DDC Final Project

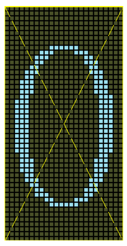
QIANYAN JIANG & HAORAN FANG

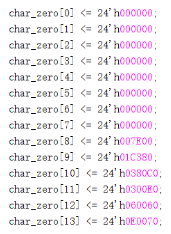
13.12.2023

Based on the moving shapes project, we have created a game similar to "Pac-Man". Players control a red square to eat the randomly generated green dots in the maze. The number of dots eaten within ninety seconds will serve as the player's score. And there will be a time bar showing the time, and a score display.

1. Characters Display

During the game process and at the result screen of the game, characters such as "score" and "game over" need to be displayed. These characters are actually produced from dot matrices. We used software to generate the corresponding dot matrices for the required characters, which has a size of m*n. Then we put them in the VGA_pattern by defining several register arrays composed of n m-bit wide elements and assigning a value to each element. For example, to show the character “0”, it is a 24x48 matrix, it will be a 1 where it has to show a dot, while a 0 means nothing. We store the matrix in heximal just because the software default generated form is in heximal.





```
char_zero[0] <= 24'h000000;
char_zero[1] <= 24'h000000;
char_zero[2] <= 24'h000000;
char_zero[3] <= 24'h000000;
char_zero[4] <= 24'h000000;
char_zero[5] <= 24'h000000;
char_zero[6] <= 24'h000000;
char_zero[7] <= 24'h000000;
char_zero[8] <= 24'h007E00;
char_zero[9] <= 24'h01C380;
char_zero[10] <= 24'h0380C0;
char_zero[11] <= 24'h0300E0;
char_zero[12] <= 24'h060060;
char_zero[13] <= 24'h0E0070;
```

2. Scaling

Scaling:

In order to increase the fun of the game, we have designed the paths in the maze to have varying widths. Therefore, we have added a function that allows the red square to successfully traverse all paths by changing its size. We control the size and position of the red square by changing the (leftX, leftY) and (rightX,rightY), which are the upper left corner and the lower right corner coordinates. By pressing the right and left button together or pressing the up and down buttons at the same time, we can change the size of the square by changing changeS. ChangeS is a signed register, and for example the exact position

leftX = 220 (starting position)-changeS+changeX.

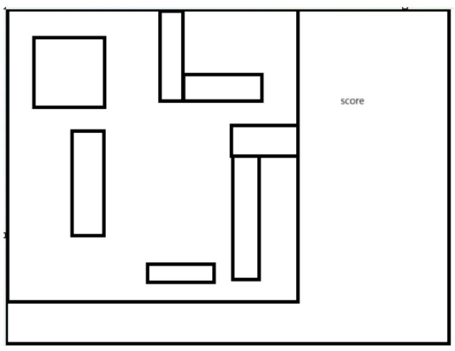
rightX <= 280+changeS+changeX;

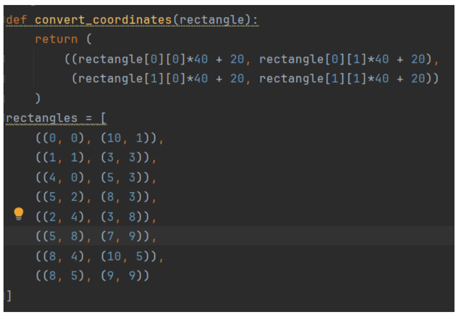
So we succeeded in enlarging and shrinking the square in this way.

