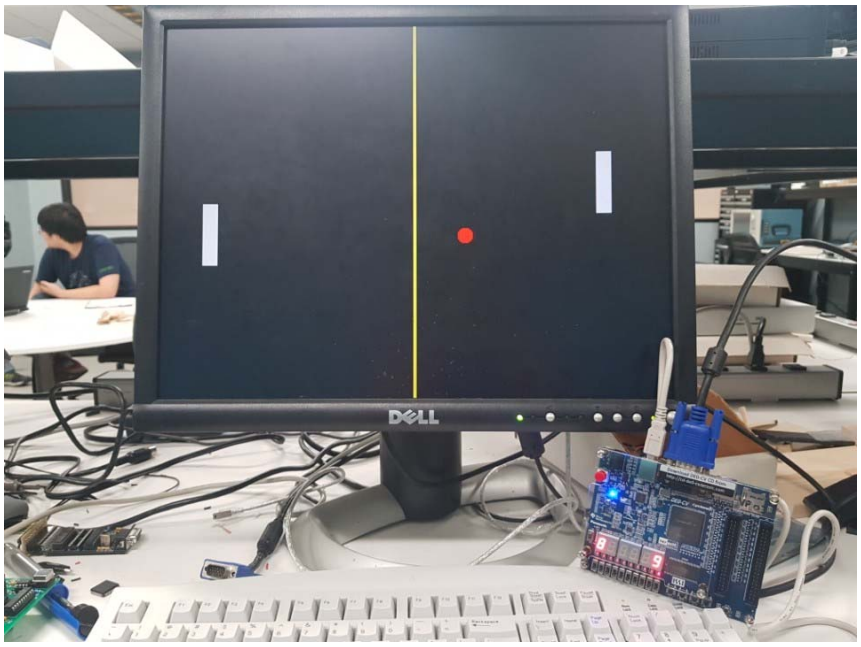
Pong with FPGA (EE125 project)

