
Designing of Google Dinosaur Game using FPGA
By:
N150178,N150194
N150997,N151087
Rajiv Gandhi University of Knowledge and Technologies

CONTENTS:

Abstract:

Computer games are mostly used for entertainment purposes. Currently, many consumer game machines are based on the operations of processors. However, in the field programmable gate array (FPGA) research field, various FPGA game solvers have been developed recently. The processing speeds of such FPGA game solvers can reach about 1000 times faster than processor-based operations. And Field Programmable Gate Array (FPGA) technology is becoming more popular among Application Specific Integrated Circuit (ASIC) developers. The ease of development and maintainability makes FPGAs an attractive option in many performance and efficiency in critical applications. So in this project we have designed an offline Google Dinosaur Game based on FPGA using Verilog HDL. Game players can move and control blocks with the USB interface keyboard . Game video is shown on a VGA monitor. Here the Real work behind this gaming project is to interface VGA monitor or projector, keyboard, interface with FPGA and control them through programming. The successful transplant of these games provides a template for the development of other visual systems in the FPGA. Here everything is implemented in hardware on FPGA.

Introduction:

1. Software and Tools:

a. Xilinx Vivado : Vivado design suite is a software suite produced by Xilinx for synthesis and analysis of HDL designs. The vivado system edition introduces high-level synthesis, with a toolchain that converts C code into programmable logic. The Vivado IP Integrator allows engineers to quickly integrate and configure IP from the later Xilinx IP library. The Integrator is also tuned for Mathworks Simulink designs built with Xilinx's System Generator and Vivado High-Level Synthesis.

b. Creation of project in Xilinx Vivado:

1)Open Vivado 2016.1 which is on the Desktop

2) Click Create New Project to start the wizard. You will see Create .A New Vivado Project dialog box. Click Next.

3)Now we will see a new window named 'New Project'. In that we have to enter the project name and Browse the project location. Click Next

4) Select RTL Project option in the Project Type form, and click Next.

5) In the Default Part form, using the Parts option and various drop-down fields select family as ARTIX-7,package as 'csg324' and, select the XC7A100TCSG324-3 part. Click Next.

6)For a new project summary Click FINISH.

7)Now one window will open in which we can write the verilog Program.

8)To write a verilog program Go to the options on the left side panel in that select Project Manager > Add Sources > Add or Create Design Sources > create file

9)Give name to the file in the file name section. Click ok and finish.

10) Now in the Tool bar Go to windows > sources. Now write the program in the file you have created.

11) Next Go the left panel and Go to the Simulation > Run simulation . Now We will see the simulation window.

12) Here by giving the values (Give right click on the variables and select the option Force Constant enter the values) we can check the simulation results and we will see the waveforms.

13) Now Go to RTL Analysis > Open Elaborated Design > Schematic . We can see the RTL Schematic. 14) And here we have to assign the pins of FPGA Board in the I/O ports > scalar ports. Here we have to give Package pins and I/O standard. Now save this file by giving some name. This is called constraint file

15) Now Go to the options in the left panel Synthesis > Open synthesized Design > Schematic . Here we will the Technology schematic.

16) Now Go to the options in the left panel Implementation> Open Implemented Design. Here placing and routing steps are taken place.

17) Finally Generate the bitstream to program our FPGA by going to the option Program and Debug > Generate Bitstream.

18)In order to configure our FPGA we have to dump this bit file into the FPGA by clicking the option Open Hardware Manager > Open target > Program Device . Now browse the bit file and click the program. Finally we can see the results on Hardware (FPGA).

9)Give name to the file in the file name section. Click ok and finish.

10) Now in the Tool bar Go to windows > sources. Now write the program in the file you have created.

11) Next Go the left panel and Go to the Simulation > Run simulation . Now We will see the simulation window.

12) Here by giving the values (Give right click on the variables and select the option Force Constant enter the values) we can check the simulation results and we will see the waveforms.

13) Now Go to RTL Analysis > Open Elaborated Design > Schematic . We can see the RTL Schematic. 14) And here we have to assign the pins of FPGA Board in the I/O ports > scalar ports. Here we have to give Package pins and I/O standard. Now save this file by giving some name. This is called constraint file

15) Now Go to the options in the left panel Synthesis > Open synthesized Design > Schematic . Here we will use the Technology schematic.

16) Now Go to the options in the left panel Implementation> Open Implemented Design. Here placing and routing steps are taken place.

17) Finally Generate the bitstream to program our FPGA by going to the option Program and Debug > Generate Bitstream.

18)In order to configure our FPGA we have to dump this bit file into the FPGA by clicking the option Open Hardware Manager > Open target > Program Device . Now browse the bit file and click the program. Finally we can see the results on Hardware (FPGA).

2. NEXYS 4 DDR FPGA

The Nexys4 DDR board is a complete, ready-to-use digital circuit development platform based on the latest Artix-7™ Field Programmable Gate Array (FPGA) from Xilinx®. With its large, high-capacity FPGA (Xilinx part number XC7A100T-1CSG324C), generous external memories, and collection of USB, Ethernet, and other ports, the Nexys4 DDR can host designs ranging from introductory combinational circuits to powerful embedded processors. Several built-in peripherals, including an accelerometer, temperature sensor, MEMs digital microphone, a speaker amplifier, and several I/O devices allow the Nexys4 DDR to be used for a wide range of designs without needing any other components.

