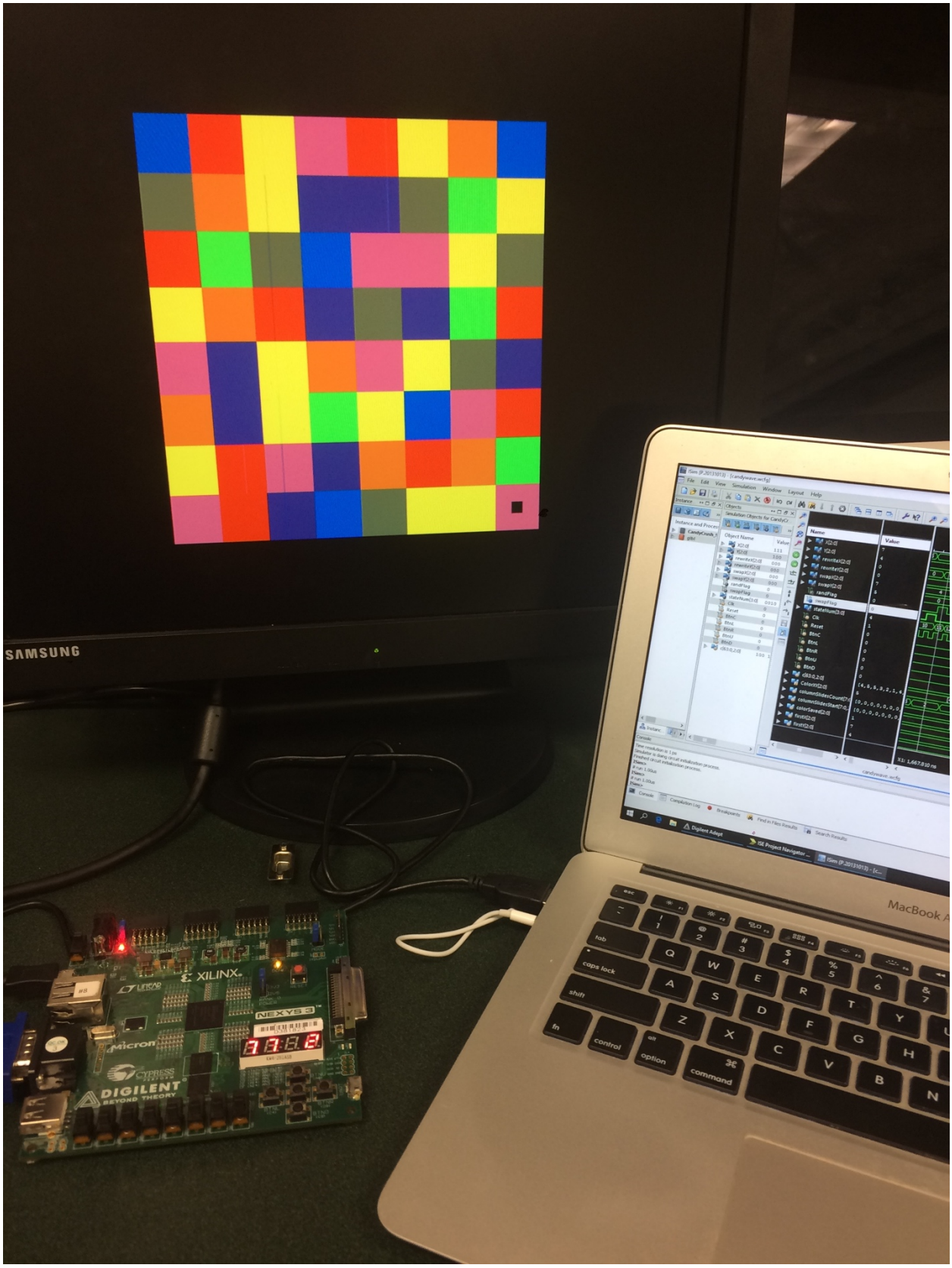
EE354 Final Project
Spring 2017
Candy Crush
By Xin Yu

Abstract:
The final project is inspired by the game of candy crush. It is designed for the user to eliminate 3 or more candies with the same color on a screen with 64 candies of 8 colors. If 3 or more candies with the same color vanish, the candies above the vanished ones will slide down and new candies will come in to fill the places. Unlike the real candy crush game, this candy crush allows the user to swap any two adjacent candies they wish without checking if the swap is qualified for vanishing candies. And it only allows the user to eliminate candies of the color same as the color of the first candy the user pick. And the user can only select the second candy from the four candies around the first candy.

