CpE 487 Digital Design Lab

Lab 6: Video Game "PONG"

1. Introduction

In this lab, we will extend the FPGA code we developed in Labs 3 and 4 (Bouncing Ball) to build a simple version of the 1970's arcade game known as "PONG". In addition to the bouncing ball, we will generate an on-screen bat which will be controlled by a potentiometer. The potentiometer generates a DC voltage which is sampled into the FPGA using an outboard analog to digital converter. The A/D converter generates a serial data stream that we will convert into the on-screen bat position. Unlike the original game, this version of PONG has only one player. The object of the game is to keep the ball in play, similar to someone hitting a ball against a practice wall.

2. ADC Interface

We will be using a 2 channel, 12-bit Analog to Digital Converter module *PmodAD1* which plugs into one of the PMOD connectors on the *Nexys2* board. The module and its connections are shown in Figure 1. The module uses two Analog Devices AD7476 ADC chips. Documentation on the module and the chip can be found on the course website.

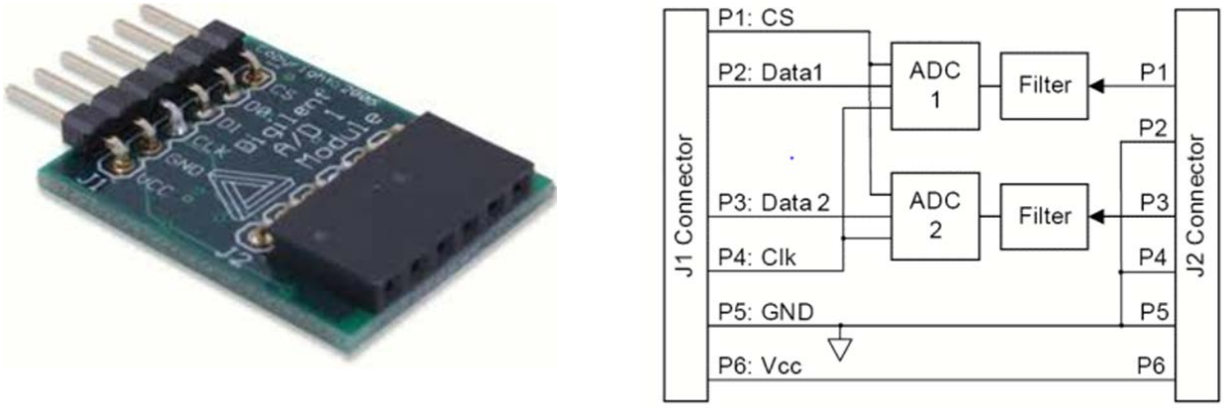

Figure 1 PModAD1 Dual 12-bit ADC Module

Each channel of the ADC module takes as input an analog voltage in the range of 0 – 3.3V. Whenever the *CS* input goes low, the ADC uses successive approximation to convert the analog input voltage into an unsigned 12-bit data value. An input of 0V generates a digital output of x"000". An input of 3.3V, generates a digital output of x"FFF". We will only be using one of these ADC channels – ADC2.

In order to minimize the number of I/O pins on the ADC chip and the module, data is output to the FPGA in serial format. Data is shifted out using the serial clock *SCLK* (just shown as *Clk* in Figure 1). In our application, *SCLK* runs at 1.56 MHz. The *CS* input to the ADC is taken low for

16 *SCLK* cycles. The ADC outputs one bit on each of these cycles. The first 4 cycles output a '0'. The remaining 12 cycles output the 12-bit data value, MSBit first

Figure 2 shows the required timing relationship between *CS*, *SCLK* and the serial data. The FPGA will be programmed to change *CS* on the rising edge of *SCLK*. Note that output data from the ADC changes on the falling edge of *SCLK*.

