Department of Electric & Electronic Engineering,
Boğaziçi University

EE 244 FINAL PROJECT REPORT

WHACK-A-MOLE

Boran Aybak Kılıç- Efe Karakoca

Project Advisor : Şenol Mutlu

5 June 2024

Table of Contents

1 INTRODUCTION

The primary objective of this project is to design and implement a simplified version of the "Whack-a-Mole" game on a Field Programmable Gate Array (FPGA) platform, utilizing the Nexys3 board. This endeavor integrates multiple fundamental concepts of digital design, including VGA signal generation, pseudo-random number generation, and real-time interactive gameplay, making it an exemplary educational exercise in the realm of digital systems and hardware description languages (HDLs).

The motivation behind this project stems from the educational value it offers in bridging theoretical knowledge with practical application. By constructing a functional game, students and hobbyists can gain a deeper understanding of FPGA architecture, VHDL programming, and the intricacies involved in designing interactive systems. Furthermore, the project demonstrates how complex digital systems can be efficiently implemented and managed on FPGA platforms, which are pivotal in modern electronics and embedded systems development.

Key concepts underpinning this project include the use of VGA drivers to render graphics on a monitor, enabling the visual representation of the game. The VGA driver module is responsible for generating the necessary timing signals to produce the correct resolution and color depth for the game display. Additionally, the project incorporates pseudo-random number generation to introduce unpredictability into the game, enhancing its challenge and replicability, which is achieved by utilizing a subtle but effective linear feedback shift register algorithm.

2 PROBLEM STATEMENT

The design of this project, which appears to be a fairly simple game, was laborious, and it required multidimensional thinking on our part. The primary reason for this was because it requires many independent components that should work in great harmony and synchronization. In addition, we have utilized almost all of the input and output ports that are covered in the context of this course (EE 244 Digital System Design) which adds another layer of dependencies. The mole should appear within the 3 by 3 grid whose position and the time of visibility, however, should be assigned randomly. The randomness can be additionally controlled by the user by changing the positions of the switches. The player should have access to the buttons on the FPGA to move the cursor on the grid in order to catch the mole before it disappears. If they manage to do so, the score should increase gradually, and it should be visible to the player on the FPGA's seven segment display, until the score reaches the value "10" which is the condition for the game to be over. The user can also restart the game by using the leftmost switch (T11) on the FPGA which is allocated to the reset signal of the design, which overall showcases the practical application of VHDL in developing interactive and visually engaging digital systems on FPGA platforms.

3 RELATED BACKGROUND

Tasks to be realized necessitate the knowledge of the efficient use of VHDL hardware description language's properties such as sequential and concurrent body structures along with the mixed use of behavioral and structural architectures. Displaying the game content to the monitor is achieved by the proper use of VGA timing signals for the screen with sizes of 640 by 480. The random vector generator uses a series of Fibonacci LFSR's (Linear Feedback Shift Register) to generate pseudo random sequences partially identified also by the input "seed" which is given by the user using slide switches on the FPGA board, whose details are mentioned in the following sections. As for buttons and switches on the FPGA, they are utilized by taking advantage of the debouncer circuits so as to prevent unwanted glitches caused by inevitable mechanical bouncing of the buttons.

4 DESIGN

The details of the functionality of the project is as follows:

A mole spawns (appears) in one the 9 squares of a 3 by 3 square grid in every 6 seconds. It stays in this square for a random time interval (until the random time assigned for it expires) before eventually disappearing after some random time that is smaller than 6 seconds. The location the mole will spawn and the time interval it will stay on the screen is randomly determined by the output of our random vector generator. Randomness and ambiguity obtained via the above-mentioned procedure making the game more challenging and more intriguing to play while the integration of a predetermined seed ensures reproducibility.

There is also a cursor element in the shape of a blue cross which represents the player side of this game. The cursor is controlled by the user using push buttons on the FPGA board. The user can move the cursor to up, down, right, left or it can click on the mole using the central push button. The resultant action of the push buttons takes place immediately after the button is released, which is handled by the edge detector. If the user manages to move the cursor to the square which mole is current on and clicks on it before it disappears, he scores a point. Furthermore, mole disappears right after the clicking regardless of the expiration of the time interval assigned and he basically scores a point. The score which is displayed on the seven-segment as a 2-digit decimal number, is then incremented by one.

All of the elements are then drawn in the top-level module using the VGA driver explained elaborately in subsection **4.2.2**. The user score is also continuously displayed on the 7- segment display. The T11 switch is used to reset the whole game.

4.1 Definitions, Acronyms and Abbreviations

FPGA: Fields Programmable Gate Array

VGA: Video Graphics Array

Seed: 7-bit user input that partially determines the random generation process.

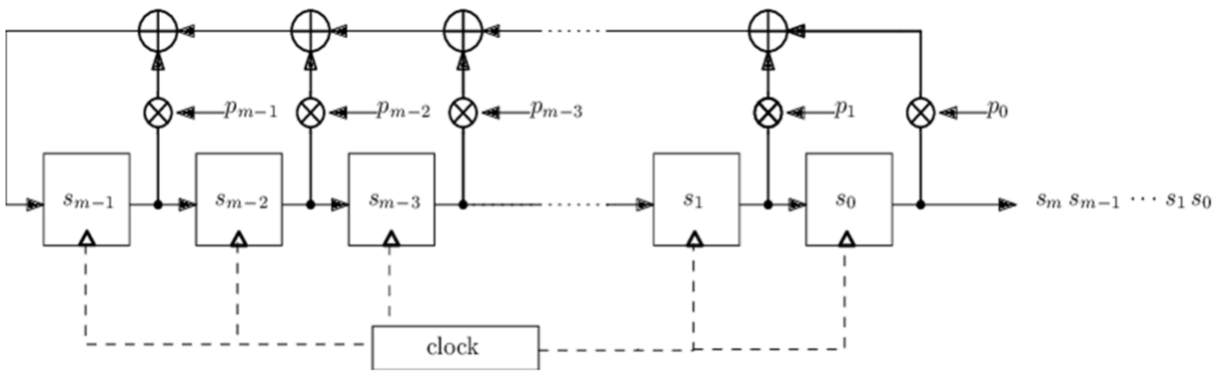
LFSR: Linear Feedback Shift Register

4.2 Components In The Design

Such a multifaceted project requires to be divided into subsections explained as the following:

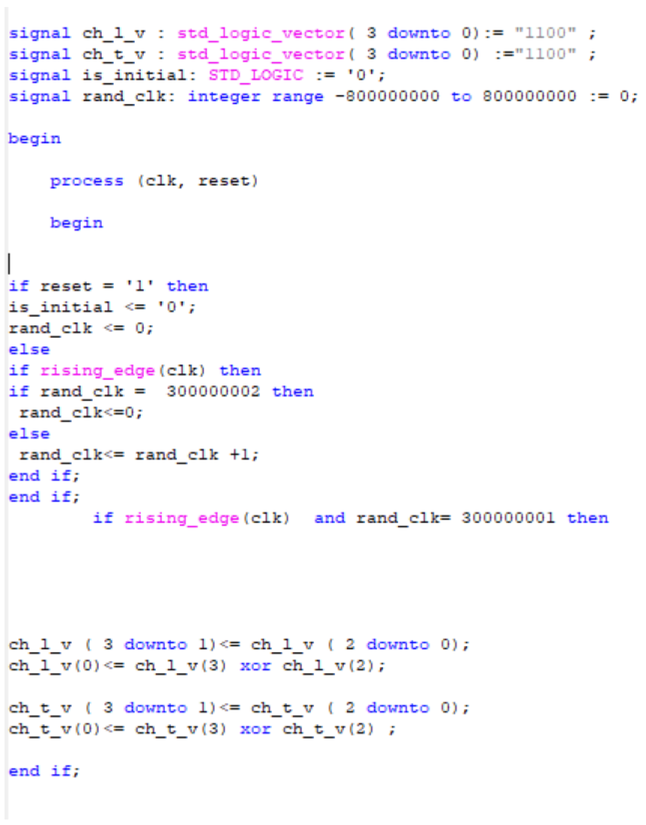
4.2.1 Random Generator

The generation of a random number is crucial for the unpredictability of such a game model, which is achieved by the invocation of the algorithm called Fibonacci Linear Feedback Shift Register (LFSR). The intricate details of mathematics involving finite field arithmetic, Galois Fields and primitive polynomials [1] are not mentioned here; however, a general overview can be given as the following: The aim is to obtain a sequence that repeats itself with the maximum period with an appearance of randomness. The methodology of the Fibonacci LFSR is based on shifting the original sequence right by one bit and applying the XOR operation of the "tap" bits. The choice of these tap bits are obtained via the polynomial representation of the binary number in the Galois field by polynomials, among which a characteristic polynomial for the specific number of n of the n-bit binary number in question. It turns out that the related polynomial should be irreducible, that is to say not being able to be represented by the multiplication of the two polynomials and primitive at the same time. The characteristic polynomial for each n can be obtained by Euler totient function $^{\varphi}$(n) [2].



