ECE385

Fall 2022

Final Project

Modified T-Rex Dinosaur Game

Haoran Yuan & Lizhuang Zheng

TA: Tianhao Yu

1. Overview

The project is a modified version of the Google Chrome T-Rex Game, with some new features added. The Google Chrome T-Rex Game is an offline game provided by the Google Chrome browser when the Internet is not connected, in which the player can press “Space” key or the “Up” arrow key on the keyboard to control the dinosaur to jump over the obstacles (like cactuses and pterosaurs) and run to get higher scores.

This project is based on FPGA DE10-Lite and Quartus 18.1 LITE Edition, and need a VGA monitor and a USB keyboard.

The top-level entity is lab62.sv, which is from ECE 385 lab 6.2, with some modifications.

To start the game, press Space. Then you can press Space or Up（↑） to control the dinosaur to jump over the obstacles like cactuses and pterosaurs. If you touched the obstacle, the game will over, and your score will be show on the screen, and your highest score will be recorded as well. Then you can press “Enter” to return to the start, and then press “Space” to run a new round. You can also press “Reset” (KEY0) on the FPGA board to reset the game, which will zero your highest score.

Sometimes there will be a heart flying towards the dinosaur, if you touched it, it would add 100 points to your score.

At some point of time, the scene will go into the night for a period of time.

There is also an easter egg: During the game, press “IKUN” in keyboard in order, your dinosaur will become a rap star with middle-part haircut, black shirts, white suspender trousers, playing with a basketball. When typing “I”, “K”, “U”, “N” one by one, the LED3~LED5 on the FPGA board will indicate which state you are in (LED[5:3] = 3’b000~3’b100). We call this mode “IKUN” mode. You can press “Esc” or “Enter” to exit “IKUN” mode.

To launch the game, you should first connect the FPGA DE10-Lite to the computer, VGA monitor, and the USB keyboard, after that you should compile the project, program to the FPGA, then go to the Eclipse, Run Configuration of the project “dino.”

2. List of Features

1) Start Page

The dinosaur is standing on the ground, the horizon does not move, the score is 0, waiting you to press “Space” to start.

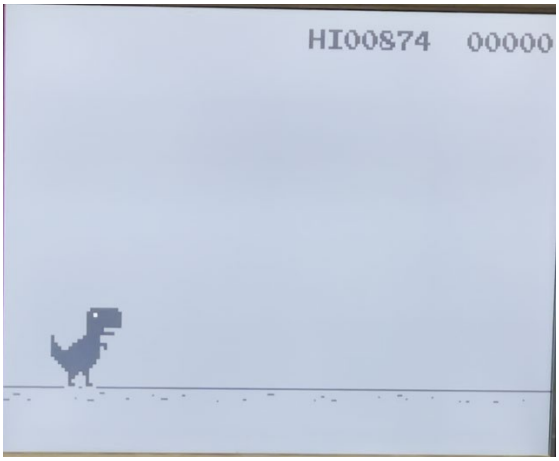


Figure 1: Start Page

2) Jump

When you press “Space,” there will be 30 frames (0.5s) of delay to judge whether the dinosaur should perform a lower jump or a higher jump. If the time you press “Space” is shorter than 15 frames (0.25s), and you release the “Space” before the 15th frame, then it will perform a lower jump, with initial velocity -10 and gravity 3. If the time you press “Space” is longer than 15 frames (0.25s), but shorter than 30 frames (0.5s), it will perform a higher jump, with initial velocity -11 and gravity 3. If the time you press “Space” is longer than 30 frames (0.5s), it will automatically perform a higher jump.

Figure 2: Jump with Two Heights

3) Collision

When the dinosaur touches a cactus or a pterosaur (has a collision with the obstacle), its Life will be changed from 1 to 0. So that the game will over. We realize this by setting 4 test points on the dinosaur, when one of those test points are inside the boxes surrounding the obstacle, it will be judged as a “collision.”

4) Animation

We add intrinsic animation for the running dinosaur, the pterosaur, the heart and the moon (with stars). And the moving of the dinosaur, the cloud, the horizon, the cactus and the pterosaur, the heart and the increment of the scores and the highest scores can also be counted as the animation. We realize the animation changing the sprite elements according to a “frame counter”, which will increase every frame. When the “frame counter” reaches a certain number, we change to another sprite element and repeat this procedure to realize the intrinsic animation. For the moving animation, we also use the “frame counter” to control, when the counter reaches a certain number, those elements will move to the left by some pixel (for the elements on the ground they are 4 pixels/frame). When game is over, the moving animation will be frozen (except the cloud), while the intrinsic animation will not be affected.

5) Obstacles

There are two types of obstacles: cactus and pterosaur. For both of them we use the pseudo-random number generated by the Linear Feedback Shift Register (LFSR). For the cactus, we use the 6-bit random number to determine the type of the cactus by judging which ranges the random number is in. There are 6 types of cactuses, with different heights and widths. For the pterosaur, we use the 6-bit random number to determine the height of it. There are 3 heights of pterosaurs. To make sure the two types of obstacles will not appear at the same place, we created two variables “ca_off” and “pt_off” to communicate between the two modules “draw_cactus.sv” and “draw_pterosaur.sv.” In draw_cactus.sv, if (Ptero_PosX + pterosaur_X > 320 || Ptero_PosX

< 0), we should not draw the cactus, so we go to the state “None” which does not draw anything and set ca_off = 1 to disable cactus_on_wr. Symmetrically, in draw_pterosaur.sv, if (Cactus_PosX + Cactus_SizeX > 320 || Cactus_PosX < 0), we should not draw the pterosaur, so we go to the state “None” which does not draw anything and set pt_off = 1 to disable pterosaur_on_wr. By doing these, the two types of obstacles will not overlap, but the drawback is that the desity of the obstacles will decrease.

Figure 3: Obstacles (Cactus and Pterosaur)

6) Score

The score will start increasing once you jump to start the game. And when the game is over, the score will stop increasing. For every 10 frames, the score will increase by 1. If the score reaches 100000, the score will be reset to 0 and increase again from 0. At every hundreds, the score will start to blink for 4 times, when blinking, the score will appear to be hundreds, and after 200 frames (the true score increases 20), the score will be increasing again following the true score. Once the dinosaur touches the red heart (buff/bonus), it will add 100 extra points into the score.

7) Highest Score

For each time you restart the game by pressing “Enter,” the highest score will be remembered by a register. The highest score will be reset to 0 only if you press KEY0 to reset. Once your current score is higher than the highest score, your current score will be loaded into the highest score, so that the highest score will be increasing simultaneously with the score. Once the game is over, the highest score will also stop and remember the current highest score.

8) Night

From score of 400, starts from every multiple of 400, we will go into the night scene, lasting for 200 of scores (i.e., 400~600, 800~1000, etc.). We realize this by giving an “isnight” signal to the palette, in order to choose from the two palettes, one is for the daytime, and another is for the nighttime. At night, there is a moon and two stars in the sky. The shape of the moon is controlled by the random number, and the stars are the animation of 3 sprite elements.

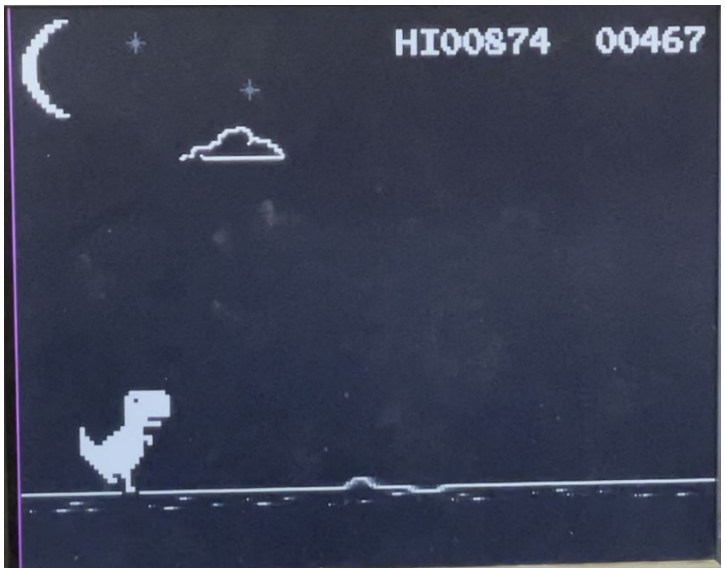


Figure 4: Night Scene

9) Heart (Buff/Bonus)

The heart will appear in the score range (score%500 > 100 && score%500 < 150), at height 256, the dinosaur needs to jump to get the heart. If the dinosaur reach the heart, it will be awarded by 100 points adding to the current score. If you missed a heart, it would reappear from the right of the screen if your score were still in the range described above. To make sure the heart will not overlap with other obstacles, we add a "flag" which are not only controlled by the score range but also controlled by the cactus position and the pterosaur position: (Cactus_PosX < 320 && (Ptero_PosX<320||Ptero_PosX>670)). After the dinosaur reaches the heart, we use a variable "contact" to turn off the heart_on_wr to make the heart disappear.