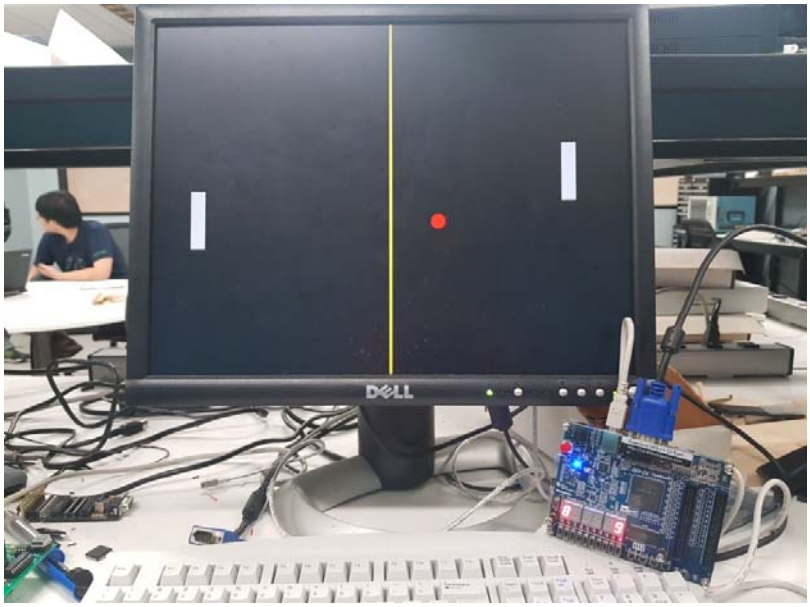


Sung Hoon Choi, Garret Sullivan

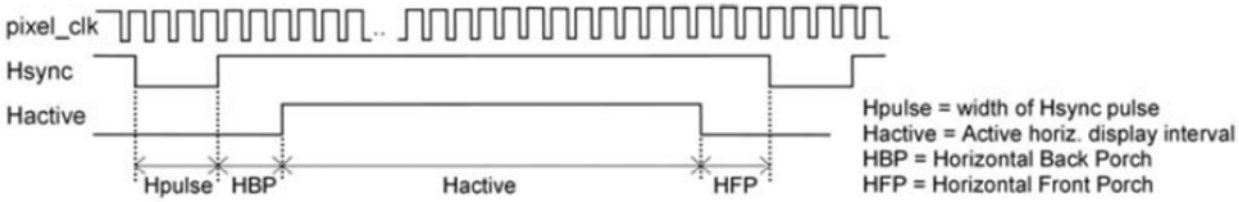
California Institute of Technology

Pong

Sung Hoon Choi, Garret Sullivan

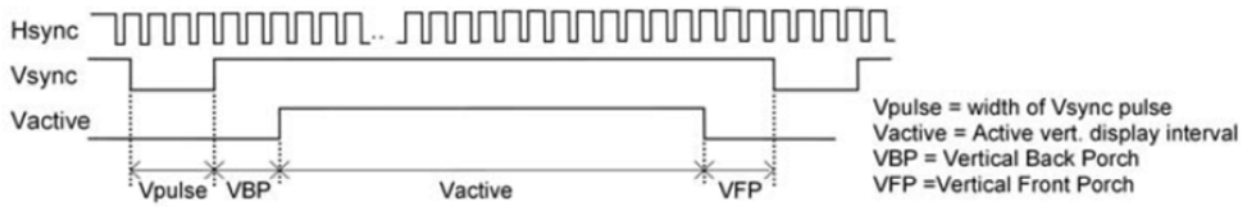


We use 640x480 resolution and 25MHz pixel clock for the VGA control of this Pong game. The timing parameters for horizontal signals and vertical signals are listed below. The figures were taken from *Circuit Design and Simulation with VHDL, Volnei A. Pedroni, 2nd edition.*



VGA mode		Refresh rate (Hz)	Pixel clock (MHz)	Horizontal timing (in clock pulses, same as pixels)				
Name	Resolution (N x M)			Hpulse (pixels)	HBP (pixels)	Hactive (pixels)	HFP (pixels)	Total pixels
VGA	640 x 480	60	25.175	96	48	640	16	800

Figure1. Horizontal time parameters for 640x480, 25MHz pixel clock.



VGA mode		Refresh rate (Hz)	Pixel clock (MHz)	Vertical timing (in lines, same as Hsync pulses)				
Name	Resolution (N x M)			Vpulse (lines)	VBP (lines)	Vactive (lines)	VFP (lines)	Total lines
VGA	640 x 480	60	25.175	2	33	480	10	525

Figure2. Vertical time parameters for 640x480, 25MHz pixel clock.

Part 1: Control Generator

The ***Gen_Pixel_Clk*** process (line 83) creates the 25MHz pixel clock by halving the 50MHz crystal clock.

The ***Gen_Horz_Ctrl*** process (line 92) generates the signals for horizontal data on the monitor, using the 25MHz pixel clock.

The ***Gen_Vert_Ctrl*** process (line 112) generates the signals for vertical data on the monitor. This process is triggered by the change of H_{sync} signal.

Part 2: Image Generator

The ***Get_Monitor_Coordinate*** process (line 139) acquires the screen coordinates in the signals x, y. Again, acquiring the x and y coordinates is driven by 25MHz pixel clock. At the rising edge of pixel clock, if H_{active} or V_{active} is set, x coordinate or y coordinate is updated, respectively.

The ***Draw_Ball_Paddles*** process (line 157) draws the ball and two paddles on the monitor screen, using the x and y coordinates acquired from *Get_Monitor_Coordinate* process. The paddle size is set to 20 x 80 pixels (Width x Height) and the ball's radius is set to 10. The process directly uses R, G, B signals to determine the color of paddles and ball (paddles: white, ball: red). The background color is set to black (R:0 G:0 B:0). The variables for the positions of paddles and balls are updated by other processes *(Move_Paddle1, Move_Paddle2, Move_Ball).*

The ***Gen_Mov_Clk*** (line 199) process generates *mov_clk,* which is used by *Move_Ball* process to control the speed of the ball.

The ***Move_Ball*** process (line 220) handles ball movement and collision detection, beginning on line 220. This process runs on the positive edge of *mov_clk*, the frequency of which effectively controls the speed of the ball. The process first updates both the ball's horizontal position and the ball's vertical position using the appropriate signals. Then, it checks for various collision scenarios.