The Artix-7 FPGA is optimized for high performance logic, and offers more capacity, higher performance, and more resources than earlier designs.

Artix-7 100T features include:

· 15,850 logic slices, each with four 6-input LUTs and 8 flip-flops

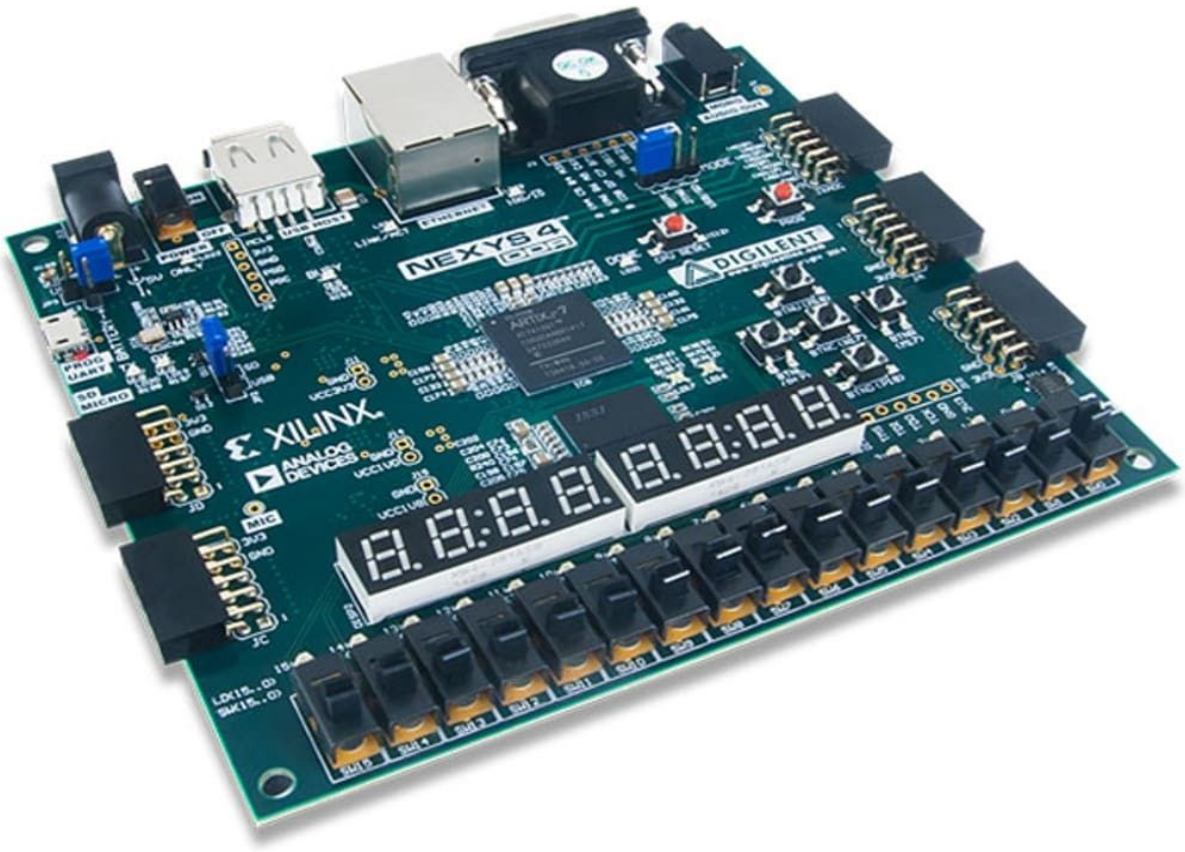
· 4,860 Kbits of fast block RAM

· Six clock management tiles, each with phase-locked loop (PLL)

· 240 DSP slices

· Internal clock speeds exceeding 450 MHz

· On-chip analog-to-digital converter (XADC)



Nexys4

The Nexys4 also offers an improved collection of ports and peripherals, including:

· 16 user switches

· USB-UART Bridge

· 12-bit VGA output

· 3-axis accelerometer

· 128MiB DDR2

· Pmod for XADC signals

· 16 user LEDs

· Two tri-color LEDs

· PWM audio output

· Temperature sensor

· Serial Flash

· Digilent USB-JTAG port for FPGA programming and communication

· Two 4-digit 7-segment displays

· Micro SD card connector

· PDM microphone

· 10/100 Ethernet PHY

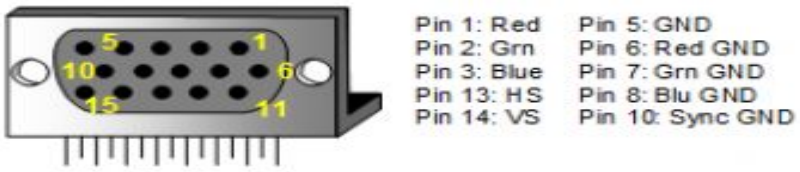
· Four Pmod ports

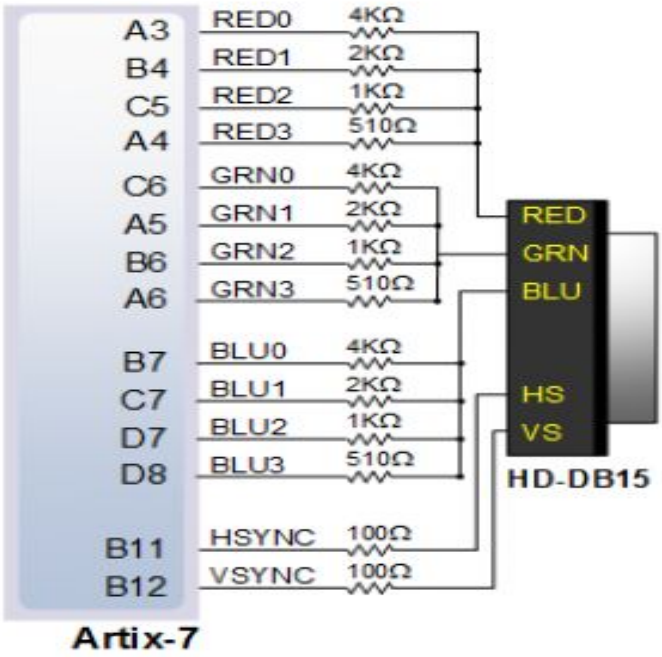
· USB HID Host for mice, keyboards and memory sticks

VGA INTERFACING

The most important thing to develop Games is the ‘Working Display System’. Most people use VGA(Video Graphic Array) in order to display Games. So we have to interface this VGA to FPGA.

VGA is an analogue video standard using a 15-pin D-sub Connector. VGA has five main signal pins: one for each of red, green, and blue and two for sync. Horizontal sync demarcates a line. Vertical sync demarcates a screen, also known as a frame. The Nexys4 board uses 14 FPGA signals to create a VGA port with 4 bits-per-colour and the two standard sync signals (HS – Horizontal Sync, and VS – Vertical Sync).



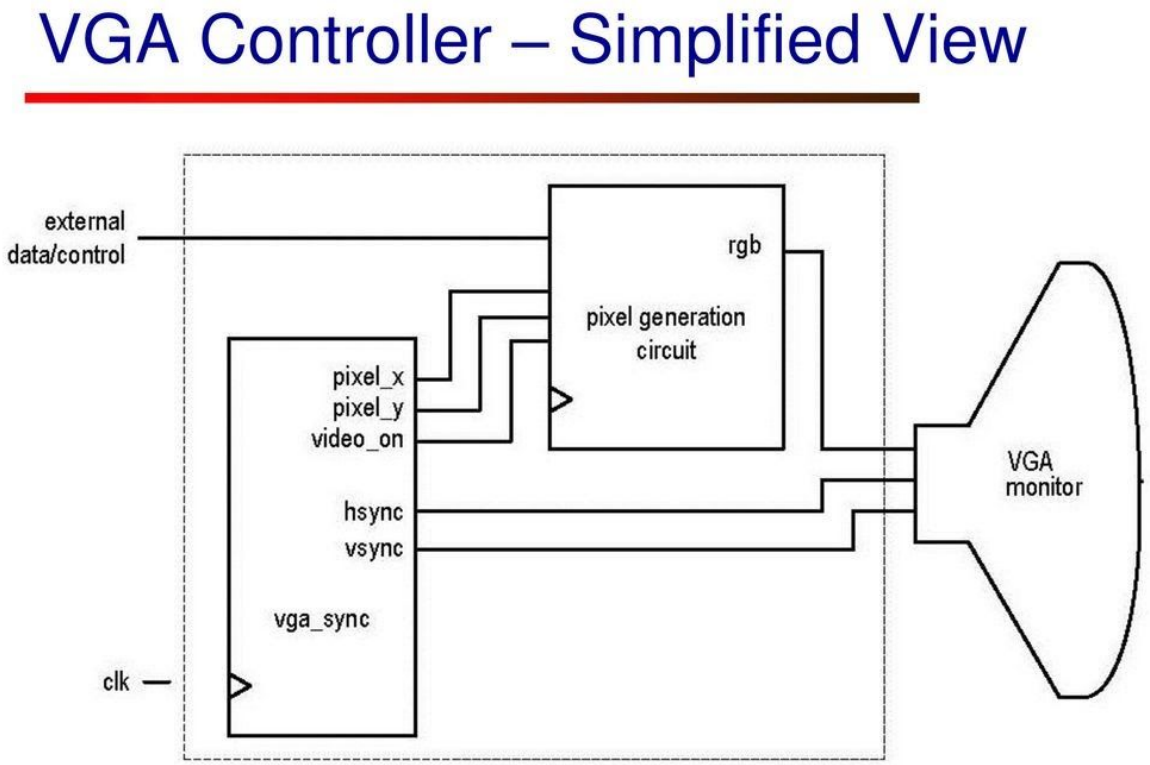


Using this circuit, 4096 different colours can be displayed, one for each unique 12-bit pattern. A video controller circuit must be created in the FPGA to drive the sync and colour signals with the correct timing in order to produce a working display system.

The following figures contain a synchronization circuit, labeled vga-synchronization circuit, and a pixel generation circuit. Finally this video controller generates the synchronization signals and outputs data pixels serially. The vga-sync circuit generates timing and synchronization signals. The HS(hsync) and VS(vsync) signals are connected to the VGA port to control the horizontal and vertical scans of the monitor. The two signals are decoded from the internal counters, whose outputs are the hcounter and vcounter signals. The hcounter and vcounter signals indicate the relative positions of the scans and essentially specify the location of the current pixel. The vga-synchronization circuit also generates the blank(~video-on) signal to indicate whether to enable or disable the display.

