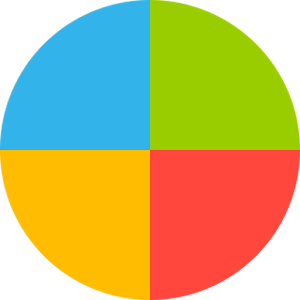
****



**SIMON**

ECEG 240 Digital Design Final Project

Dunni Adenuga  
Brooke Bullek  
Bobby Cao  
Andrew Capuano

Table of Contents

1. Introduction **Page 2**

2. Description of the Simon Game **Page 2**

3. Objectives **Page 3**

4. Delegation of Work **Page 3**

5. Block Diagram **Page 4**

6. Initial FSM State Transition Diagram **Page 6**

7. Five Modules **Page 7**

1. Simon\_FSM Page 7
2. Display Module Page 12
3. Debounce Buttons Page 15
4. Validate Buttons Page 17
5. Sound Page 18

8. Final State Transition Diagram **Page 20**

9. Challenges we overcame **Page 22**

10. Future Work **Page 22**

11. Innovation **Page 25**

12. FPGA board overview **Page 26**

13. Lessons Learned **Page 28**

14. Conclusions **Page 29**

15. Appendices **Page 30**

1. **Introduction**

Our team was very eager to tackle this project because it would be an opportunity to utilize everything we learned this semester. It was a unanimous decision when we decided to implement Simon, a popular memory game. The FPGA board was perfect for replicating Simon's gameplay because of the four buttons and the 7-segment display. We all became familiar with the gameplay mechanics in order to collaboratively come up with a vision for our version of the game. Each of us was able to contribute to the design and creation of which modules would be useful. We were very ambitious and we wanted to add a sound and VGA display module. Of course, with a pressuring deadline some of those ambitions were not accomplished, but they are discussed in detail in Future Work (Section 10).

Being very enthusiastic we started early to implement our models through Verilog. Making our respective modules was a task in and of itself because we ran into errors and unavoidable warnings with some of our modules. Through trial and error and with help from the TAs and Meriel, we fixed our modules and eventually had Version 0 of our project working. Version 0 was just the FPGA board showing a sequence of length one on the 7-segment display and the player would repeat the simple sequence by pressing the respective button. The win, lose, and playing LEDs, along with a reset switch, were also functional in Version 0. For our Version 1, we implemented more rounds of the game in addition to implementing the sound module, having a message displayed on the 7-segment display, and having the LEDs and state machine logic fully working.

Finally with accomplishing this project we were proud of what we put together. We met our goal of replicating the game of Simon on the FPGA board and having it playable. We also surpassed a few of our goals; for example, we initially wanted to have a welcome sound to entice the player, but we thought sound would be more useful if we were to output a tone during a button press. There were many lessons learned and many things we would do differently if we had a clean slate and if we worked on this project at the start of the semester.

1. **Description of Simon**

Simon is an electronic memory game. The device creates a series of tones and colored lights and requires the player to repeat the series by pressing colored buttons. If the user succeeds in repeating the series, the next level will repeat the series while adding another step to the end of it. It becomes progressively longer and more complex, and if the player fails to reproduce the series, the game ends. If the player wishes to start over the series will start with one tone and color and will continue from there. Most Simon games have four colored buttons, with each producing a particular tone when it is pressed or activated by the device. There are variations of the game which include more buttons and colors than the standard four. There are also different sounds or an on/off mode that toggles sound for a challenging session.

1. **Objectives**

The objective was choosing a game that we have played before and one that we knew would be challenging to emulate. For this reason, we unanimously decided to implement the game Simon. The reason for this decisions boiled down to the rules and the structure of the game. This project would simulate the Simon game using the FPGA board. Just like the real life counterpart, our game will start as soon as we turn on the game. The player will use the set of 7-segment displays and buttons on the board to play the game. When starting the game one of the 4 segments of the display shows a signal indicating which button to press. These hard coded sequences subsequently appear on the appropriate segments of the display. This process continues until the player loses the game by failing to repeat the correct sequence (press the wrong button), at which point a loss signal is displayed. However, if the player is successful, the game advances to the next round. A user who successfully completes all levels the player will be greeted with a win message.   
 To complete this project, we plan to make use of the 7-segment display, buttons, and LEDs on the Nexys2 board. Then we planned to develop a top-level Verilog module to contain the game logic and bridge the gap between the other modules. We quickly realized that many challenges would ensue from linking up the various display and button modules with our game logic. However, these issues would later be resolved and we found ourselves well on our way to our final goal.

