TETRIX

Carlton Knox   Po Hao Chen   Basil Ng

https://github.com/carltonknox/TETRIX

Goal / Motivation

Tetris the puzzle video game classic that we all know and love. The player stacks a sequence of “tetrominoes” into a 10x20 grid.

Goal: We set out to re-implement the game logic on a 32 x 64 matrix with support for multiplayer.

Motivation: this challenge to work with the LED matrix and to do something cool with it.

Applications: Our design can be easily adapted to other display, controllable by the user, such as a digital clock, and more.

Short Functionality

>> Player controls the blocks via keyboard PS2 input (Po Hao & Basil & Carlton)

>> Input goes into the game logic

>> Tetris Game Logic (Basil & Po Hao & Carlton)

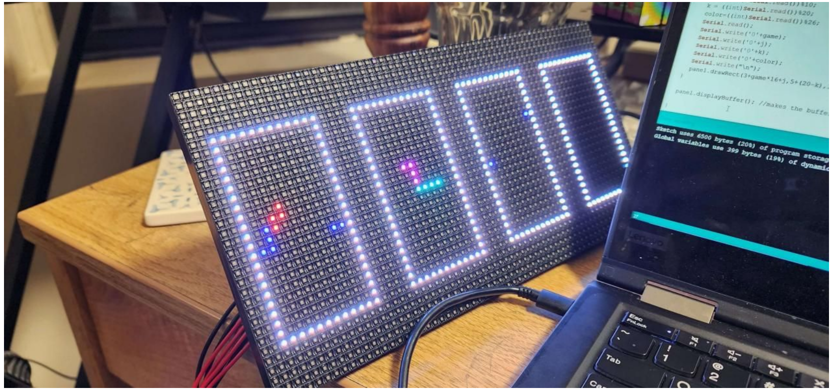
>> Falling Block FSMs (Carlton & Basil)

>> FPGA outputs Tetris graphics (Arduino/Matrix or VGA)

(Carlton)

>> Game Ends When Fall Fails

(Po Hao & Basil)



```
      Serial.read())%10;
    k = ((int)Serial.read())%20;
    color=((int)Serial.read())%26;
    Serial.read();
    Serial.write('0'+game);
    Serial.write('0'+j);
    Serial.write('0'+k);
    Serial.write('0'+color);
    Serial.write("\n");
    panel.drawRect(3+game*16+j,5+(20-k),
  }

  panel.displayBuffer(); //makes the buffe
}
```

```
Sketch uses 6500 bytes (20%) of program storag
Global variables use 399 bytes (19%) of dynamic
```

Short Specification

1. Reduce high clock frequency of falling blocks (11~12 ns delay),

   Solution: Divide the clock to increase the allowed delay. (20 ns)

2. UART controller (Graphix_printer) (> 20ns)

   Solution: Divide the clock twice to increase the allowed delay. (40 ns)

Detailed Functionality: Tetris FSM

S_INIT = 0, S_FALL = 1, S_SET = 2, S_BREAK = 3, S_OVER=4;

S_INIT: sets block at top of the screen

S_FALL: 1. falls block until it reaches the bottom (or a different block)

2. NO_INPUT = 4'b0000, RIGHT = 4'b0001, LEFT = 4'b1000, INSTA_FALL = 4'b0100, FAST_FALL = 4'b0010, ROTATE_RIGHT = 4'b0011, ROTATE_LEFT = 4'b1100;

