T-REX Run! Implemented in VHDL

Matthew James Bellafaire, Khaled Jarrah, Conner McInnes
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
mbellafaire@oakland.edu, kjarrah@oakland.edu, cmcinnes@oakland.edu

***Abstract*—Through the use of VHDL the aim of this project is to create a version of the Google Chrome "T-Rex Run" game on an FPGA. Using a single button to control all inputs, we seek to employ many parts to ensure proper function of the game's logic and visual display. The game has two primary modes, a game active mode and a game inactive mode, input is received from an external button. Depending on the state of the game, blocks are enable or disabled in accordance to ensure proper functioning.**

I. Introduction

"T-Rex Run!" is a simple platformer game available on google chrome when there is no internet connection available. The game is played by the player controlling a dinosaur on the screen and jumping over obstacles which move from left to right. In order to gain a higher score the player must press a button to make the character jump on the screen and avoid the obstacles. If a player fails to jump over an obstacle the game is reset and the hi-score is recorded and displayed on the screen next to the current score.

The goal of this project is to create this simple game using VHDL implemented on an FPGA. The user can control the game by pressing a large external button connected to the PMOD headers of the Nexys-A7 board. The game itself is displayed on an external VGA monitor which is driven directly from the FPGA development board. Images of the game objects are stored directly on the FPGA through the use of LUTs generated by an Octave script. A central Finite State Machine (FSM) determines the state of the game and interfaces with game-critical components. This project seeks to emulate the "T-Rex Run!" game as closely as possible in functionality.

II. Methodology

A. Top Level

To better facilitate testing and development it was decided to split the project among as many different components as possible. For this reason, the design of the top file of this game relies heavily on blocks which are able to interface with as few inputs as possible. A high level block diagram of the project can be seen in Figure 1.

The only input to this system is the single button which the user uses to tell the on-screen character to jump. Deboucing of the button is handled by the movement algorithm of the dinosaur block itself, allowing for the exclusion of an external debouncing component. On screen game objects, such as the player's character and the obstacles to be avoided are controlled by their own independent blocks which track their location. The movement of these game objects is primarily controlled by these blocks, however they take inputs from the game control FSM reset or freeze their position.

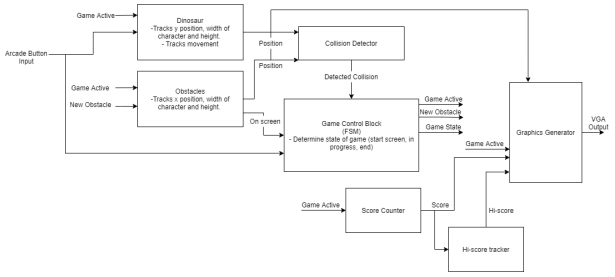

Fig. 1. Top File block diagram of "T-Rex Run!" FPGA implementation

Collision detection is handled autonomously by a collision detection block which is able to determine whether or not the hit-boxes of any of the game objects are touching. This collision detection block outputs a "detected collision" signal whenever the hitboxes of two game objects overlap. The "detected collision" signal is read by the FSM and used to determine the game state.

Score is maintained in two accumulators which constantly tick up while the game is in motion. When a high score is achieved the "hi-score tracker" block captures the score and maintains it until a higher score is registered. These score counters are controlled by the central FSM and only increase when the game is in an active state.

Finally to simplify the overall design of the project the "Graphics Generator" block directly handles the drawing of game objects on the screen. The "Graphics Generator" handles all parts of drawing the game objects to the screen and requires only the x and y coordinates of each game object then draws them to the screen. In addition, the graphics generator is able to display scores provided in binary values as base 10 score's on the VGA display. The goal of this block was to encapsulate the drawing of objects on screen as much as possible to simplify the design of the other blocks in the final system.

B. VGA Output

The VGA output block outputs all game objects to the VGA output. This block is built on top of the vga_ctrl_simple component provided by Daniel Llamocca. The vga_ctrl_simple block is used to draw pixels individually to the VGA display

and to obtain the current pixel being drawn on the screen. This information is used by logic in the VGA output block to determine the desired color of the pixel on screen at the given position. For inputs the VGA output block uses the clock and reset signals as inputs, and a single signal which determines whether or not objects on screen are animated.