1. **Delegation of Work**

Andrew and Bobby implemented the buttons via a button module to work with player inputs. Then they debounced all four buttons using a debouncing module from lab. Dunni and Brooke implemented the display module to show the sequence of button presses required for the series, and Dunni presented game feedback to the user on the FPGA’s seven segment display during the win and lose states. Brooke implemented the game logic and game rules and some parts of the finite state machine governing our game in our main *Simon FSM* module along with the help from Bobby hard coding three rounds for the user to play. Additionally, Dunni was instrumental in creating some other necessary states to make the finite state machine more robust and responsive. We then connected the logic between the display and the buttons to complete the game functions. Brooke incorporated the sound to go along with each button press to make the game intuitive and user-friendly.

Everyone then worked on combining the modules. This was a struggle but with each group member’s contribution, we were able to do this in a shorter amount of time than we had planned for. Adequate communication between each team member concerning the required inputs and expected outputs made the combination of the modules seamless.

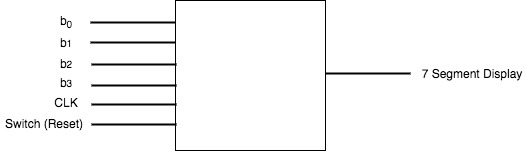
Our group used a version of Scrum, a form of agile development. We had daily 10 minute sprints and we discussed our progress. GroupMe was an essential element of our collaboration. It served as a Scrum board containing our goals and timeline, and it also helped us set appropriate meeting times for everyone. At every meeting, we always had short sessions beforehand to discuss the work that we want to complete before our next meeting (short sprints). Also, we had long sprints, about one week in length, for certain complex features of the game. For example, merging all the implemented modules from the shorter sprints took a longer period of time.

User input and feedback is a major part of agile development. While we didn’t have an official user, Professor Meriel Huggard and our fellow ECEG 240 classmates served as our de facto users. We shared our progress (both triumphs and tribulations) intermittently with Meriel and our classmates through presentations and sometimes, inviting Meriel to our meetings. We were able to receive useful feedback which we incorporated into our Simon game design. For example, when we shared our two possible game designs, Meriel gave feedback, from a user’s perspective of just wanting a working game, that helped us make a decision.

There were a number of struggles during our endeavor with this project so sometimes, we had different group members try to solve the problems. For example, to meet up with a very tight deadline for a working version 0, we split our team into two. Team 1, consisting of Dunni and Brooke, worked on getting a simplistic version of the Simon game of one round with hard-coded sequences while team 2, Andrew and Bobby, worked on minimizing the amount of warnings such as latches and combinatorial loops generated by the system.

1. **Block Diagram**

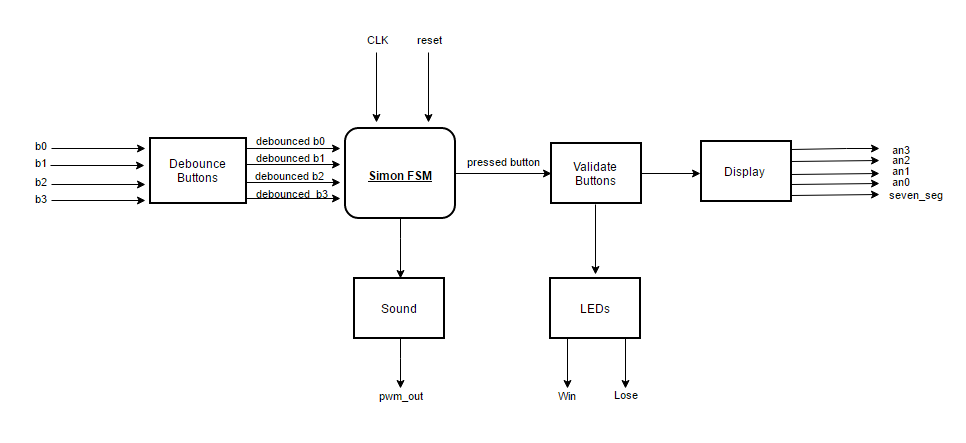
Below, in Figure 1 we see our group’s block diagram for version 0. It was extremely simple because we were having issues with our design that held us back from what we wanted to achieve with our initial implementation (see Section 9).



