CANDY CRUSH

EE354 FINAL PROJECT

MACKENZIE COLLINS, DANNY PAN

Table of Contents

Project Proposal

We dedicated to pay homage to a popular application back during our high school day called Candy Crush. We're going to implement a simplified version of it on an 8x8 Grid of color blocks displayed on a VGA display. The modified version will allow users to selected a block and swap it with other next to it. However, the color will be sampled only from the original selected block and compared to the North, South, East, and West blocks. As long as much as matches occurs, the colors will be consumed until a different color is reached. The game board will then shift all remaining colors down to close the gaps created by consuming color blocks and repopulate the empty areas with new random blocks.

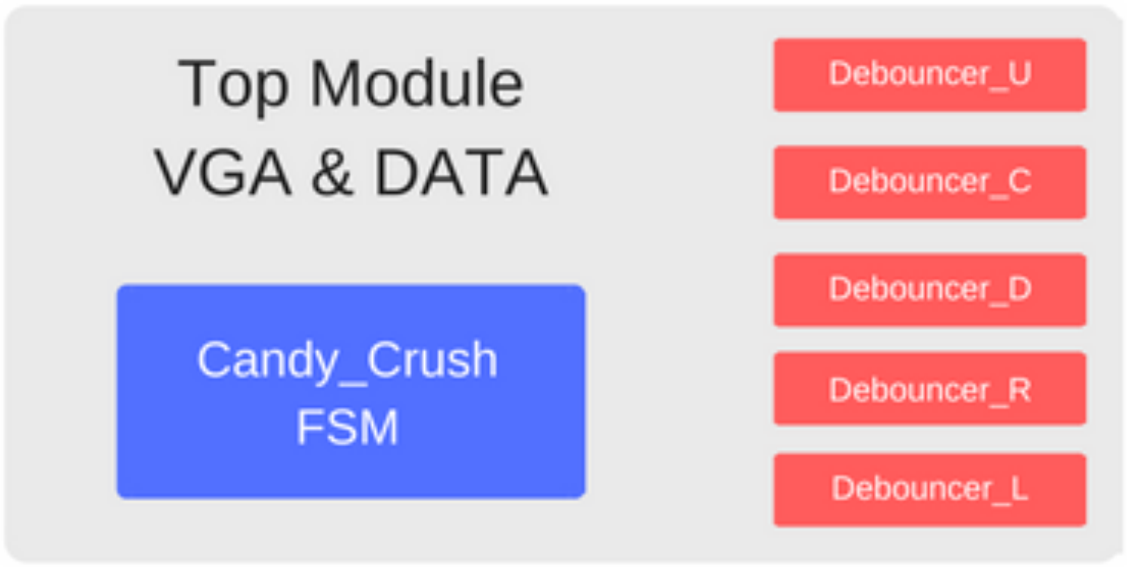
The original version of Candy Crush requires a 3 in a row scenario to initiate block consumptions, but we'll be using 2 to simply the project in order to make the workload reasonable. The game flow is conducted using the button on the Nexsys 3 board, with a VGA driver providing the visual representation on the monitor.

Verilog Code Design

The top module is derived from the VGA demo code given to us from the lab resources. We made this design decision due to time constraints, so we decided to build off of existing code even though the functionalities aren't 100 percent built for our program. Therefore, small modifications would only needed to prepare the top module for use.

The top module houses the data variables for the 8x8 grid along with any writing operations done onto the grid. This is accomplished by a synchronous always block checking for certain output flags by the Candy Crush FSM to signal for specifics memory writes or swaps at specified locations given the Candy Crush FSM as output variables. In addition, 5 de-bouncing modules are used produce pulses for each of the five buttons the Nexsys 3. All initialization of game variables for board along with display variables are conducted by always blocks with flags to check for completion.

Housed within the Top Module is a Candy Crush FSM that conducts all the game logic. All cursor movement and the selection of blocks for swaps and alterations are outputted as variables from the Candy Crush FSM. As noted above, the Candy Crush FSM will send flags to signal the top module to conduct an alteration of the 8x8 grid based on the flag.