Individual onscreen components such as the obstacles, clouds, and player-controlled character are created as separate components. Each of the onscreen object components take as inputs the current horizontal and vertical pixel of the VGA output and the ”animate” signal. For components that move according to inputs given to the VGA output block the x and y position is also taken in by the display component. The on-screen object components output a simple output-mask and a RGB signal, when the display component is considered ”non-transparent” the mask will give a value of 0x000 and the RGB output will give the current RGB value of this pixel. For each game object shown on the screen a image was created using paint and then translated into a LUT by a Octave script. The matlab script created by Daniel Llamocca to create text-files from images was modified first to run in Octave[1], then to output the text as VHDL code that could be copied into the component directly. In order to reduce the size of the LUTs the Ocatave script does not output pixels with an RGB value of 0x000, this value indicates transparency of the pixel, thus the default value of the output is set as 0x000. The generated LUTs serialize the RGB values of the image, in order to obtain the RGB value of a pixel at a given (x,y) position in an image we use Equation 1 where w is the image's width in pixels.

$$P(x, y) = w * y + x \tag{1}$$

The vga_ctrl_simple component provides horizontal and vertical (h,v) position of the current pixel being written to the VGA monitors. In essence each component uses these (h,v) coordinates to determine whether or not they are drawing to that pixel then outputs either a 12-bit RGB value for the pixel or 0x000 to indicate transparency. In the case of translation, objects can be translated on screen using the coordinate system shown in Figure 2. For a component with an image translated by coordinates (x,y) the current RGB output is found by Equation 2 using the horizontal and vertical position of the current pixel (h,v). If the required pixel is outside of the image's size then the component will simply output an RGB value of 0x000 and a mask value of 0xFFF to indicate transparency.

$$RGB(v, h) = P(h - x, v - y) \tag{2}$$

To write multiple independent images to the screen an insert approach was used to give certain components priority over others. Each game object component has an RGB and Mask output, the Mask output is 0x000 when a pixel is being used and 0xFFF when it is not. When a pixel at an indicated (h,v) position is to be written to by a game object the previous component's RGB value is masked off and the new RGB value is inserted onto the output. The result is that all components are essentially placed in a ”chain” as shown in Figure 3, with items in the foreground being found closer to the final RGB output of the chain.

[1]Ocatave does not have support currently for fixed point arithmetic, therefore the conversion of images to LUTs needed to be changed.

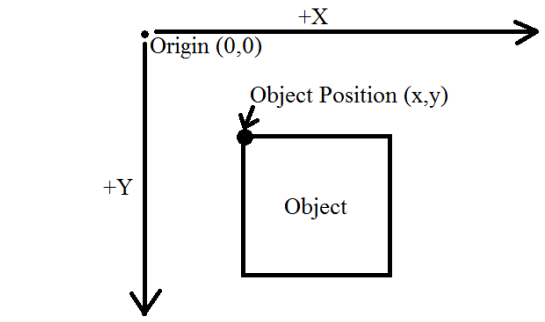

Fig. 2. Coordinate system of the VGA output block, coordinates are referenced to the upper left corner of the screen.

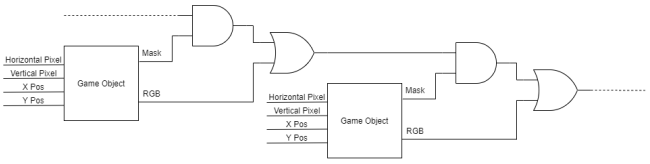

Fig. 3. Connections to individual game objects inside the VGA output block.

In order to show numbers on the VGA display the ”numberDigit” component was created which utilizes the methods previously outlined. The numberDigit uses VHDL parameters in order to set its fixed position and takes in a 4-bit BCD digit which is then displayed on the screen. The numberDigit extracts the numbers to be displayed on screen from the image shown in Figure 4. In order to display the numbers on screen the numberDigit component crops out the required digit from the image and draws that to the screen with black pixels being treated as transparent.