Files:
CandyCrush_Simple.v
CandyCrush_Simple_tb.v
CandyCrushVGA.v
LFSR.v
hvsync_generator.v
ee201_debounce_DPB_SCEN_CCEN_MCEN.v
nexys3.ucf

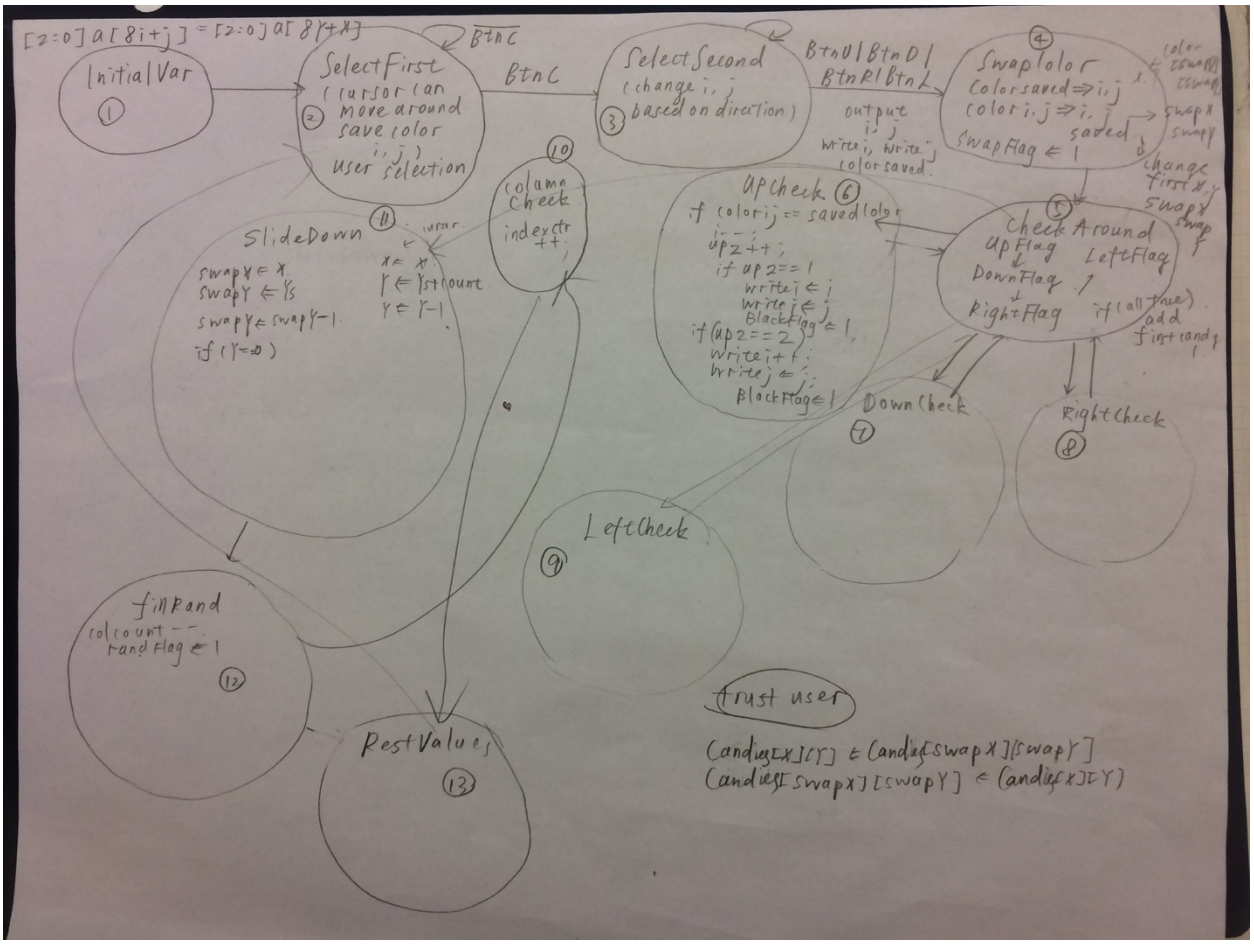
The Design:
In the VGA file, there are three major sections: VGA control, Candy Crush Initialization, and SSD control. The VGA control part refers to the vga_demo.v and uses hvsync_generator.v to generate two of the five vga signals, CounterX and CounterY for assigning the positions of the candies. The five button inputs are debounced by ee201_debounce_DPB_SCEN_CCEN_MCEN.v. Each candy is of the size of 40x40 on a 640x480 screen. The grid of each candy is assigned by allocating a certain value from CounterX and CounterY. Also a cursor is assigned a black grid of size 5x5 to track user move. The 8 colors are defined according to 8 of the different combinations 3Red, 3Green, 2Blue. vga_r[2:0], vga_g[2:0], and vga_b[1:0] are the other three vga signals that corresponds to 3R,3G,2B. Each of the signals takes the control of the respective color of the whole screen, this case, all the candies. So it would be necessary to give the signal the 64 candies and their respective color bits at once. The Candy Crush part instantiates the candy crush state machine, declares and initializes the input variables, and implements swapping or filling in new candies. It uses LFSR to generate random colors (index of the color array). It initializes the 64 candies by the method of counting machine. And it is important to disenable the if block after finishing loading the candies inside the always block, otherwise the last candy would be consistently loaded causing flashing grey color. The SSD part outputs the current X and Y, and also the current state. But when playing the game, the states later than 3 are changing fast and are not seen because the state