The fundamental principle of functionality of the Fibonacci LFSR

This methodology is applied to our design with minor modifications: Since we should accommodate 10 random numbers for time interval during which mole is visible and 9 numbers for the location of appearance, we should generate 4-bit random vector whose characteristic polynomial is for n=4 x^4+ x^3+1 which corresponds to 1100 sequence to be initiated along with applying XOR to the first and second most significant bits. Moreover, an inherent frequency divider is implemented in the random generator block which enables us to get the samples of the randomly generated numbers at such points that do not coincide with the edges of the maximum refreshment period of 6 seconds which also contributes to the randomness of the samples. The code depicting the explanation is as the following:

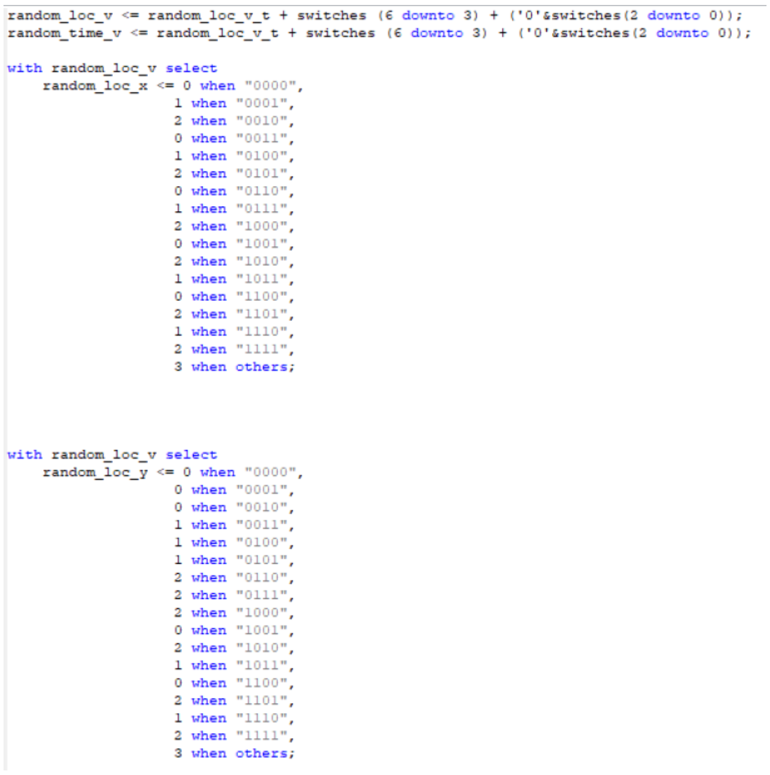


```vhdl
signal ch_l_v : std_logic_vector( 3 downto 0):= "1100" ;
signal ch_t_v : std_logic_vector( 3 downto 0) :="1100" ;
signal is_initial: STD_LOGIC := '0';
signal rand_clk: integer range -800000000 to 800000000 := 0;

begin

    process (clk, reset)

    begin

if reset = '1' then
is_initial <= '0';
rand_clk <= 0;
else
if rising_edge(clk) then
if rand_clk =  300000002 then
 rand_clk<=0;
else
 rand_clk<= rand_clk +1;
end if;
end if;
        if rising_edge(clk)  and rand_clk= 300000001 then



ch_l_v ( 3 downto 1)<= ch_l_v ( 2 downto 0);
ch_l_v(0)<= ch_l_v(3) xor ch_l_v(2);

ch_t_v ( 3 downto 1)<= ch_t_v ( 2 downto 0);
ch_t_v(0)<= ch_t_v(3) xor ch_t_v(2) ;

end if;
```

Code snippet of the Random Generator

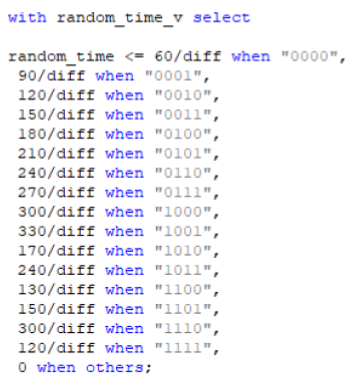
Here we initiate the time and the location parameters to the characteristic “1100” vector which is updated via the LFSR methodology at every edge of the main clock divided by the 300000002, which denotes nearly 3 + δ seconds which denotes sampling at a reasonable rate. (In effect, a larger number such as 500000000 was intended to be preliminarily opted for, which albeit exceeded the FPGA gate capacities, which in all likelihood resulted in the attenuation of the VGA signals, preventing monitoring the display outputs.)

Moreover random numbers ranging from 0 to 15 is treated in the main code such that the contribution of the 6 bit seed extracted from the switches (*V8, U8,N8, M8, V9, T9,T10*) is also implemented along with the selections of the time and location parameters to be used in the process of drawing mole and in the game engine processes, which are all defined in the main block of the game as the following code snippet represents as the following:



```
random_loc_v <= random_loc_v_t + switches (6 downto 3) + ('0'&switches(2 downto 0));
random_time_v <= random_loc_v_t + switches (6 downto 3) + ('0'&switches(2 downto 0));

with random_loc_v select
    random_loc_x <= 0 when "0000",
                    1 when "0001",
                    2 when "0010",
                    0 when "0011",
                    1 when "0100",
                    2 when "0101",
                    0 when "0110",
                    1 when "0111",
                    2 when "1000",
                    0 when "1001",
                    2 when "1010",
                    1 when "1011",
                    0 when "1100",
                    2 when "1101",
                    1 when "1110",
                    2 when "1111",
                    3 when others;



with random_loc_v select
    random_loc_y <= 0 when "0000",
                    0 when "0001",
                    0 when "0010",
                    1 when "0011",
                    1 when "0100",
                    1 when "0101",
                    2 when "0110",
                    2 when "0111",
                    2 when "1000",
                    0 when "1001",
                    2 when "1010",
                    1 when "1011",
                    0 when "1100",
                    2 when "1101",
                    1 when "1110",
                    2 when "1111",
                    3 when others;
```

With- Select code for the position



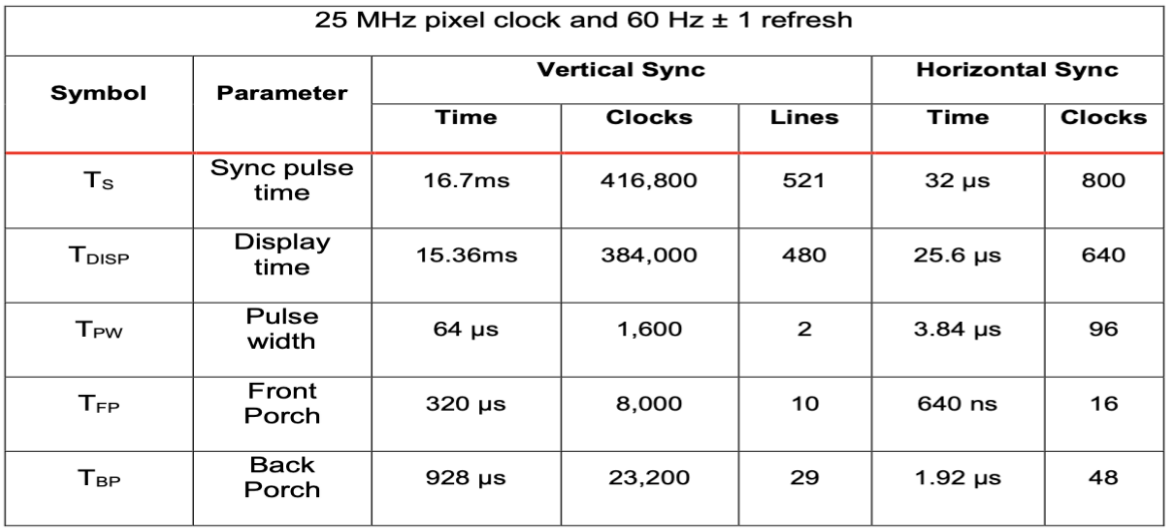
```
with random_time_v select

random_time <= 60/diff when "0000",
 90/diff when "0001",
 120/diff when "0010",
 150/diff when "0011",
 180/diff when "0100",
 210/diff when "0101",
 240/diff when "0110",
 270/diff when "0111",
 300/diff when "1000",
 330/diff when "1001",
 170/diff when "1010",
 240/diff when "1011",
 130/diff when "1100",
 150/diff when "1101",
 300/diff when "1110",
 120/diff when "1111",
 0 when others;
```