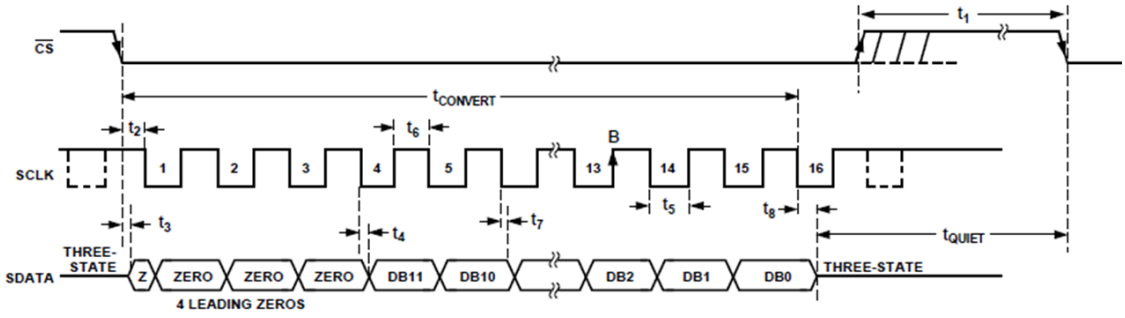

Figure 2 AD7476 Serial Timing Diagram

3. Hardware Setup

Plug the VGA monitor into the VGA port on the *Nexys2* board as shown in Figure 4.

A 5 *k*Ω potentiometer is used to control the bat position by delivering a varying voltage to the ADC as shown in Figure 3. Plug the *PmodAD1* module into PMOD jack *JA1* on the *Nexys2* boards as shown in Figure 4. Note that the *PmodAD1* has only 6 pins whereas the jack JA1 has 12 sockets (2 rows of 6). Plug the module into the top row of sockets on JA1. Please be careful not to bend the pins on the *PmodAD1* module. Now plug the cable from the potentiometer into the J2 connector on the *PmodAD1* module, so that the white orientation symbol is on the left as shown in the Figure.