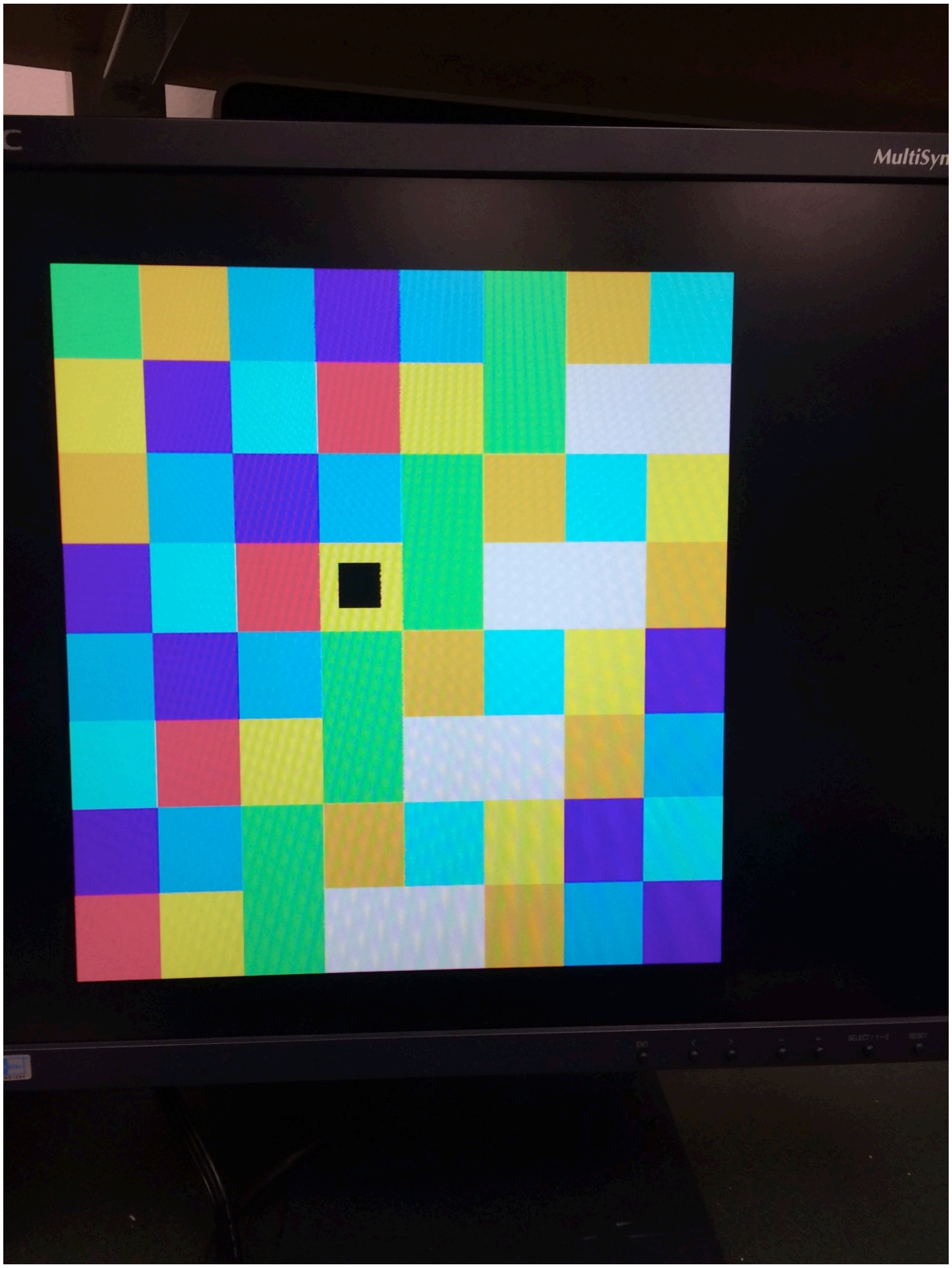
VGA Design

The files that we control the VGA display are *hsync_generator.v, nexys3.ucf*, and, *CandyCrush_VGA.v*. The first file synchronizes the vertical and horizontal positions on the display, defines the display area to be 480x640, and generates two counters to keep track of the x and y positions. This file was largely unchanged from the demo file provided to us.