The first conditional, on line 228, checks if the center of the ball is in a small rectangle surrounding the right side of the left paddle and ensures the last collision was not with this paddle. If the conditions are met, a collision with the left paddle is registered. The horizontal direction of the ball is set to the right, and the vertical speed of the ball is modified slightly based on where on the paddle the collision was detected: a collision on the upper half of the paddle adjusts the angle of the ball slightly upwards while a collision on the lower half of the paddle adjusts the angle of the ball slightly downwards. Additionally, the *last_collision* signal is updated to show that the last collision was with the left paddle; this is to prevent the ball from becoming trapped in the collision area of a single paddle and repeatedly "colliding" back and forth within the paddle. The next conditional, on line 245, operates very similarly, except it checks for a collision on the left side of the right paddle and sets the horizontal direction of the ball to the left if a collision is detected.

The process then checks for collisions with the top and bottom edges of the screen. The corresponding conditionals, on lines 262 and 266, check if the ball's vertical position is within a few pixels of the top or bottom of the screen. Additionally, the ball must be moving upwards to trigger a collision with the top of the screen and downwards to trigger a collision with the bottom; this is to prevent the ball from being trapped within the collision detection area and bouncing back and forth at the top or bottom of the screen. If a collision is detected, the vertical speed of the ball is negated, causing the ball to start moving in the opposite vertical direction.

Finally, the process checks for collisions with the left and right edges of the screens using the conditionals on lines 272 and 286. These collisions correspond to a player scoring, as a collision with these edges means that the other player was unable to return the ball using their paddle. The conditionals just check if the ball is within a few pixels of the left or right edges of the screen; since the ball's position is reset if a collision is detected, there is no need to worry about the ball becoming trapped at the edge of the screen. If a collision is detected, the ball is reset back to the middle of the screen with zero vertical speed and with horizontal

direction set so that the ball is moving towards the player who scored. The value of *last_collision* is also updated so that the ball can collide with either paddle. Additionally, the score of the player who scored is incremented if it is not already at its maximum value.
If none of these conditionals are met, there was no collision in this frame, and the ball is moving normally.

The ***Move_Paddle1*** process (line 304) moves the position of paddle 1 (left paddle). Pressing the up-key or down-key allows the player to move the paddle up and down. The keys are debounced. The paddles are limited to move vertically within the monitor screen. The paddles won't move further when they reach the upper edge or lower edge of the monitor screen.

The ***Move_Paddle2*** process (line 334) is exactly same as *move_paddle1* except that it moves paddle 2 (right paddle) instead of paddle1.

The ***Display_Score*** process (line 364) uses case statements to display the score of player 1 and player 2 on the SSDs on the DE0-CV board. As already mentioned above, the player gets a point if the ball reaches the vertical edge of the opponent's side, and the score is capped at 9.