The scoreboard which displays in the upper right hand corner of the screen uses a unique approach in the VGA output block in order to display numbers on the screen. For simplicity the scoreboard was written as a parametric component which can be synthesized to display up to a specified ”max score”, this parameter is used to determine the number of base 10 digits required to display a given max score. The scoreboard generates the required number of numberDigit components to display the given maxScore. A VHDL process was created to convert unsigned binary values given as an input to the VGA output block into BCD digits which could be individually displayed. Using the numeric.std library the binary input of the scoreboard is converted to BCD for the n^{th} digit D with binary

Fig. 4. Numbers image used to draw digits in numberDigit component

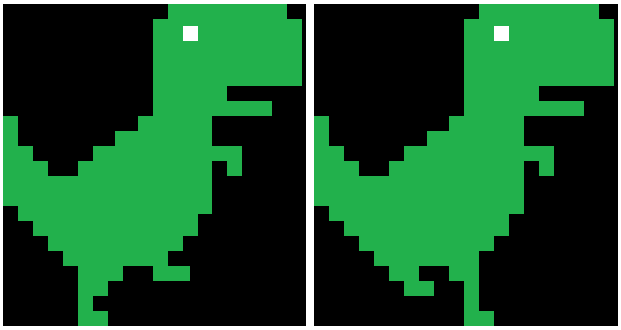

Fig. 5. Frames cycled between by the T-rex visual component when the animate signal is equal to '1'

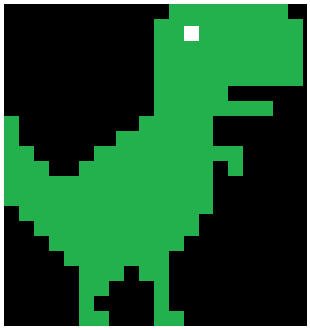

Fig. 6. The dinosaur image displayed by the game when the animate signal is equal to '0'

input z using Equation 3. The output of this VHDL process is then used as an input by the number digits which display the Base 10 score on the VGA screen. This approach allows for the score to be easily provided by external components which are directly controlled by the FSM.

$$D_n = \frac{z \; mod \; 10^{n+1}}{n} \tag{3}$$

In some cases objects will have their own internal animations, display components such as the dinosaur and the clouds have an animation cycle controlled by the "animate" signal input to the VGA display block. In the case of the clouds the visual component moves from the right to the left of the screen whenever the animate input is '1' and stop when that input is '0'. The dinosaur also uses this methodology to create a walking or standing animation depending on the animate input. When the animate input is '1' the dinosaur will display a walking animation by switching between the two frames shown in Figure 5 at a slow rate. When animate input is '0' the dinosaur displays only a single frame with both feet on the ground as shown in Figure 6. The advantage to these self-contained visual components is that they require minimal external interfacing and allow for the player to have an intuitive indication of the current game state.

C. Collision

The collision detection process based on overlap detection between the T-Rex and the cactus. Viewing Figure 7 whenever

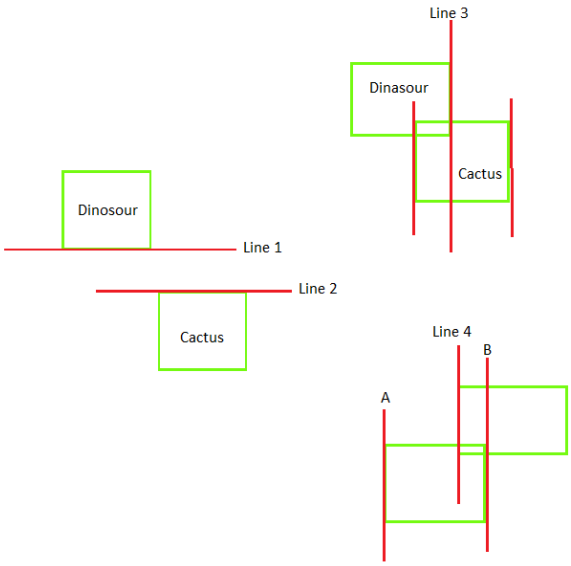

Fig. 7. Collision detector

Line 1 becomes below Line 2 , the possibility of the overlapping will be high. In order to check if there is a collision, Line 3 or line 4 should be located between the cactus X-Left and X-right coordinates. If that so , the collision signal will be set and the game will be ended.