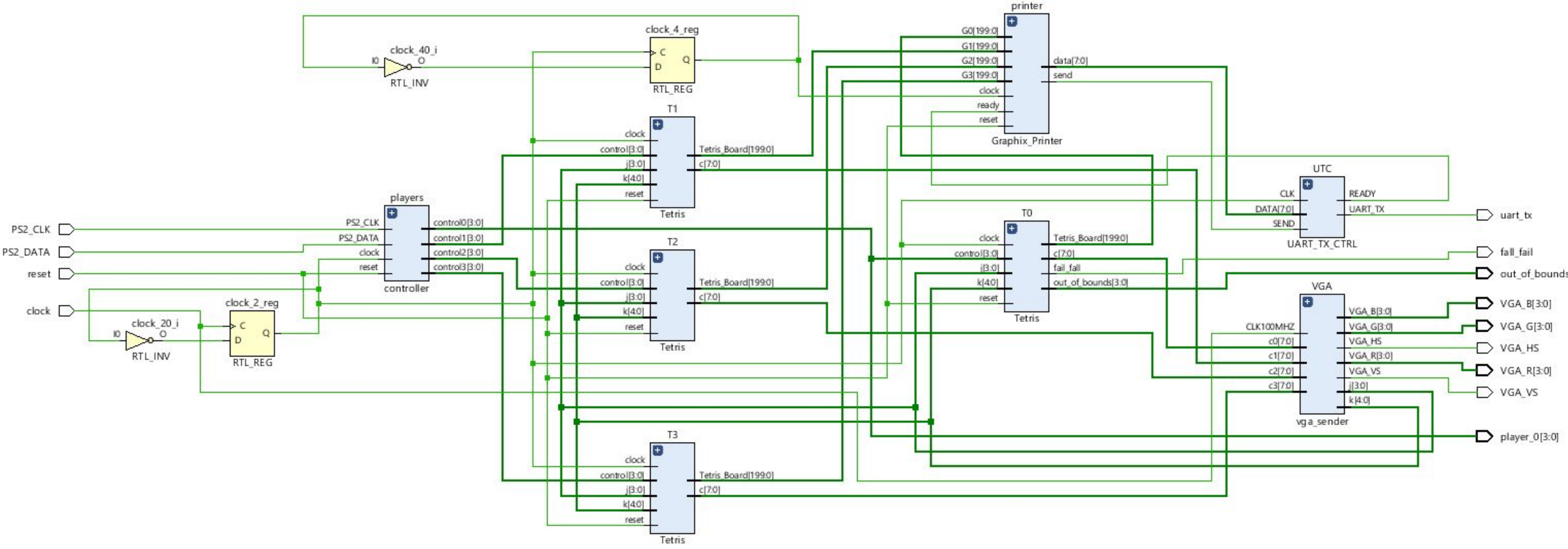
S_SET: 1. once collision is detected, set the block in place

2. This state also checks if the block hits to top boundary (to S_OVER)

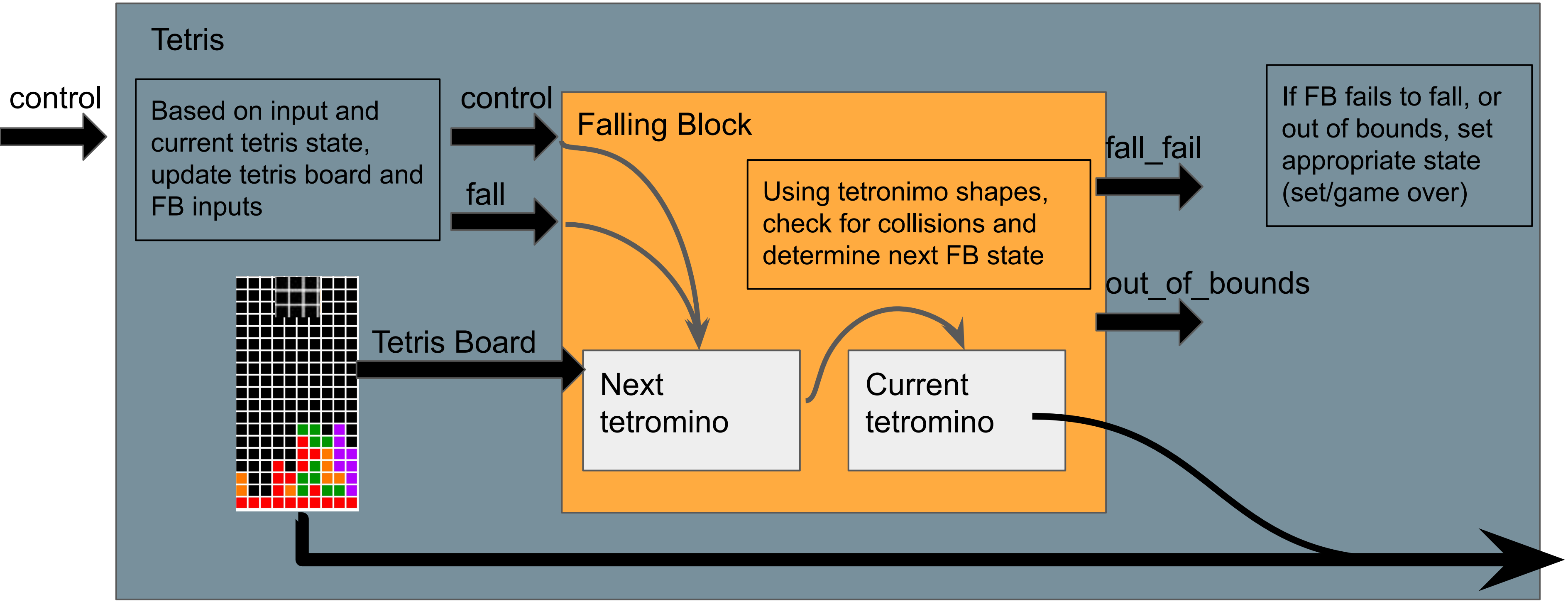
S_BREAK: shifts out all filled line, and replace to with zero on top

