11 downto 0);
   signal r_next1:  unsigned(11 downto 0);
   signal led_addr: unsigned(6 downto 0);
```

```
    signal led_next: unsigned(6 downto 0);
    signal ram_data: STD_LOGIC_VECTOR(7 downto 0);
    signal clk_20k:  STD_LOGIC;
    signal clk_10k:  STD_LOGIC;
    signal Bclk_10k: STD_LOGIC;
    signal led_data: STD_LOGIC;

    type ram_type is array (0 to 15) of std_logic_vector(7 downto 0);
    signal RAM: ram_type := (
--      X"81", X"82", X"84", X"88",     -- initial data, pattern 1
--      X"44", X"42", X"41", x"40",
--      X"81", X"82", X"84", X"88",
--      X"44", X"42", X"41", x"40"

--      X"01", X"02", X"04", X"08",     -- initial data, pattern 2
--      X"10", X"20", X"40", x"80",
--      X"80", X"40", X"20", x"10",
--      X"08", X"04", X"02", x"01"

         X"00", X"00", X"00", X"00",       -- initial data, pattern 3
        X"00", X"00", X"00", x"00",
        X"00", X"00", X"00", x"00",
        X"00", X"00", X"00", x"00"
    );

begin

-------------------------------------------------------------
--    Scan clock generator - 10khz output to run LED update
-------------------------------------------------------------
    process(mclk,r_reg1)  -- create 20000hz signal
    begin
        if (mclk'event and mclk='1') then
            if (r_reg1 = 2499) then
                r_reg1 <= (others=>'0');
            else
                r_reg1 <= r_next1;
            end if;
        end if;
    end process;
    r_next1 <= r_reg1 + 1;
    clk_20k <= '1' when r_reg1 = 2499 else '0';

    process(mclk,clk_20k)  -- create square 10khz clock
    begin
        if (mclk'event and mclk='1') then
```

```
            if (clk_20k = '1') then
               clk_10k <= NOT clk_10k;
            end if;
         end if;
      end process;

      Clk_Buffer: BUFG        -- create 10khz buffered clock
         port map ( I => clk_10k, O => Bclk_10k);

-------------------------------------------------------------
-- Dual ported RAM that holds an image of what is to be displayed
-- synchronous write and asynchronous read
-------------------------------------------------------------
      process(sys_clk)
      begin
         if (sys_clk'event and sys_clk = '1') then
            if (we = '1') then
               RAM(to_integer(unsigned(wrt_addr))) <= data_in;
            end if;
         end if;
      end process;
      ram_data <= RAM(to_integer(unsigned(led_addr(6 downto 3))));

-------------------------------------------------------------
-- LED address generator, a 7 bit binary counter
-------------------------------------------------------------
      process(Bclk_10k)
      begin
         if (Bclk_10k'event and Bclk_10k='1') then
            if (led_addr = 127) then
               led_addr <= (others=>'0');
            else
               led_addr <= led_next;
            end if;
         end if;
      end process;
      led_next <= led_addr + 1;
      out_to_display(7 downto 1) <= std_logic_vector(led_addr);

-------------------------------------------------------------
-- Multiplexer to get one led data bit from a data byte
-------------------------------------------------------------
      process(led_addr, ram_data)
      begin
         case led_addr(2 downto 0) is
            when "000" =>
```

```
            led_data <= ram_data(7);
         when "001" =>
            led_data <= ram_data(6);
         when "010" =>
            led_data <= ram_data(5);
         when "011" =>
            led_data <= ram_data(4);
         when "100" =>
            led_data <= ram_data(3);
         when "101" =>
            led_data <= ram_data(2);
         when "110" =>
            led_data <= ram_data(1);
         when others =>
            led_data <= ram_data(0);
      end case;
   end process;
   out_to_display(0) <= led_data;
end Behavioral;
```

Appendix IIId – VHDL Code – Seg7Driver.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    15:35:14 12/12/2021
-- Design Name:
-- Module Name:    Seg7Driver - BasedOnLab5
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
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 Seg7Driver is
    port(   mclk : in std_logic;
            p1_score, p2_score : in unsigned(3 downto 0);
            to_anode2, to_anode4 : out std_logic;
            to_cathode : out std_logic_vector(7 downto 0)   );
end Seg7Driver;

architecture BasedOnLab5 of Seg7Driver is
    --10kHz TFF used to toggle between 2 anodes
    signal cnt_reg, cnt_next : unsigned(12 downto 0);   --2^13=8192
    signal t_reg, t_next : std_logic;

    --Routing
    signal binary_under_1001 : std_logic_vector(3 downto 0);

    --Components
        component bcd2_7seg
```

```
        port (  data : in  STD_LOGIC_VECTOR (3 downto 0);
                cath_out : out  STD_LOGIC_VECTOR (7 downto 0)   );
    end component;
begin
  --1 bit counter
    --Register
    process(mclk)
    begin
      if (mclk'event and mclk='1') then
        cnt_reg <= cnt_next;
        t_reg <= t_next;
      end if;
    end process;

    --NS Logic
    cnt_next <= (others=>'0') when (cnt_reg=2499) else (cnt_reg+1);
    t_next <= (not t_reg) when (cnt_reg=2499) else t_reg;          --10kHz

  --Generate to_cathode
    binary_under_1001 <= std_logic_vector(p1_score) when (t_reg='1') else
std_logic_vector(p2_score);
    bcdTo7SegDecoder: bcd2_7seg port map (binary_under_1001,to_cathode);

  --Generate to_anode
    to_anode2 <= t_reg;         --p2, pulled down to 0(activated) when cnt_reg=0which selects
p2_score
    to_anode4 <= not t_reg;     --p1, pulled down to 0(activated) when cnt_reg=1 which selects
p1_score

end BasedOnLab5;
```