With- Select code for the time

4.2.2 VGA DRIVER

A VGA (Video Graphics Array) driver is a module that generates the necessary signals to display images on a monitor. It works by creating precise timing signals for horizontal and vertical synchronization, which dictate the refresh rate and resolution of the display. The driver converts digital pixel data into corresponding analog signals for the red, green, and blue (RGB) channels, synchronizing these with the monitor's refresh cycles. By controlling these signals, the VGA driver can display the desired graphics or images on the screen.

Our VGA driver operates, somewhat, slightly differently than the one we've used in lab 7. It operates as an independent module that can paint the current pixel in the designated color and gives the current pixels horizontal and vertical components as output for later usage in the top module.



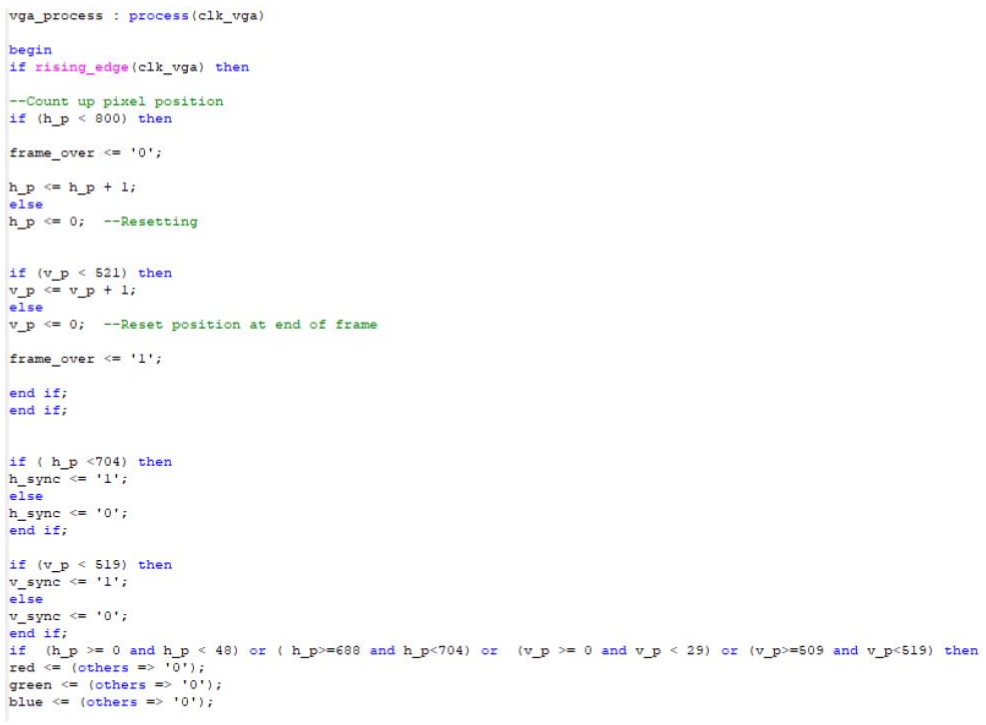
25 MHz pixel clock and 60 Hz ± 1 refresh						
Symbol	Parameter	Vertical Sync			Horizontal Sync	
		Time	Clocks	Lines	Time	Clocks
T_S	Sync pulse time	16.7ms	416,800	521	32 µs	800
T_{DISP}	Display time	15.36ms	384,000	480	25.6 µs	640
T_{PW}	Pulse width	64 µs	1,600	2	3.84 µs	96
T_{FP}	Front Porch	320 µs	8,000	10	640 ns	16
T_{BP}	Back Porch	928 µs	23,200	29	1.92 µs	48

VGA signal timing table

VGA driver takes 25 MHz clock as input. It generates a horizontal synchronization signal of 800 clock cycles (represented as *clk_vga* in our code) and a vertical synchronization signal of 521 horizontal synch. periods. We also need to account for the back porch and front porch times by assigning '0' to red, green, blue outputs which demands the electron gun to stay off during these intervals.

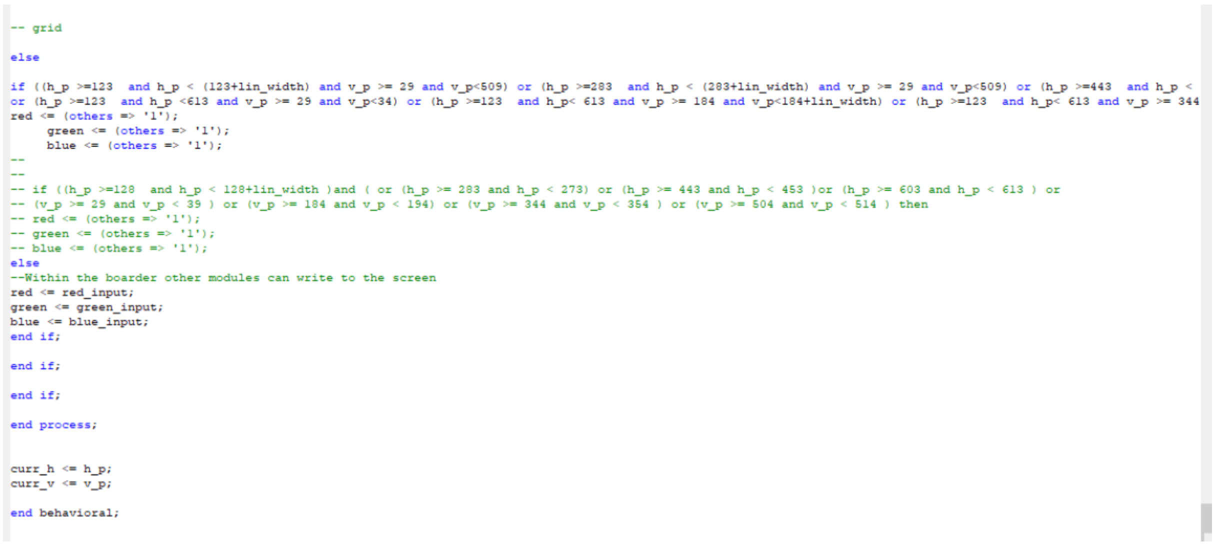
- If horizontal pulse is between 0-48 (back-porch) or 688-704 (front-porch) => display black
- If vertical pulse is between 0-29 (back-porch) or 509-519 (front-porch) => display black