transitions do not need user inputs.



(A) Explanation of the state machine:

The state machine diagram is shown as in the following picture. It takes in inputs of clock, reset, five button pulses, the color of the candy at (X,Y) which can be changed by both the user and the state machine, and enable to enable the state machine after the display of candies is ready. And it outputs the two sets of coordinates to for the vga to swap and the coordinates for the vga to fill in new candies, and the flags to indicate when the vga should swap colors or fill in new candies. For testing, it also outputs the state number for displaying the current state on the 7-segment display.



Explanation of each state:

InitialVar:

The state is to set the initial coordinates of the cursor (3,3) for the user to begin the game. This would be user friendly since the cursor begins in the middle of the screen for the user to move around. And it sets the variables used in the state machine to zero, such as coordinates of the first candy picked by the user, coordinates of the two candies to be swapped, and coordinates of the new candies. The state also initializes the two arrays that save the number of candies to slide down and from which position to slide down to zero.

SelectFirst:

In this state, the user can move around by pressing up/down/right/left button and then confirm the selection of the first candy by pressing the center button. Then the coordinates of the first candy are saved as well as the color for future use. When the user presses the center button, it transitions to SelectSecond.

SelectSecond:
The state takes in one button press and changes X and Y accordingly to select the second candy. It then saves the coordinates of the first candy into swapX and swapY to prepare for the swap between the first candy and the second candy [(swapX,swapY) with (X,Y)]. And it also sets the swapFlag to high when the two sets of coordinates are ready. And the state transitions to swapColor when the user has selected the second candy by pressing any to the four buttons (up/down/right/left).

SwapColor:
The swapFlag is set to zero after swap is done. Since the swap is finished, the coordinates of the first candy should be updated. And the state transitions to CheckAround unconditionally.

CheckAround:
The state serves as a command center for the four states: UpCheck, DownCheck, RightCheck, LeftCheck. It makes sure that every direction of the first candy is checked by setting flags of four directions and change X and Y accordingly. And the state transitions to any of the four states if the respective flag is not set and to ColumnCheck if the four states have been visited and saves the Y-coordinate of the first candy if there are no 3 or more candies with the same color in the down direction but in other directions.

UpCheck:
Checks if the candy at (X,Y) has the same color as the first candy. If it does, increment the count of the candies to slide down/fill in in the array according to the its column number. And saves the Y-coordinate if it is lower than the already saved one. And then goes up to check the upper candies. If it does not, then reset X and Y to the coordinates of the first candy and go back to CheckAround.

DownCheck:
Checks if the candy at (X,Y) has the same color as the first candy. If it does, increment the count of the candies to slide down/fill in in the array according to the its column number. And saves the Y-coordinate. And then goes down to check the lower candies. If it does not, then reset X and Y to the coordinates of the first candy and go back to CheckAround.

RightCheck:
Checks if the candy at (X,Y) has the same color as the first candy. If it does, increment the count of the candies to slide down/fill in in the array according to the its column number. And

saves the Y-coordinate. And then goes right to check the candies at right. If it does not, then reset X and Y to the coordinates of the first candy and go back to CheckAround.