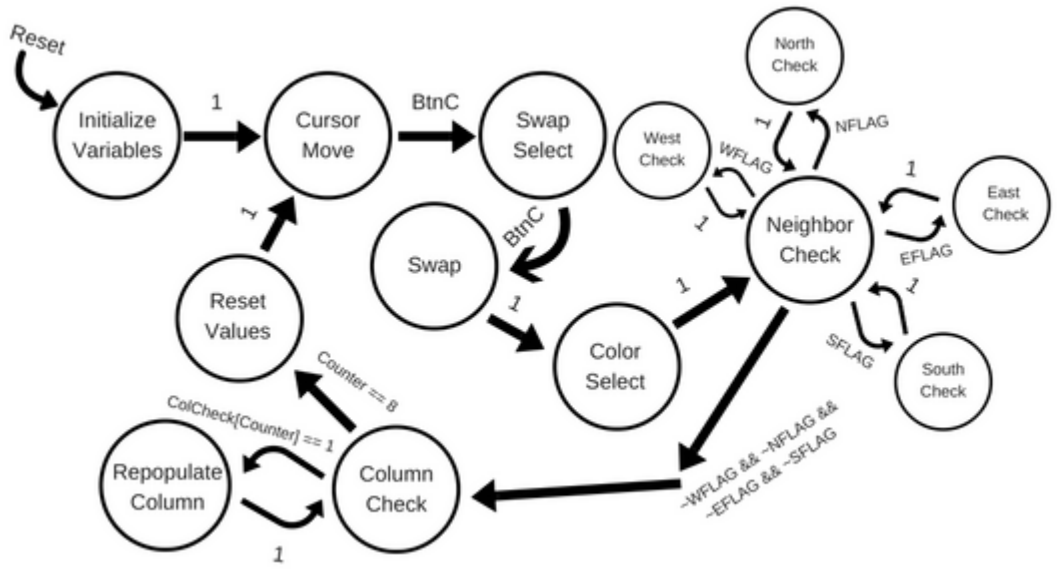
The UCF file contains the interface specifications and timing constraints. To increase the number of different colors we were able to display, we initialized all eight pins on the VGA port; 3 pins for red and green, and 2 pins for blue. This allowed us to display up to 265 different 8-bit colors.

Lastly, *CandyCrush_VGA.v* contains the actual logic we used to display a randomly generated 8x8 grid of 8 colors. We chose 50x50 for the size of a square. From that, we stored the x and y intervals corresponding to each square in two arrays (one for x, one for y). That way, we could just use Boolean arguments to designate which square(s) we wanted the red/green/blue pins to signal. From this point, we struggled in figuring out the most efficient way to create the grid. Ideally, we could loop through the array of 64 random numbers between 1 and 8, assign each number to a color, and build the grid that way. Our issue was that we wanted to display all the colors at the same time, which requires one statement at the end of the logic to set each pin. The looping algorithm would reset the pins after each square was displayed, erasing the previous square. Ultimately, we had to simply hard-code the 64 square dimensions into each color pin. We used the randomly generated Grid[][] in Boolean expressions within these hard-coded pin statements to turn on/off the bits corresponding to the color we wanted to display.

While our method to display the grid was not the most code-efficient, we successfully created a random grid of 8 different colors.


MultiSyn

State Machine Design



Our Candy Crush FSM contains a 13 state mealy machine to conduct all the game logic. Above is a state diagram of the transitions. Since the actual internal logic in each state is complex. I'll provide a description of each state with its logic and transition conditions.

Initialize Variables

Transitions From:
- Any State when reset is high

Transition To:
- Unconditional transition to Cursor Move State

Description: The initial state that sets all the variables values to 0 and the cursor to (3,3) on the grid.

Cursor Move

Transitions From:

- Initialize Variables

Transition To:

- Swap Select when Button C is pressed

Description: The cursor move state allows the user to move their cursor around the board to select a block they want to swap from. Once Button C is pressed, the coordinates are saved in order to reference the original square and it transitions to the swap select.

Swap Select

Transitions From:

- Cursor Move

Transition To:

- Swap when Button C is pressed

Description: The user can now choose an adjacent square from the block that was selected from the Cursor Move state. The choices are limited to the North, East, South, and West blocks that are directly adjacent to the original selected block. If the user does not move the block in the Swap Select state, they can select to not swap the block by just pressing Button C. Once the Button C is pressed, the program sets the variables and signals the swap flag to initiate the swap.

Swap

Transitions From:

- Swap Select

Transition To:

- Color Selected Unconditionally

Description: The swap flag is reset and the coordinates of X and Y is set to the original values from the Cursor Select. This allows sampling of the color value inside the original value for the color Select state. The program then unconditionally moves to Color Select.