3. Maze & Maze Walls Collisions Determination

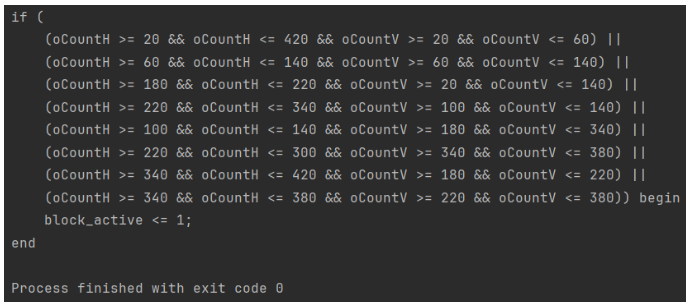
Maze:

We divided the maze walls into multiple rectangles, and the showing logic is the same as showing a square. We design a maze map and change it into Verilog using python.





```python
def convert_coordinates(rectangle):
    return (
        ((rectangle[0][0]*40 + 20, rectangle[0][1]*40 + 20),
         (rectangle[1][0]*40 + 20, rectangle[1][1]*40 + 20))
    )
rectangles = [
    ((0, 0), (10, 1)),
    ((1, 1), (3, 3)),
    ((4, 0), (5, 3)),
    ((5, 2), (8, 3)),
    ((2, 4), (3, 8)),
    ((5, 8), (7, 9)),
    ((8, 4), (10, 5)),
    ((8, 5), (9, 9))
]
```



```
if (
    (oCountH >= 20 && oCountH <= 420 && oCountV >= 20 && oCountV <= 60) ||
    (oCountH >= 60 && oCountH <= 140 && oCountV >= 60 && oCountV <= 140) ||
    (oCountH >= 180 && oCountH <= 220 && oCountV >= 20 && oCountV <= 140) ||
    (oCountH >= 220 && oCountH <= 340 && oCountV >= 100 && oCountV <= 140) ||
    (oCountH >= 100 && oCountH <= 140 && oCountV >= 180 && oCountV <= 340) ||
    (oCountH >= 220 && oCountH <= 300 && oCountV >= 340 && oCountV <= 380) ||
    (oCountH >= 340 && oCountH <= 420 && oCountV >= 180 && oCountV <= 220) ||
    (oCountH >= 340 && oCountH <= 380 && oCountV >= 220 && oCountV <= 380)) begin
    block_active <= 1;
end

Process finished with exit code 0
```

Maze Walls Collisions:

Whether it is to enlarge or shrink the square, or to move in any direction, it is necessary to first determine whether the maze walls will hinder this change before the actual change occurs. We have defined an "en_change" register to enable the scaling and moving functions, which is decided by a next state logic. Every move will change the next_state element, and check if the next_state is in the legal area. If yes, then we put the next_state value to the current state. If not, we assign the next_state back to the current state, and the square remains in its original position and size before it touches any walls.

4. Dots Generation & Eating dots

Dots Generation:

We use two 11-bit registers, “foodX” and “foodY”, to represent the center coordinates of the dots, and define two 11-bit registers, “randomX” and “randomY”, as well as a two-bit register, “randomtimer”. To generate dots at random positions, we define that within a certain horizontal and vertical range, whenever the value of “randomX” is incremented by 2, the value of “randomY” is incremented by 1. In order to prevent the dots from appearing within the maze walls, we introduce a register called "legal". When either the value of “randomX” or “randomY” falls within the range of the maze walls, the value of "legal" is set to 0, and the values of “randomX” and “randomY” are not assigned to “foodX” and “foodY”.