S_OVER: ends game

Top-level Block Diagram



Tetris Block Diagram



Tetris

control

Based on input and current tetris state, update tetris board and FB inputs

control

fall

Tetris Board

Falling Block

Using tetronimo shapes, check for collisions and determine next FB state

Next tetromino

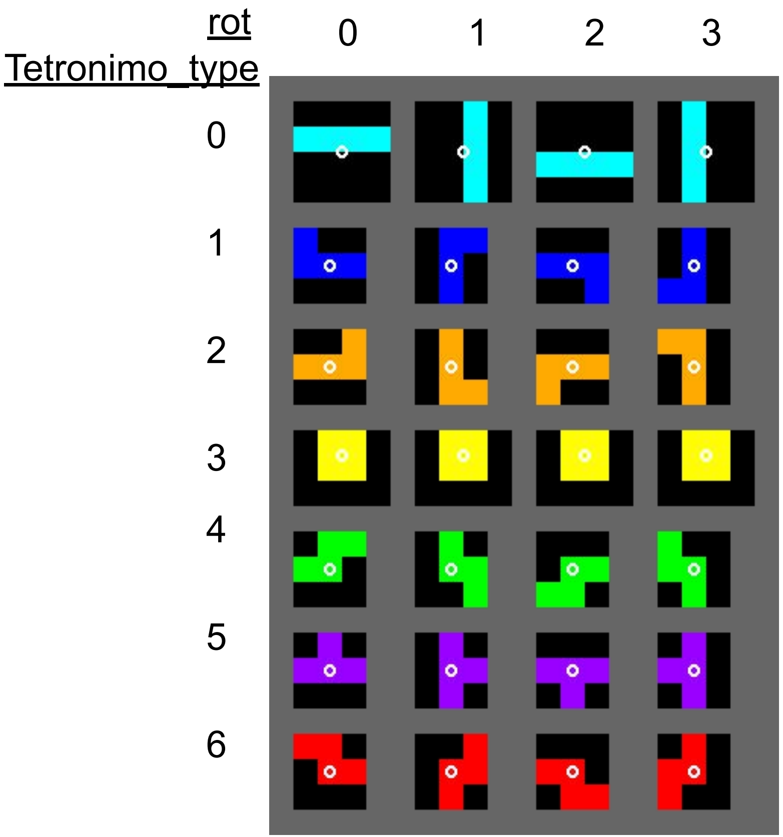
Current tetromino

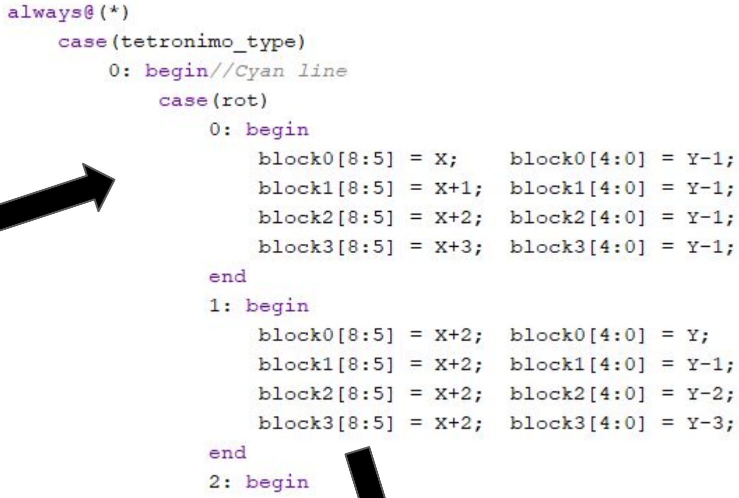
fall_fail

out_of_bounds

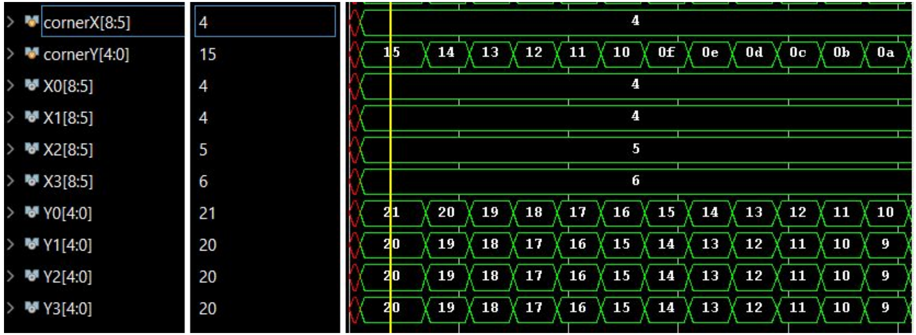
If FB fails to fall, or out of bounds, set appropriate state (set/game over)

Tetromino





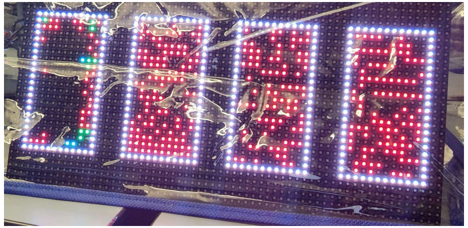
```
always@(*)
    case(tetronimo_type)
        0: begin//Cyan line
            case(rot)
                0: begin
                    block0[8:5] = X;    block0[4:0] = Y-1;
                    block1[8:5] = X+1;  block1[4:0] = Y-1;
                    block2[8:5] = X+2;  block2[4:0] = Y-1;