The pixel generation circuit generates the three video signals, which are referred to as the RED, GREEN,BLUE signals. A colour value is obtained according to the current coordinates of the pixel (the hcounter and vcounter signals) and the external control and data signals.

i.VGA SYNCHRONIZATION



The video synchronization circuit generates the hsync signal, which specifies the required time to traverse (scan) a row, and the vsync signal, which specifies the required time to traverse (scan) the entire screen. Subsequent discussions are based on a 640-by-480 VGA screen with a 25-MHzpixel rate, which means that 25M pixels are processed in a second. Note that this resolution is also known as the VGA mode. The screen of a CRT monitor usually includes a small black border, as shown at the top of the figure. The middle rectangle is the visible portion. Note that the coordinate of the vertical axis increases downward. The coordinates of the top-left and bottom-right comers are (0,0) and (639,479), respectively.

a.Horizontal synchronization

- A detailed timing diagram of one horizontal scan is shown in the above Figure. A period of the hsync signal contains 800 pixels and can be divided into four regions:
- Display :It is the region where the pixels are actually displayed on the screen. The length of this region is 640 pixels.
- Retrace: It is the region in which the electron beams return to the left edge. The video signal should be disabled (i.e., blank), and the length of this region is 96 pixels.
- Right border: It is the region that forms the right border of the display region. It is also known as the front porch (i.e., porch before retrace). The video signal should be disabled, and the length of this region is 9 pixels.
- Left border : It is the region that forms the left border of the display region. It is also known as the back porch (i.e., porch after retrace). The video signal should be disabled, and the length of this region is 55 pixels.
- Note that the lengths of the right and left borders may vary for different brands of monitors.

The hsync signal can be obtained by a special mod-800 counter and a decoding circuit. The counts are marked on the top of the HS(hsync) signal in the Figure . We intentionally start the counting from the beginning of the display region. This allows us to use the counter output as the horizontal (x-axis) coordinate. This output constitutes the hcounter signal. The hsync signal goes low when the counter's output is between 648 and 744.

Note that the CRT monitor should be blank in the right and left borders and during retrace. We used the blank(~h-video-on) signal to indicate whether the current horizontal coordinate is in the displayable region. It is asserted only when the pixel count is smaller than 640.

b.Vertical synchronization

During the vertical scan, the electron beams move gradually from top to bottom and then return to the top. This corresponds to the time required to refresh the entire screen. The format of the vsync signal is similar to that of the hsync signal, as shown in Figure.

The time unit of the movement is represented in terms of horizontal scan lines. A period of the vsync signal is 525 lines and can be divided into four regions: · Display: It is the region where the horizontal lines are actually displayed on the screen. The length of this region is 480 lines. · Retrace: It is the region that the electron beams return to the top of the screen.

The video signal should be disabled, and the length of this region is 2 lines. · Bottom border: It is the region that forms the bottom border of the display region. It is also known as the front porch (i.e., porch before retrace). The video signal should be disabled, and the length of this region is 3 lines. · Top border: It is the region that forms the top border of the display region. It is also known as the back porch (i.e., porch after retrace). The video signal should be disabled, and the length of this region is 40 lines.

As in the horizontal scan, the lengths of the top and bottom borders may vary for different brands of monitors. The VS(vsync) signal can be obtained by a special mod-525 counter and a decoding circuit.

Again, we intentionally start counting from the beginning of the display region. This allows us to use the counter output as the vertical (y-axis)

coordinate. This output constitutes the vcounter signal. The VS(vsync) signal goes low when the line count is 482 or 484. As in the horizontal scan, we use the v-video-on(~blank) signal to indicate whether the current vertical coordinate is in the displayable region or not. It is asserted only when the line count is smaller than 480.

iii. HDL IMPLEMENTATION OF VGA CONTROLLER CIRCUIT

We implemented VGA controller circuit for the VGA interfacing. Here pixel_clk is taken as the input and remaining HS,VS,hcounter,vcounter,blank are the outputs. Hcounter and the vcounter are the internal counters. For every positive edge of the pixel_clk the hcounter is incremented by one ,whenever it reaches to HMAX(800 pixels) it will be reset and vcounter is incremented by one.

Likewise when vcounter is reaches to VMAX(525 pixels) vcounter becomes reset(value become zero). While this counting process is running whenever the hcounter is in between the range of 648(HFP) and 744(HSP) the hsync(HS) signal is loaded with zero otherwise it will be one.In the same way when the vcounter is in between the range of 482(VFP) and 484(VSP) the vsync(VS) signal is loaded with zero otherwise it will be one. Now when the hcouter and vcounter is in the range of display region.