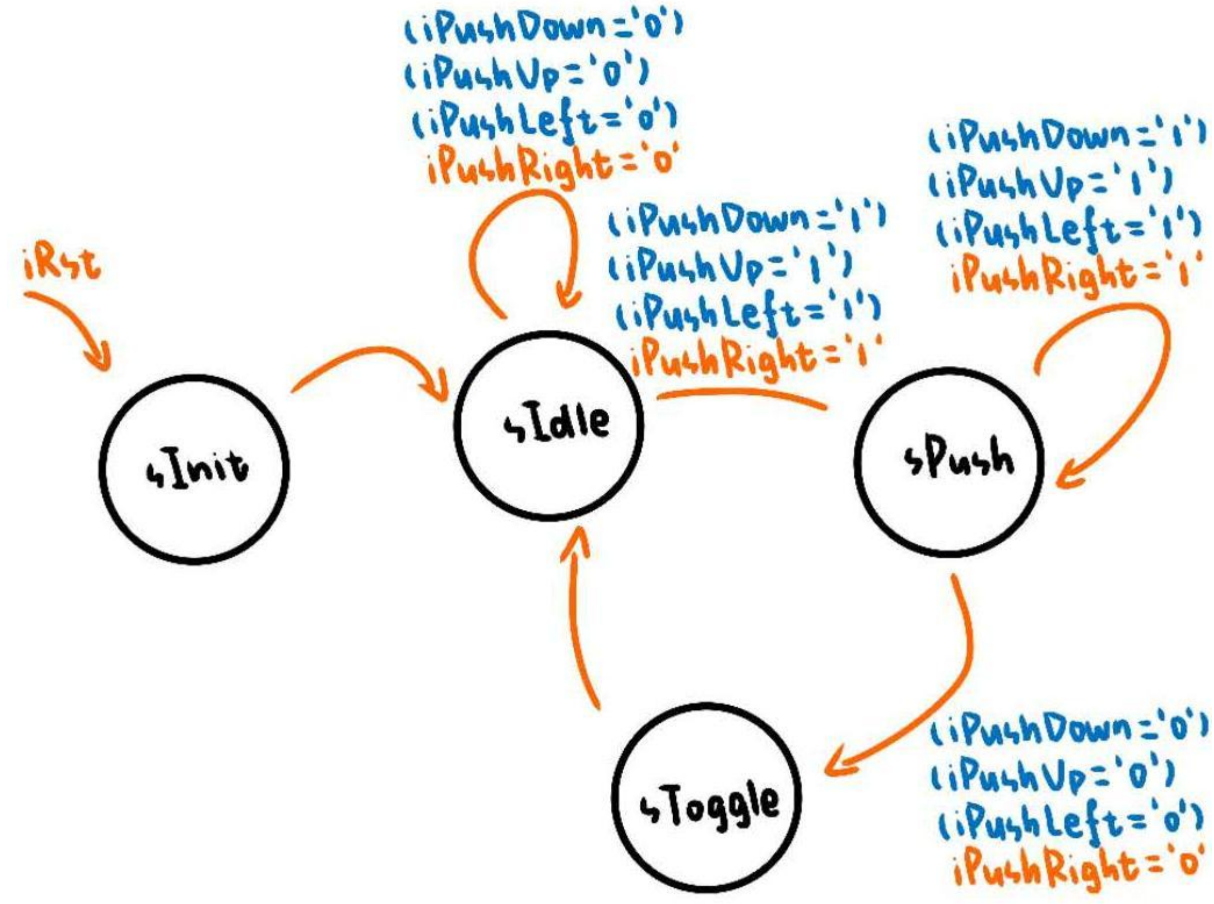
Eating Dots:

The basic condition for dots to be eaten is that the center of the dot is within the same horizontal or vertical range of the square. With this condition satisfied, when any side of the square touches the dot, it is considered that the square eats the dot, and a new dot will appear in another position following the logic of dots generation.

5.State Transition Diagram

Button state FSM:

