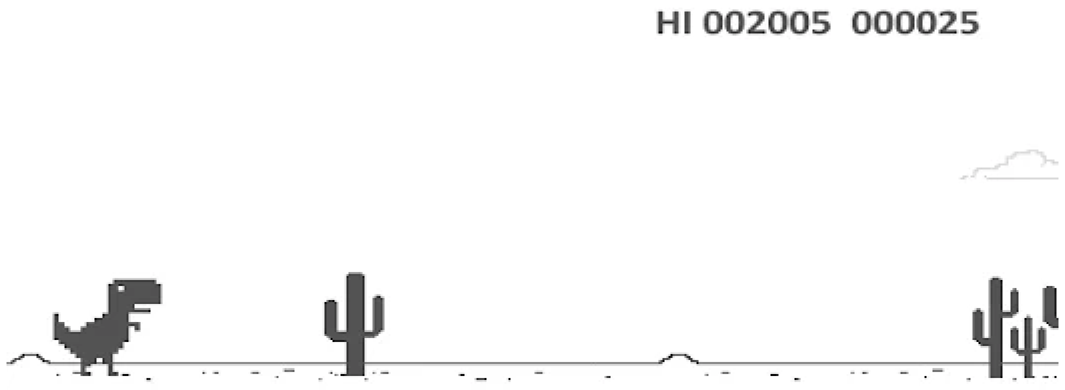
# T-Rex Game Implementation on FPGA

Linus Lei <linuslei@usc.edu>
Way Zheng <wayzh@usc.edu>

## I. Abstract & Introduction

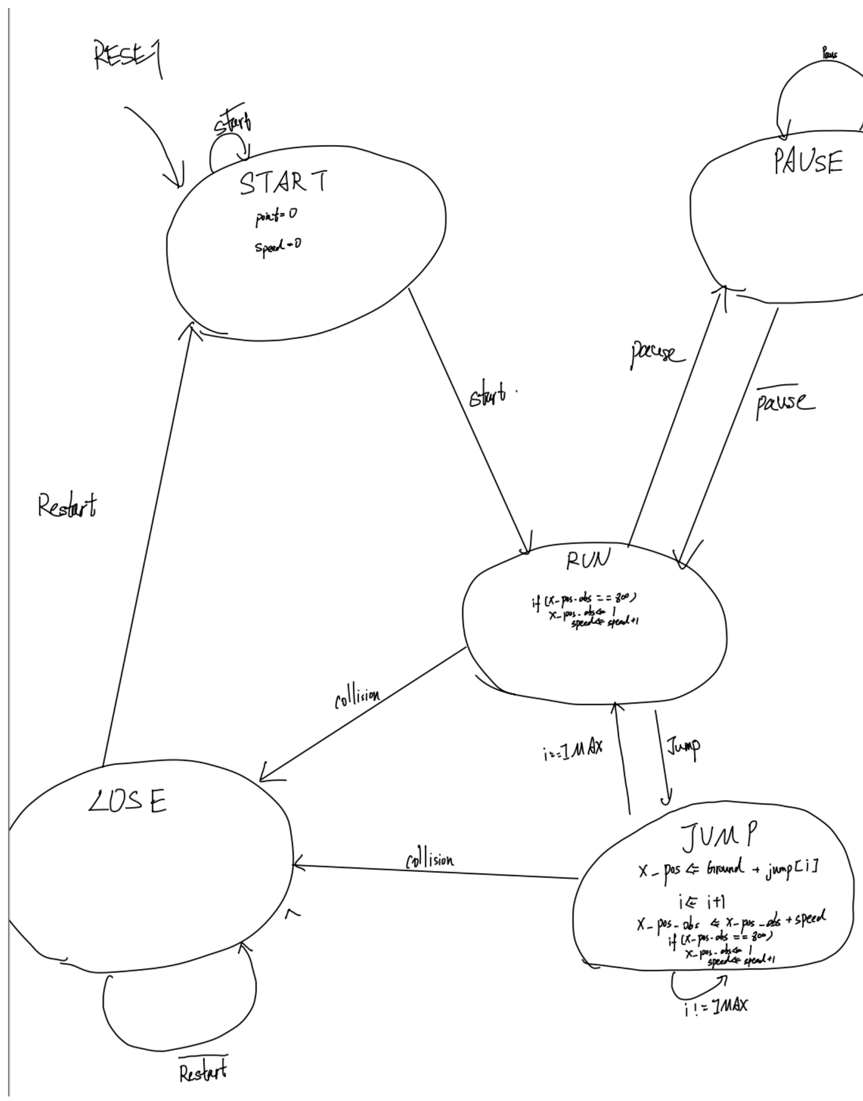
For our project, we want to design a simple game that is similar to the T-Rex game that pops up every time chrome loses its internet connection. We plan to have a similar layout, with the T-Rex being moving in the y-direction, stationery in the x-direction, and the obstacles moving in the x-direction and stationery in the y-direction. We will try to design a randomly generated course that rolls with an ever-increasing speed.