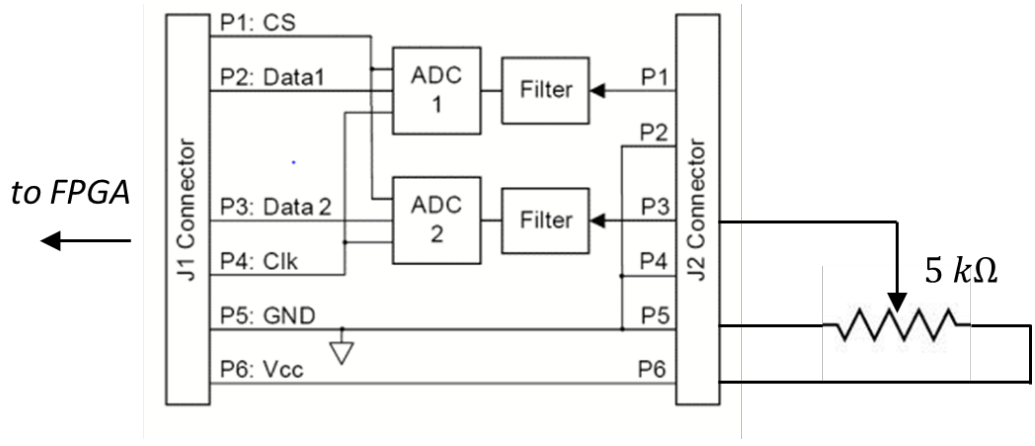

Figure 3 Potentiometer used to generate varying voltage input to ADC

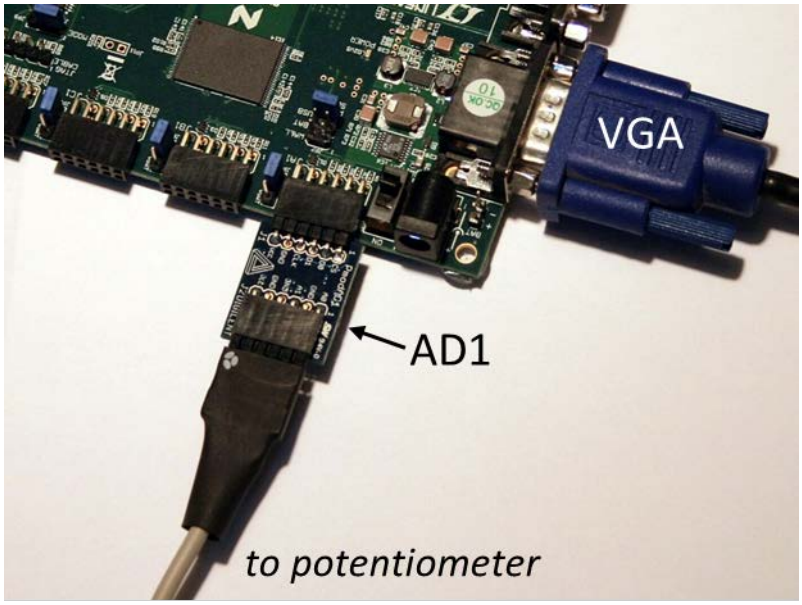

Figure 4 PmodAD1 Module inserted in PMOD jack JA1 with connector cable to potentiometer

4. Configuring the FPGA

4.1 Create a New Project

Use the Xilinx ISE software to create a new project named *Pong* using the same *project settings* as in Labs 1 and 2.

4.2 Add Source for "vga_sync"

Create a new VHDL source module called *vga_sync* and load the following source code into the edit window. This is the same code we used in Labs 3 and 4 to generate VGA sync and timing signals. Expand the **Synthesize** command in the *Process* window and run **Check Syntax** to verify that you have entered the code correctly.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity vga_sync is
    Port ( clock_25MHz : in  STD_LOGIC;
           red : in  STD_LOGIC;
           green : in  STD_LOGIC;
           blue : in  STD_LOGIC;
           red_out : out  STD_LOGIC;
```

```
        green_out : out  STD_LOGIC;
        blue_out : out  STD_LOGIC;
        hsync : out  STD_LOGIC;
        vsync : out  STD_LOGIC;
        pixel_row : out  STD_LOGIC_VECTOR (9 downto 0);
        pixel_col : out  STD_LOGIC_VECTOR (9 downto 0));
end vga_sync;

architecture Behavioral of vga_sync is
signal h_cnt, v_cnt: STD_LOGIC_VECTOR (9 DOWNTO 0);
begin
sync_pr:        process
variable video_on: STD_LOGIC;
        begin
                wait until rising_edge(clock_25MHz);

                -- Generate Horizontal Timing Signals for Video Signal
                -- h_cnt counts pixels across line (800 total = 640 active + extras for sync and blanking)
                -- Active picture for 0 <= h_cnt <= 639
                -- Hsync for 659 <= h_cnt <= 755
                if h_cnt >= 799 then
                        h_cnt <= "0000000000"; else
                        h_cnt <= h_cnt+1;
                end if;
                if (h_cnt >= 659) and (h_cnt <= 755) then
                        hsync <= '0'; else
                        hsync <= '1';
                end if;

                -- Generate Vertical Timing Signals for Video Signal
                -- v_cnt counts lines down screen (525 total = 480 active + extras for sync and blanking)
                -- Active picture for 0 <= v_cnt <= 479
                -- Vsync for 493 <= h_cnt <= 494
                if (v_cnt >= 524) and (h_cnt = 699) then
                        v_cnt <= "0000000000";
                elsif h_cnt = 699 then
                        v_cnt <= v_cnt+1;
                end if;
                if (v_cnt >= 493) and (v_cnt <= 494) then
                        vsync <= '0'; else
                        vsync <= '1';
                end if;

                -- Generate Video Signals and Pixel Address
                if (h_cnt <= 639) and (v_cnt <= 479) then
                        video_on := '1'; else
                        video_on := '0';
                end if;
                pixel_col <= h_cnt;
                pixel_row <= v_cnt;
```