                    block3[8:5] = X+3;  block3[4:0] = Y-1;
                end
                1: begin
                    block0[8:5] = X+2;  block0[4:0] = Y;
                    block1[8:5] = X+2;  block1[4:0] = Y-1;
                    block2[8:5] = X+2;  block2[4:0] = Y-2;
                    block3[8:5] = X+2;  block3[4:0] = Y-3;
                end
                2: begin
```

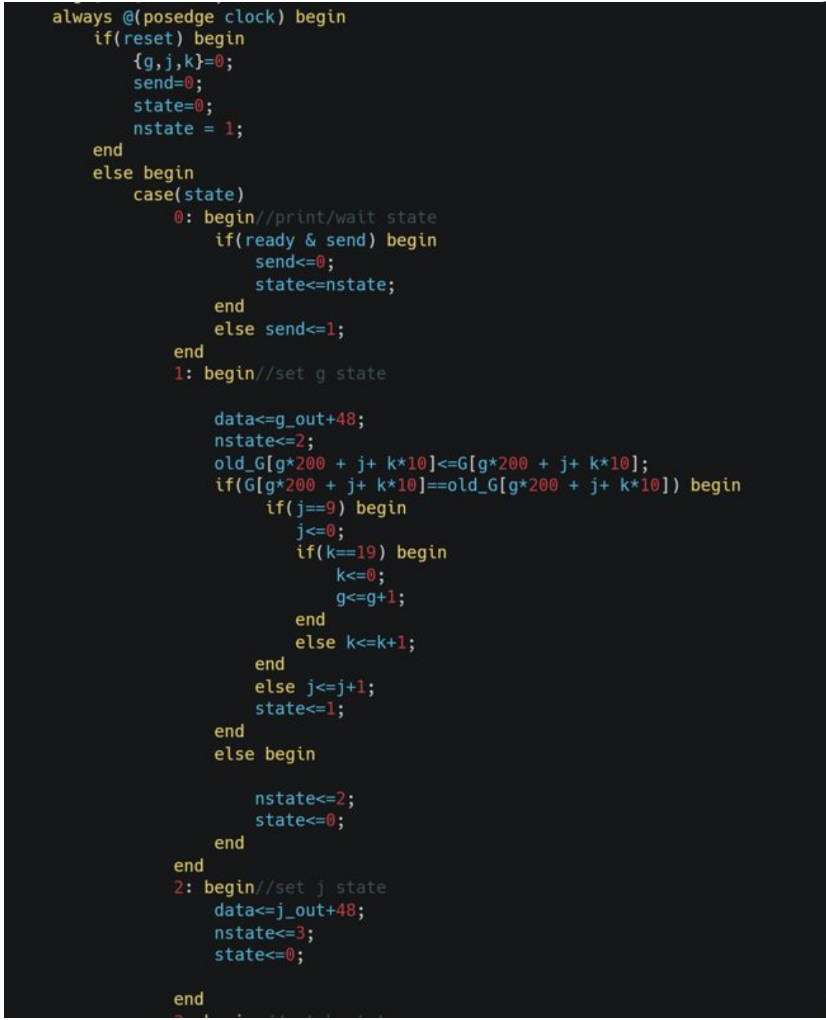


```
cornerX[8:5]  4   4
cornerY[4:0]  15  15  14  13  12  11  10  0f  0e  0d  0c  0b  0a
X0[8:5]       4   4
X1[8:5]       4   4
X2[8:5]       5   5
X3[8:5]       6   6
Y0[4:0]       21  21  20  19  18  17  16  15  14  13  12  11  10
Y1[4:0]       20  20  19  18  17  16  15  14  13  12  11  10  9
Y2[4:0]       20  20  19  18  17  16  15  14  13  12  11  10  9
Y3[4:0]       20  20  19  18  17  16  15  14  13  12  11  10  9
```