The Code

```
001: --------------------------------------------------------------------------------
002: library ieee;
003: use ieee.std_logic_1164.all;
004: use ieee.numeric_std.all;
005: --------------------------------------------------------------------------------
006: entity pong is
007:    generic (
008:            Ha: integer := 96;  --Hpulse
009:            Hb: integer := 144; --Hpulse+HBP
010:            Hc: integer := 784; --Hpulse+HBP+Hactive        640 Width
011:            Hd: integer := 800; --Hpulse+HBP+Hactive+HFP
012:            Va: integer := 2;   --Vpulse
013:            Vb: integer := 35;  --Vpulse+VBP
014:            Vc: integer := 515; --Vpulse+VBP+Vactive        480 Height
015:            Vd: integer := 525);--Vpulse+VBP+Vactive+VFP
016:    port (
017:            clk: in std_logic;  --50MHz in our board
018:            paddle1_down_key, paddle1_up_key, paddle2_down_key, paddle2_up_key: in std_logic;
019:            pixel_clk: buffer std_logic;
020:            Hsync, Vsync: buffer std_logic;
021:            R, G, B: out std_logic_vector(3 DOWNTO 0);
022:            Score1_SSD: out std_logic_vector(0 to 6);
023:            Score2_SSD: out std_logic_vector(0 to 6));
024: end pong;
025: --------------------------------------------------------------------------------
026: architecture pong of pong is
027:
028:    -------------------------------
029:    -- VGA definitions
030:    -------------------------------
031:    signal Hactive, Vactive, dena: std_logic;
032:    signal x: natural range 0 to Hd; --X Coordinate on screen
033:    signal y: natural range 0 to Vd;--Y Coordinate on screen
034:
035:    -------------------------------
036:    -- Game logic definitions
037:    -------------------------------
038:    constant BALL_MAX_SPEED: integer := 5; --pixels per frame
039:    constant BALL_RADIUS: integer := 10; --pixels
040:
041:    constant BALL_MOVE_DELAY: integer := 1000000; --clock cycles
042:    constant PADDLE_MOVE_DELAY: integer := 1000000; --clock cycles
043:
```

```
044:    constant PADDLE_HORZ_SIZE: integer := 20; --pixels
045:    constant PADDLE_VERT_SIZE: integer := 80; --pixels
046:
047:    --Clock signal for moving the ball
048:    signal mov_clk: std_logic;
049:
050:    --Position of the (top left of) the left paddle
051:    signal paddle1_horz_pos: integer range 0 to 650 := 50; --pixels
052:    signal paddle1_vert_pos: integer range 0 to 650 := 240; --pixels
053:
054:    --Position of the (top left of) the right paddle
055:    signal paddle2_horz_pos: integer range 0 to 650 := 550; --pixels
056:    signal paddle2_vert_pos: integer range 0 to 650 := 240; --pixels
057:
058:    --Movement direction of the paddles
059:    signal paddle1_direction: std_logic;
060:    signal paddle2_direction: std_logic;
061:
062:    --Position of the (center of) the ball
063:    signal ball_horz_pos: integer range 0 to 640 := 320; --pixels
064:    signal ball_vert_pos: integer range 0 to 480 := 240; --pixels
065:
066:    --Horizontal/vertical direction/speed of the ball
067:    signal ball_horz_dir: integer range -1 to 1 := 1;
068:    signal ball_vert_speed: integer range -5 to 5 := 0;
069:
070:    --Side of the last paddle collision; used to prevent multiple collisions
071:    signal last_collision: integer range -1 to 1 := 0;
072:
073:    --Scores of the left and right players
074:    signal Score1, Score2: integer range 0 to 9;
075:
076: begin
077:
078:    -------------------------------------------------------
079:    --VGA Part 1: CONTROL GENERATOR
080:    -------------------------------------------------------
081:
082:    --Create pixel clock (50MHz->25MHz)
083:    Gen_Pixel_Clk: process (clk)
084:    begin
085:            if (clk'event and clk='1') then
086:                    pixel_clk <= not pixel_clk;
087:            end if;
088:    end process;
089:
090:
091:    --Horizontal signals generation
092:    Gen_Horz_Ctrl: process (pixel_clk)
093:    variable Hcount: integer range 0 to Hd;
094:    begin
095:            if (pixel_clk'event and pixel_clk='1') then
096:                    Hcount := Hcount + 1;
097:                    if (Hcount=Ha) then
098:                            Hsync <= '1';
099:                    elsif (Hcount=Hb) then
100:                            Hactive <= '1';
101:                    elsif (Hcount=Hc) then
102:                            Hactive <= '0';
103:                    elsif (Hcount=Hd) then
104:                            Hsync <= '0';
105:                            Hcount := 0;
106:                    end if;
107:            end if;
108:    end process;
109:
110:
```

```
111:    --Vertical signals generation
112:    Gen_Vert_Ctrl: process (Hsync)
113:            variable Vcount: integer range 0 to Vd;
114:    begin
115:            if (Hsync'event and Hsync='0') then