```
            -- Register video to clock edge and suppress video during blanking and sync periods
            red_out <= red and video_on;
            green_out <= green and video_on;
            blue_out <= blue and video_on;
        end process;
end Behavioral;
```

4.2 Add Source for "bat_n_ball"

Create a new VHDL source module called *bat_n_ball* and load the following source code into the edit window. This is a modified version of the ball module that we used in Labs 3 and 4. Expand the **Synthesize** command in the *Process* window and run **Check Syntax** to verify that you have entered the code correctly.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity bat_n_ball is
    Port ( v_sync : in  STD_LOGIC;
           pixel_row : in  STD_LOGIC_VECTOR(9 downto 0);
           pixel_col : in  STD_LOGIC_VECTOR(9 downto 0);
           bat_x : in STD_LOGIC_VECTOR (9 downto 0);  -- current bat x position
           serve: in STD_LOGIC;                       -- initiates serve
           red : out  STD_LOGIC;
           green : out  STD_LOGIC;
           blue : out  STD_LOGIC);
end bat_n_ball;

architecture Behavioral of bat_n_ball is

constant bsize: integer:=8;                 -- ball size in pixels
constant bat_w: integer:=20;                -- bat width in pixels
constant bat_h: integer:=3;                 -- bat height in pixels

-- distance ball moves each frame
constant ball_speed: STD_LOGIC_VECTOR (9 downto 0) := CONV_STD_LOGIC_VECTOR (6,10);

signal ball_on: STD_LOGIC;                  -- indicates whether ball is at current pixel position
signal bat_on: STD_LOGIC;                   -- indicates whether bat at over current pixel position
signal game_on: STD_LOGIC := '0';           -- indicates whether ball is in play

-- current ball position - intitialized to center of screen
signal ball_x: STD_LOGIC_VECTOR(9 downto 0):= CONV_STD_LOGIC_VECTOR(320,10);
signal ball_y: STD_LOGIC_VECTOR(9 downto 0):= CONV_STD_LOGIC_VECTOR(240,10);
```

```
-- bat vertical position
constant bat_y: STD_LOGIC_VECTOR(9 downto 0):= CONV_STD_LOGIC_VECTOR(400,10);

-- current ball motion - initialized to (+ ball_speed) pixels/frame in both X and Y directions
signal ball_x_motion, ball_y_motion: STD_LOGIC_VECTOR(9 downto 0):= ball_speed;

begin
	red <= not bat_on;		-- color setup for red ball and cyan bat on white background
	green <= not ball_on;
	blue <= not ball_on;


	-- process to draw round ball
	-- set ball_on if current pixel address is covered by ball position
balldraw: process (ball_x, ball_y, pixel_row, pixel_col) is
	variable vx, vy: STD_LOGIC_VECTOR (9 downto 0);
	begin
		if pixel_col <= ball_x then			-- vx = |ball_x - pixel_col|
			vx := ball_x - pixel_col; else
			vx := pixel_col - ball_x;
		end if;
		if pixel_row <= ball_y then			-- vy = |ball_y - pixel_row|
			vy := ball_y - pixel_row; else
			vy := pixel_row - ball_y;
		end if;

		if((vx*vx) + (vy*vy)) < (bsize*bsize) then		-- test if radial distance < bsize
			ball_on <= game_on; else
			ball_on <= '0';
		end if;
	end process;

	-- process to draw bat
	-- set bat_on if current pixel address is covered by bat position
batdraw: process (bat_x, pixel_row, pixel_col) is
	variable vx, vy: STD_LOGIC_VECTOR (9 downto 0);
	begin
		if ((pixel_col >= bat_x - bat_w) or (bat_x <= bat_w)) and
			pixel_col <= bat_x + bat_w and
			pixel_row >= bat_y - bat_h and
			pixel_row <= bat_y + bat_h then bat_on <= '1';
			else bat_on <= '0';
		end if;
	end process;

-- process to move ball once every frame (i.e. once every vsync pulse)
mball: process
	variable temp: STD_LOGIC_VECTOR (10 downto 0);
	begin
		wait until rising_edge(v_sync);
```

```
            if serve = '1' and game_on = '0' then                -- test for new serve
                game_on <= '1';
                ball_y_motion <= (not ball_speed) + 1;  --  set vspeed to (- ball_speed) pixels
            elsif ball_y <= bsize then                           -- bounce off top wall
                ball_y_motion <= ball_speed;    --  set vspeed to (+ ball_speed) pixels
            elsif ball_y + bsize >= 480 then             -- if ball meets bottom wall
                ball_y_motion <= (not ball_speed) + 1;  --  set vspeed to (- ball_speed) pixels
                game_on <= '0';                          -- and make ball disappear
            end if;

            -- allow for bounce off left or right of screen
            if ball_x + bsize >= 640 then                -- bounce off right wall
                ball_x_motion <= (not ball_speed) +1;   -- set hspeed to (- ball_speed) pixels
            elsif ball_x <= bsize then                   -- bounce off left wall
                ball_x_motion <= ball_speed;    -- set hspeed to (+ ball_speed) pixels
            end if;

            -- allow for bounce off bat
            if (ball_x + bsize/2) >= (bat_x - bat_w) and
                (ball_x - bsize/2) <= (bat_x + bat_w) and
                (ball_y + bsize/2) >= (bat_y - bat_h) and
                (ball_y - bsize/2) <= (bat_y + bat_h) then
                ball_y_motion <= (not ball_speed) + 1;  -- set vspeed to (- ball_speed) pixels
            end if;

                -- compute next ball vertical position
                -- variable temp adds one more bit to calculation to fix unsigned underflow problems
                -- when ball_y is close to zero and ball_y_motion is negative
            temp := ('0' & ball_y) + (ball_y_motion(9) & ball_y_motion);
            if game_on = '0' then ball_y <= CONV_STD_LOGIC_VECTOR(440,10);
            elsif temp(10) = '1' then ball_y <= (others=>'0');
            else ball_y <= temp(9 downto 0);
            end if;

                -- compute next ball horizontal position
                -- variable temp adds one more bit to calculation to fix unsigned underflow problems
                -- when ball_x is close to zero and ball_x_motion is negative
            temp := ('0' & ball_x) + (ball_x_motion(9) & ball_x_motion);
            if temp(10) = '1' then ball_x <= (others=>'0');
            else ball_x <= temp(9 downto 0);
            end if;

        end process;

end Behavioral;
```