LeftCheck:
Checks if the candy at (X,Y) has the same color as the first candy. If it does, increment the count of the candies to slide down/fill in in the array according to the its column number. And saves the Y-coordinate. And then goes left to check the candies at left. If it does not, then reset X and Y to the coordinates of the first candy and go back to CheckAround.

ColumnCheck:
Check each column. If the value in the respective array element is larger than zero, then the column has to slide down and has random candies coming in. The state transitions to SlideDown. The way to slide down is by swapping the candies starting with the lowest candy to be eliminated and the candy just above the highest candy to be eliminated. Use the values stored in the two arrays to implement this, and outputs swapX and swapY, X and Y for swap in vga. If all the columns have been checked, the state transitions to ResetValues.

SlideDown:
Perform repetitive swaps for the column until all the candies to be eliminated move to the top positions and prepare for FillRand when the swaps are finished by starting a counter.

FillRand:
Increment the counter to indicate the places where new candies should come and set the randFlag to high for vga. And the state transitions to ColumnCheck when it finishes filling the new candies.

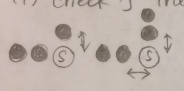
RestValues:
Reset the output variables and intermediate variables to zero to get ready for next user selection. And set X and Y to the last user selection and then goes back to SelectFirst.

(B) Challenges

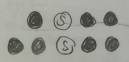
I first tried to implement the state machine according to the rules of the actual candy crush game. And I found out it requires much work and time to design the state machine since the real candy crush game can eliminate candies of colors of the first candy and the second candy user pick. And it does not allow user to swap if there are not any patterns for potential elimination. I wrote down the state machine diagram and I found out I can not finish the pattern checking for the two candies when I reach 700 lines of code. So I decided to cut the features and make sure I have a working project. After having discussion with TA, I was able to change some rules of the game, debug and finish the project. And my lab partner was always “sick” during regular lab sessions.

Candy Crush State Machine:

1. 6 colors: blue, green, yellow, red, orange, purple (probably different shapes)
2. three or more candies of the same colors get eliminated
3. the whole screen slides down, randomly generated new candies to fill the empty spots after every elimination
4. user can swap two candies for potential elimination, only if rule 2 can be met. change the screen, otherwise don't do anything.
   (i) check if the four candies around (primitive ones)



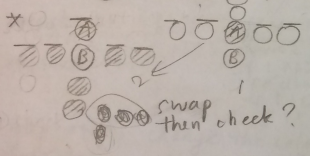
2 adjacent directions occupied by 1 or 2 candies with same color



2 cross directions occupied by 1 or 2 candies with same color

3 of the 4 directions occupied by only 1 candy with same color

the 3 situations combined

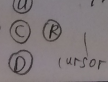


swap then check?

criteria to apply swap
(i) A, B different colors
(ii) both A, B should go through the combination checks

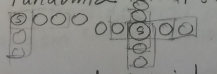
① check A, B different color
② check if A, B satisfy * (user)
else go back wait for another A, B

~~if A, B are adjacent~~



5. after effective swap, followed by immediate elimination
   3, 4, 5 candies with same color to be eliminated
   save the column number → columns to drop down
   row number → rows above to drop down