D. Movement

We analyze two-dimensional Dinosaur jump motion by breaking it into two independent one-dimensional motions along the vertical and horizontal axes. The horizontal motion is simple, because ax=0 and vx is thus constant. However, The velocity in the vertical direction begins to decrease as the Dinosaur rises; at its highest point, the vertical velocity is zero. As the object falls towards the Earth again by the Gravity force, the vertical velocity increases again in magnitude but points in the opposite direction to the initial vertical velocity. The Y position can be extracted to give the exact position and height at any given point on the trajectory. However, if the Dinosaur is not in the jump state, he will move with a constant velocity vx. The derived kinematic equations to draw the dinosaur motion are shown by Equations 4 and 5, where $Y_{Acceleration}$ is a constant value.

$$Y_{velocity} = Y_{velocity} + Y_{Acceleration} \tag{4}$$

$$Y_{Position} = Y_{Position} + Y_{velocity} \tag{5}$$

The cactus will move with a negative constant speed relative to the Dinosaur movement direction. The cactus is moved exclusively to the left without any vertical velocity.

E. Game Logic

The game logic block controls the status of the game using two processes that will ensure proper functioning of the FSM. It uses a process named transitions to handle the various states

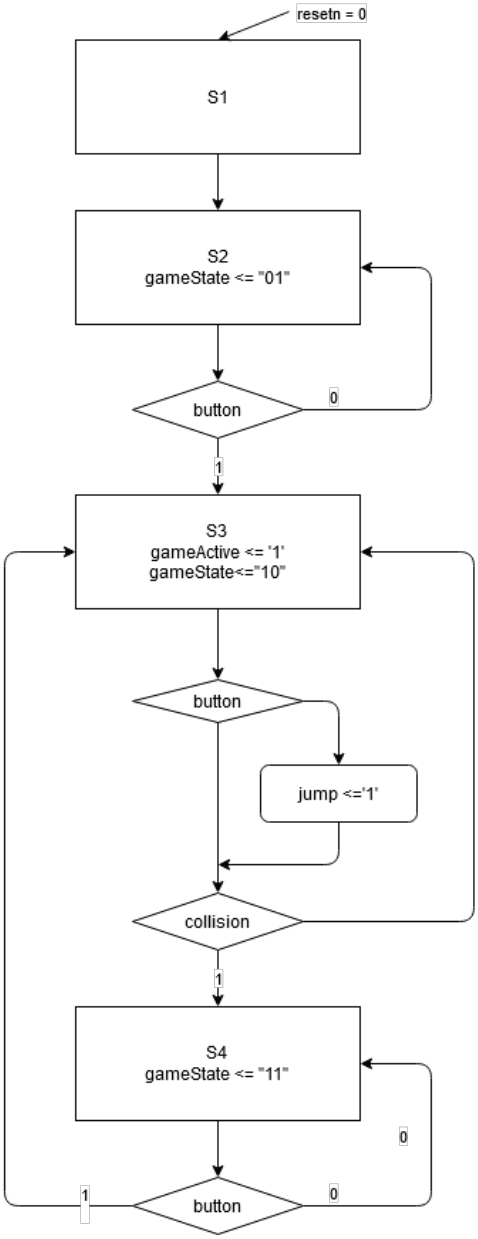

Fig. 8. Fig. 2. Diagram of the multiple states and conditions used for the Game Logic's FSM