This module draws the bat and ball on the screen and also causes the ball to bounce (by reversing its speed) when it collides with the bat or one of the walls. It also uses a variable *game_on* to indicate whether the ball is currently in play. When *game_on* = ‘1’, the ball is visible and

bounces off the bat and/or the top, left and right walls. If the ball hits the bottom wall, *game_on* is set to '0'. When *game_on* = '0', the ball is not visible and waits to be served. When the *serve* input goes high, *game_on* is set to '1' and the ball becomes visible again.

4.3 Add Source for "adc_if"

Create a new VHDL source module called *adc_if* and load the following source code into the edit window. Expand the **Synthesize** command in the *Process* window and run **Check Syntax** to verify that you have entered the code correctly.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adc_if is
   Port ( SCK: in  STD_LOGIC;          -- serial clock that goes to ADC
          SDATA1 : in  STD_LOGIC;      -- serial data channel 1
          SDATA2 : in  STD_LOGIC;      -- serial data channel 2
          CS: in  STD_LOGIC;           -- chip select that initiates A/D conversion
          data_1 : out STD_LOGIC_VECTOR(11 downto 0);    -- parallel 12-bit data ch1
          data_2 : out STD_LOGIC_VECTOR(11 downto 0));   -- parallel 12-bit data ch2
end adc_if;

architecture Behavioral of adc_if is
signal pdata1, pdata2: std_logic_vector (11 downto 0); -- 12-bit shift registers
begin

      -- this process waits for CS=0 and then clocks serial data from ADC into shift register
      --  MSBit first. After 16 SCK's, four leading zeros will have fallen out of the most significant
      -- end of the shift register and the register will contain the parallel12-bit data
adpr: process
         begin
                  wait until falling_edge (SCK);
                  if CS='0' then
                           pdata1 <= pdata1 (10 downto 0) & SDATA1;
                           pdata2 <= pdata2 (10 downto 0) & SDATA2;
                  end if;
         end process;

      -- this process waits for rising edge of CS and then loads parallel data
      -- from shift register into appropriate output port
sync: process
                  begin
                           wait until rising_edge (CS);
                           data_1 <= pdata1;
                           data_2 <= pdata2;
                  end process;
```