**Figure 1: The block diagram for our system's version 0**

Our crude design involved 4 buttons that were not debounced, a clock and a reset switch to restart the game. The status of the game was outputted to the 7-segment display using a marker that showed us whether or not the game had been completed successfully. The design was very simple and involved us reworking our original, overly ambitious design.

Our design for version 1 was much more sophisticated because once we figured out a smarter design we could use to build more levels, we made leaps and bounds in terms of the game that we produced. Our updated block diagram can be see below in Figure 2.

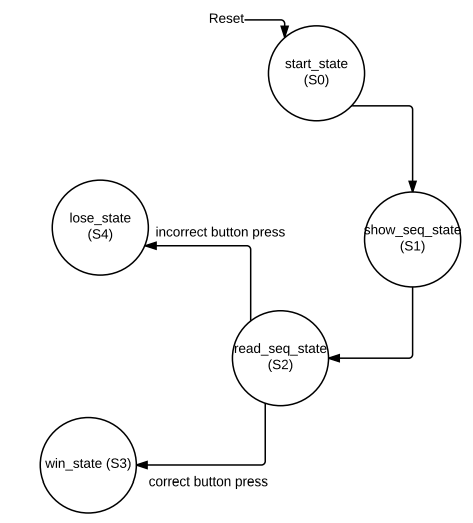


**Figure 2: The block diagram for our system's Version 1**

The flow of the block diagram begins when the 4 buttons from the FPGA are inputted to a module named *debounce\_btns* (see Section 7.d). They come out debounced nicely, which is important because we wanted to create a game where every button press was as accurate as possible. If the user pressed the button once, the system should only register one button press, as two presses could end the game (if the second button press did not match the next intended pattern). These buttons were then inputted to the *Simon FSM*, our top-level module that housed all of the game logic (see Section 7.a). That module also takes inputs of a universal clock and reset switch, and from that module comes a sound for each button pressed (see Section 7.e). The *Simon FSM* module also figures out which of the 4 debounced buttons was the one pressed, which is then an input to the *validate\_btns* module (see Section 7.b). This module checks whether the button pressed matches that from the sequence which was outputted to the display. From this module, LEDs on the FPGA board can signify whether the user has won, lost, or remains in "playing mode" (defined as the states in which the game is actively expecting the player to press a button). If the game is merely showing the user the pattern via the 7-segment display, no LEDs are illuminated. Finally, if that button press is validated, it shows a flashing marker on one of the 4 anodes of the 7-segment display. There is also the possibility that the 7-segment display lights up with a message that tells the user of their status (see Section 7.c) to give them a result that matches that of the win or lose LED.

1. **Initial State Transition Diagram**

Our initial state transition diagram consisted of five states that encapsulated the core functionality of our proposed system. It was vital that our Simon implementation carry out the logic that showed the player a sequence (S1) before reading the player's imitation of the sequence (S2), at which point the player would either advance or lose the game following the entry of an incorrect sequence (S4). In our system's version 0, our game had only one round (effectively, a sequence of length one), and completing this round successfully would land the player in the win state (S3). Finally, a start state would permit the system to be reset if the player wanted to try again (S0).

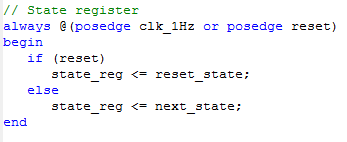


**Figure 3: The state transition diagram for our system's version 0.**

Figure 3 shows our initial state transition diagram. The branching paths to the win and lose states depended on the validation of a button press; the draft of this diagram alerted us to the necessity of a Verilog module that would allow the 7-segment display to "talk to" the four button inputs from the board in order to validate the pattern. Otherwise, the correctness of a button press would be indeterminate. Despite its simplicity, this V0 diagram steered us in the right direction toward building a system that could be iteratively improved upon throughout the duration of the project.