Figure 5: Heart

10) Game Over Page

When the dinosaur touches obstacle (a cactus or a pterosaur), the game will over, the dinosaur, horizon, obstacles, heart, and the score (perhaps the highest score) will all be frozen. A line of "GAME OVER" will be printed in the screen and a picture of the restart button will appear, and

your current score will also be shown under the restart button. Then you can press “Enter” to back to the start page.

Figure 6: Game Over Page

11) Easter Egg

The Easter Egg can be triggered by typing “I” “K” “U” “N” in order during the game. When the Easter Egg is triggered, the dinosaur will change into another costume, dribbling a basketball. See the figure below. If you want to exit the Easter Egg mode, you can press “Esc” or “Enter” on the USB keyboard, so that the dinosaur will take off the costume and back to normal.

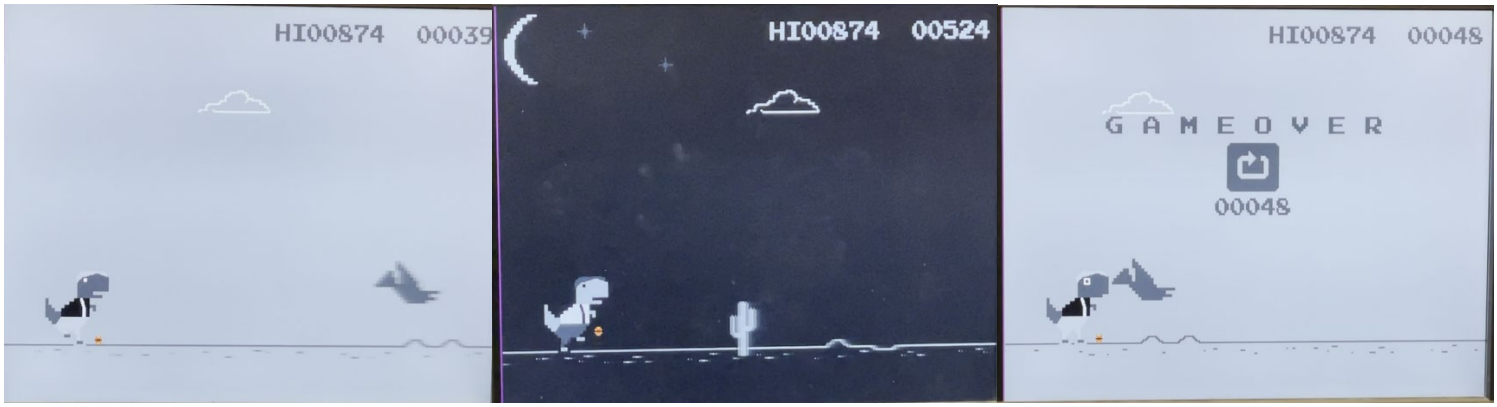


Figure 7: Easter Egg

3. Block Diagrams

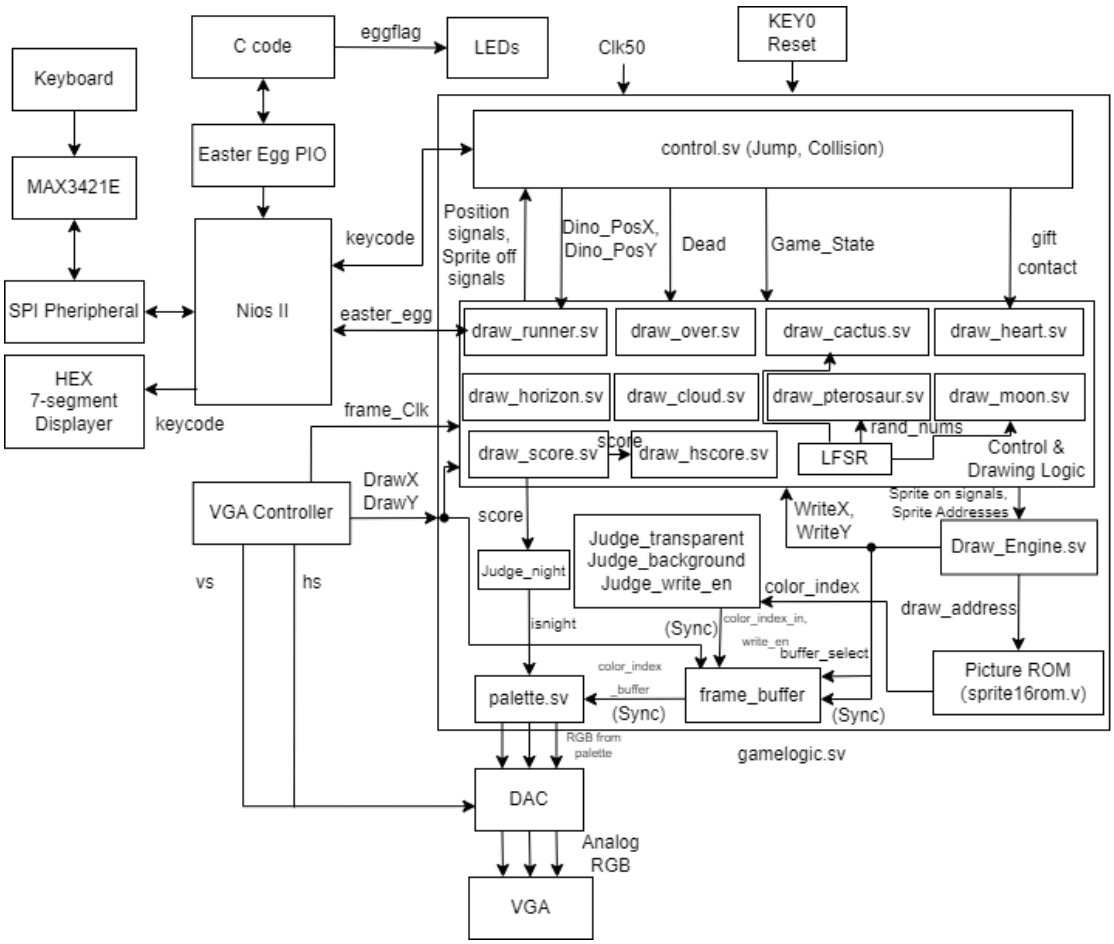


Figure 8: Final Project Block Diagram

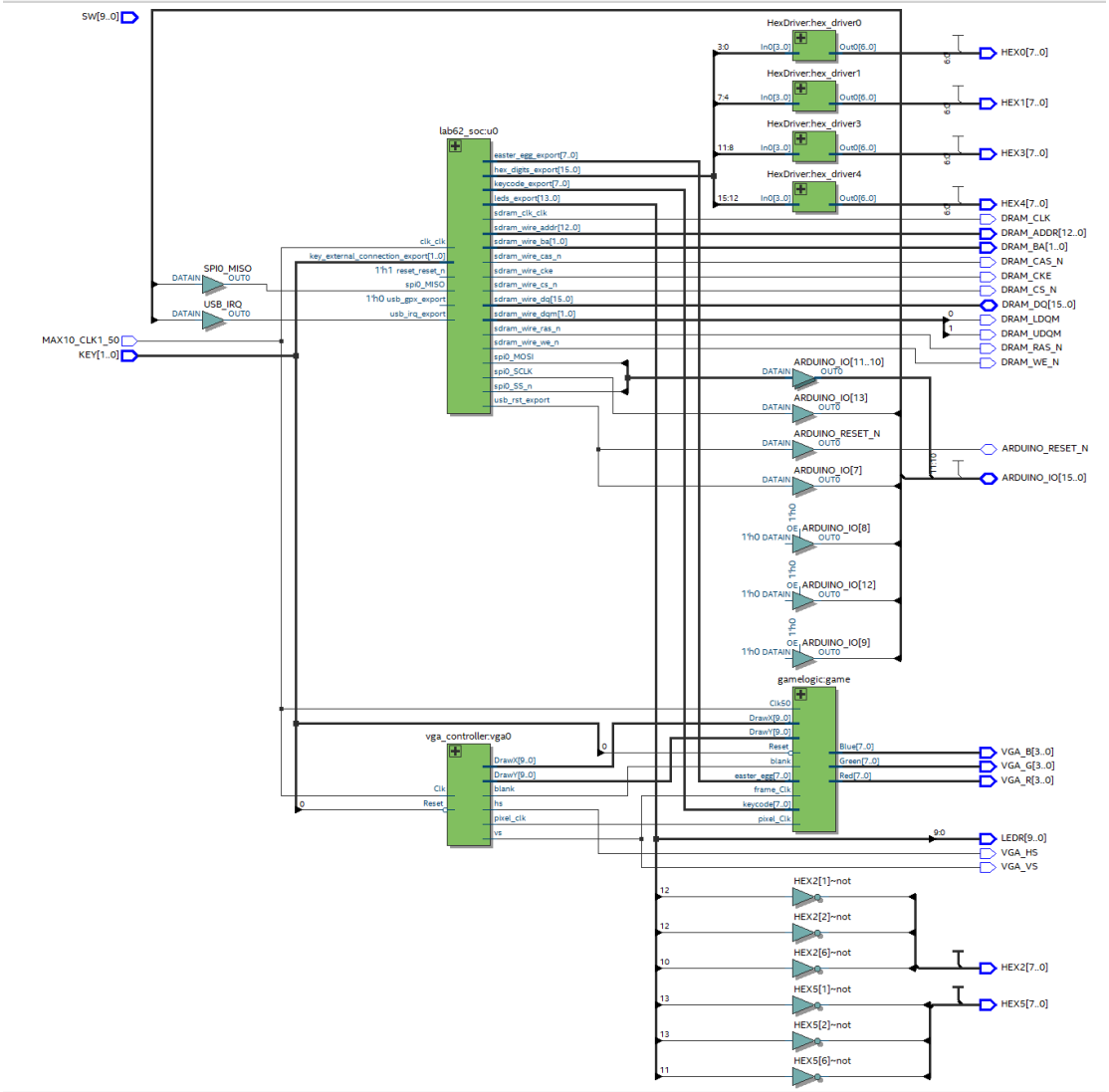


Figure 9: Top Level RTL Block Diagram

4. Description of .sv Modules

1) Module: lab62.sv

```
input                MAX10_CLK1_50,
//////// KEY ////////
input      [ 1: 0]   KEY,
//////// SW ////////
input      [ 9: 0]   SW,
//////// LEDR ////////
output     [ 9: 0]   LEDR,
//////// HEX ////////
output     [ 7: 0]   HEX0,
output     [ 7: 0]   HEX1,
output     [ 7: 0]   HEX2,
output     [ 7: 0]   HEX3,
```

```
output      [ 7: 0]       HEX4,
output      [ 7: 0]       HEX5,
//////// SDRAM ////////
output                    DRAM_CLK,
output                    DRAM_CKE,
output      [12: 0]       DRAM_ADDR,
output      [ 1: 0]       DRAM_BA,
inout       [15: 0]       DRAM_DQ,
output                    DRAM_LDQM,
output                    DRAM_UDQM,
output                    DRAM_CS_N,
output                    DRAM_WE_N,
output                    DRAM_CAS_N,
output                    DRAM_RAS_N,
//////// VGA ////////
output                    VGA_HS,
output                    VGA_VS,
output      [ 3: 0]       VGA_R,
output      [ 3: 0]       VGA_G,
output      [ 3: 0]       VGA_B,
//////// ARDUINO ////////
inout       [15: 0]       ARDUINO_IO,
inout                     ARDUINO_RESET_N
```

Purpose: This is the top level of the project. We instantiate the NIOS II SOC, VGA controller and our Game Logic.

Description: The NIOS II module is used as USB driver and implement the control logic of the easter egg. The keycode information from the keyboard will be transmitted through NIOS II into the Game Logic module to do further logic control.

2) Module: gamelogic.sv

Inputs: Clk50, pixel_Clk, frame_Clk, Reset, blank,
[9:0] DrawX, DrawY,
[7:0] keycode,
[7:0] easter_egg,

Output: [7:0] Red, Green, Blue

Purpose: Realize the main part of the game logic.

Description: The integration of drawing engine, row buffer, drawing logic and control logic of collision and movement.

3) Module: control.sv

input Reset, frame_Clk,
input pt_off, ca_off,
input int Ptero_PosX, Ptero_PosY,
input int Fire_PosX, Fire_PosY,
input int Buff_PosX, Buff_PosY,
input int Cactus_PosX, Cactus_PosY,

input int Cactus_SizeX, Cactus_SizeY,
input int heart_PosX, heart_PosY,
input logic heart_off,
input [7:0] keycode,
output logic [9:0] Dino_PosX, Dino_PosY,
output logic Dead,
output logic contact,
output logic gift,
output logic [1:0] Game_State

