2048 Game Project

Anish Prakriya

Introduction and Requirements

In this design project, our team will create a version of the popular game 2048 using the Nexys3 Spartan-6 FPGA Board and connect the board to a VGA output to display the game.

To implement the game, we will use four push buttons on the FPGA board for user input: right, left, up, and down. These buttons will control the movement of the numbered tiles on the game board. For example, if the player selects the "left" button, all of the tiles on the board will slide to the left and combine if necessary. Additionally, there will be a "reset" button which can be used to start a new game with a randomly generated grid containing only one "2" and one "4" tile. The reset button can be used at any point during the game.

To render the game board, we will use the VGA display controller to write each pixel's red, green, and blue values to create the image of a 3x3 grid on the screen. We will be using patterns of pixels with specific colors to create numbers inside each tile of the grid. The game will end when there are no more possible moves left, at which point the player loses and the screen turns red.

In addition to our master clock, we will use three additional clocks to control different aspects of the game, including handling user input buttons, updating the grid, and displaying to VGA.

To ensure the smooth operation of the game, we will be using debouncers to remove the bouncing effect of buttons on the circuit and to control the necessary speed of the user's clicks. This will help ensure the game responds accurately and reliably to user input.

Design Description

We modularized our design into ten different modules, each with its own various role.

Our first module, 'clk_n,' is a clock divider that generates a clock signal with a frequency that is a fraction of the input clock frequency. The "clk_n" module takes an input clock signal "clk" and outputs a clock signal "clk_out" with a frequency determined by the "freq" parameter. The "max_count" variable is calculated based on the desired frequency, and "count" is incremented on each clock cycle until it reaches "max_count - 1", at which point the "clk_out" signal is toggled, and "count" is reset to 0.

Our next module interacts directly with the previous module. The clk_gen module generates three output clocks, namely i_clk, d_clk, and u_clk, with frequencies of 100 Hz, 25 MHz, and 5 Hz, respectively. These frequencies are determined by the instantiation of three clk_n modules in the clk_gen module, where each clk_n module generates a clock signal with a specific

frequency. The c100 instance generates a clock signal with a frequency of 100 Hz, the c25M instance generates a clock signal with a frequency of 25 MHz, and the c5 instance generates a clock signal with a frequency of 5 Hz.

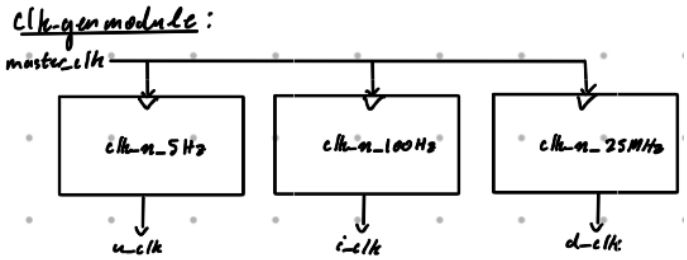

Figure 1: clk_gen module schematic

Our next two modules also work together: "btn_debouncer" and "debouncer." "btn_debouncer" is a module that takes an input clock signal "clk" and a button input signal "btn", and outputs a debounced signal "out." Debouncing is the process of filtering out multiple signals generated by a single button press due to mechanical vibrations or bouncing. The module uses a shift register with two elements to filter out any bouncing, and the final output signal is the logical AND of the two elements of the shift register. The "debouncer" module takes five button input signals "btnRST", "btnR", "btnL", "btnU", "btnD", and the clock signal "clk" as inputs. It also outputs five debounced signals "rst", "r", "l", "u", and "d". The debounced signals are generated using instances of the "btn_debouncer" module for each button input, where the debounced output of each instance is connected to the corresponding output port of the "debouncer" module.

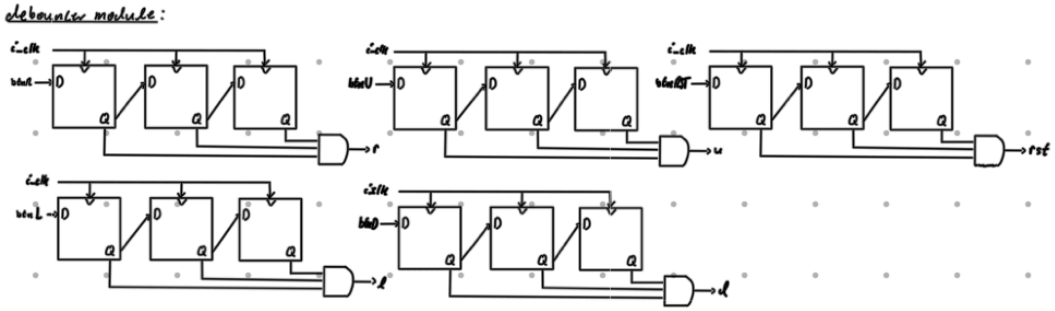

Figure 2: Debouncer module schematic

The next module, “game_state_update” is quite important, and represents the game state update logic. It takes input signals clk, rst, r, l, u, and d and outputs grid and lose. The grid output is a 27-bit vector that represents the current state of the game board, where each 3-bit segment represents the value in one of the nine cells in the 3x3 grid (0 corresponds to 0, 1 corresponds to 2^1, 2 corresponds to 2^2, and so on). The lose output is a single-bit wire that is set to 1 when the game is over (i.e., the player has lost).

