**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.

1. **ADC Interface**

|  |
| --- |
|  |
| **Figure 1 PModAD1 Dual 12-bit ADC Module** |

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.

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** |

1. **Hardware Setup**

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

A 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** |

1. **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:

1. 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.
2. 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.
3. Count the number of successful hits (after each serve) and display the count (in binary) on the LEDs on the *Nexys2* board.