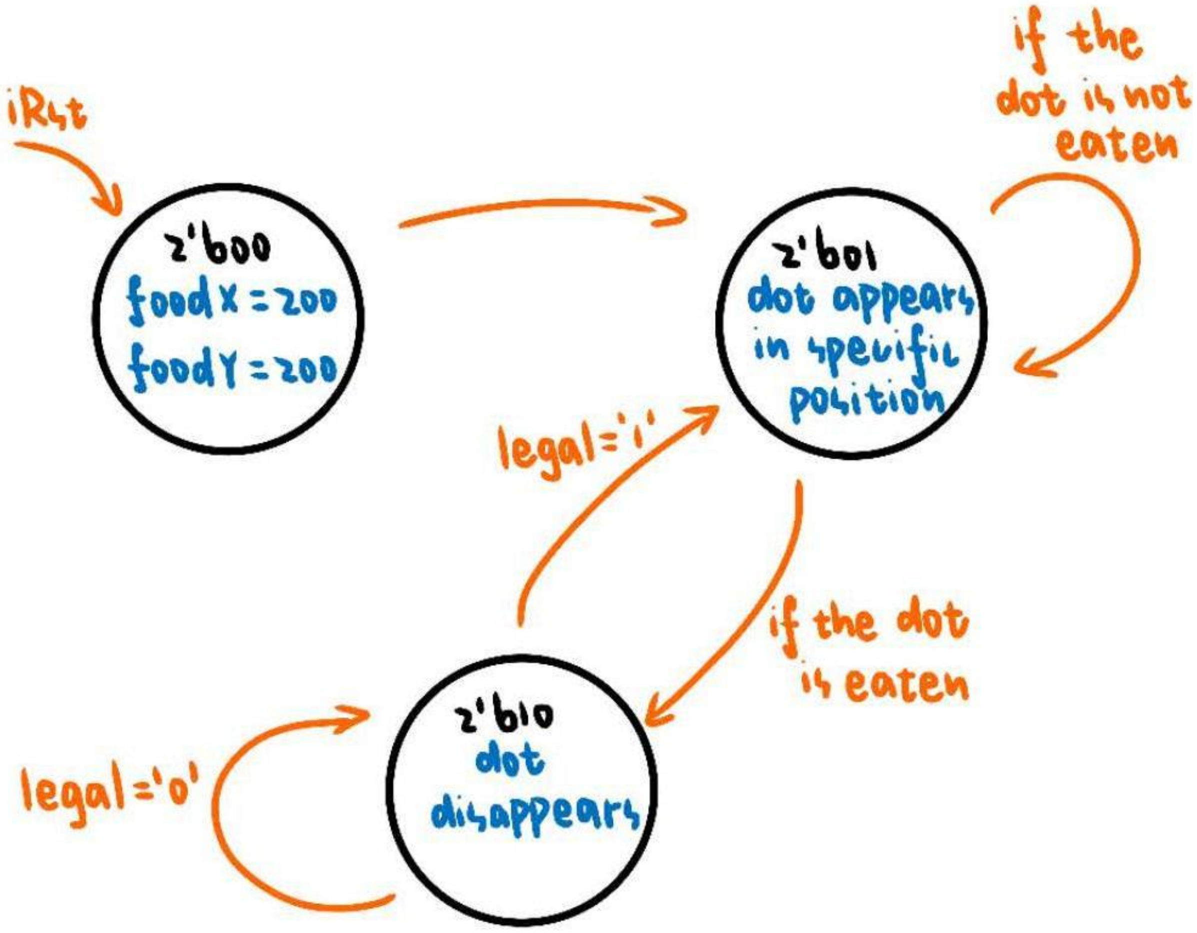
There are four states in FSM to decide the state of the button. When the program starts, it will go into sInit, which means an initial state. Then it will go in sIdle, waiting for a typical button being pushed. if any button is pressed, it will go into sPush while the button is kept triggering. After the button is released, it will go into sToggle for 1 clock cycle, which marks a state change between being pressed and released. Then it will go back to sIdle and start a next round of waiting.