If h*_p* and *v_p* (horizontal and vertical positions of the current pixel) are not in these bounds then the pixel is painted according to the red, green, blue inputs provided by the higher level entity. In response, the module outputs a sequence of red, green, blue signals in synchronization with horizontal and vertical pulses. The VGA driver module outputs another signal named "frame_over" which tells the higher-level modules that the electron gun has finished scanning the entire frame. Note that it is '1' for a single *vga_clock* period and otherwise '0'. Additionally, the grid which is a static entity (remaining always on the screen) is drawn inside the VGA driver module. The code for the VGA driver is given below:



```
vga_process : process(clk_vga)

begin
if rising_edge(clk_vga) then

--Count up pixel position
if (h_p < 800) then

frame_over <= '0';

h_p <= h_p + 1;
else
h_p <= 0;  --Resetting


if (v_p < 521) then
v_p <= v_p + 1;
else
v_p <= 0;  --Reset position at end of frame

frame_over <= '1';

end if;
end if;


if ( h_p <704) then
h_sync <= '1';
else
h_sync <= '0';
end if;

if (v_p < 519) then
v_sync <= '1';
else
v_sync <= '0';
end if;
if  (h_p >= 0 and h_p < 48) or ( h_p>=688 and h_p<704) or  (v_p >= 0 and v_p < 29) or (v_p>=509 and v_p<519) then
red <= (others => '0');
green <= (others => '0');
blue <= (others => '0');
```

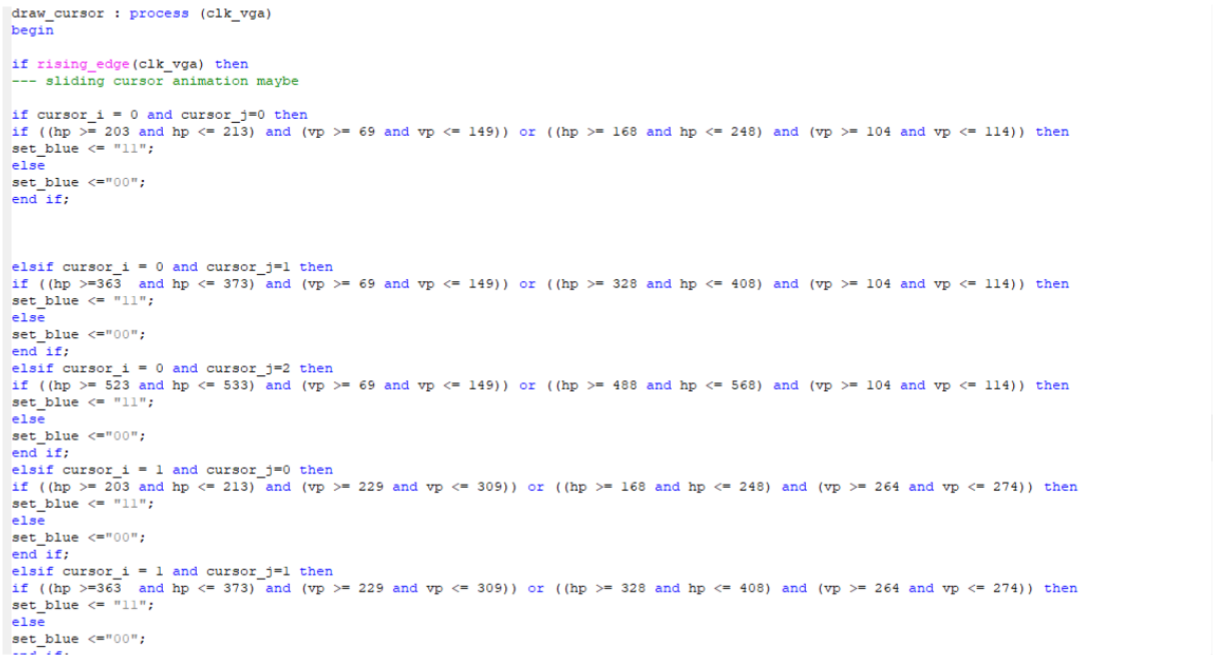
The code snippet of the VGA driver module showing timing signal arrangement



```
-- grid

else

if ((h_p >=123  and h_p < (123+lin_width) and v_p >= 29 and v_p<509) or (h_p >=283  and h_p < (283+lin_width) and v_p >= 29 and v_p<509) or (h_p >=443  and h_p <
or (h_p >=123  and h_p <613 and v_p >= 29 and v_p<34) or (h_p >=123  and h_p< 613 and v_p >= 184 and v_p<184+lin_width) or (h_p >=123  and h_p< 613 and v_p >= 344
red <= (others => '1');
     green <= (others => '1');
     blue <= (others => '1');
--
--
-- if ((h_p >=128  and h_p < 128+lin_width )and ( or (h_p >= 283 and h_p < 273) or (h_p >= 443 and h_p < 453 )or (h_p >= 603 and h_p < 613 ) or
-- (v_p >= 29 and v_p < 39 ) or (v_p >= 184 and v_p < 194) or (v_p >= 344 and v_p < 354 ) or (v_p >= 504 and v_p < 514 ) then
-- red <= (others => '1');
-- green <= (others => '1');
-- blue <= (others => '1');
else
--Within the boarder other modules can write to the screen
red <= red_input;
green <= green_input;
blue <= blue_input;
end if;

end if;

end if;

end process;


curr_h <= h_p;
curr_v <= v_p;

end behavioral;
```

Ever Present Grid drawn in the VGA module

We display the shapes we want to draw to the screen by carefully adjusting which pixels we want to draw in which color. For example, the process for drawing the cursor is given below:



```vhdl
draw_cursor : process (clk_vga)
begin

if rising_edge(clk_vga) then
--- sliding cursor animation maybe

if cursor_i = 0 and cursor_j=0 then
if ((hp >= 203 and hp <= 213) and (vp >= 69 and vp <= 149)) or ((hp >= 168 and hp <= 248) and (vp >= 104 and vp <= 114)) then
set_blue <= "11";
else
set_blue <="00";
end if;


elsif cursor_i = 0 and cursor_j=1 then
if ((hp >=363  and hp <= 373) and (vp >= 69 and vp <= 149)) or ((hp >= 328 and hp <= 408) and (vp >= 104 and vp <= 114)) then
set_blue <= "11";
else
set_blue <="00";
end if;
elsif cursor_i = 0 and cursor_j=2 then
if ((hp >= 523 and hp <= 533) and (vp >= 69 and vp <= 149)) or ((hp >= 488 and hp <= 568) and (vp >= 104 and vp <= 114)) then
set_blue <= "11";
else
set_blue <="00";
end if;
elsif cursor_i = 1 and cursor_j=0 then
if ((hp >= 203 and hp <= 213) and (vp >= 229 and vp <= 309)) or ((hp >= 168 and hp <= 248) and (vp >= 264 and vp <= 274)) then
set_blue <= "11";
else
set_blue <="00";
end if;
elsif cursor_i = 1 and cursor_j=1 then
if ((hp >=363  and hp <= 373) and (vp >= 229 and vp <= 309)) or ((hp >= 328 and hp <= 408) and (vp >= 264 and vp <= 274)) then
set_blue <= "11";
else
set_blue <="00";
```

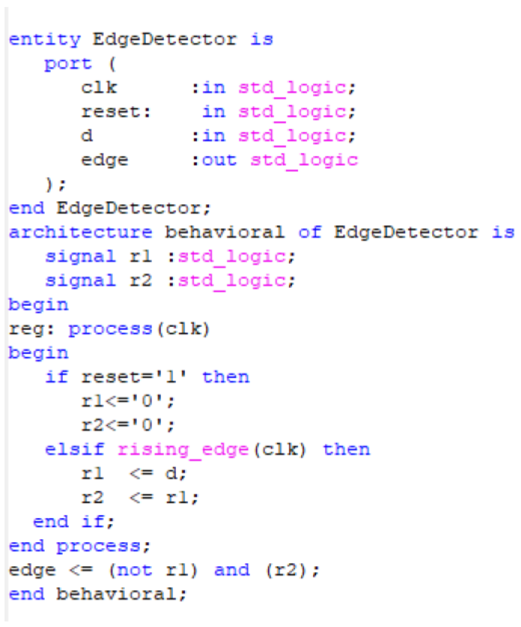
The code depicting the generation od the cursor via the color assignment depending on the VGA signal positions

If the outputs of the VGA driver module *h_p* & *v_p* representing the current pixel location are in some predetermined intervals, we assign the *set_blue* signal (which is an input of our VGA driver component) to "11". Essentially, this corresponds to drawing a blue rectangle on the screen. Drawing more complex such as circles and curved lines are harder to implement via this design. Luckily, rectangles were enough to accomplish our goals. However, defining the exact boundaries for each shape was a tedious process.