Purpose: The module implements the movement control of the little dinosaur under the commands from the keyboard and the collision detection between the little dinosaur and the obstacles.

Description: We used a jump counter to memorize the time that the space button was pressed to indicate the initial velocity when the dragon leaves the ground. And we set a gravity parameter to simulate the free-fall body motion of the little dinosaur. Also we select four points from the body of the dinosaur as test points for collision detection. We use rectangular shapes as the collision region of the cactuses and the pterosaurs. When any of the test points is in the region of the obstacles, the little dinosaur will be dead and the game is over.

4) Module: Draw_Engine.sv
input Clk50, Reset,
input Draw_Back, Draw_Ground, //layer_1
input Draw_Cloud, Draw_Moon, //layer_2
input Draw_Cactus, Draw_Buff, Draw_Rock, Draw_Pterosaur, //layer_3
input Draw_Score, Draw_Fire, Draw_Runner, Draw_Highscore, Draw_Over, //layer_4
input [17:0] address_Back, address_Ground,
input [17:0] address_Cloud, address_Moon,
input [17:0] address_Cactus, address_Buff, address_Rock, address_Pterosaur,
input [17:0] address_Score, address_Fire, address_Runner, address_Highscore, address_Over,
input [9:0] DrawX, DrawY,
output [17:0] draw_address,
output [9:0] write_X, write_Y,
output [2:0] write_which_layer

Purpose: The drawing engine generate the row image consisting of four layers. The drawing engine plays an important role in the case that objects from different layers have an intersection.

Description: The drawing engine is implemented by a state machine. The states and state transition are shown in the state transition diagram below (in Section 6). If one pixel has several layer, the state machine will transmit the address of the corresponding objects to the output port from the lowest layer to the highest. And the write row buffer can use the address to overwrite and generate row image.

5) Module: frame_buffer.sv
input Clk50, Reset, write_en,
input [3:0] write_data,
input [9:0] write_X, read_X,
input [9:0] write_Y, read_Y,

input select,
output logic [3:0] read_data
Purpose: The frame buffer consists of 2 line buffers.
Description: The module has two row buffers. One is for write buffer, the other is for read buffer. Whenever the Write_Y increments by one, the read buffer and write buffer will swap. The row buffers are implemented by the two RAMs in the on-chip memory.

6) Module: struct_declaration.sv
Purpose & Description: Declare the structure of the little dinosaur.

7) Module: palette.sv
input logic [3:0] color,
input logic isnight,
output logic [7:0] Red, Green, Blue
Purpose: Translate color index into the RGB values for VGA.
Description: The colors are coded with 4-bit indices that occupies less space in on-chip memory than using the full RGB values. The module contains two palettes. One is for the day mode, the other is for the night mode. Given the color index, the palette will output the corresponding RGB values.

8) Module: LFSR.sv
input Clk, Enable,
input Load_Seed,
input [length-1:0] Seed,
output logic [length-1:0] Out,
output logic Done
Purpose: Generate pseudo-random numbers with certain "length."
Description: Given a seed value with length $length$ and high-active enable signal, the module can generate a random number in the range of $[0, 2^{length} - 1]$ on each rising edge of the given clock.