```
end Behavioral;
```

This module converts the serial data from both channels of the ADC into 12-bit parallel format. When the *CS* line of the ADC is taken low, it begins a conversion and serially outputs a 16-bit quantity on the next 16 falling edges of the ADC serial clock. The data consists of 4 leading zeros followed by the 12-bit converted value. These 16 bits are loaded into a 12-bit shift register from the least significant end. The top 4 zeros fall off the most significant end of the shift register leaving the 12-bit data in place after 16 clock cycles. When CS goes high, this data is synchronously loaded into the two 12-bit parallel outputs of the module.

4.4 Add Source for top level “pong”

Create a new VHDL source module called *pong* and load the following source code into the edit window. Expand the **Synthesize** command in the *Process* window and run **Check Syntax** to verify that you have entered the code correctly.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pong is
   Port ( clk_50MHz : in  STD_LOGIC;                              -- system clock
          VGA_red : out  STD_LOGIC_VECTOR (2 downto 0);    -- VGA outputs
          VGA_green : out  STD_LOGIC_VECTOR (2 downto 0);
          VGA_blue : out  STD_LOGIC_VECTOR (1 downto 0);
          VGA_hsync : out  STD_LOGIC;
          VGA_vsync : out  STD_LOGIC;
          ADC_CS : out  STD_LOGIC;            -- ADC signals
          ADC_SCLK : out  STD_LOGIC;
          ADC_SDATA1 : in  STD_LOGIC;
          ADC_SDATA2 : in  STD_LOGIC;
          btn0 : in STD_LOGIC);                    -- button to initiate serve
end pong;

architecture Behavioral of pong is

signal ck_25: STD_LOGIC := '0';    -- 25 MHz clock to VGA sync module

     -- internal signals to connect modules
signal S_red, S_green, S_blue: STD_LOGIC;
signal S_vsync: STD_LOGIC;
signal S_pixel_row, S_pixel_col: STD_LOGIC_VECTOR (9 downto 0);
signal batpos: STD_LOGIC_VECTOR (9 downto 0);
signal serial_clk, sample_clk: STD_LOGIC;
signal adout: STD_LOGIC_VECTOR (11 downto 0);
```

```
signal count: STD_LOGIC_VECTOR (9 downto 0);    -- counter to generate ADC clocks

component adc_if is
   Port ( SCK: in  STD_LOGIC;
          SDATA1 : in  STD_LOGIC;
          SDATA2 : in  STD_LOGIC;
          CS: in  STD_LOGIC;
          data_1 : out STD_LOGIC_VECTOR(11 downto 0);
          data_2 : out STD_LOGIC_VECTOR(11 downto 0));
end component;

component bat_n_ball is
   Port ( v_sync : in  STD_LOGIC;
          pixel_row : in  STD_LOGIC_VECTOR(9 downto 0);
          pixel_col : in  STD_LOGIC_VECTOR(9 downto 0);
          bat_x : in STD_LOGIC_VECTOR (9 downto 0);
          serve : in STD_LOGIC;
          red : out  STD_LOGIC;
          green : out  STD_LOGIC;
          blue : out  STD_LOGIC);
end component;

component vga_sync is
   Port (  clock_25MHz : in  STD_LOGIC;
          red : in  STD_LOGIC;
          green : in  STD_LOGIC;
          blue : in  STD_LOGIC;
          red_out : out  STD_LOGIC;
          green_out : out  STD_LOGIC;
          blue_out : out  STD_LOGIC;
          hsync : out  STD_LOGIC;
          vsync : out  STD_LOGIC;
          pixel_row : out  STD_LOGIC_VECTOR (9 downto 0);
          pixel_col : out  STD_LOGIC_VECTOR (9 downto 0));
end component;

begin

     --  Process to generate clock signals
ckp:      process
          begin
                    wait until rising_edge(clk_50MHz);
                    ck_25 <= not ck_25;     -- 25MHz clock for VGA modules
                    count <= count+1;        -- counter to generate ADC timing signals
          end process;

          serial_clk <= not count(4);       -- 1.5 MHz serial clock for ADC
          ADC_SCLK <= serial_clk;
          sample_clk <= count(9);           -- sampling clock is low for 16 SCLKs
          ADC_CS <= sample_clk;
```

```
        -- Multiplies ADC output (0-4095) by 5/32 to give bat position (0-640)
        batpos <= ('0'& adout(11 downto 3)) + adout(11 downto 5);

        -- set least significant bits of VGA  video to '0'
        VGA_red(1 downto 0) <= "00";
        VGA_green(1 downto 0) <= "00";
        VGA_blue(0) <= '0';

adc: adc_if port map (                  -- instantiate ADC serial to parallel interface
                SCK => serial_clk,
                CS => sample_clk,
                SDATA1 => ADC_SDATA1,
                SDATA2 => ADC_SDATA2,
                data_1 => OPEN,
                data_2 => adout );

add_bb: bat_n_ball port map(            --instantiate bat and ball component
                v_sync => S_vsync,
                pixel_row => S_pixel_row,
                pixel_col => S_pixel_col,
                bat_x => batpos,
                serve => btn0,
                red => S_red,
                green=> S_green,
                blue => S_blue);

vga_driver: vga_sync port map(          --instantiate vga_sync component
                clock_25MHz => ck_25,
                red => S_red,
                green => S_green,
                blue => S_blue,
                red_out => VGA_red(2),
                green_out => VGA_green(2),
                blue_out => VGA_blue(1),
                pixel_row => S_pixel_row,
                pixel_col => S_pixel_col,
                hsync => VGA_hsync,
                vsync => S_vsync);

                VGA_vsync <= S_vsync;           --connect output vsync

end Behavioral;
```