of which there are four. The process requires a clock event to run the if statements that make it up. Importantly this process solely handles the change of states, not the outputs. A separate process entitled output is what utilized these states to determine the outputs of the component. Most of the game logic is handled through an FSM that has many outputs that ensures all components will receive the correct inputs depending on the game's status. A diagram for the FSM is illustrated in in Figure 8. The FSM has to major outputs in gameState and gameActive. gameState is one bit signal that is set high when the game is in a state where the dinosaur can jump and cacti are moving. gameActive is a two bit signal has the following significance: "00" is the startup, "01" is the title, "10" is the game play, and "11" is the game over. Next an overview of the state transitions and outputs will be ran through. State one is the startup phase, not much happens here and will immediately change to state two after one clock cycle. All the outputs are set to zero in state one and it is invoked when resetn = '0'. State two is the title for the game, the screen will load up but no animation will be occurring. The gameState is set to "01" and the state will transition to state three if button is high during the next clock cycle. If it is low it will stay in state two waiting for the user to click the button to ready the game. State three is where the game play occurs. gameActive is high and gameState is set to "10". When button is high jump will go high, but this will have no effect on the states. If the Game Logic receives the signal collision as high, that means a collision has occurred and the game should go to its end state and progress to state four. When collision is low the FSM stays in state three. State four is the final state that exists in the FSM and it is just the end state. gameState is set to "11" and until button goes high, the FSM will stay indefinitely in state four. When the button finally recieves a high signal the FSM will loop back to state three and repeat the process until the FPGA is turned off or reset.

The other part of the game logic is a counter that will send signals to generate cacti or increment the score on two second timers. The signals obstacleEnable1 and obstacleEnable2 indicate when one of the cacti should be set to the edge of the screen to traverse it horizontally. Whereas a signal named score is sent every two seconds that will indicate for the score tracker to raise the score. This creates a scoring system that increases as the player stays alive longer. A process called timer will dictate how much time will pass between when these signals are set to one or zero. An unsigned internal signal named time is incremented on every clock event by one. When time reaches the value of 50,000,000 obstacleEnable1 is set to '1' for the clock cycle and reverted back down to zero after. After time reaches 100,000,000 obstacleEnable2 is set to '1', score is set to '1', and finally time is set back to zero. The values of 50,000,000 and 100,000,000 were chosen because we are utilizing the 50Mhz clock built to the Nexys A7 board, each clock pulse will increment time by one so after 50,000,000 clock pulses one second will have passed. An additional statement exists that will reset the time variable when the game is no longer active back to zero.

F. Score Tracker

This block receives a command to increment the score by 100 and will do so if and only if the game is active. In addition it is responsible for storing and replacing the hiscore when a new hiscore has been achieved. It uses the clock event and multiple if statements inside of a process to to increment the score correctly. When the game has ended and gameActive = '0', a signal named scoreNum is compared against hiscoreNum and replaced if it is greater. Following this scoreNum is set back to zero and awaits for the game to restart before it is incremented again.

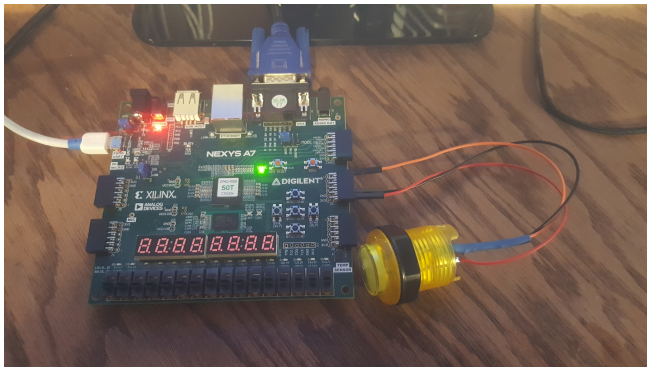

Fig. 9. Experimental setup of the FPGA board using an external 30mm arcade button with a 10kΩ pull-down between the read pin and the JA[1] connector on the Nexys A7, and VGA output connected to an external monitor.

III. Experimental Setup

As the game is really hard to be simulated in the real game time, we simulated the entities over milliseconds to insure the functionality of the game. We faced several problems in this case from denouncing, high pulse duration and animation ratio problems. However, we could solved several problems on this level and for that we saved a lot of time and we simulated over the real times just few times.

The simulation was to test all the functionality, signals and how the FSM controls the game. we were able to following the signals one by one and analyzing every signal in the game to make sure that this signal did its duty perfectly without affecting any other signal functionality.

During these small simulations, we could imagine the functionality of the game, and expand it on the real game time which is in seconds. By this scenario, we were able to build the game on a very short period. Testing for the final game consisted primarily of playing the game and searching for any unexpected behavior. When unexpected behavior was found the system was re-simulated when possible to correct issues or collaboration between the group was used to determine the source of problems.