9) Module: draw_cloud.sv
input frame_Clk, Reset,
input [9:0] WriteX, WriteY,
output logic cloud_on_wr,
output logic [17:0] address
Purpose: To draw the cloud moving in the sky from the right to the left of the screen.
Description: The module contains the motion logic and drawing logic. The motion logic is simply diving the cloud moving leftward and pull it back to the right side when the cloud is completely invisible from the left side of the screen. The drawing logic is simply when Write_X is in the rectangle region of the cloud, the module will give a cloud_on signal.

10) Module: draw_runner.sv
input frame_Clk, Reset,
input [9:0] WriteX, WriteY,
input [9:0] PosX, PosY,
input Dead,
input [1:0] Game_State,
input int score,

input [7:0] easter_egg,
output logic runner_on_wr,
output logic [17:0] address
Purpose: To draw the dinosaur (runner) in the screen.
Description: The module contains the drawing logic of the little dinosaur. To realize the intrinsic animation of the running dinosaur, we use a counter to manipulate the switch between the two sprites. And when the module receives the easter_egg signal, the little dinosaur will put on the clothes of Kunkun.

11) Module: draw_score.sv
input frame_Clk, Reset,
input [9:0] WriteX, WriteY,
input Dead, gift,
input [1:0] Game_State,
output logic [2:0] score_on_wr,
output logic [17:0] address,
output int score_out
Purpose: To calculate the score and draw the score on the top-right corner of the screen.
Description: The module memorizes the present score and every time the score reaches the multiple of 100, the score will blink for 20 points.

12) Module: draw_horizon.sv
input frame_Clk, Reset,
input [9:0] WriteX, WriteY,
input Dead,
input [1:0] Game_State,
output logic horizon_on_wr,
output logic [17:0] address
Purpose: To circularly draw the horizon, moving from right to left.
Description: The module contains the drawing logic of the horizon. We use a moving window that is 640 pixels wide and circularly right-shift. When Write_X is in the rectangle region of the ground, the module will give a horizon_on signal.

13) Module: draw_cactus.sv
input frame_Clk, Reset,
input [9:0] WriteX, WriteY,
input int Ptero_PosX, Ptero_PosY,
input Dead,
input [1:0] Game_State,
output logic cactus_on_wr,
output logic [17:0] address,
output int Cactus_PosX, Cactus_PosY,
output int Cactus_SizeX, Cactus_SizeY,
output logic ca_off
Purpose: To generate the cactuses and draw them on the screen, moving from right to left.
Description: The module contains the generation logic of various cactuses determined by pre-set possibilities among different kinds of cactuses, and the corresponding drawing logic.

14) Module: draw_pterosaur.sv
    input frame_Clk, Reset,
    input Dead,
    input [1:0] Game_State,
    input [9:0] WriteX, WriteY,
    input int Cactus_PosX, Cactus_PosY,
    input int Cactus_SizeX, Cactus_SizeY,
    output logic pterosaur_on_wr,
    output int Ptero_PosX, Ptero_PosY,
    output logic [17:0] address,
    output logic pt_off
    Purpose: To generate the pterosaurs and draw them on the screen, moving from right to left.
    Description: The module contains the generation logic of pterosaurs of different height determined by pre-set possibilities among different heights, and the corresponding drawing logic.
15) Module: draw_hscore.sv
    input frame_Clk, Reset,
    input [9:0] WriteX, WriteY,
    input Dead,
    input [1:0] Game_State,
    input int score,
    output logic [2:0] hscore_on_wr,
    output logic [17:0] address
    Purpose: To record the highest score and display the highest score on the top of the screen.
    Description: The module contains the control logic of the highest score, that is to update the highest score if the current score exceeds the previous highest score. The corresponding drawing logic will display the highest score on the top of the screen and on the Game Over page.
16) Module: draw_heart.sv
    input frame_Clk, Reset,
    input Dead, contact,
    input int score,
    input [1:0] Game_State,
    input [9:0] WriteX, WriteY,
    input int Cactus_PosX, Cactus_PosY,
    input int Ptero_PosX, Ptero_PosY,
    output logic heart_on_wr,
    output int heart_PosX, heart_PosY,
    output logic [17:0] address,
    output logic heart_off
    Purpose: To control the generation and disappearance of the heart, moving from right to left.
    Description: The module contains the control logic of the buff heart, that is: if the dinosaur contacts the heart, there would be an addition of 100 points to the total score. Then there are the corresponding drawing logic.
17) Module: draw_over.sv

input frame_Clk, Reset,
input [9:0] WriteX, WriteY,
input Dead,
input [1:0] Game_State,
input int score,
output logic [3:0] over_on_wr,
output logic [17:0] addres
Purpose: Draw the Game Over page.
Description: Draw “GAME OVER,” restart icon and the current score to the Game Over page when the dinosaur is dead.

18) Module: draw_moon.sv
input frame_Clk, Reset,
input [9:0] WriteX, WriteY,
input isnight,
output logic [1:0] moon_on_wr,
output logic [17:0] address
Purpose: Draw the moon together with the stars at night.
Description: Draw the moon and the stars onto the background in the night mode. The type of the moon is determined by the random number in a similar way as mentioned in generating other obstacles.

19) Module: HexDriver.sv
input [3:0] In0,
output logic [6:0] Out0
Purpose: To drive the 7-segment displayer on FPGA. Unrelated with the main part of our final project.
Description: With HexDriver, we are able to show the two keycodes we received in the 7-segment displayer, the same as in Lab 6.2.

20) Module: lab62_soc.v

input wire	clk_clk,	//clk.clk
output wire [7:0]	easter_egg_export,	//easter_egg.export
output wire [15:0]	hex_digits_export,	//hex_digits.export
input wire [1:0]	key_external_connection_export,	// key_external_connection.export
output wire [7:0]	keycode_export,	//keycode.export
output wire [13:0]	leds_export,	//leds.export
input wire	reset_reset_n,	//reset.reset_n
output wire	sdram_clk_clk,	//sdram_clk.clk
output wire [12:0]	sdram_wire_addr,	//sdram_wire.addr
output wire [1:0]	sdram_wire_ba,	//.ba
output wire	sdram_wire_cas_n,	// .cas_n
output wire	sdram_wire_cke,	//.cke
output wire	sdram_wire_cs_n,	//.cs_n
inout wire [15:0]	sdram_wire_dq,	//.dq
output wire [1:0]	sdram_wire_dqm,	//.dqm
output wire	sdram_wire_ras_n,	//.ras_n

output wire sdram_wire_we_n, //.we_n
input wire spi0_MISO, //spi0.MISO
output wire spi0_MOSI, //.MOSI
output wire spi0_SCLK, //.SCLK
output wire spi0_SS_n, //.SS_n
input wire usb_gpx_export, //usb_gpx.export
input wire usb_irq_export, // usb_irq.export

Purpose: Our SOC module.

Description: Verilog file generated by the Platform Designer, which is the hardware basement of the NIOS II.

21) Module: buffer.v

input aclr;
input [9:0] address;
input clock;
input [3:0] data;
input wren;
output [3:0] q;

Purpose: Used as the line buffer (row buffer) in the frame buffer.

Description: It is a 4-bit wide RAM with 1024 addresses. When writing into the RAM, it uses "wren" as the write enable signal, if wren=1, then it can write "data" into the current address. When reading from the RAM, it will load the content in current address into "q." We can use appropriate address (WriteX and DrawX) and wren to realize the line buffer.

22) Module: spriterom16.v

input [15:0] address_a;
input [15:0] address_b;
input clock;
input [15:0] data_a;
input [15:0] data_b;
input wren_a;
input wren_b;
output [15:0] q_a;
output [15:0] q_b;

Purpose: Used as the sprite ROM in the on-chip memory, storing the sprite pictures.

Description: It is a 16-bit wide 2-Port RAM with 65536 addresses to perform the task of a ROM, but actually we only used one set of port. In each address, there are 16 bits, which consists of four 4-bit color indices. For convenience, we use the 18-bit "pixel address" outside the ROM, and inside the ROM, we use the higher 16-bit of the pixel address as the "ROM address," and then use the lower 2 bits to locate the actual 4-bit color index we want: 00 = q[15:12], 01 = q[11:8], 10 = q[7:4], 11 = q[3:0].