FSM of Four Buttons

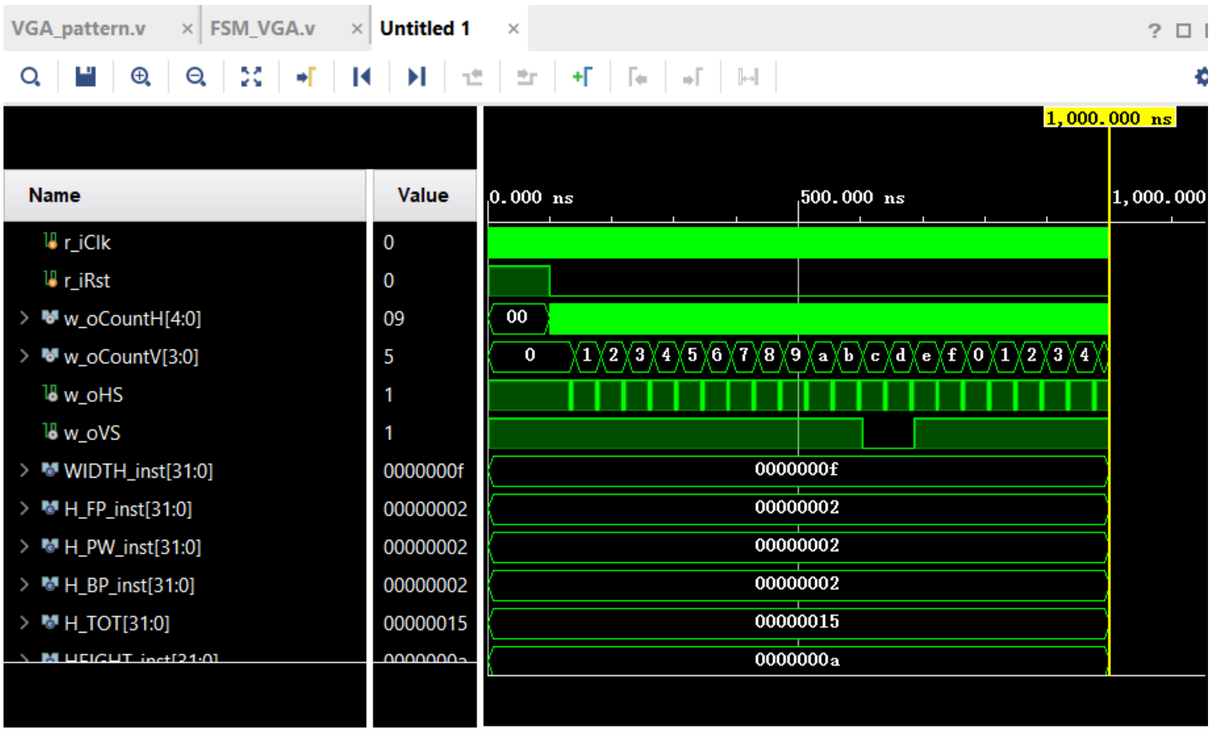
Dots state FSM:

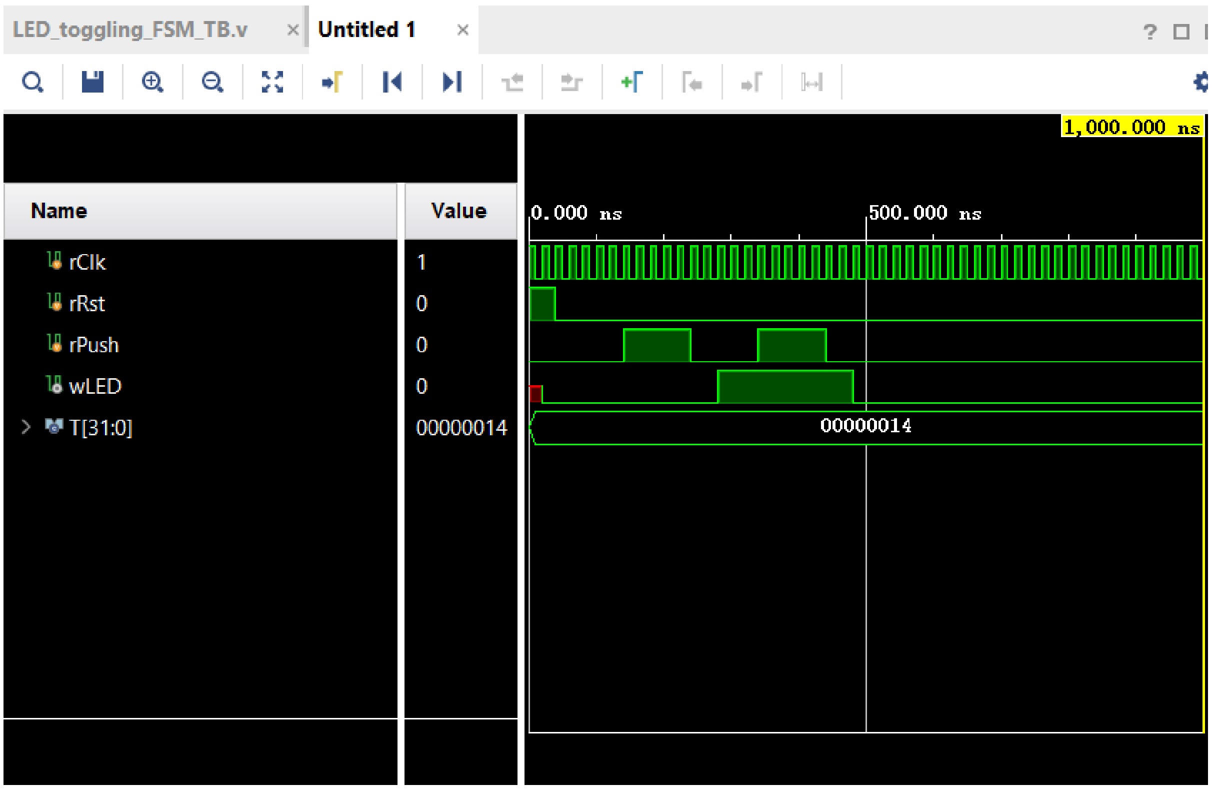
The dot(food) in the programme has 3 different states. When the programme begins, it will go into 2’b00 state, which gives foodX and foodY a initial position at (200,200), then it will automatically go into 2’b01 and start waiting. If the dot is being eaten (decided by the relative position between the red square and the dot) the FSM will go into 2’b10. And in 2’b10 state, it will refresh the position of the dot (checking if the current randomX and randomY is legal).