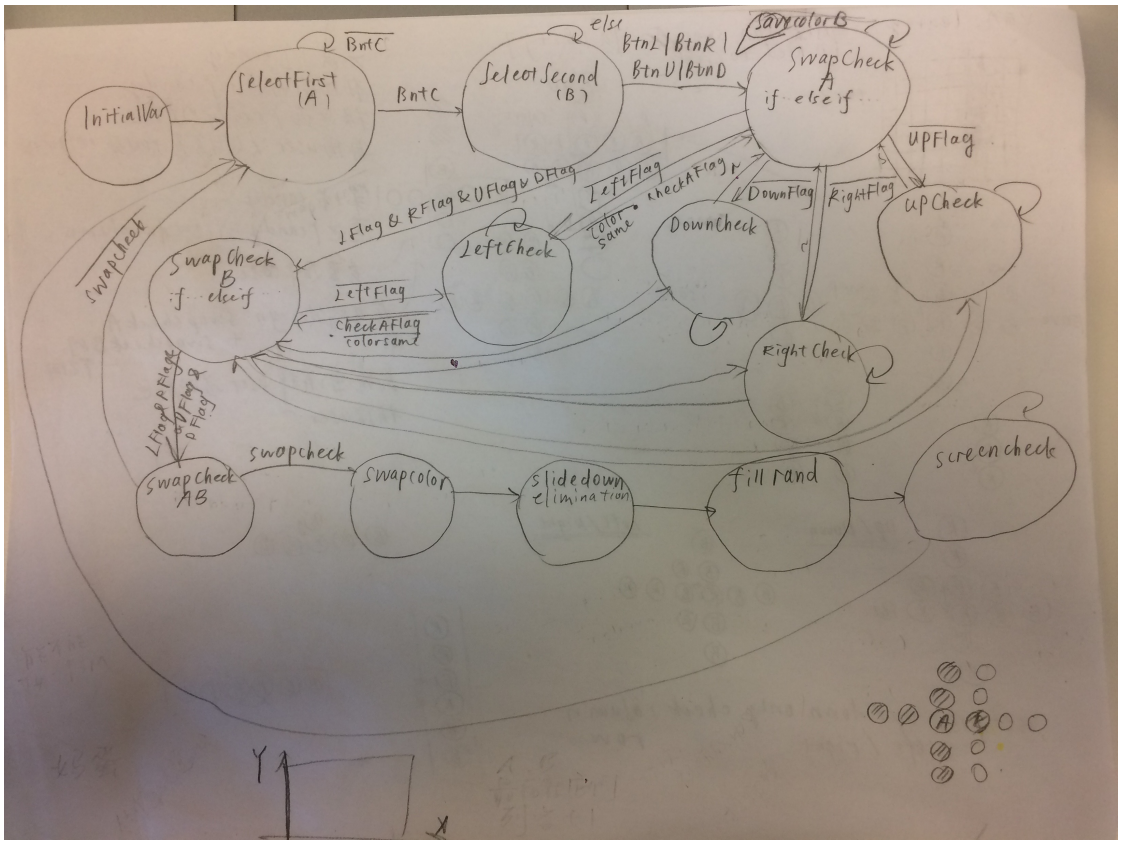
6. randomization of candies to be filled in
   initial screen setup
   how to implement "random" in state machine? → mathematical equations?
   1° randomize initial screen, do an immediate screen check.



check 2 direction enough?

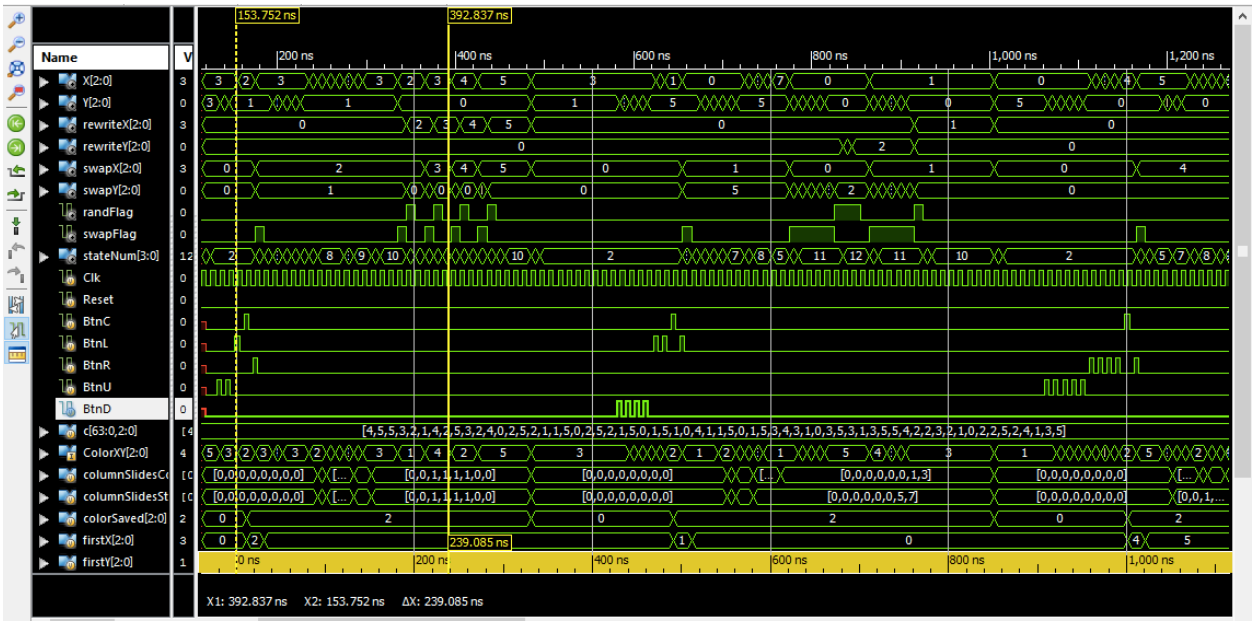
   2° hardcode initial screen? → play same game?
   3° another state machine for randomization?
   given number of candies (row, column)