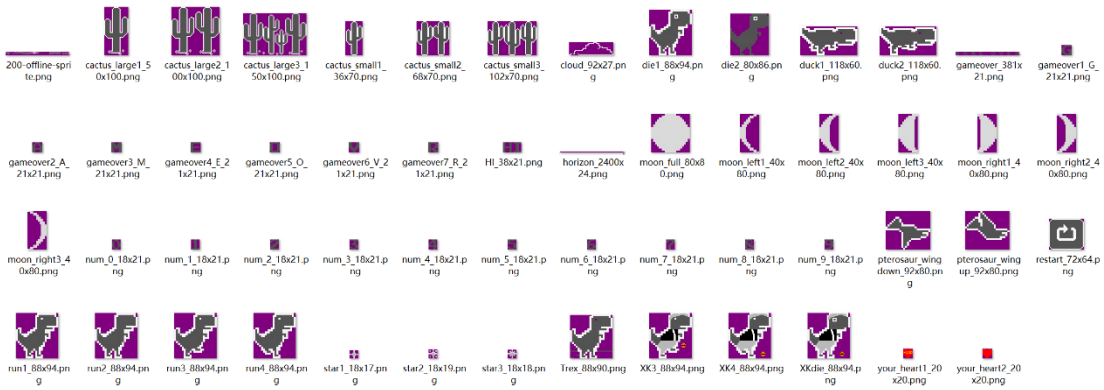


Figure 10: Sprite Pictures Stored in the Sprite ROM

5. Description of SOC Components

1) Component: clk_0
   The 50MHz clock, which exports clk and reset signals.
2) Component: nios2_gen2_0
   The NIOS II Processor block, a 32-bit embedded processor for FPGA. It functions as the CPU in this project, which controls the data and instructions in the SOC, and it is the master of other peripheral components.
3) Component: onchip_memory2_0
   The On-Chip Memory block, it has 200K resources to use. In this project, we place the Sprite ROM into the on-chip memory by creating a RAM from Megafunction IP.
4) Component: sdram
   The SDRAM block, it has much larger spaces than on-chip memory. In this project we store the software codes here.
5) Component: sdram_pll
   The SDRAM Phase Lock Loop block. It outputs the “controller clock” c0 and “SDRAM clock” c1, which are both needed by SDRAM.
6) Component: sysid_qsys_0
   The System ID Checker block. It is used to check whether the hardware and software versions are compatible.
7) Component: jtag_uart_0
   The JTAG UART peripheral block. It is used to communicate your computer terminal with NIOS II, so that you can use the console to debug.
8) Component: keycode
   The PIO clock for keycodes, used to get the keycodes from the keyboard and export the 8-bit “keycode” which contains the keycode of the key we pressed.
9) Component: usb_irq
   The PIO block for the USB Interrupt Request signal.
10) Component: usb_gpx
    The PIO block for the USB General-Purpose Multiplexed Push-Pull signal.
11) Component: usb_rst
    The PIO block for the USB Reset signal.

12) Component: hex_digits_pio

    The PIO block for HEX digits, which are used to drive the 7-segment displayer. In this project it works the same as in Lab 6.2, which can display 2 keycodes in decimal.

13) Component: leds_pio

    The PIO block for LEDs. Here we use LEDR[5:3] to indicate which stage of the Easter Egg are we in. It ranges from 3'b000 to 3'b100. When LEDR[5:3]=3'b100, we enter the Easter Egg mode.

14) Component: key

    The PIO block for the two buttons KEY0 and KEY1. Here KEY0 is used to reset.

15) Component: timer_0

    The Interval Timer Intel FPGA IP block. It is used to track various timeouts (pulses) USB requires. It can also assert interrupt request (irq).

16) Component: spi_0

    The Serial Peripheral Interface (SPI) port peripheral block. Here the SPI is in Master mode, we can use it to control the USB. It can also assert interrupt request (irq). Here it still works in Master mode.

17) Component: easter_egg

    The PIO block for the Easter Egg flag signal "easter_egg." It has an output port "easter_egg," we reserved 8-bit because we are not sure whether we need more bits from software for other features. Here only the least significant bit is needed for the Easter Egg, it is a flag signal telling whether we should go into the Easter Egg mode.