In the final testing procedure a 30mm arcade button was connected to the Nexys A7 board and used to drive the input of the dinosaur. The button was connected with a 10kΩ pull-down resistor between the button's lead and the GND pin of the JA header on the board. An image of the testing setup can be seen in Figure 9, showing the VGA connected to the FPGA and external button utilized. For much of the testing of the program the BTNC button on the FPGA board was utilized to facilitate collaboration between the group working on this project.

IV. Results

The results of this project were overall highly positive. even though if we faced several issues in the design, in connecting several entities together or even dealing with libraries that has no synthesis, but we were able to build the T-rex game and getting high score in it by one button input.

As the game entities is strongly controlled by the FSM which controls all the signals sent or received by the other classes in the game, we would able to make the game so realistic on a low level environment. The Dinosaur will keep running and the score will be increased until the collision happened. Once the collision happened, the score will persists if it is higher than the registered high score and every thing will be frozen. when the button is pressed a gain, then the game will restart again to get other high score.

Finally, after all testing, simulations and effort, we could build a real game on FPGA connecting to the VGA. In the final implementation the game was tested by being played to find any unexpected behavior. The final game ran without any significant issues at the end of development. Animations, such as the dinosaur's walking animation and the cloud movement, were able to operate properly while the game was in an active state and did not continue when the game was not running. The score accumulator was able to accumulate score as expected, counting by 100 for each 2 seconds the player avoided an obstacle, and loading the high-score to the proper scoreboard at the end of the game. Finally the collision and obstacle blocks were able to determine when a collision had occurred between the player and an obstacle, resulting in a game-over status.

The game was able to be played without resetting the FPGA or any special input, the FSM was able to control the game state without any significant issues. In the active state the game displayed cloud and character animations and moved the obstacles from right to left on the screen as shown in Figure 11. Whereas in the inactive state the game was able to pause all on-screen animations and move initiate the game when the player pressed the input button as shown in 10.

Fig. 10. T-rex game in start-state before the player presses the input button

V. Conclusions

Overall the project turned out in a state as good or even better than we imagined. That's not to say at numerous points during the development that there were issues. In fact, a lot of important concepts about VHDL and implementation of parts and process was made apparent. One of which was the how significant it is to include every variable you read from in process in the sensitivity list. Lots of debugging time could

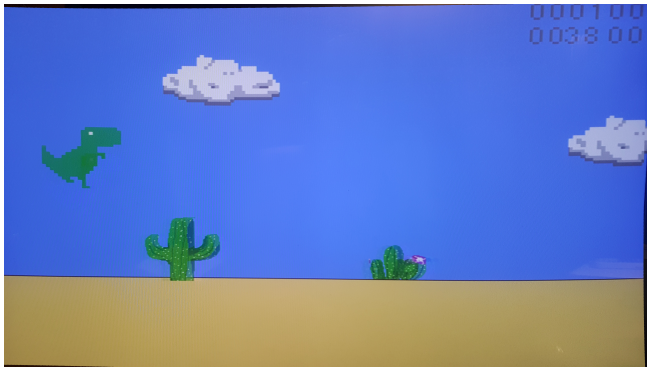

Fig. 11. T-Rex Run game in active state, note score counter in the upper right accumulating score by 100 and the saved high-score

have been saved had the sensitivity list been implemented properly on the first time of synthesizing it. In addition the use of libraries that utilized floats complicated the project when it came time to synthesizing. These libraries would not synthesize and thus the code had to be completely rewritten and debugged very late into the development. This caused issues with port mapping the component to other components and threw back numerous errors until it was finally fully debugged. Compartmentalizing the code into distinct blocks that would handle calculations locally allowed for each part to rely on one another without any one component becoming to burdened with a lot of code. The use of LUTs to store the object data also enlightened us to how intensive using large LUTs can be when trying to synthesize, implement, and generate a bit-stream. One additional thing is, since the components were made to work with real-time conditions, simulating the code become a challenging without rewriting or redefining numerous variables to work at a more efficient time scale for simulation.