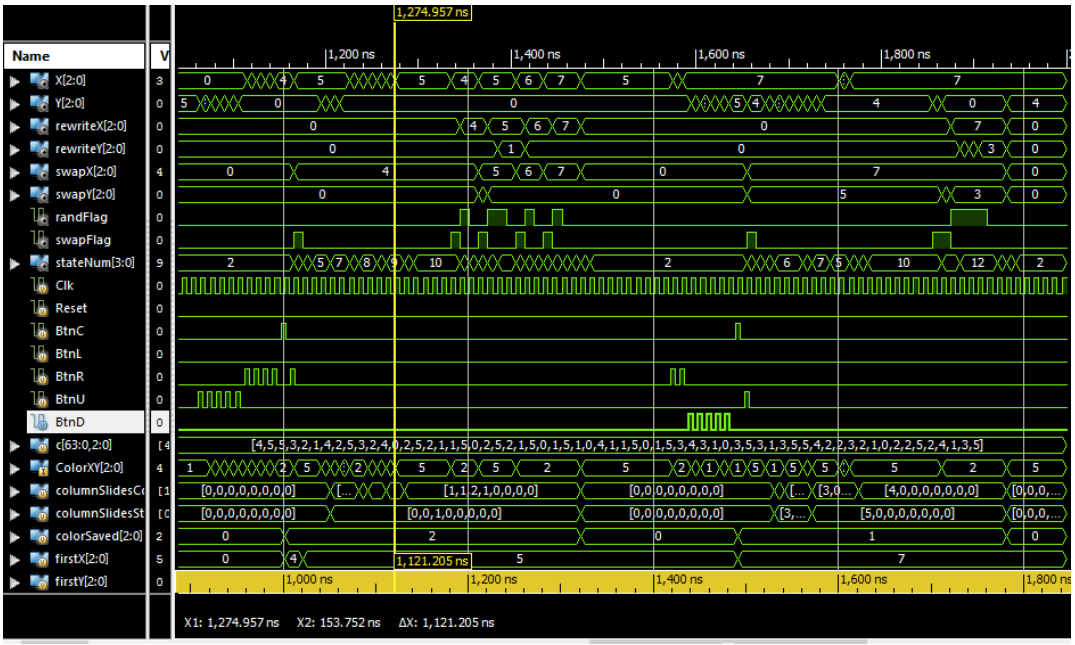
7. after steps elimination, the new generated screen can have sequences of candies to be eliminated.



Test Methodology:

To test the state machine, it is important to have different test cases including corner cases. In this case, I tested the state machine with four cases. Test 1: yellow purple yellow yellow at row 2 from column 2 to 4; Test 2: yellow yellow at column 0 from row 6 to 7 and yellow at column 1 row 5; Test 3: yellow orange yellow yellow at row 0 from column 4 to 7; Test 4: red red orange red at column 7 from row 2 to 5; I was able to modify my state machine based on waveform. When I tested on VGA, I found that the cases of test 3 and 4 will not work. So I added two tests, and found the mistakes in the state machine (typed <0 instead of >0, missing condition for candies at row 0).





To test the VGA, I found it is important to look at the warnings when generating the bit file. The color signals were not working because I forgot to uncomment and modify the respective lines in the ucf file. By looking at the warnings of unconnected signals, truncations or optimizations, I was able to find the places where I typed wrong. The button debouncers were not working because I used wrong combination of the file and N_dc. And then I found reference from one of the labs.

Conclusion and Future Work:

The current state machine relies on trusting user for some cases. Because of the limited time for the final project, I was not able to implement the final check for qualified patterns for elimination. So the case where there is only one candy with the same color with first candy, the 2 candies will vanish as well. The cases like yellow red yellow would eliminate the two yellow candies after swap. For future, I should try to implement the real candy crush game using the more complicated state machine and do the final check. And I can change the number of candies by making the state machine modular. For state machine, I could have passed in the 2d array of candies so the candies can be swapped during the state. And it would be easier to debug on the waveform. If the 2d array can be not be passed in the modules, I can transform it into 1d. I was more comfortable with 2d array since it is easier to write code for both vga and state machine.