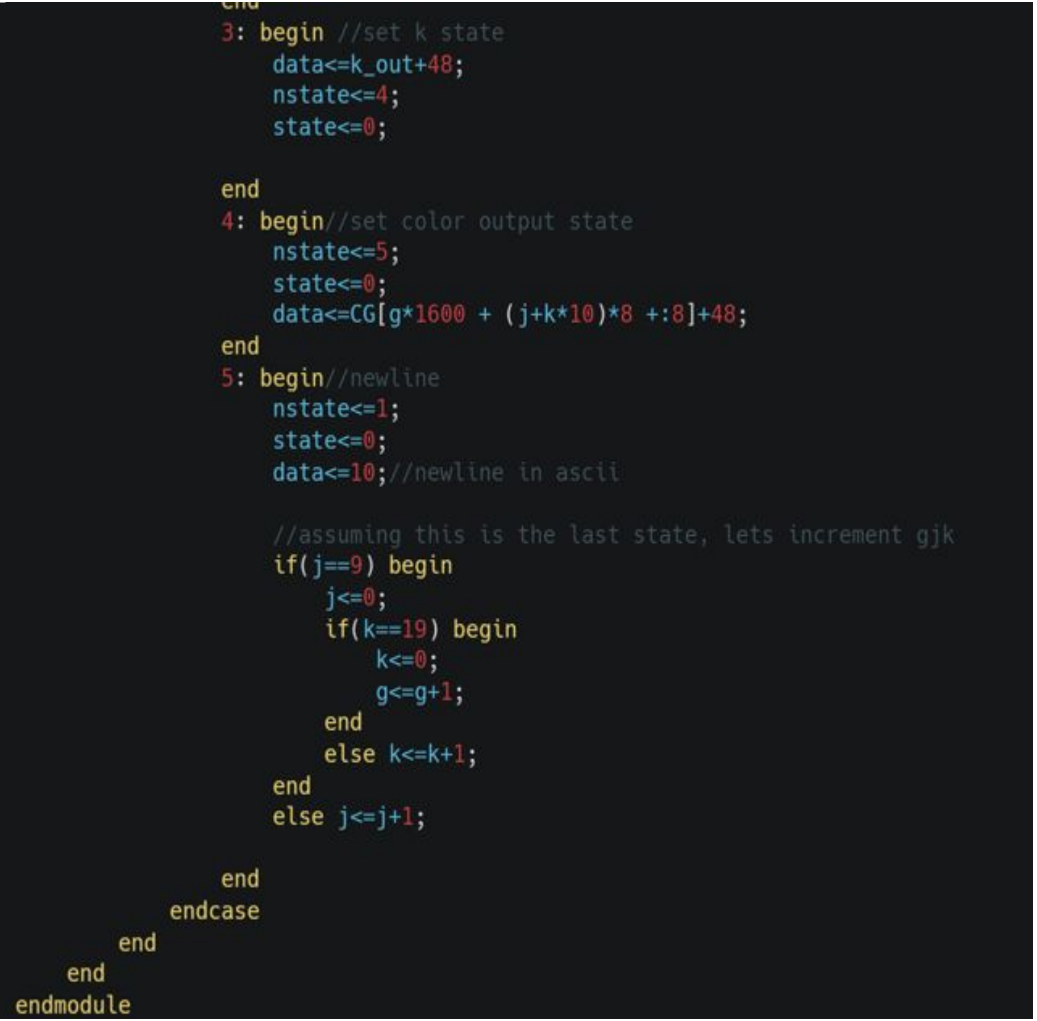
FPGA->UART->Arduino->RGB Matrix

- Arduino
  - Receive 4 bytes via UART, corresponding to game, x,y, and color.
  - Program one pixel at a time
- Graphix_printer module (FPGA)
  - State machine to print out one pixel at a time over UART
  - Original implementation: Send all 800 pixels each loop
    - too much data flooded serial bus
  - “Optimized” algorithm: only send pixels that need to be updated
    - Greatly reduced frequency of print,
    - However, resulted in some “trailing” and unprinted pixels





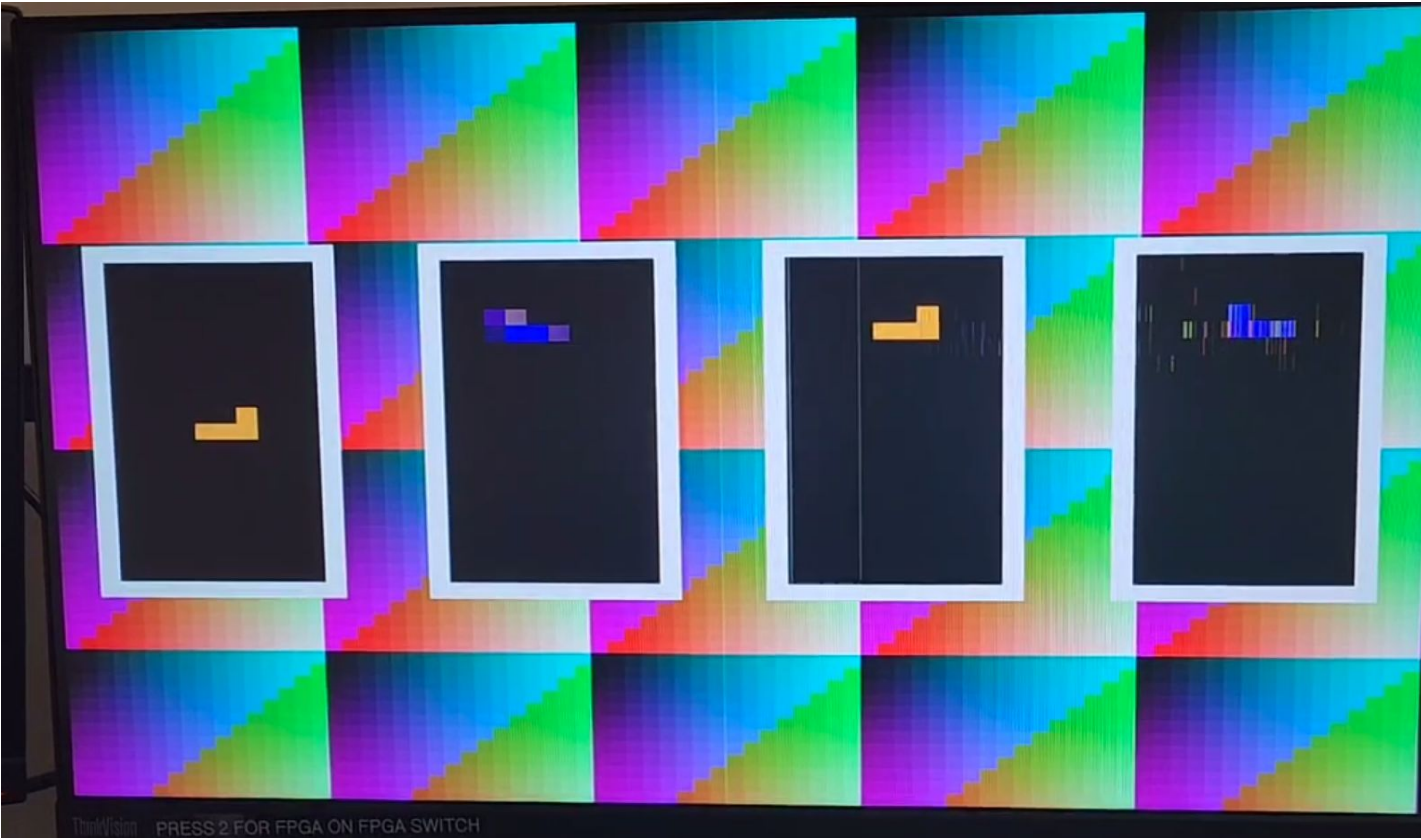
```
always @(posedge clock) begin
    if(reset) begin
        {g,j,k}=0;
        send=0;
        state=0;
        nstate = 1;
    end
    else begin
        case(state)
            0: begin//print/wait state
                if(ready & send) begin
                    send<=0;
                    state<=nstate;
                end
                else send<=1;
            end
            1: begin//set g state

                data<=g_out+48;
                nstate<=2;
                old_G[g*200 + j+ k*10]<=G[g*200 + j+ k*10];
                if(G[g*200 + j+ k*10]==old_G[g*200 + j+ k*10]) begin
                     if(j==9) begin
                        j<=0;
                        if(k==19) begin
                            k<=0;
                            g<=g+1;
                        end
                        else k<=k+1;
                    end
                    else j<=j+1;
                    state<=1;
                end
                else begin

                    nstate<=2;
                    state<=0;
                end
            end
            2: begin//set j state
                data<=j_out+48;
                nstate<=3;
                state<=0;

            end
```



```
                end
                3: begin //set k state
                    data<=k_out+48;
                    nstate<=4;
                    state<=0;

                end
                4: begin//set color output state
                    nstate<=5;
                    state<=0;
                    data<=CG[g*1600 + (j+k*10)*8 +:8]+48;
                end
                5: begin//newline
                    nstate<=1;
                    state<=0;
                    data<=10;//newline in ascii

                    //assuming this is the last state, lets increment gjk
                    if(j==9) begin
                        j<=0;
                        if(k==19) begin
                            k<=0;
                            g<=g+1;
                        end
                        else k<=k+1;
                    end
                    else j<=j+1;

                end
            endcase
        end
    end
endmodule
```