116:                    Vcount := Vcount + 1;
117:                    if (Vcount=Va) then
118:                            Vsync <= '1';
119:                    elsif (Vcount=Vb) then
120:                            Vactive <= '1';
121:                    elsif (Vcount=Vc) then
122:                            Vactive <= '0';
123:                    elsif (Vcount=Vd) then
124:                            Vsync <= '0';
125:                            Vcount := 0;
126:                    end if;
127:            end if;
128:    end process;
129:
130:    ---Display enable generation:
131:    dena <= Hactive and Vactive;
132:
133:
134:    ------------------------------------------------
135:    --VGA Part 2: IMAGE GENERATOR
136:    ------------------------------------------------
137:
138:    --Process: get screen coordinates in signal x, y
139:    Get_Monitor_Coordinate: process (pixel_clk, Hsync, dena)
140:    begin
141:            ----Count columns:--------
142:            if (pixel_clk'event and pixel_clk='1') then
143:                    if (Hactive='1') then x <= x + 1;
144:                    else x <= 0;
145:                    end if;
146:            end if;
147:            ----Count lines:----------
148:            if (Hsync'event and Hsync='1') then
149:                    if (Vactive='1') then y <= y + 1;
150:                    else y <= 0;
151:                    end if;
152:            end if;
153:    end process;
154:
155:
156:    --Process: draw the ball and paddles
157:    Draw_Ball_Paddles: process (pixel_clk, Hsync, dena)
158:    BEGIN
159:            if (dena='1') then
160:                    if ((x > paddle1_horz_pos and x < paddle1_horz_pos+PADDLE_HORZ_SIZE) and (y >
paddle1_vert_pos and y < paddle1_vert_pos+PADDLE_VERT_SIZE)) then
161:                            --drawing the left paddle
162:                            R <= (others => '1');
163:                            G <= (others => '1');
164:                            B <= (others => '1');
165:                    elsif ((x > paddle2_horz_pos and x < paddle2_horz_pos+PADDLE_HORZ_SIZE) and (y >
paddle2_vert_pos and y < paddle2_vert_pos+PADDLE_VERT_SIZE)) then
166:                            --drawing the right paddle
167:                            R <= (others => '1');
168:                            G <= (others => '1');
169:                            B <= (others => '1');
170:                    elsif ((x-ball_horz_pos)**2+(y-ball_vert_pos)**2 < BALL_RADIUS**2) then
171:                            --drawing the ball
172:                            R <= (others => '1');
173:                            G <= (others => '0');
174:                            B <= (others => '0');
175:                    elsif (x>318 AND x<323) then
```

```
176:                            --drawing the center dividing line
177:                            R <= (others => '1');
178:                            G <= (others => '1');
179:                            B <= (others => '0');
180:                    else
181:                            --drawing the background
182:                            R <= (others => '0');
183:                            G <= (others => '0');
184:                            B <= (others => '0');
185:                    END if;
186:            else
187:                    R <= (others => '0');
188:                    G <= (others => '0');
189:                    B <= (others => '0');
190:            end if;
191:    end process;
192:
193:
194:    ------------------------------------------------
195:    --GAME LOGIC:
196:    ------------------------------------------------
197:
198:    --Generate a clock signal that controls the speed of the ball movement
199:    Gen_Mov_Clk: process (pixel_clk)
200:            variable mov_timer: integer range 0 to BALL_MOVE_DELAY+1;
201:    begin
202:            if (rising_edge(pixel_clk)) then
203:                    --increment the timer and reset it to 0 if appropriate
204:                    mov_timer := mov_timer+1;
205:                    if (mov_timer >= BALL_MOVE_DELAY) then
206:                            mov_timer := 0;
207:                    end if;
208:
209:                    --generate the scaled clock signal
210:                    if (mov_timer <= BALL_MOVE_DELAY/2) then
211:                            mov_clk <= '1';
212:                    else
213:                            mov_clk <= '0';
214:                    end if;
215:            end if;
216:    end process;
217:
218:
219:    --Move the ball and handle collisions
220:    Move_Ball: process (mov_clk)
221:    begin
222:            if (rising_edge(mov_clk)) then
223:                    --move the ball both horizontally and vertically
224:                    ball_horz_pos <= ball_horz_pos+BALL_MAX_SPEED*ball_horz_dir;