4.2.3 Button Circuit

Input signals we receive from the five push buttons (*BTNL*, *BTNR*, *BTNU*, *BTND and BTN*S) are unsteady and "bouncy" demonstrating a lot of irregular peaks until it stabilizes. This is due to the oscillation of the electronic and mechanical structure of the buttons. We use the debouncer circuit that we've implemented in lab 6 [3]. which is essentially four FDCE's stacked serially.

Upon obtaining a steady signal we use the edge detector entity that detects the falling edge. It outputs '1' for a single period of the top-level clock after it's been released instead of staying at '1' for the whole duration when the button is pressed down. This approach prevents the misconception (on the FPGA side) that the button is pressed multiple times in the meantime. Such an edge detection can be obtained by the following behavioral description in VHDL:



```
entity EdgeDetector is
   port (
      clk      :in std_logic;
      reset:    in std_logic;
      d        :in std_logic;
      edge     :out std_logic
   );
end EdgeDetector;
architecture behavioral of EdgeDetector is
   signal r1 :std_logic;
   signal r2 :std_logic;
begin
reg: process(clk)
begin
   if reset='1' then
      r1<='0';
      r2<='0';
   elsif rising_edge(clk) then
      r1  <= d;
      r2  <= r1;
  end if;
end process;
edge <= (not r1) and (r2);
end behavioral;
```

Debouncer Circuit

In this behavioral code at each rising edge of the clock the values of d input shifts to the *r1* and that of *r1* shifts to the *r2*, whose difference yields an output (edge) of "1" only when there is a falling signal level received from the debouncer circuit, which allows the user to move the cursor only when the button is released.

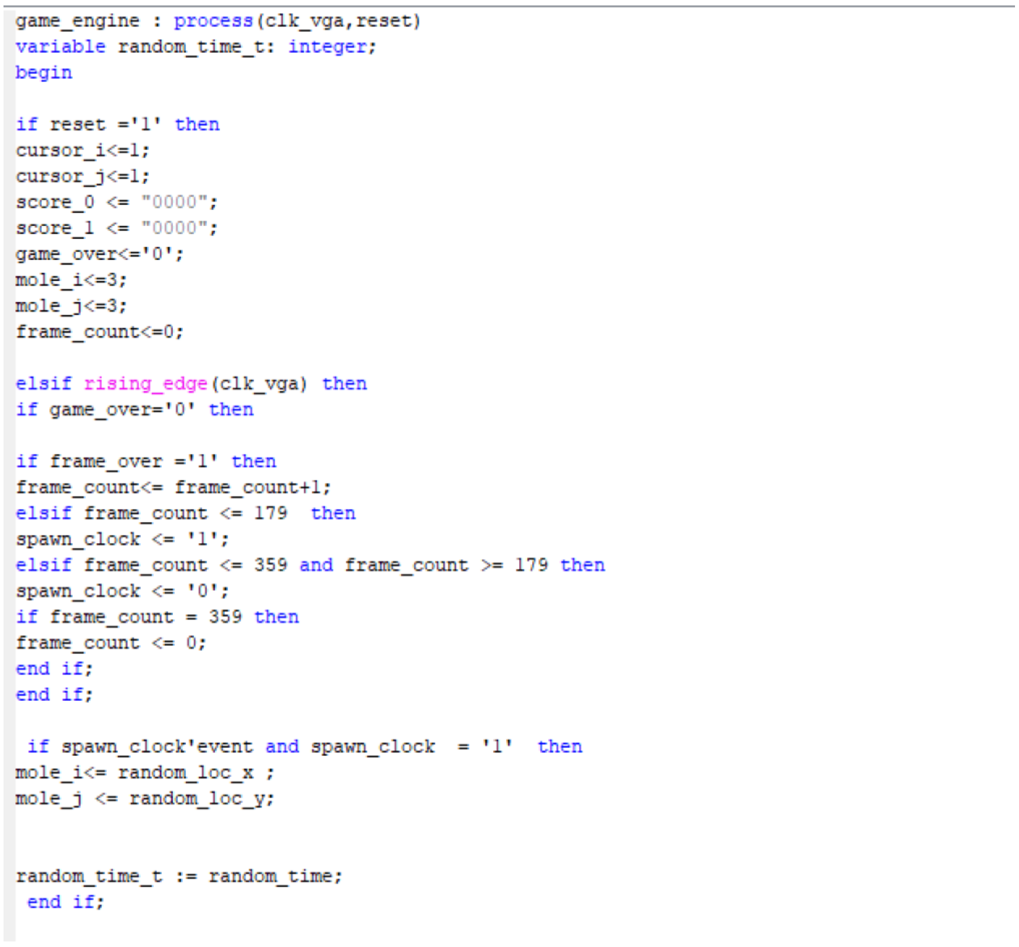
4.2.4 Seven Segment

The implementation of the seven segment which holds the score of the player is greatly similar to the methods applied in *lab 5* which involves obtaining the 7-segment representation of each number in the range of 0 to 9, which becomes the input of a sequential seven segment driver operation *nexys3_sseg_driver [4]* which itself has a specific clock division rate for 2 GHz refresh rate.

4.3 Game Engine

Game engine is a process defined in the top module that is essentially the brains of the game. Every possible condition is checked, and corresponding signals are assigned in accordance. Signals (of type integer) *mole_i*, *mole_j*, *cursor_i* and *cursor_j* determines the location of the mole and the cursor in the 3 by 3 grid. Range of these signals are 0-2. Thus, assigning 3 to either i'th or j'th position is no different than making the element disappear which we've commonly utilized throughout the process. *game_over* signal (of type std_logic) tracks whether the user has reached the score of 10 which ends the game. *frame_count* signal (of type integer) keeps track of how many frames have passed.