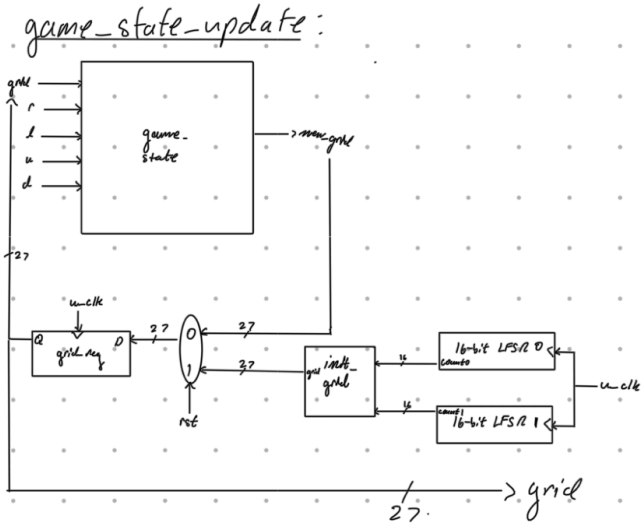

Figure 3: Game state update module schematic

The module uses two linear feedback shift registers (LFSRs) to generate two pseudo-random numbers, rand_num0 and rand_num1. These numbers are used to place new game pieces on the board in a random fashion. The module also includes an instantiation of the game_state module to update the grid, and an instantiation of the “lose_check” module to calculate if the player has lost. Later, I will talk about these in more depth. Finally, there are two always blocks in this module. The first always block updates the LFSRs (count0 and count1) on each positive clock edge. The second always block updates the state of the game board (grid) on each positive clock edge if any of the input signals r, l, u, or d is high. If rst is high, the game board is reset to its initial state with two new pieces randomly placed on the board.

Now, let’s talk about the “lose_check” module in more detail. It takes as input a 27-bit grid (9 blocks each consisting of 3 bits) and outputs a single bit lose. The module checks if the game is over (i.e., the player has lost) by first checking if there are any empty blocks and then checking if there are any adjacent blocks with the same value.

The “game_state” module takes as input a 27-bit grid, a 16-bit random number counter, and four control bits (r, l, u, d) that specify the direction of the move. It outputs a new 27-bit grid and a 4-bit count of the number of empty blocks in the new grid. The module deals with the movement of the grid by instantiating “extract”, “inverter_3”, “row_ops_3”, and “cols_to_rows” modules.

Here's a brief description of each of those modules:

- extract: This module extracts the values of tiles from a given grid array and assigns them to three separate t0, t1, and t2 variables. If rc is 1, the values are extracted from the rows of the grid, otherwise, they are extracted from the columns.
- inverter_3: This module takes three rows of tiles and an inv bit as input and outputs three inverted rows if inv is 1 and simply returns the input rows if inv is 0. It instantiates three inverter modules and connects them to the inputs and outputs.

- row_ops_3: applies the row_ops module to each row of a 3x3 grid of values. The input registers in0, in1, and in2 contain the rows to be operated on, and the corresponding output registers out0, out1, and out2 contain the results of the operation.
- cols_to_rows: This module converts a given set of three columns into three rows. If rc is 1, it assumes that the input columns are already in row form and simply assigns them to the output rows. If rc is 0, it performs the actual conversion.

First, “game_state” extracts the blocks from the specified row or column using the extract module, then performs an inversion operation on the extracted blocks using the inverter_3 module, then performs a row operation on the inverted blocks using the row_ops_3 module, and then inverts the result again using inverter_3. Finally, the module transposes the result back into the grid using the cols_to_rows module. If any movement occurs (i.e., any control bit is high), the new grid is output, otherwise, the original grid is output.