<u>Color Select</u>

Transitions From:
- Swap

Transition To:
- Color Selected Unconditionally

Description: The swap flag is reset and the coordinates of X and Y is set to the original values from the Cursor Select. This allows sampling of the color value inside the original value for the color Select state. The program then unconditionally moves to Color Select.

<u>Neighbor Check</u>

Transitions From:
- Color Selected
- North Check
- West Check
- South Check
- East Check

Transition To:
- North Check if North Flag = 0
- West Check if West Flag = 0
- South Check if South Flag = 0
- East Check if East Flag = 0
- Column Check if all directional flags are high

Description: The program checks a series of flags to see if the all four directions have been checked. If any flag is equal to 0, we set the flag to one and see if it possible to check that directions (within the grid). Then we transition into that directional check state that sees if there are block that are adjacent with the same color as the origin. The directional check states will eventually return to the Neighbor Check again. We continue to do this until all directions have been checked and all flags are set to high. Finally, we check to see if there was an adjacent block that was the same color as the origin by using a consume origin flag that is set within those directional check states. If it is high, we consume the center block and transition to the Column check state to start the repopulation.

West Check

Transitions From:

- Neighbor Check

Transition To:

- Neighbor Check if the color at coordinates != selected color

Description: The program walks down to West of the selected square block. If the color at the selected blocked is the same as the selected color set in the previous state, then the program sets the color middle flag, marks the column as changed using the column check array, writes that block as the color black, and continues West. When we reach a block that is not the same color the program transition back to neighbor check.

North Check

Transitions From:

- Neighbor Check

Transition To:

- Neighbor Check if the color at coordinates != selected color

Description: The program walks up to North of the selected square block. If the color at the selected blocked is the same as the selected color set in the previous state, then the program sets the color middle flag, marks the column as changed using the column check array, writes that block as the color black, and continues North. When we reach a block that is not the same color the program transition back to neighbor check.

South Check

Transitions From:

- Neighbor Check

Transition To:

- Neighbor Check if the color at coordinates != selected color

Description: The program walks down to South of the selected square block. If the color at the selected blocked is the same as the selected color set in the previous state, then the program sets the color middle flag, marks the column as changed using the column check array, writes that block as the color black, and continues South. When we reach a block that is not the same color the program transition back to neighbor check.

East Check

Transitions From:

- Neighbor Check

Transition To:

- Neighbor Check if the color at coordinates != selected color

Description: The program walks down to East of the selected square block. If the color at the selected blocked is the same as the selected color set in the previous state, then the program sets the color middle flag, marks the column as changed using the column check array, writes that block as the color black, and continues east. When we reach a block that is not the same color the program transition back to neighbor check.

Column Check

Transitions From:

- Neighbor Check
- Repopulate Column

Transition To:

- Reset Values if Counter == 8
- Transitions to Repopulate Column if column has been changed by previous states

Description: We iterate through all the columns and using a column check array that was set in the previous color consumption states, we see if the column has been altered by the previous states. If it is altered we transition to the repopulate column state, which will move the colors down and repopulate the grid with new colors. The state then transitions to Reset Values once the counter is 8, which all columns have been checked.

Repopulate Column

Transitions From:
- Repopulate Column

Transition To:
- Repopulate Column once all colors are filled

Description: I'm going to break apart this state's functionalities into two paragraphs so I can organize it into the reorganization and finally the repopulation phase.

The state begins by starting at the very bottom block and checking to see if its color is black. If not, it continues upwards until it finds the first black block. Then it marks the black block and sets the repopulate flag. The program continues walking up until it finds a block that is not black. We then start swapping the first marked black block with that first colored block. This continues until the cursor gets to the top of the grid.

Once the cursor reaches Y = 0. It is then set to where the next black block is after all the swaps. Then we mark a market a flag that writes a random value into the block and we continue upwards until the whole column is repopulated. Then the state transitions back to the Repopulate Column state. At the beginning of the repopulate column state, all flags are reset.

Finally there's an edge case where the only block to repopulated is the top block, which is handled with an if statement that checks to see if the variable used to market blocks that are black is unchanged.

Reset Values

Transitions From:
- Repopulate Column

Transition To:
- Cursor Move unconditionally

Description: All values that were used in the previous operations are reset. The Cursor is reset to the original selected values and the state transitions to the cursor select state, which allows us to restart the whole process.