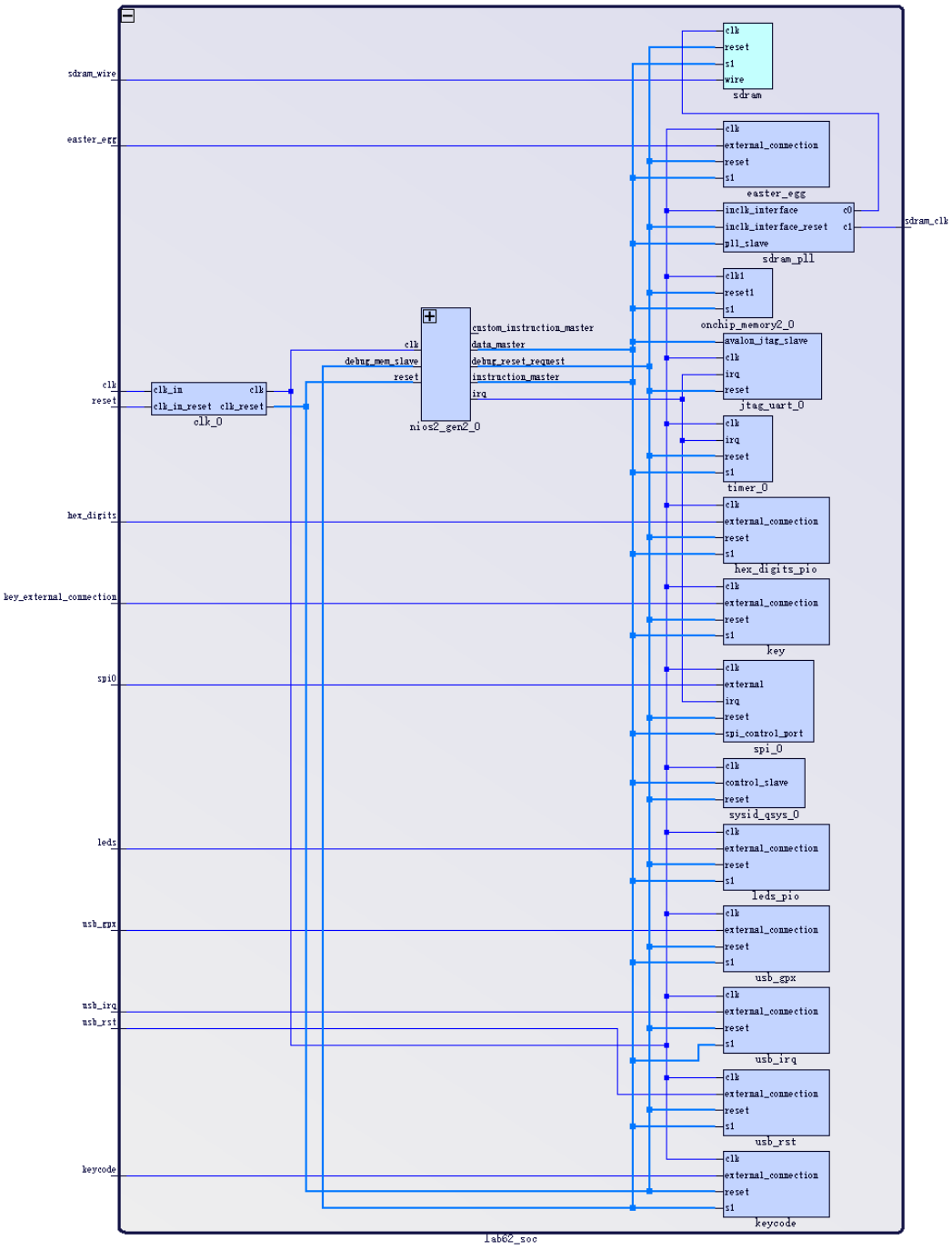


Figure 11: The Schematics of the SOC lab62_soc

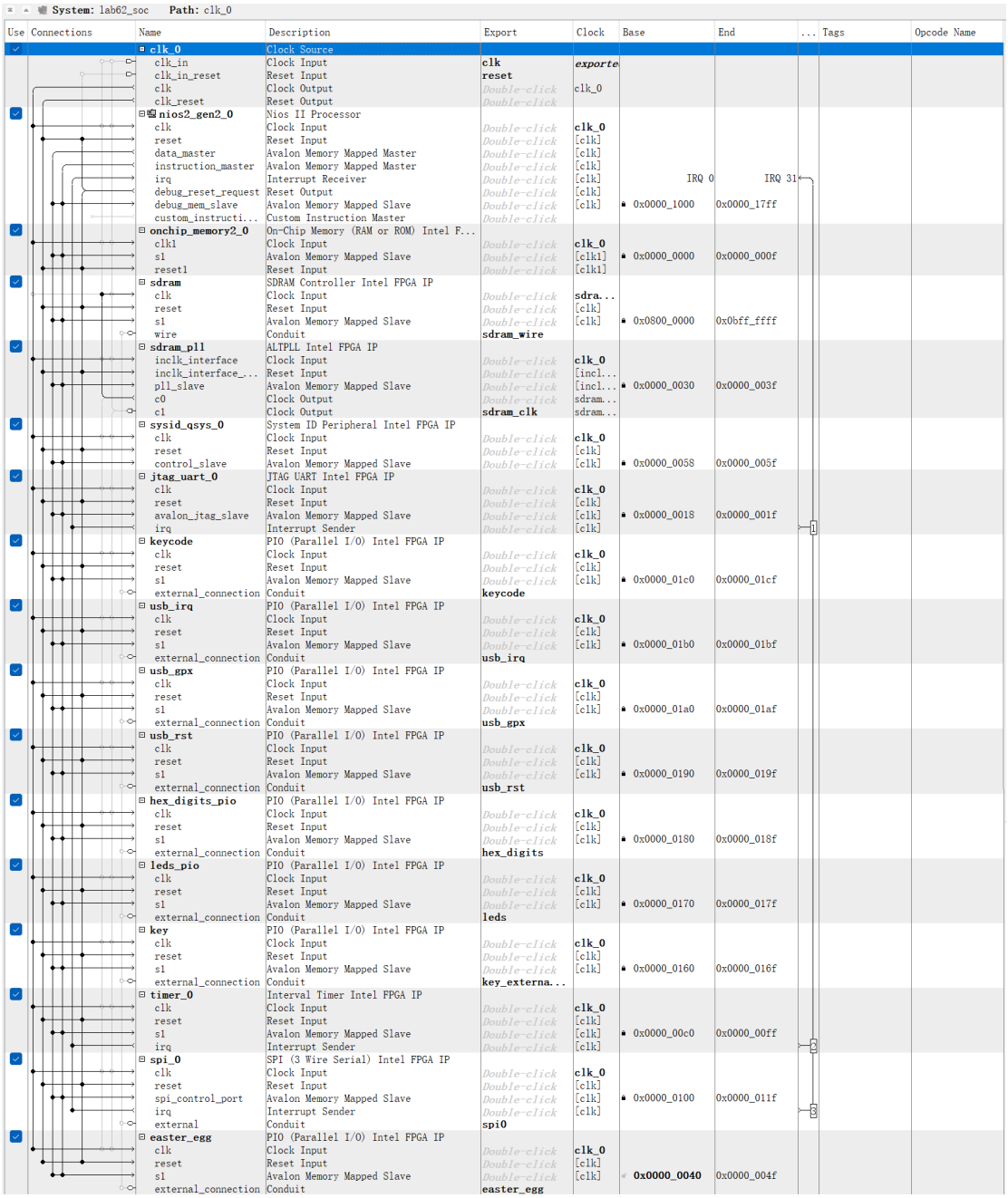


Figure 12: The System View of the SOC lab62_soc

6. State Machines

(1) The State Machine for Writing into Buffer (in Draw_Engine.sv)

The most important state machine in our project is the state machine in **Draw_Engine.sv** which determines the layers of sprite components and the order of drawing them in each pixel. The "XYZ" ={Layer_2_on, Layer_3_on, Layer_4_on} number near the transition lines means if there are components in a layer. X=Whether there is something in Layer2, Y=Whether there is something in Layer3, Z=Whether there is something in Layer4. Each of X, Y, Z = 1 means there is something in this layer, and X, Y, Z = 0 means there is nothing to draw in this layer. Signal "Smaller" means whether WriteX reaches the end of this line, i.e., if WriteX $\leq$ 640, then Smaller = 1, else Smaller = 0, which means we need to stop writing and waiting for DrawY and WriteY to change and begin

to write next line. For each pixel to write into the line buffers, it will check whether each layer has a sprite element, and output the pixel address to the ROM to get the color index, and finally write the color index into the line buffers. After each pixel is finished, it is necessary to go into a state PIXEL_FINISH, where all the output is stable and we can tell that a pixel is already written. Finally if this line of pixel is finished writing, we will go to REST state, and wait for DrawX = 0 to shift the read and write line buffers and to start writing the new line.

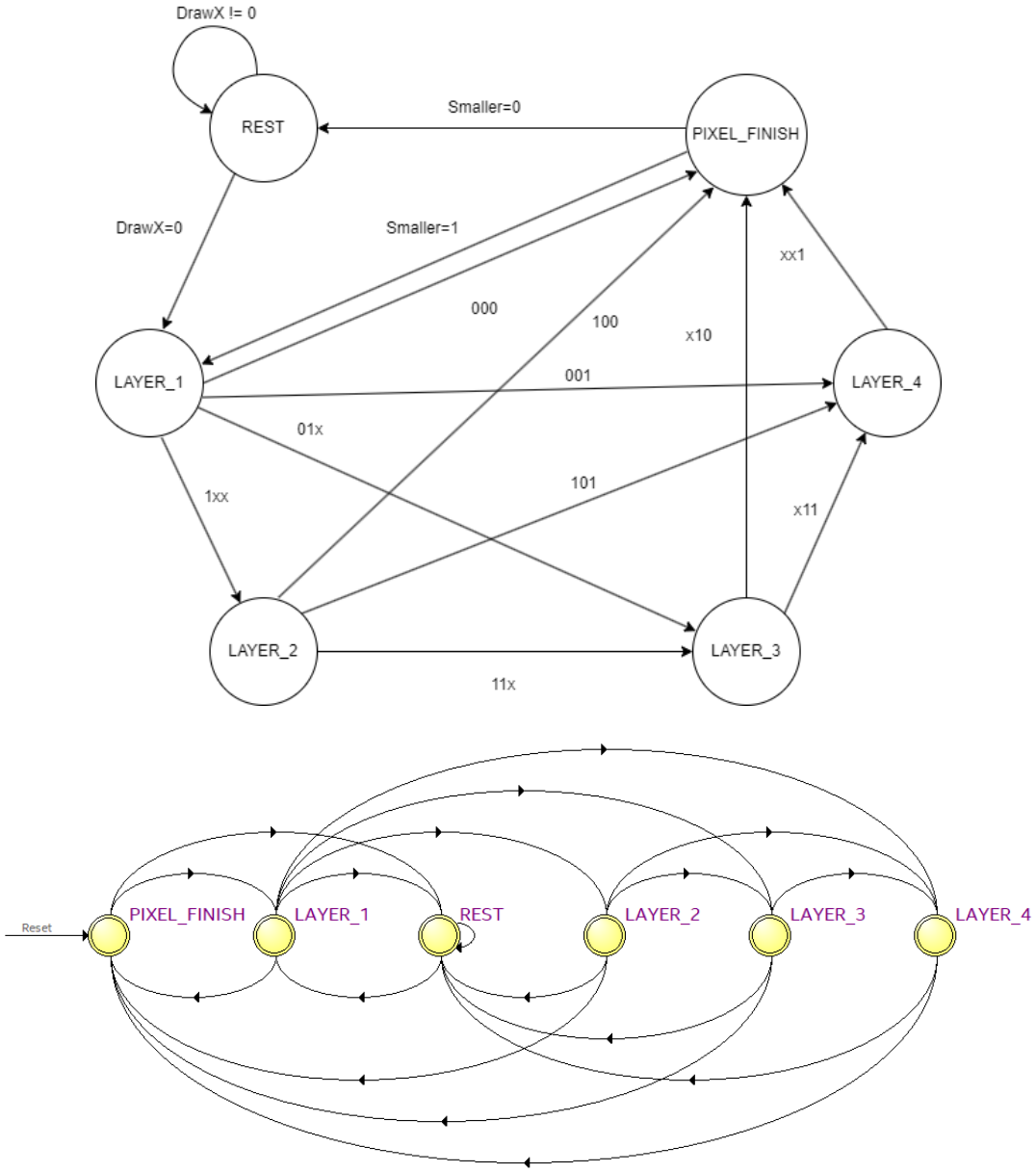


Figure 13: State Machine for Draw_Engine.sv

(2) The Game State Machine (in control.sv)

The state machine we used in the control.sv is used to control the game state. We use the current "game state" to determine whether we should freeze the screen to wait for the start or go into the game over page. At the beginning of the game, we are in "Start" state by default, if the keycode received the Space key or the Up-arrow key, the state machine will go into the "Game" state. In the "Game" state if you crashed into a cactus or a pterosaur, mydragon.Life will be set to 0, and the game state will go into "Over" state, and will use the output "game_state" to freeze the animation

and show the “GAME OVER” interface. Then if you want to restart the game, press “Enter,” the state machine can loop back to “Start” state.