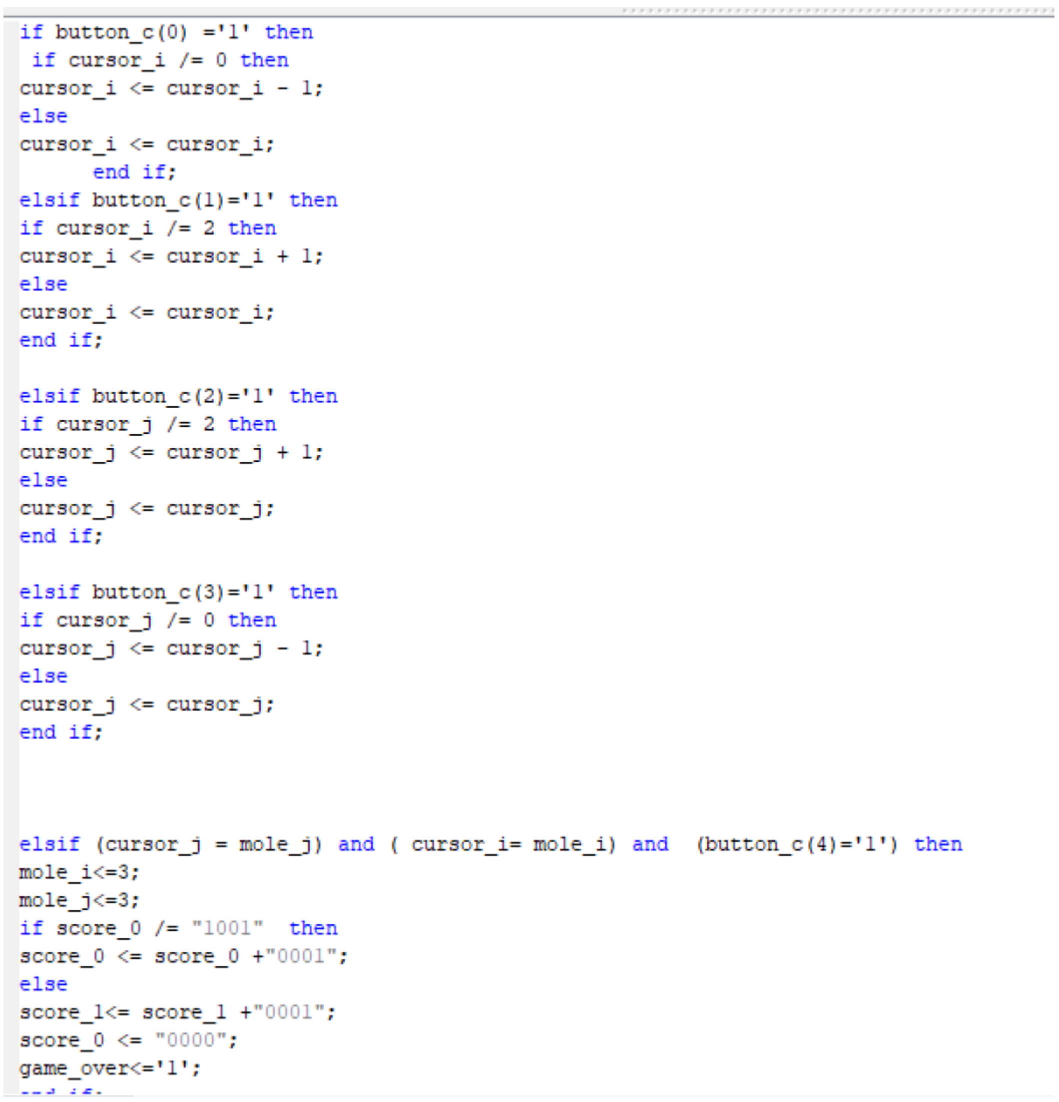
Firstly, the process checks whether or not reset is '1'. If so, it initializes all the parameters. Then, if "*game_over*" = '0' (the game has not yet ended) at the end of every frame it increments the frame count. After 360 frames or 6 seconds, frame count is reset back to zero. This is because all of our timing design is based on the 6 second intervals mole will spawn in. From the frame count signal, we generate an additional clock (for design purposes and reliability) named *spawn_clock*. At the rising edge of the *spawn_clock*, we assign a random location and random time for the mole. Note that random time is defined in terms of frame counts.



```
game_engine : process(clk_vga,reset)
variable random_time_t: integer;
begin

if reset ='1' then
cursor_i<=1;
cursor_j<=1;
score_0 <= "0000";
score_1 <= "0000";
game_over<='0';
mole_i<=3;
mole_j<=3;
frame_count<=0;

elsif rising_edge(clk_vga) then
if game_over='0' then

if frame_over ='1' then
frame_count<= frame_count+1;
elsif frame_count <= 179  then
spawn_clock <= '1';
elsif frame_count <= 359 and frame_count >= 179 then
spawn_clock <= '0';
if frame_count = 359 then
frame_count <= 0;
end if;
end if;

 if spawn_clock'event and spawn_clock  = '1'  then
mole_i<= random_loc_x ;
mole_j <= random_loc_y;


random_time_t := random_time;
 end if;
```

Game engine with the synchronization signal for the mole spawning on the screen

Thereafter, whether random time assigned for the mole has expired should be dealt with. If so, it assigns *mole_i*, *mole_j* <= (3,3) which effectively makes the mole disappear. There remains to check whether the buttons are pressed. If one of the direction buttons (up, down, left, right) is pressed *cursor_i* , *cursor_j* are incremented or decremented accordingly. It of course needs to ensure that the cursor will not go out of bounds. For example, if the cursor is already in the upper row and the user presses the up button, it will stay in the same position regardless. Finally, if the central button (*BTNS*) is pressed and if the mole and cursor locations are the same; first and foremost, it assigns *mole_i*, *mole_j* <= (3,3) which makes the mole disappear and secondly it increments the score by one. The code of the explanation is as the following:



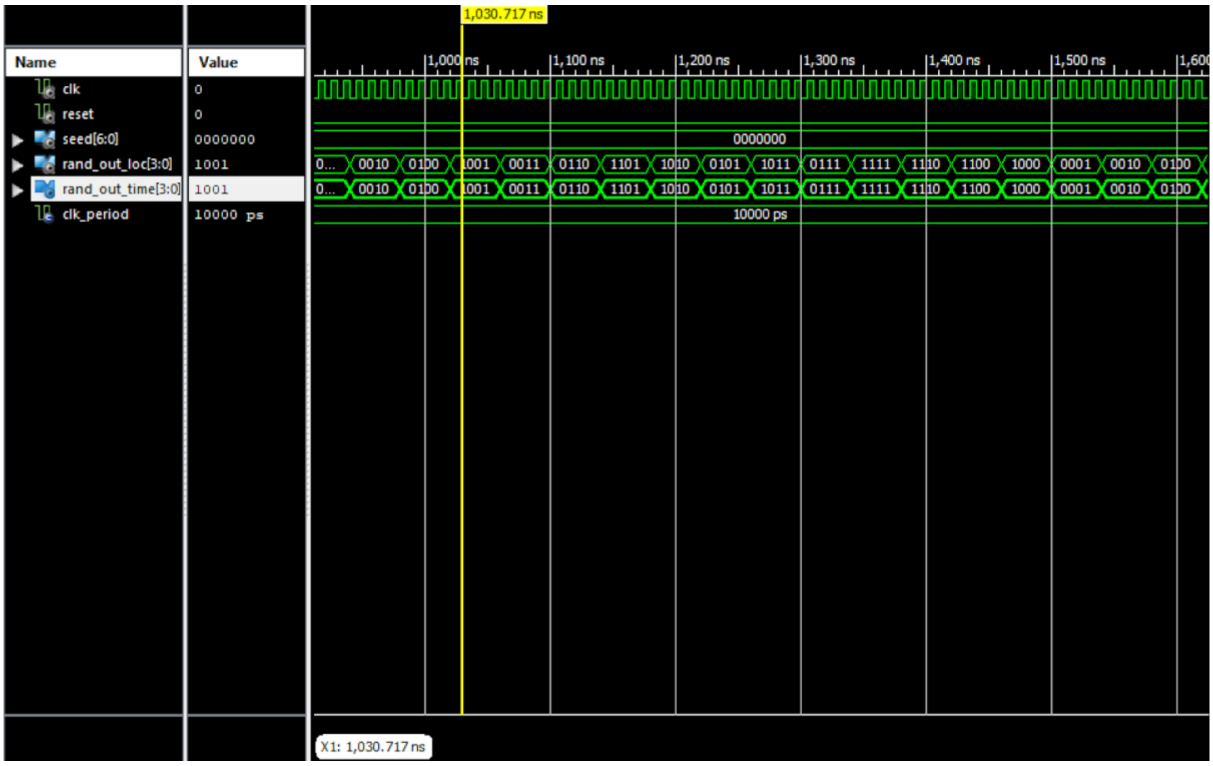
```
if button_c(0) ='1' then
 if cursor_i /= 0 then
cursor_i <= cursor_i - 1;
else
cursor_i <= cursor_i;
      end if;
elsif button_c(1)='1' then
if cursor_i /= 2 then
cursor_i <= cursor_i + 1;
else
cursor_i <= cursor_i;
end if;

elsif button_c(2)='1' then
if cursor_j /= 2 then
cursor_j <= cursor_j + 1;
else
cursor_j <= cursor_j;
end if;

elsif button_c(3)='1' then
if cursor_j /= 0 then
cursor_j <= cursor_j - 1;
else
cursor_j <= cursor_j;
end if;



elsif (cursor_j = mole_j) and ( cursor_i= mole_i) and  (button_c(4)='1') then
mole_i<=3;
mole_j<=3;
if score_0 /= "1001"  then
score_0 <= score_0 +"0001";
else
score_1<= score_1 +"0001";
score_0 <= "0000";
game_over<='1';
```

The code describing the behavior of the push buttons in the game engine

5 RESULTS

The preliminary results are acquired by the simulations of certain critical subsections, which are as the following:

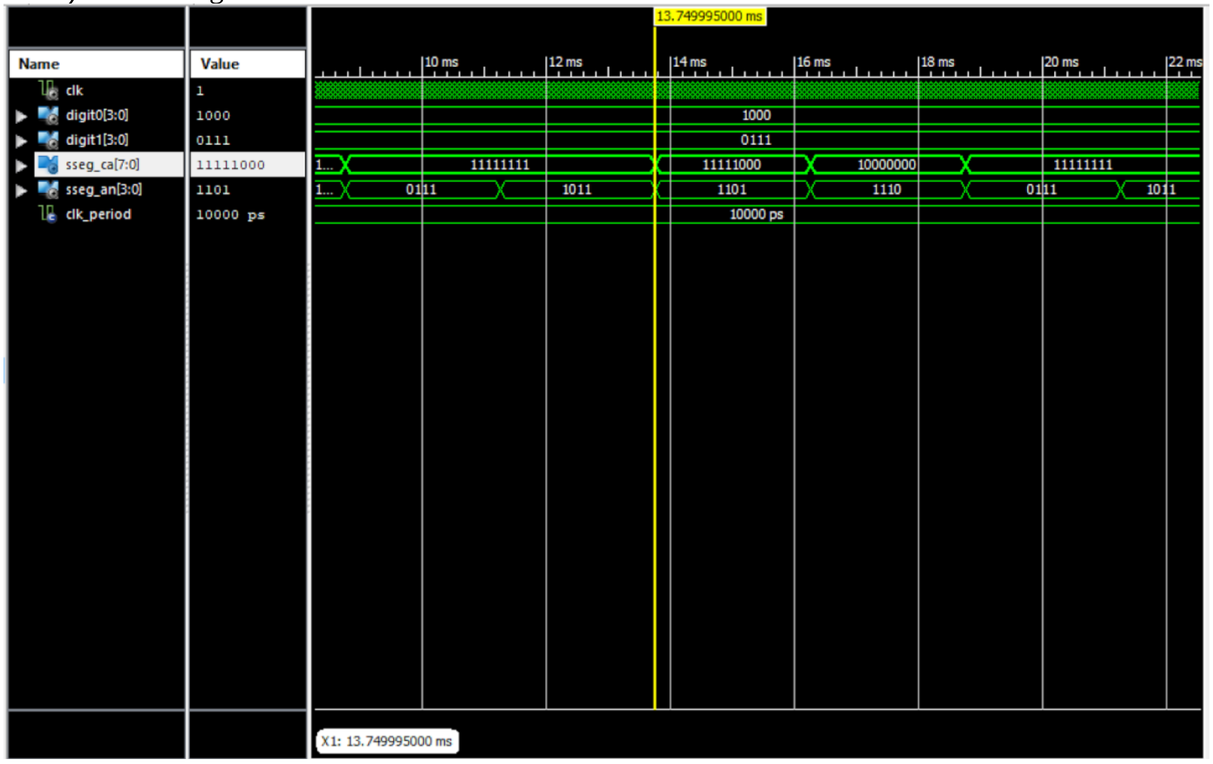
1) Random Generator



Simulation Results for the Random Generator

As expected, it is observed that the results do not converge although fairly long time has passed and 4-bit numbers appear to be random to those who are not informed about the algorithm.

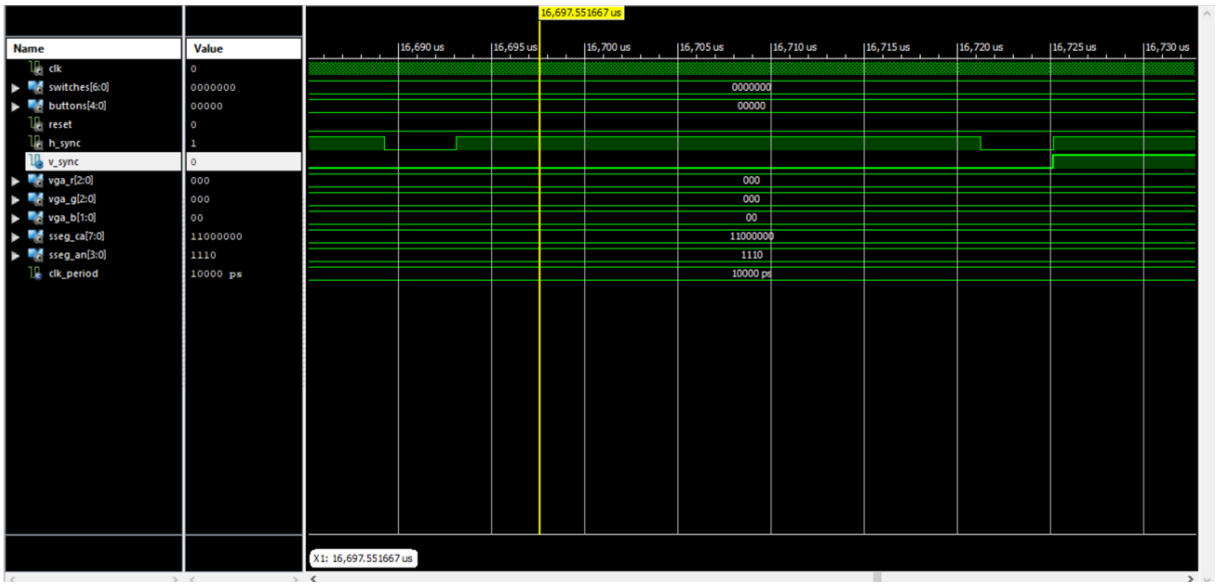
2) Seven Segment



Simulation results for the seven-segment code

As can be seen from the above simulation snippet, seven segment display outputs sweep across the LED’s as expected according to each binary digit which are supposed to be displayed

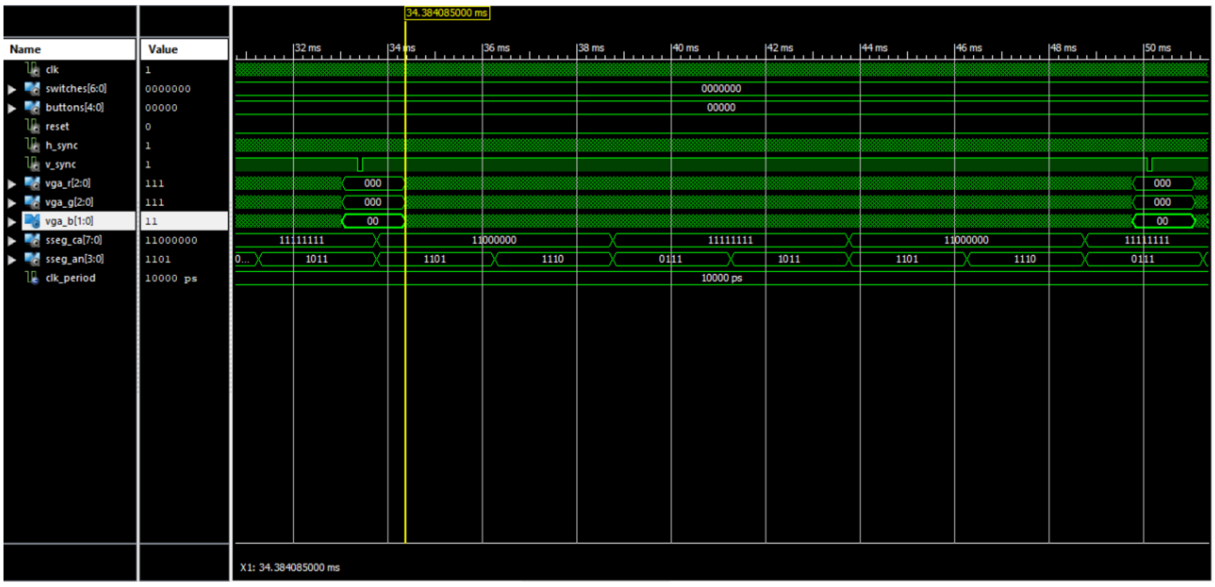
3) Overall Design



Name	Value
clk	0
switches[6:0]	0000000
buttons[4:0]	00000
reset	0
h_sync	1
v_sync	0
vga_r[2:0]	000
vga_g[2:0]	000
vga_b[1:0]	00
sseg_ca[7:0]	11000000
sseg_an[3:0]	1110
clk_period	10000 ps

Simulation results of the overall game (close-up)

We have generated a horizontal synchronization signal of 8000 ns as can be confirmed from the simulation