FPGA->VGA

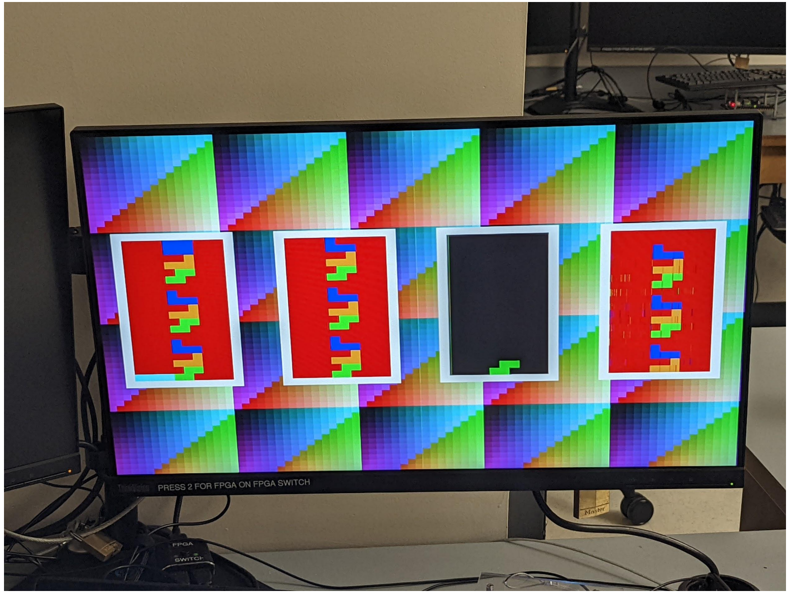
- Much easier



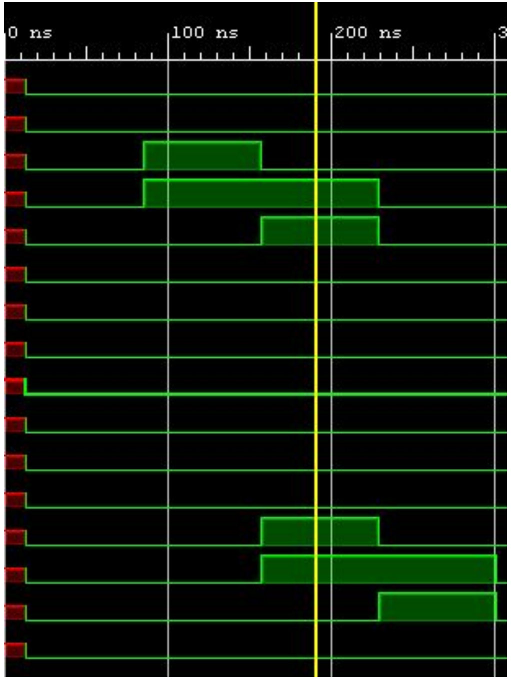
Success

Success

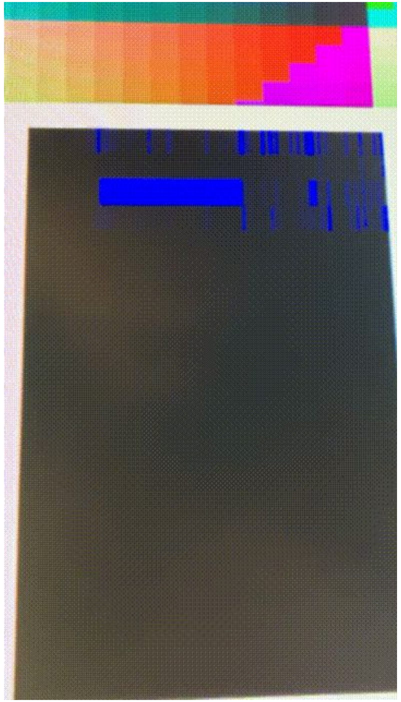
1. Functional basic game logic
    a. Slow fall does not lock block: solved
2. Functional controller
    a. Instant left right error: solved
3. Mostly functional VGA Display