This is the top level that hooks it all together. *BTN0* on the *Nexys2* board is used to initiate a serve. The process *ckp* is used to generate timing signals for the VGA and ADC modules. The output of the *adc_if* module drives *bat_x* of the *bat_n_ball* module.

4.5 Synthesis and Implementation

Highlight the *pong* module in the *Hierarchy* window and execute the **Synthesize** command in the *Process* window.

Add an Implementation Constraint source file *pong.ucf* and enter the following data into the edit window:

```
NET     "clk_50MHz"    LOC = B8;

NET     "vga_hsync"         LOC = T4;
NET     "vga_vsync"       LOC = U3;

NET     "vga_red[0]"     LOC = R9;
NET     "vga_red[1]"     LOC = T8;
NET     "vga_red[2]"     LOC = R8;
NET     "vga_green[0]" LOC = N8;
NET     "vga_green[1]" LOC = P8;
NET     "vga_green[2]" LOC = P6;
NET     "vga_blue[0]"    LOC = U5;
NET     "vga_blue[1]"    LOC = U4;

NET    "ADC_SDATA1"   LOC = K12;
NET    "ADC_SDATA2"   LOC = L17;
NET    "ADC_SCLK"     LOC = M15;
NET    "ADC_CS"       LOC = L15;

NET    "btn0"         LOC = B18;

NET "ck_25" TNM_NET = ck_25_net;
TIMESPEC TS_ck_25 = PERIOD "ck_25_net" 40 ns HIGH 50%;
```

Now highlight the *pong* module in the Hierarchy window and run **Implement Design** followed by **Generate Programming File** (don't forget to change the *FPGA Start-up Clock* to be the *JTAG Clock*).

4.6 Download and Run

Use the *Adept* software to download your configuration file *pong.bit* and check out the result. You should see a white screen with a blue bat whose position can be changed using the potentiometer connected to the ADC. Now push *BTN0* and the red ball should start bouncing around the screen. Use the bat to keep the ball in play.

4.7 Now let's make some changes …

Modify your VHDL code to do one or more of the following:

(a) The ball speed is currently 6 pixels per video frame. Use the slide switches on the *Nexys2* board to program the ball speed in the range of 1-32 pixels per frame. (Avoid setting the speed to zero as the ball will then never reach the bat or wall). See how fast you can move the ball and still keep it in play.
(b) Double the width of the bat (makes it really easy). But now modify the code so that the bat width decreases one pixel each time you successfully hit the ball (and then resets to starting width when you miss). See how many times you can hit the ball in a row as the bat slowly shrinks.
(c) Count the number of successful hits (after each serve) and display the count (in binary) on the LEDs on the *Nexys2* board.