Similar to the original game, the T-Rex needs to jump across the Cactus and the wide Cactus. The point will be the number of obstacles the T-Rex jumped. Whenever a T-Rex hits an obstacle the games stops and resets.

## II. Design

State Diagram:



### State Explanation and Main Feature Implementation

Initial State:
In the initial state, all game variables are set to 0. This includes points, speed, and obstacle detection (collision). The state will persist until the game is initiated with the 'start' action.

Run State:
Once the game starts, the character begins to run. The speed is set to a non-zero default value, indicating motion, and points start to accumulate over time. If an obstacle is detected (collision = 0), the game continues in this state, with points incrementally increasing. The game can

transition to a 'Pause' state if the pause action is initiated, or it could end up in the 'Lose' state if the obstacle is hit (collision = 1).

Pause State:
The game enters this state when the player pauses the game. The current points and speed are maintained, but the game is in a suspended state. The game can either be resumed, returning to the 'Run' state, or reset, which brings it back to the 'Initial State.'

Jump State:
This state is a subset of the 'Run' state, where the character jumps over an obstacle. Points continue to accumulate, and the speed remains constant unless adjusted by the game's logic (not shown in the flowchart). After the jump, the game returns to the 'Run' state.

Collision Detect:
The way we are doing is we take collision as a wire that is assigned to when the obstacle hits the T-Rex. To start off, the x_pos, y_pos and x_pos_obs, y_pos_obs are all originated as the top right corner of the sprite or the obstacle Since we used a sprite for the T-Rex, we know the size of the sprite is 50*23. Hence we could detect to see if the obstacle is in bound with T-Rex sprite. Also, sicen the T-Rex is in a fixed x_pos, we only need to worry about the y_pos.

Speed Control:
The speed control is incremented by 1 everytime the obstacle hits one end and needs to be regenerated.

Jumping with gravity:
We encountered many difficulties trying to calculate the x_pos while jumping in real-time. Especially due to the fact that if most operations are done integer-wise, it would be hard to scale back to the VGA display output which is only 800 pixels in height. Since we are not considering the force and duration of the jump button, we used a hard-coded, pre-calculated array with 40 entries, each of which is the y_pos of the sprite at that clock (starting from detecting the jump). We used a slowed-down clock of 0.02 seconds per clock (slowed down using a mix of divclk and counter), and loaded the y_pos from the array to achieve a 0.8-second jump. The overall effect was pretty smooth and reliable.

## IO handling & Display handling

IO Handling:

For our game the input is BtnC for reset and start, a switch for pause and the Up button for jumping. The Up button, normally should be implemented with some kind of debouncing scheme. However, since we are using the array method for jumping, the debouncing is automatically done since pressing the button meant triggering a 0.8-second script which serves debouncing purposes.

VGA Display Handling:

For the VGA display handling, we used the vga_moving_block as the base. We kept the display_controller as our scanning program, and we implemented the sprites using a bit map. The logic is similar to block fill. We load a bit map in the form of array into the program. Then we calculated the relative x,y pos from the display to the array and we use the similar method as blockfill to create the dinosaur sprite. As for the obstacles, they were taken care of using a simple blockfill.

SSD Display:

The SSD display was used to display the speed of the dinosaur, which is also the score in our case.

## III. Test Methodology

To test our module, we developed based on the VGA display example, we ensured an image output as first priority and base our tests around it. We consider the edge case where the dinosaur happened to be on the two corners, we set the dinosaur's position manually in our code to see if it triggers. Also we left a backdoor that allows us to control the dinosaur like a block in our code for better debugging]

## IV. Conclusion

In conclusion, the T-Rex project was able to capture the main essence of the original game. The jumping with gravity and collision control worked smoothly and beautifully. Some steps that could be taken to take the game further include, the random generation of the obstacles, a scrolling background, and the ducking feature which is not implemented in the game. Also the jumping array could be longer to create a smoother jump.