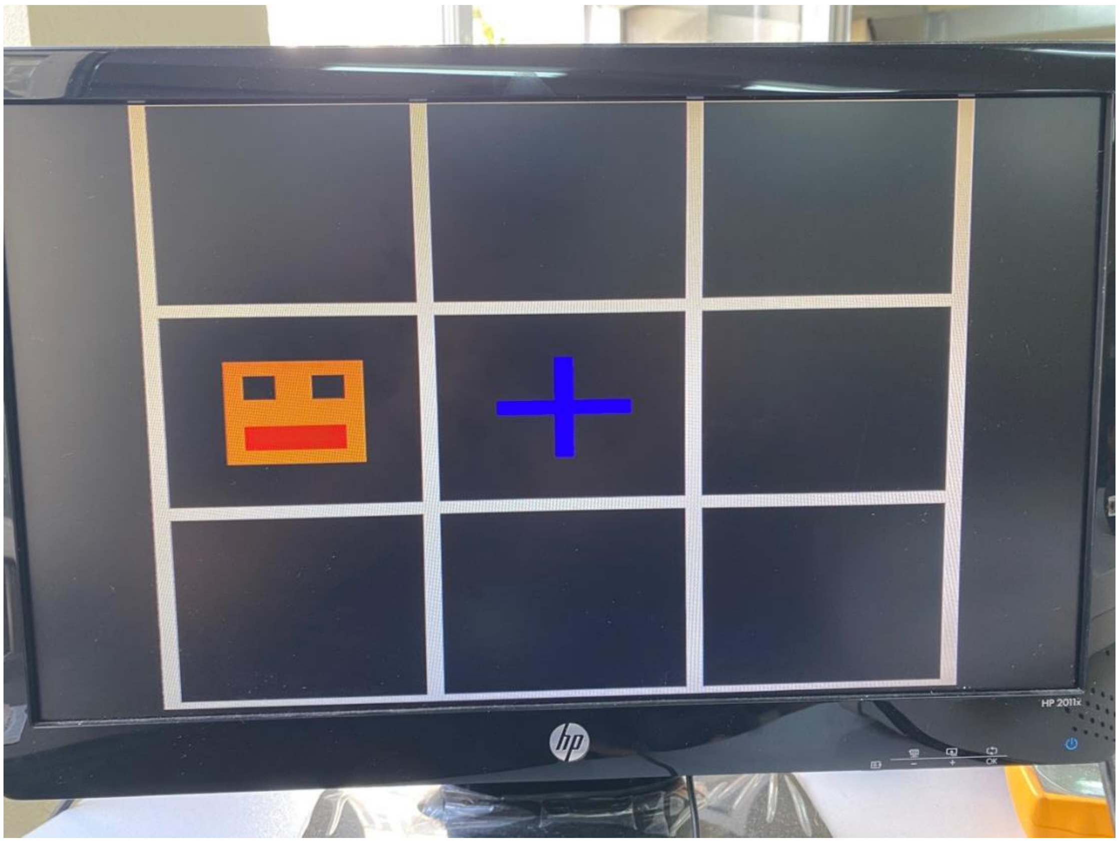


Name	Value
clk	1
switches[6:0]	0000000
buttons[4:0]	00000
reset	0
h_sync	1
v_sync	1
vga_r[2:0]	111
vga_g[2:0]	111
vga_b[1:0]	11
sseg_ca[7:0]	11000000
sseg_an[3:0]	1101
clk_period	10000 ps

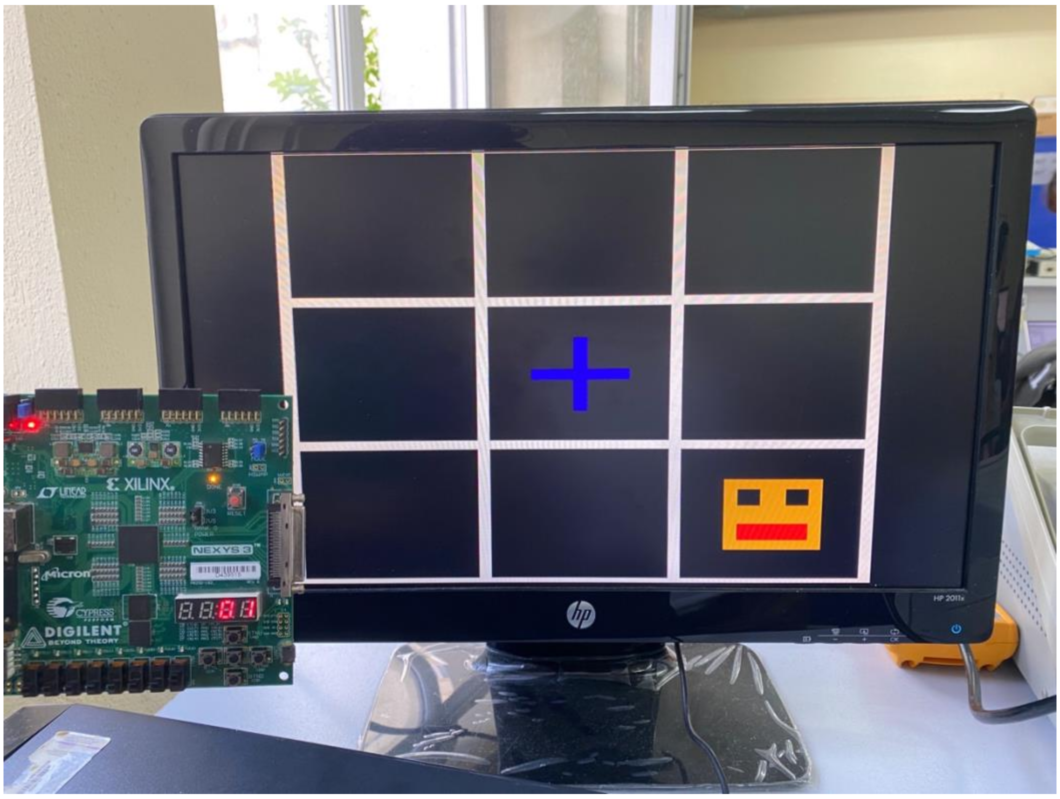
Simulation results of the overall game

We observe the front and back porch and necessary signal synchronizations as expected. Around the pulse width, color output signals is reinitialized to”000”

The functionality of the game can be observed from the VGA monitor whose photographs are provided below:



The picture depicting the mole and the cursor



The photo depicting the increased score, mole and the cursor

6 CONCLUSION

The whack-a-mole game realized through the VHDL hardware language description code provides an example of how FPGA's can be programmed to achieve the certain tasks, even a game, which makes use of various aspects of FPGA from seven segment LEDs to the VGA output property to build an endearing and interactive game as such. Crucial mathematical techniques such as Fibonacci LFSR along with the efficient implementation of displaying necessary content to the monitor is successfully achieved by Xilinx ISE (Integrated Synthesis Environment) tool, which enables us to write the structural and behavioral code, error debugging , as well as simulating the components before the real life implementation of the project

7 REFERENCES

[1] https://www.moria.us/articles/demystifying-the-lfsr/
[2] https://en.wikipedia.org/wiki/Linear-feedback_shift_register
[3]-[4] EE 244 Lab Manuals

APPENDIX

A READ ME

- In order to implement the overall design, the user should have a Nexys3™ FPGA board, according to which the design is optimized. Upon loading the FPGA board with the bit file, one can opt for resetting the game in advance with the T5 switch on the FPGA which also allows reinitializing the game.
- Switches called V8 U8 N8 M8 V9 and T10 constitute the seed inputs whereby the randomization can be chosen before starting the game. The reset is achieved by the activation of T5 switch.
- Push buttons have the following functionalities:

  -> BTNU: moving the cursor upwards

  -> BTND: moving the cursor downwards

  -> BTNL: moving the cursor to the left

  ->BTNR: moving the cursor to the right

  ->B8: clicking on the mole

B USERS' MANUAL

1. Obtain the bit-file either by synthesizing the VHDL files for the project or take the already synthesized one
2. Transfer the bit-file to Nexys3 Board by Digilent Adept software
3. Program the device
4. Take a 2-sided HDMI VGA cable, connect one side to the output port of the Nexys3 Board and the other to the input driver of a 800x640 monitor with 69 GHz refresh rate
5. Follow the instructions on ***Read Me*** section to control the player
6. If you receive no input signal on the monitor, try resetting the design by sliding the T11 switch up & down.

7. If you still don't get any signal check the compatibility of your monitor and your FPGA board.