    a. Oscillations (will talk about later)



Failure (Oscillation)

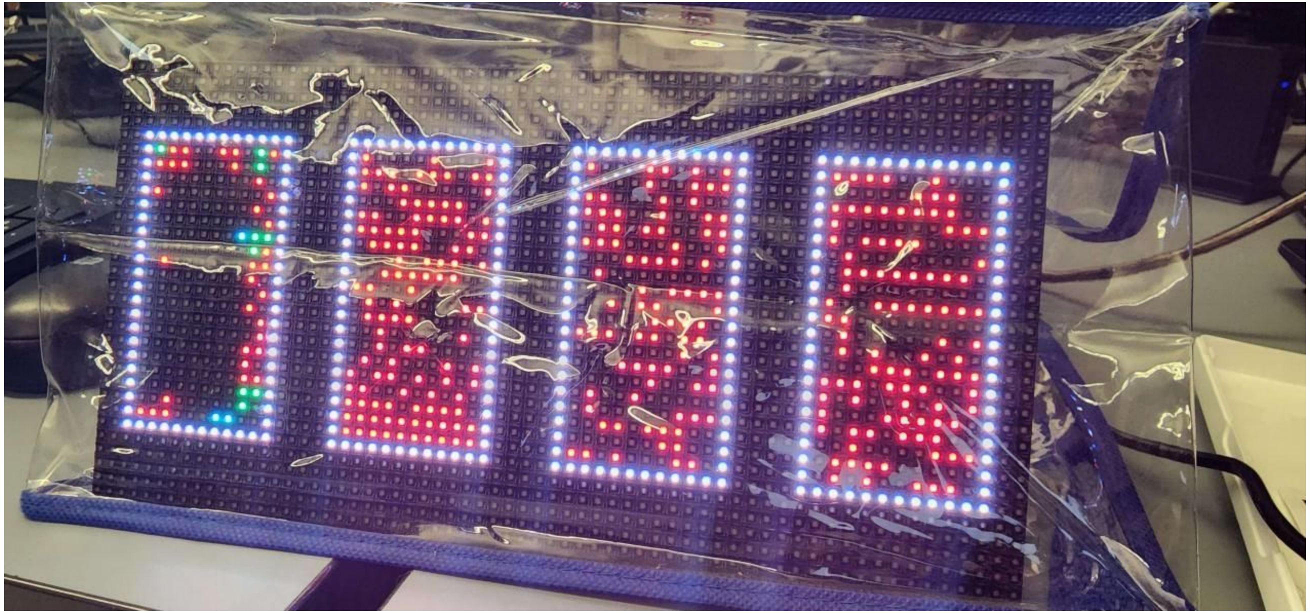


How it’s supposed to be



Oscillation

On the RGB Matrix



Leftmost panel: artifacts left behind the trail
Right panels: supposed to be unplugged -> all should be red but only some pixels are red

Thank You