225:                    ball_vert_pos <= ball_vert_pos+ball_vert_speed;
226:
227:                    --check for a collision with the left-side paddle
228:                    if (ball_horz_pos >= paddle1_horz_pos+25 and ball_horz_pos <= paddle1_horz_pos+30
and ball_vert_pos >= paddle1_vert_pos and ball_vert_pos <= paddle1_vert_pos+80 and last_collision/=1) then
229:                            --Set the horizontal direction to the right
230:                            ball_horz_dir <= 1;
231:
232:                            --Adjust the vertical speed based on where the ball hit the paddle
233:                            if (ball_vert_pos > (paddle1_vert_pos+40)) then
234:                                    if ball_vert_speed /= BALL_MAX_SPEED then
235:                                            ball_vert_speed <= ball_vert_speed + 1;
236:                                    end if;
237:                            elsif (ball_vert_pos < (paddle1_vert_pos+40)) then
238:                                    if ball_vert_speed /= -BALL_MAX_SPEED then
239:                                            ball_vert_speed <= ball_vert_speed - 1;
240:                                    end if;
241:                            end if;
```

```
242:                            last_collision <= 1;
243:
244:                    --check for a collision with the right-side paddle
245:                    elsif (ball_horz_pos <= paddle2_horz_pos-5 and ball_horz_pos >= paddle2_horz_pos-
10 and ball_vert_pos >= paddle2_vert_pos and ball_vert_pos <= paddle2_vert_pos+80 and last_collision/=-1)
then
246:                            --set the horizontal direction to the left
247:                            ball_horz_dir <= -1;
248:
249:                            --adjust the vertical speed based on where the ball hit the paddle
250:                            if (ball_vert_pos > (paddle2_vert_pos+40)) then
251:                                    if ball_vert_speed /= BALL_MAX_SPEED then
252:                                            ball_vert_speed <= ball_vert_speed + 1;
253:                                    end if;
254:                            elsif (ball_vert_pos < (paddle2_vert_pos+40)) then
255:                                    if ball_vert_speed /= -BALL_MAX_SPEED then
256:                                            ball_vert_speed <= ball_vert_speed - 1;
257:                                    end if;
258:                            end if;
259:                            last_collision <= -1;
260:
261:                    --check for a collision with the top edge of the screen
262:                    elsif (ball_vert_pos <= 20 and ball_vert_speed < 0) then
263:                            --reverse the vertical speed/direction
264:                            ball_vert_speed <= ball_vert_speed * (-1);
265:
266:                    --check for a collision with the bottom edge of the screen
267:                    elsif (ball_vert_pos >= 460 and ball_vert_speed > 0) then
268:                            --reverse the vertical speed/direction
269:                            ball_vert_speed <= ball_vert_speed * (-1);
270:
271:                    --check for the ball reaching the left edge of the screen (right player scores)
272:                    elsif (ball_horz_pos <= 20) then
273:                            --reset the ball to the center moving right
274:                            ball_horz_pos <= 320;
275:                            ball_horz_dir <= 1;
276:                            ball_vert_pos <= 240;
277:                            ball_vert_speed <= 0;
278:                            last_collision <= 0;
279:
280:                            --increment the right player's score
281:                            if (Score2 /= 9) then
282:                                    Score2 <= Score2+1;
283:                            end if;
284:
285:                    --check for the ball reaching the right edge of the screen (left player scores)
286:                    elsif (ball_horz_pos >= 620) then
287:                            --reset the ball to the center moving left
288:                            ball_horz_pos <= 320;
289:                            ball_horz_dir <= -1;
290:                            ball_vert_pos <= 240;
291:                            ball_vert_speed <= 0;
292:                            last_collision <= 0;
293:
294:                            --increment the left player's score
295:                            if (Score1 /= 9) then
296:                                    Score1 <= Score1+1;
297:                            end if;
298:                    end if;
299:            end if;
300:    end process;
301:
302:
303:    --Move the left paddle
304:    move_paddle1: process (pixel_clk)