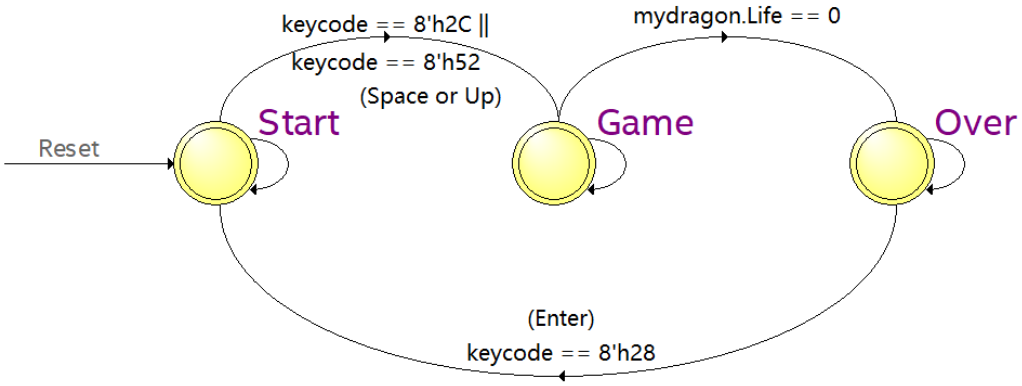
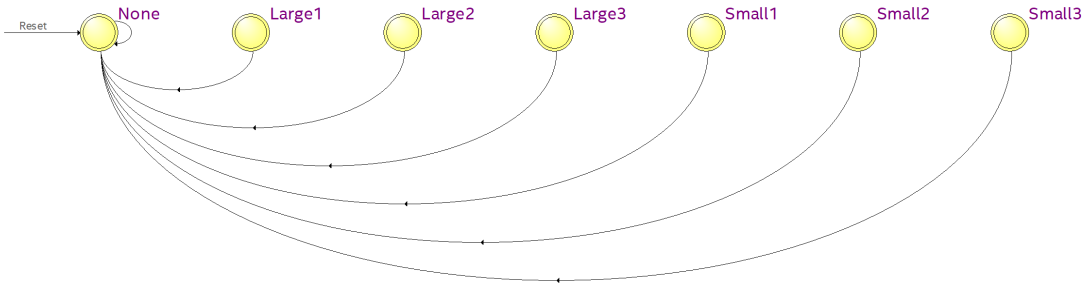


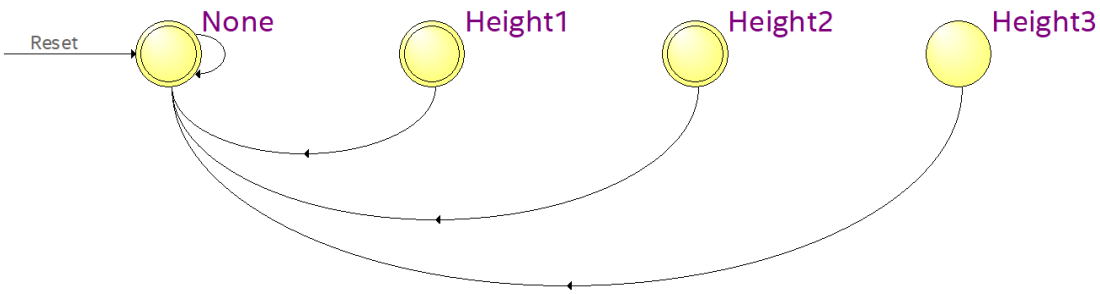
Figure 14: State Machine for control.sv

(3) Random State Machines

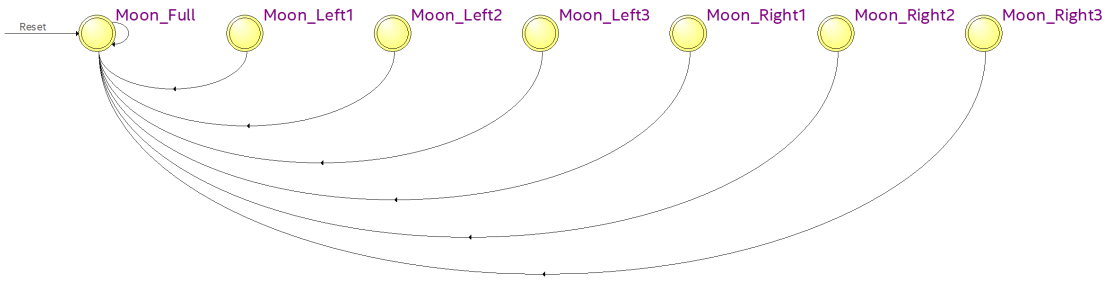
Besides, there are 3 state machines which are controlled by the random numbers. They are: the state machine that determines the size and type of the cactus in the module draw_cactus.sv; the state machine that determines the height of the pterosaur in the module draw_pterosaur.sv, and the state machine that determines the shape of the moon at night in the module draw_moon.sv.



(a) The State Machine in draw_cactus.sv



(b) The State Machine in draw_pterosaur.sv



(c) The State Machine in draw_moon.sv

Figure 15: State Machines controlled by the random numbers

7. Simulation

a) When testing the Drawing Engine, we found that the cactus cannot display correctly, we refer to the SignalTap and find errors in our state transitions.

b) After the Drawing Engine was fixed, when generating the cactuses, we found that there were some glitches with the cactuses. We supposed it was out of metastability. Instead of changing the type of the cactus on the rising edge (posedge) of the frame clock (frame_Clk), we adjusted it to the falling edge (negedge) of the frame clock and the strange cactus was fixed.

c) When debugging, we found that the cactuses and scores had jagged edges, which did not match their sprites in the memory. We supposed that it was caused by the asynchronism of the inputs of the frame buffer. Therefore, we added two buffers (registers as synchronizers) driven by the frame clock on the inputs and outputs of the frame buffer respectively, and then the correct images were restored.

d) When testing the motion control of the cloud, we found that the cloud would disappear on contact with the left edge of the screen, instead of completely merging into the left screen. We identified that this was because “logic” was actually unsigned data type. We change all the data type from “logic” to “int” when it came to control logic, which was similar to coding in C, and the problem was solved.

e) When debugging the motion control codes of the draw_heart module, we found some unexpected behavior that was different from serial execution in C. We then followed the criteria of using “if” and “else” in pairs, and the problem was solved. Additionally, we noticed that the mix use of always_ff and always_comb when implementing control logic of one variable can easily cause error. It is better either to use always_ff or to use always_comb.

f) When displaying the “GAME OVER” in the screen, there are some lines of gaps on the “GAME OVER” sprite. After observing the SignalTap, we found that it is because the sprite is too large so that it takes a long time for the Drawing Engine to write into the line buffer. Since it is there are too many pixels that need to draw 2 layers and wait for another cycle for PIXEL_FINISH, the total time consumed for drawing one line exceed the time between swapping the read buffer and the write buffer. So the gaps will appear. The solution is that we cut the large sprite element into small letters and put them in the right place in the sprite ROM, and then when drawing these small letters, we will save a lot of time, and thus we can eliminate the gaps.