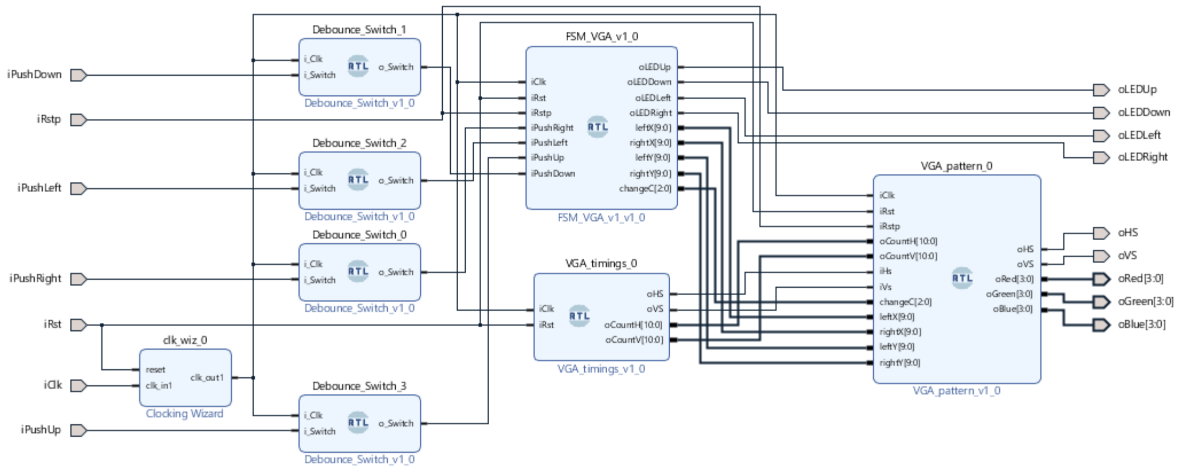
State Transition of Dots

6.Test bench of VGA_timmings and LED_toggling

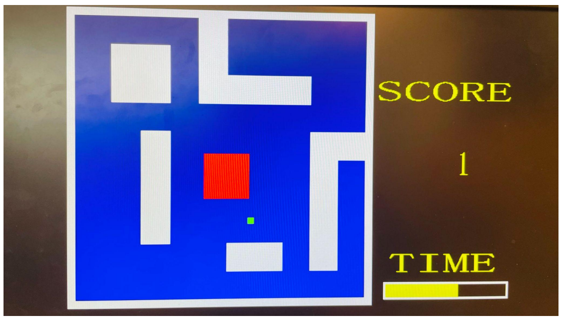




7.Block desgin



8.Final UI



9.Extension

The project is uploaded to:

https://github.com/mengmengdm/PAC_MAN-GAME-base-onFPGA/tree/main