Verilog Code :

```
module vga_controller (pixel_clk,HS,VS,hcounter,vcounter,blank);

    input pixel_clk;        // It is the 25MHz clk to drive the VGA

    output reg HS, VS;    // synchronization pulses to VGA

    output reg blank;      // To indicate the blank region
```

```
output reg [10:0] hcounter, vcounter;
Parameter HMAX = 800 ; // Maximum Horizontal pixel count
Parameter VMAX = 525 ; // Maximum vertical pixel count


Parameter HLINES = 640 ; // Horizontal display pixels
Parameter VLINES = 480; //Vertical display pixels
Parameter HSP = 744; // Horizontal sync pulse  pixel count
Parameter HFP = 648 ; //  Horizontal sync pulse porch  pixels
Parameter VFP = 482 ; // Horizontal Front porch  pixels
Parameter VSP = 484 ; // Vertical Front porch  pixels
Parameter SSP = 0 ; // Vertical sync pulse pixels
Wire video_enable
----------------------------------------------------------------
//Indicating the region where video is disabled(blank region)
----------------------------------------------------------------
always@(posedge pixel_clk)
begin
blank <= ~video_enable;
end
----------------------------------------------------------------
// Incrementation of horizontal count
```

```
----------------------------------------------------------------

always@(posedge pixel_clk)

begin

if(hcounter == HMAX)

hcounter <= 0;

else

hcounter <= hcounter + 1'b1;

end

----------------------------------------------------------------

// Incrementation of vertical count

----------------------------------------------------------------

always@(posedge pixel_clk)

begin

if(hcounter == HMAX)

begin

if(vcounter == VMAX)

vcounter <= 0;

else

vcounter <= vcounter + 1'b1;

end

end

----------------------------------------------------------------
```

```
// For Horizontal sync pulse
-----------------------------------------------------------------
always@(posedge pixel_clk)
begin
if(hcounter >= HFP && hcounter < HSP)
HS <= SPP;
else
HS <= ~SPP;
End
-----------------------------------------------------------------
// For vertical sync pulse
-----------------------------------------------------------------
always@(posedge pixel_clk)
        Begin
                if(vcounter >= VFP && vcounter < VSP)
                        VS <= SPP;
                else
                        VS <= ~SPP;
                end
-----------------------------------------------------------
//For video enable region
-----------------------------------------------------------
```

```
        assign video_enable = (hcounter < HLINES && vcounter < VLINES)

        ? 1'b1 : 1'b0;

        endmodule
```

HDL IMPLEMENTATION FOR MOVING BLOCK

For the movement of the block we have to design a counter such that it should generate frequency lower than 100Mhz(Board frequency), because we can't see the movement of the block which runs on 100MHz clock and we can see the movement if it runs on lower frequency. So here we designed a 26 bit counter to generate the clock frequency of 30Hz. By using this clock we have to increment the positions of the block so that the block can move.

Verilog Code :

```
module move(output reg[3:0]B, output reg [3:0]R, output VS,

        output HS, output reg [3:0]G,input clk,input [4:0]control);

        wire [10:0] hcount,vcount;

        reg [10:0]x,y;

        wire clk_25;

        wire blank;

        reg slow_clk;

        reg [25:0]slow_count;

----------------------------------------------------------------
```

```
//Generation of 25MHZ clock using clock wizard

------------------------------------------------------------------

clk_wiz_0 instance_name

(

// Clock in ports

 .clk_in1(clk),                    // input clk_in1
 // Clock out ports
 .clk_out1(clk_25));

 // output clk_out1
initial

begin

x = 11'd200;

y = 11'd200;

end

------------------------------------------------------------------

//Counter used to Generate a slow_ck with the frequency of 30hz

------------------------------------------------------------------

always @ (posedge clk)

begin

slow_count = slow_count + 1'b1;

slow_clk = slow_count[25];

end

------------------------------------------------------------------
```

```
//Condition to display 10X10 block                    (10 units length 10 units width)
And assigning the values of RGB
-----------------------------------------------------------------

always@(posedge clk_25)

begin

if(~blank & (hcount >= x & hcount <= x+ 11'd10 & vcount >= y &

vcount <= y+ 11'd10))

begin

R<=4'b1111;

G<=4'b1111;

B<=4'b1111;

end

else

begin

R<=4'b0000;

G<=4'b0000;

B<=4'b0000;

end

end

-----------------------------------------------------------------

//Conditions to move the block

-----------------------------------------------------------------
```



```
always@(posedge slow_clk)
```

```
begin

case(control)

5'b00000:

 begin                    //Since X and Y values are constant
                          block will remain in Current position

x = x;

y = y;

end

5'b01000:

 begin                    //Since X value is decrementing and Y
                          value is constant Block will move left

x = x- 11'd10;

y=y;

end

5'b00100:

begin

//Since X value is incrementing and Y

value is constant Block will move right

x = x + 11'd10;

y=y;

end

5'b00010:

 begin                   //Since Y value is decrementing and X
                         value is constant Block will move up
```

```
x = x ;

y= y-11'd10;

end

5'b00001:

 begin              //Since Y value is incrementing and X
                    value is constant Block will move down

x = x ;

y = y+11'd10;

end

endcase

end

vga_controller u1(clk_25,HS,VS,hcount,vcount,blank);

endmodule
```

HDL IMPLEMENTATION FOR BIT MAPPING :

Generally to display one block on VGA we have to specify the hcount and vcount values for that specific block,so in that case we can only display the rectangular blocks. But if we want to display one Non-rectangular object it is difficult to specify hcount and vcount values for that. In order to overcome that problem we can use the technique called BIT MAPPING.

In this we have to specify the object pattern and should generate the RGB signals according to that map. To implement this scheme, we need to include a pattern ROM to store the bit map and an address

mapping circuit to convert the scan coordinates to the ROM's row and column. First,we have to define a pattern ROM for the non-rectangular object using a case statement. After that we have to assign the boundaries of the non-rectangular object.