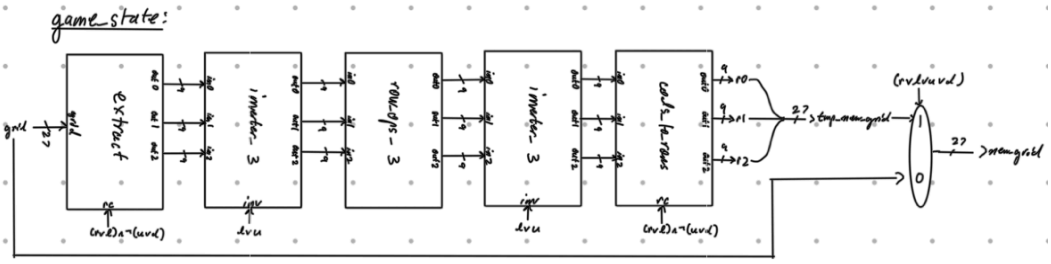

Figure 4: Game state module schematic

The final submodule deals with our UI creation. It is our VGA controller module. It takes a 27-bit input signal called "grid", which is used to control what is displayed on the screen. There are also input signals for a pixel clock ("d_clk"), an asynchronous reset ("clr"), and a signal called "lose". The output signals are the horizontal sync ("hsync"), vertical sync ("vsync"), and three color signals for red, green, and blue, each with a different number of bits. The code uses nested for loops to iterate over a 3x3 grid of tiles on the screen, and for each tile, it uses a case statement to determine which digits to display based on the value of the input signal "grid". The digits are then displayed on the screen using a series of if statements, which set the appropriate color values for each pixel in the digit. The color values are stored in registers called "red", "green", and "blue", and are updated on each clock cycle. If the “lose” input is 1, then the background of the screen turns red indicating that the player has lost the game.

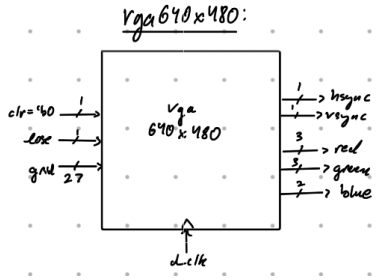

Figure 5: VGA hardware implementation

The main module, named “five12.v” instantiates all of the necessary submodules. Its input ports are "clk" (clock), "btnRST" (reset button), "btnR" (right button), "btnL" (left button), "btnU" (up button), and "btnD" (down button). The output ports are "hsync" (horizontal synchronization), "vsync" (vertical synchronization), "red" (red color value), "green" (green color value), and "blue" (blue color value). The module instantiates three other modules: "clk_gen", "debouncer", and "game_state_update". Finally, the module instantiates a "vga640x480" module, which generates the VGA output based on the current game state.

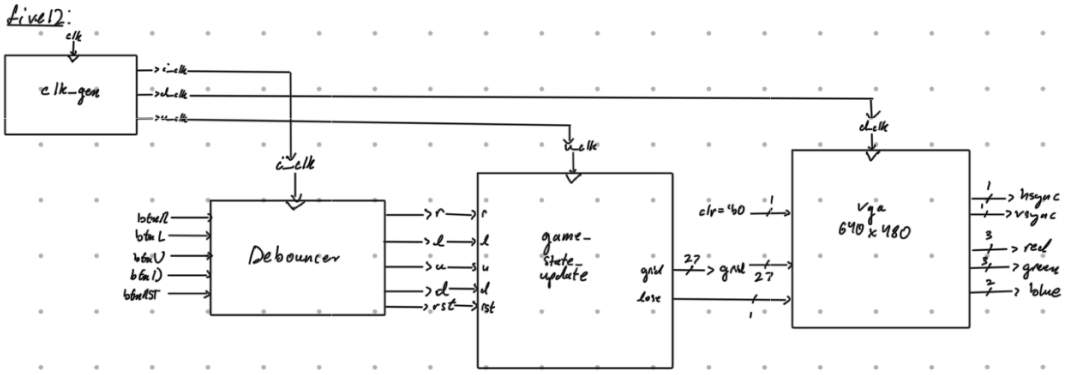

Figure 6: Outer-level game logic hardware implementation

After explaining all the modules, it becomes evident how each module is interconnected and how they build on one another in a perfect modular fashion. Each module serves a specific function and communicates with other modules through input and output signals, creating a hierarchical system. For example, the debouncer module debounces the input buttons and sends signals to the game_state_update module, which updates the game state and outputs to the VGA display through the vga640x480 module. The modular design allows for easy maintenance and modification of each module without affecting the overall functionality of the system.

Conclusion

Overall, our design includes several modules, including a debouncer for input control, a game state manager to update the game state, and a VGA controller to display the game on the monitor. The design was challenging at times, and one of the main difficulties we encountered was with the VGA controller. It was a new concept to us and we were unsure how to update it with each grid update. We overcame this challenge by conducting extensive testing and research to find a solution. Overall, we believe that we did a good job creating a specification for the lab that balanced time and difficulty. As a general suggestion for improving the lab, it would be helpful to provide more detailed instructions during class on how to implement the VGA controller to reduce the learning curve for students.