305:            --debounce both buttons for moving the paddle
306:            variable up_debounce_timer: integer range 0 to PADDLE_MOVE_DELAY+1;
```

```
307:            variable down_debounce_timer: integer range 0 to PADDLE_MOVE_DELAY+1;
308:    begin
309:            if (rising_edge(pixel_clk)) then
310:                    up_debounce_timer := up_debounce_timer+1;
311:                    down_debounce_timer := down_debounce_timer+1;
312:
313:                    --check if we should move the paddle up
314:                    if (paddle1_up_key = '0' and up_debounce_timer > PADDLE_MOVE_DELAY) then
315:                            --prevent the paddle from moving off the top of the screen
316:                            if (paddle1_vert_pos >= 5) then
317:                                    paddle1_vert_pos <= paddle1_vert_pos-5;
318:                            end if;
319:                            up_debounce_timer := 0;
320:
321:                    --check if we should move the paddle down
322:                    elsif (paddle1_down_key = '0'and down_debounce_timer > PADDLE_MOVE_DELAY) then
323:                            --prevent the paddle from moving off the bottom of the screen
324:                            if (paddle1_vert_pos <= 395) then
325:                                    paddle1_vert_pos <= paddle1_vert_pos+5;
326:                            end if;
327:                            down_debounce_timer := 0;
328:                    end if;
329:            end if;
330:    end process;
331:
332:
333:    --Move the right paddle
334:    move_paddle2: process (pixel_clk)
335:            --debounce both buttons for moving the paddle
336:            variable up_debounce_timer: integer range 0 to PADDLE_MOVE_DELAY+1;
337:            variable down_debounce_timer: integer range 0 to PADDLE_MOVE_DELAY+1;
338:    begin
339:            if (rising_edge(pixel_clk)) then
340:                    up_debounce_timer := up_debounce_timer+1;
341:                    down_debounce_timer := down_debounce_timer+1;
342:
343:                    --check if we should move the paddle up
344:                    if (paddle2_up_key = '0' and up_debounce_timer > PADDLE_MOVE_DELAY) then
345:                            --prevent the paddle from moving off the top of the screen
346:                            if (paddle2_vert_pos >= 5) then
347:                                    paddle2_vert_pos <= paddle2_vert_pos-5;
348:                            end if;
349:                            up_debounce_timer := 0;
350:
351:                    --check if we should move the paddle down
352:                    elsif (paddle2_down_key = '0'and down_debounce_timer > PADDLE_MOVE_DELAY) then
353:                            --prevent the paddle from moving off the bottom of the screen
354:                            if (paddle2_vert_pos <= 395) then
355:                                    paddle2_vert_pos <= paddle2_vert_pos+5;
356:                            end if;
357:                            down_debounce_timer := 0;
358:                    end if;
359:            end if;
360:    end process;
361:
362:
363:    --Display each player's score on the seven-segment displays
364:    Display_score: process(all)
365:    begin
366:            case Score1 is
367:                    when 0 => Score1_SSD <= "0000001";
368:                    when 1 => Score1_SSD <= "1001111";
369:                    when 2 => Score1_SSD <= "0010010";
370:                    when 3 => Score1_SSD <= "0000110";
371:                    when 4 => Score1_SSD <= "1001100";
372:                    when 5 => Score1_SSD <= "0100100";
373:                    when 6 => Score1_SSD <= "0100000";
```

```
374:                    when 7 => Score1_SSD <= "0001111";
375:                    when 8 => Score1_SSD <= "0000000";
376:                    when 9 => Score1_SSD <= "0000100";
377:                    when others => Score1_SSD <= "1111110";
378:            end case;
379:
380:            case Score2 is
381:                    when 0 => Score2_SSD <= "0000001";
382:                    when 1 => Score2_SSD <= "1001111";
383:                    when 2 => Score2_SSD <= "0010010";
384:                    when 3 => Score2_SSD <= "0000110";
385:                    when 4 => Score2_SSD <= "1001100";
386:                    when 5 => Score2_SSD <= "0100100";
387:                    when 6 => Score2_SSD <= "0100000";
388:                    when 7 => Score2_SSD <= "0001111";
389:                    when 8 => Score2_SSD <= "0000000";
390:                    when 9 => Score2_SSD <= "0000100";
391:                    when others => Score2_SSD <= "1111110";
392:            end case;
393:    end process;
394:
395: end pong;
396: ----------------------------------------------------------------------------------
```