While scanning is taking place we have to check whether it is in the region of given boundary or not. If it so, the signal sqareball_on signal will become high.Now we should consider the rom_addr and rom_column. If the scan coordinates are within the square ball region, subtracting the four LSBs of top boundary (i.e.,ball_up) from the vcount provides the corresponding ROM row (i.e., rom-addr) this subtraction leads a 4 bit counter here, and subtracting the four LSBs of left boundary (i.e., ball_left) from the hcount provides the corresponding ROM column (i.e., rom-col).Now we can access rom_bit using indexing operation(rom_data[rom_col]).Finally we should assign RGB values only when the square_ball_on and rom_bit values are high.

Verilog Code:

```
module obstacle(input clk, output reg [3:0]R, output reg [3:0]G,output reg [3:0] B, output HS,VS);

    reg [4:0]rom_addr, rom_col; reg [31:0] rom_data; wire [10:0] hcount,vcount; wire blank,clk_25; reg [10:0]bALL_left,bALL_right,bALL_up,bALL_down;

    reg square_bALLon,rom_bit;

    initial

    begin  bALL_left=300;  bALL_right = 332;  bALL_up =200 ;  bALL_down=232;

    end

    clk_wiz_0 instance_name

        (

        // Clock in ports
```

```
.clk_in1(clk),    // input clk_in1

// Clock out ports

.clk_out1(clk_25)

);         // output clk_out1

always@(* )

begin

case(rom_addr)

5'd0:rom_data=32'b00000011110000001111100111100111;

5'd1:rom_data=32'b00000011110000001111100111100111;

5'd2:rom_data=32'b00000011110011101111100111100111;

5'd3:rom_data=32'b00000011110011101111100111100111;

5'd4:rom_data=32'b00000011110011101111100111100111;

5'd5:rom_data=32'b00000011110011101111100111100111;

5'd6:rom_data=32'b11110011110011101111100111100111;

5'd7:rom_data=32'b11110011110011101111100111100111;

5'd8:rom_data=32'b11110011110011101111100111110111;

5'd9:rom_data=32'b11110011110011101111100111100111;

5'd10:rom_data=32'b11110011111111101111100111100110;

5'd11:rom_data=32'b11110011110000001111100111101100;

5'd12:rom_data=32'b11110011110000001111100111111000;

5'd13:rom_data=32'b11110011110000001111111111100000;

5'd14:rom_data=32'b11111111110000001111111111100000;

5'd15:rom_data=32'b11111111110000000001111111100000;
```

```
5'd16:rom_data=32'b11111111110000000000011111100000;

5'd17:rom_data=32'b00000111110000011100000111100000;

5'd18:rom_data=32'b00000111110000011100000111100000;

5'd19:rom_data=32'b00000111110000011100000111100000;

5'd20:rom_data=32'b00000011110000011100000111100000;

5'd21:rom_data=32'b00000011110000011100000111100000;

5'd22:rom_data=32'b00000011110111011100000111100000;

5'd23:rom_data=32'b00000011110111011100000111100000;

5'd24:rom_data=32'b00000011110111011100000111100000;

5'd25:rom_data=32'b00000011110111011100000111100000;

5'd26:rom_data=32'b00000011110111011100110111100000;

5'd27:rom_data=32'b00000011110111111100110111100000;

5'd28:rom_data=32'b00000011110111111100110111100000;

5'd29:rom_data=32'b00000011110011111100110111100000;

5'd30:rom_data=32'b00000011110000011100110111100000;

5'd31:rom_data=32'b00000011110000011111110111100000;

endcase

end

always@(posedge clk_25)

begin

        if(square_bALLon=(hcount >= bALL_left) && (hcount <= bALL_right)
&& (vcount >=       bALL_up)&& (vcount <= bALL_down);

rom_addr = vcount[4:0]- bALL_up[4:0];

rom_col = hcount[4:0] - bALL_left[4:0];
```

```
rom_bit =rom_data[rom_col];
if(     square_bALLon && rom_bit)
        begin
        R <= 4'b1111;
        G <= 4'b0000;
        B <= 4'b0000;
        end
        else
        begin
        R <= 4'b0000;
        G <= 4'b1111;
        B <= 4'b1111;
        end
        end
vga_controller u1(clk_25,HS,VS,hcount,vcount,blank);

endmodule
```

KEYBOARD INTERFACING:

Many keyboards connect to the computer through a cable with a PS/2 or USB(Universal Serial Bus) connector. Regardless which type of connector is used, the cable must carry power to the keyboard, and it must carry signals from the keyboard to the board. A keyboard consists of a matrix of keys and an embedded microcontroller that monitors (i.e., scans) the activities of the keys and sends scan code accordingly, where scan code is nothing but the unique code assigned to each key which is sent whenever a key is pressed.

KeyBoard with scan codes

The keyboard can send data to the host only when both the data and clock lines are at high (or idle). Before driving the bus, the keyboard must check the status of the host (whether it is sending any data or not) and here the host is the bus master. To facilitate this, the clock line is used as a “clear” to send signal. If the host drives the clock line low, the keyboard must not send any data until the clock is released. The keyboard sends data to the host in 11-bit word which contains `0' as a start bit, followed by

8-bits of scan code (LSB first), followed by an odd parity bit and terminated with `1' as a stop bit. The keyboard generates 11 clock transitions (at 20 to 30KHz) when the data is sent, and data is valid on the falling edge of the clock.

i.USB HID Host:

The Auxiliary Function microcontroller (Microchip PIC24FJ128) provides the Nexys4 with USB HID host capability. After power-up, the microcontroller is in configuration mode, either downloading a bitstream to the FPGA, or waiting to be programmed from other sources. Once the FPGA is programmed, the microcontroller switches to application mode, which is USB HID Host in this case. Firmware in the microcontroller can drive a mouse or a keyboard attached to the type A USB connector at J5 labeled "USB Host." Hub support is not currently available, so only a single mouse or a single keyboard can be used. The PIC24 drives several signals into the FPGA { two are used to implement a standard PS/2 interface for communication with a mouse or keyboard, and the others are connected to the FPGA's two-wire serial programming port, so the FPGA can be programmed from a file stored on a USB pen drive or microSD card.

ii. HDL IMPLEMENTATION OF KEYBOARD INTERFACING:

For this interfacing circuit inputs are vga_clk, kb_clk, kb_data and the output is the code. Here vga_clk is nothing but the 25MHZ clock which is used in the VGA Interfacing. We should take the data from the keyboard on the basis of vga_clk pulse. Here we need to synchronize this keyboard with VGA monitor, because we need to use the data from the keyboard for the movements of the blocks on VGA monitor. Kb_clk,

kb_data are the outputs from the keyboard ,which are nothing but inputs to the interfacing circuit. Here the final output is the scancode stored in the register called code.

Here based on the shift register 0th index value we will start the loading of the keyboard data. Intially Shift_reg[0] loaded with zero, so endbit becomes one.Whenever endbit becomes one the 11 bit shift register is loaded with 'ff'. If the shift_reg[0] is one then the endbit becomes zero then shifting of kb_data is takes place. This shifting of data is also depened upon kb_clk, so we have stored that clock pulse into the reg Q0, and immediately passed that to Q1,so when (~Q0 & Q1) is high the shift will enable.Then the data will pass to the shift register.Finally the middle 8-bit data in the shift_reg stored into the code.And this code is used for the controlling.

Verilog Code :

```
module kbInput(VGA_clk,KB_clk,KB_data,code);

        input VGA_clk;

        input KB_clk;

        input KB_data;

        reg [30:0]count;

        output reg [7:0] code;

        reg Q0, Q1;

        reg [10:0] shreg;

        wire endbit;

        assign endbit = ~shreg[0];

        assign shift = Q1 & ~Q0;
```

```
always @ (posedge VGA_clk)

begin

Q0 <= KB_clk;

Q1 <= Q0;

shreg <= (endbit) ? 11'h7FF : shift ? {KB_data, shreg[10:1]} :

shreg;

if (endbit)

code <= shreg[8:1];

else

count=count+1'b1;

end

endmodule
```

Game Description:

Using Verilog Hardware Description Language, code a program to recreate the game 'Dino Run' by Google Chrome (available whenever internet service is down). . The display is to be on a 640X480 resolution using a VGA connection.

The game is simple: the scenery is shifted right to left and the only input is to jump or restart the game. The user must create a pixelated 'dinosaur' whose motion mimics running in place while the obstacles are coming dinosaur's way. If the dinosaur is an contact with any obstacle, then the game is over and the score is displayed. As the player accumulates more

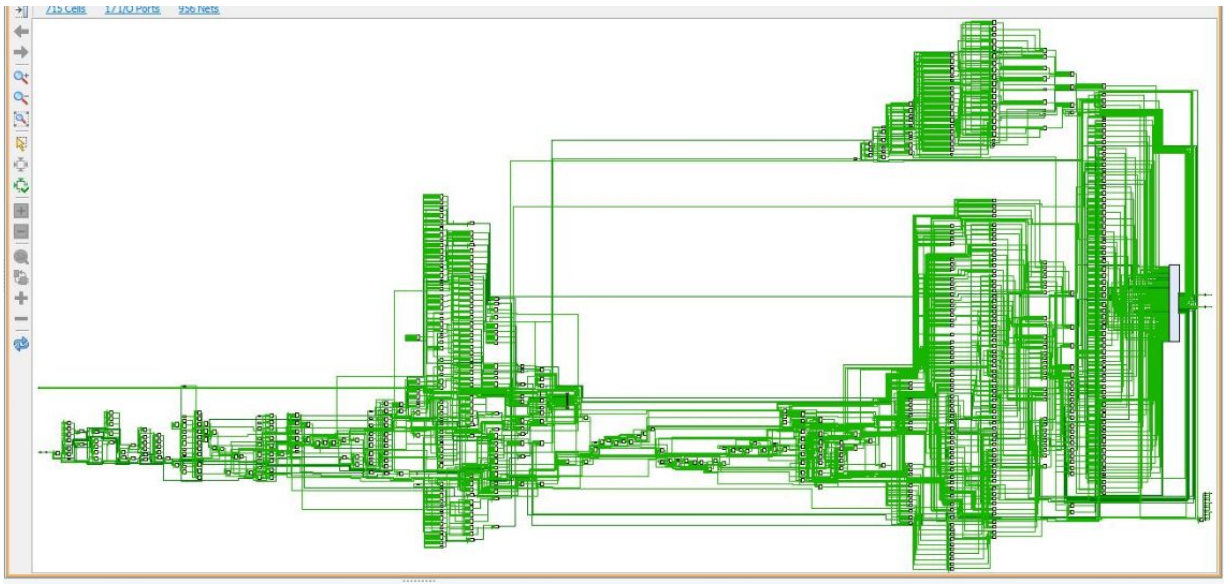
points, the rate at which the environment shifts must increase to make the game more difficult.

Some display aspects of the game include creating an initial start screen, displaying points per obstacle jumped, randomly spacing obstacles, and displaying a 'game over' sign with the player's final score.