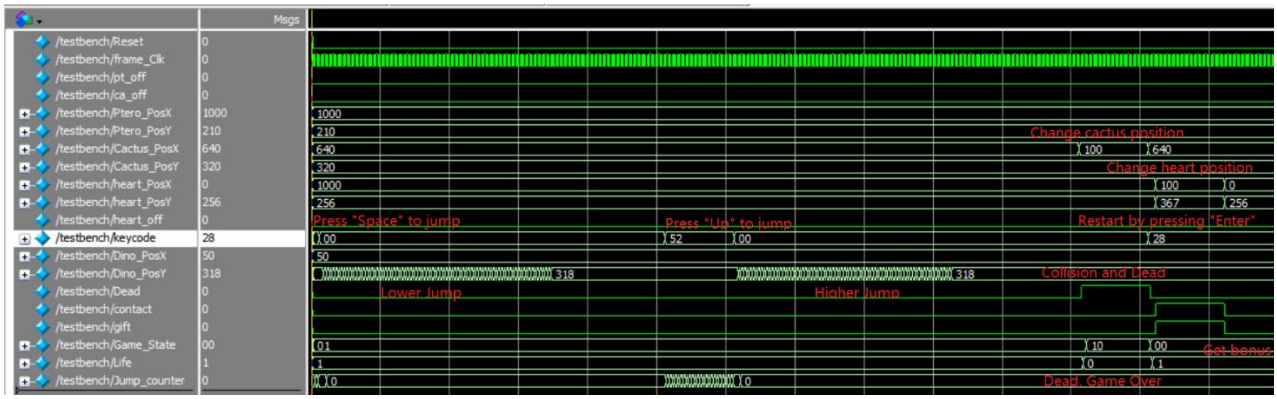


Figure 16: Test Waveform for control.sv

8. Software

There are two parts of software codes in C language in our project. First part is the USB driving code, and the second part is for the output the easter egg flag signal "easter_egg" from PIO to hardware.

The USB driving codes are the same as we used in Lab 6.

The additional codes used for the easter egg are as follows:

```
141 int easter_egg(WORD keycode, int flag)
142 {
143     WORD pio_val = IORD_ALTERA_AVALON_PIO_DATA(EASTER_EGG_BASE);
144     switch (flag){
145     case 0:
146         if (keycode == 0x0c){       //keycode gets "I" (12)
147             flag = 1;
148         }
149         break;
150     case 1:
151         if (keycode == 0x0e){       //keycode gets "K" (14)
152             flag = 2;
153         }else if (keycode != 0x00){flag = 0;}
154         break;
155     case 2:
156         if (keycode == 0x18){       //keycode gets "U" (24)
157             flag = 3;
158         }else if (keycode != 0x00){flag = 0;}
159         break;
160     case 3:
161         if (keycode == 0x11){       //keycode gets "N" (17)
162             flag = 4;
163         }else if (keycode != 0x00){flag = 0;}
164         break;
165     case 4:
166         pio_val = 0x01;
167         if (keycode == 0x29 || keycode == 0x28){        //keycode gets "Esc" (41) or "Enter" (40)
168             flag = 0;
169             pio_val = 0x00;
170         }
171         break;
172     }
173
174     IOWR_ALTERA_AVALON_PIO_DATA(EASTER_EGG_BASE, pio_val);
175     return flag;
176 }
```

(a) The Software Code of the function easter_egg in main.c

```
if (GetUsbTaskState() == USB_STATE_RUNNING) {
    if (!runningdebugflag) {
        runningdebugflag = 1;
        setLED(9);
        device = GetDriverandReport();
    } else if (device == 1) {
        //run keyboard debug polling
        rcode = kbdPoll(&kbdbuf);
        if (rcode == hrNAK) {
            continue; //NAK means no new data
        } else if (rcode) {
            printf("Rcode: ");
            printf("%x \n", rcode);
            continue;
        }
        printf("keycodes: ");
        for (int i = 0; i < 6; i++) {
            printf("%x ", kbdbuf.keycode[i]);
        }
        setKeycode(kbdbuf.keycode[0]);
        printSignedHex0(kbdbuf.keycode[0]);
        printSignedHex1(kbdbuf.keycode[1]);
        printf("\n");

        eggflag = easter_egg(kbdbuf.keycode[0], eggflag);
        switch (eggflag){
        case 0: //000
            clearLED(3);
            clearLED(4);
            clearLED(5);
            break;
        case 1: //001
            setLED(3);
            clearLED(4);
            clearLED(5);
            break;
        case 2: //010
            clearLED(3);
            setLED(4);
            clearLED(5);
            break;
        case 3: //011
            setLED(3);
            setLED(4);
            clearLED(5);
            break;
        case 4: //100
            clearLED(3);
            clearLED(4);
            setLED(5);
            break;
        }
    }
```

(b) The Software Code of calling the function easter_egg in main() function of main.c

```
67    always_comb
68    begin
69       if (Game_State == 2'b00)
70          start = runner1;
71       else
72       begin
73          if (easter_egg == 8'b00000001)
74          begin
75             if (Game_State == 2'b10 || Dead)
76                start = XKdie;
77             else
78             begin
79                if (draw_run3)
80                   start = XK3;
81                else
82                   start = XK4;
83             end
84          end
85          else
86          begin
87             if (Game_State == 2'b10 || Dead)
88                start = die1;
89             else
90             begin
91                if (draw_run3)
92                   start = runner3;
93                else
94                   start = runner4;
95             end
96          end
97       end
98       offset = DistY*SizeX + DistX;
99    end
100   assign address = start + offset;
```

(c) The Hardware Code in draw_runner.sv

Figure 17: Software Codes and a Part of Hardware Codes for the Easter Egg

Here the function “easter_egg” accepts an 8-bit WORD “keycode” and an integer “flag,” and finally returns the updated “flag” as an output. “flag” ranges from 0 to 4, 0 means no keycode matches, 1 means “I” detected, 2 means “IK” detected, 3 means “IKU” detected, 4 means “IKUN” detected. And once we reach flag=4, it means the easter egg is triggered, so we need to change the output value of the PIO “pio_val” into 1. And if we receive “Esc” keycode or “Enter” keycode, we

should reset “flag” to 0 and “pio_val” back to 0 as well, which means we exit the easter egg mode.

For every update, the function will first check “flag,” in each case of “flag,” we then check if the keycode matches, if the next keycode is the desired one, “flag” will go to the next stage, else it will back to 0 due to the wrong keycode. Note that if you do not press any key, you still have keycode = 0x00, for which you should not zero the flag.

In main() function, we call the easter_egg function to update “eggflag,” and we use this “eggflag” to control the LED5~LED3 on FPGA to indicate which stage we are in (in binary).

After the PIO output the value from the software to the hardware, we use this “easter_egg” signal and input it into “draw_runner.sv,” if easter_egg=1, we choose another set of sprite pictures for the dinosaur (the runner), which realizes our easter egg.

9. Design Resources and Statistics

LUT	19417
DSP	20
Memory (BRAM) / 1677312	1253376 (75%)
Flip-Flop	17423
Frequency (MHz)	22.09
Static Power (mW)	98.01
Dynamic Power (mW)	370.62
Total Power (mW)	494.19

Table 1: Design Resources and Statistics for Final Project

10. Problems Unsolved

There is a bug that we have no time to fix: sometimes the sprite pictures of the cactus will be distorted, in a low possibility. This maybe because the metastability when changing the cactus types according to the random number, where the random number changes causing the cactus type to change, but the corresponding “cactus_on” region does not change due to some reasons. We suppose this can be fixed by turning "cactus_off=1" when we find the current cactus type is mismatched with the current cactus width.

Figure 18: Bug

11. Conclusion

After one month's hard-working, we successfully realized the baseline in of our final project proposal, which is similar to the original Google T-Rex game, and then we add the bonus (heart) and the easter egg as new features. Due to the time limit, we are unable to complete all the features we proposed in our proposal, such as spitting fire and breaking high rocks, and pitifully there is still a little bug appears in low frequency. Although there is still a little problem about metastability, we tried our best to present the game on FPGA.

You can check for our complete codes at https://github.com/xiaoniaogangsi/final_project.git and see the demo video at https://www.bilibili.com/video/BV1MP411T77a.

By doing this final project, we learnt a lot on how to organize a large project, and understood the importance of having plans ahead of time. We also gained more experience on on-chip memory usage, the game logic design, SystemVerilog coding and debugging. We also learnt how to analyze a bug by observing SignalTap and the VGA monitor. We think it is a good chance for us to practice and collaborate, and it is a precious experience we will remember for long.