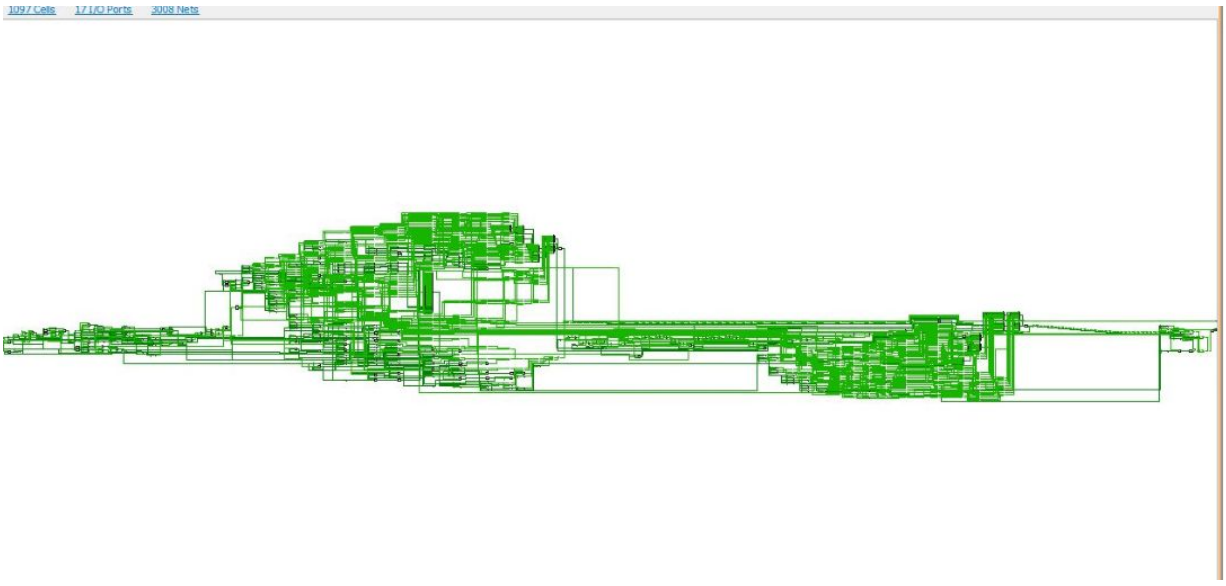
HDL IMPLEMENTATION:

The dino game is implemented with two interfaces VGA Interfacing, Keyboard Interfacing respectively. From these interfaces we can display the blocks, and can control the movement of blocks. So in this Dino Game we instantiated these two interfaces. In each and every game score plays a major role, so we displayed the score by creating the segments on the screen like seven segment displays.And we converted the binary number which is coming from the counter(which counts score) into the BCD by using the logic of binary to BCD conversion. Initially we have stored 40 segments of the snake so whenever the snake eats food the segments will increase. And the player loses the game if the dinosaur touches the obstacle(cactus) and gains a point if the dinosaur runs without touching the obstacles.

RTL Schematic :



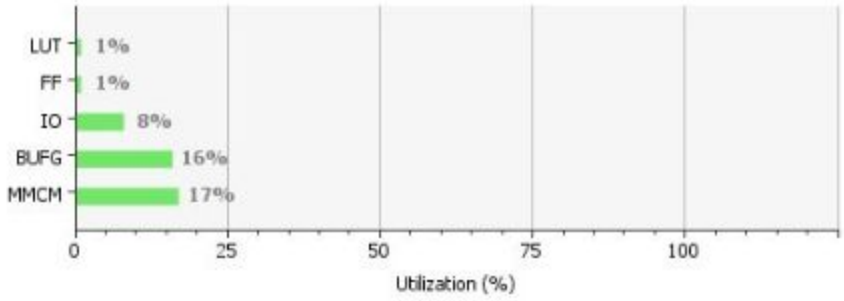
Implemented Schematic :



Synthesis Report :

1. Utilization Report :

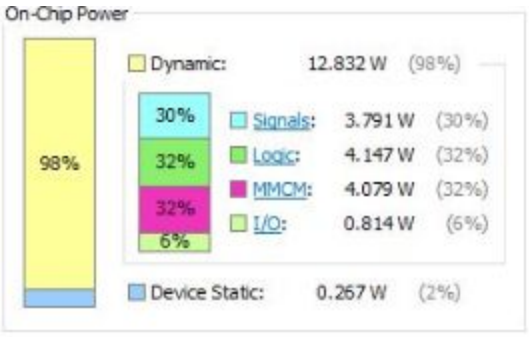
Resource	Utilization	Available	Utilization %
LUT	813	63400	1.28
FF	272	126800	0.21
IO	17	210	8.10
BUFG	5	32	15.63
MMCM	1	6	16.67



2. Power Report :

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**13.099 W**
Junction Temperature:	**84.8 °C**
Thermal Margin:	0.2 °C (0.1 W)
Effective ΘJA:	4.6 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low



3.Timing Report :

Future Scope :

Here this game is operated through a keyboard and further it can be implemented by “Accelerometer Sensor”. By using this sensor we can operate the game’s display according to the movement of the FPGA board. Whether the FPGA board is flipped according to the board game’s display also flipped, And After reaching some score the difficulty level can be increased,the colour the dinosaur and obstacles can be changed,other obstacles will interface with dinosaur.