1. **Modules**

**7.a. Simon FSM Module**  
 *(Authors: Bobby Cao, Brooke Bullek, Dunni Adenuga, Andrew Capuano)*  
  
 Simon is our comprehensive module and contains all of the game logic. One of its most important tasks is directing the game throughout several states via a finite state machine (FSM). Each state represents a concrete stage of the game, including displaying a pattern to the user, reading a pattern from the user, or alerting the user to the fact that they have either won or lost. In general, this FSM is driven by two variables: *state\_reg* (the current state) and *next\_state*. A short always block (shown in Figure 4) updates the current state every second (or 1 Hz clock cycle) by setting it to the next state.



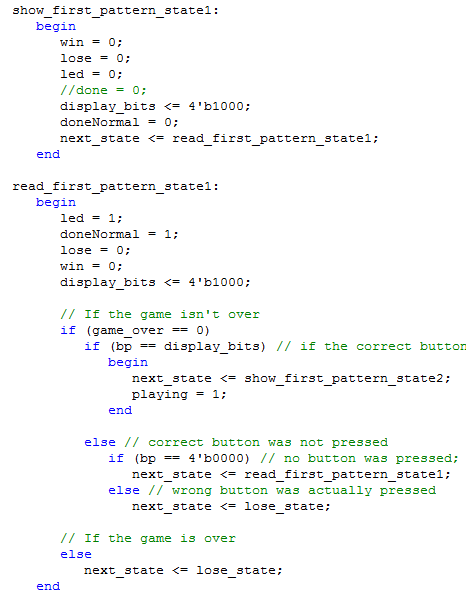
**Figure 4: The heart of the sequential logic that drives the system's FSM.**

Shown in Figure 5are our two important classifications of states: show and read states. The show state interacts with the display module in order to show the player the sequence/pattern on the 7-segment display. The read state reads the user’s input and checks it if it correctly matches the pattern that was displayed. We hard coded 3 levels in our game. For each level, we used a new set of show and read states.

The naming conventions for each state are a bit unorthodox. We have six show and six read states in total. For each level, the previous show and read state are repeated, but we named the repeated states as *state2* and *state3*. We could not just reassign our next\_state register to a previously called state because that would have created a loop in our game logic, so we created new states that consisted of repetitive code. The naming convention followed *“show\_nth\_pattern\_state#” or “read\_nth\_pattern\_state#* ”, representing the nth pattern (first, second, or third—since our game has only three rounds). The “*state#*” token indicates the number of times the pattern had been shown at that point; for example, “*show\_first\_pattern\_state3”* shows the first pattern for the 3rd time, which is associated with the third level of Simon. The full list of show and read states is listed below, and outlines the chronological progression of a successful Simon game:

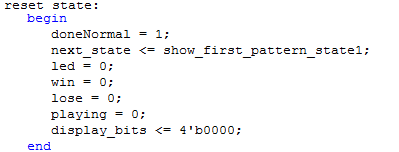
1. show\_first\_pattern\_state1
2. read\_first\_pattern\_state1
3. show\_first\_pattern\_state2
4. show\_second\_pattern\_state1
5. read\_first\_pattern\_state2
6. read\_second\_pattern\_state1
7. show\_first\_pattern\_state3
8. show\_second\_pattern\_state2
9. show\_third\_pattern\_state1
10. read\_first\_pattern\_state3
11. read\_second\_pattern\_state2
12. read\_third\_pattern\_state1

Only one show and read state comprised the first pattern. For the second state, we added an additional set of show and read states. In this second set of show and read states, we copied the *show\_first\_pattern\_state* and *read\_first\_pattern\_state* from the first round, but we adjusted the *next\_state* appropriately. We initially had a more intuitive way of adding levels, but we could not make it functional within the allotted time. Although the newer yet more cumbersome state machine did work, our old design was more innovative and will be discussed later in Future Work (Section 10) and Lessons Learned (Section 13).

****

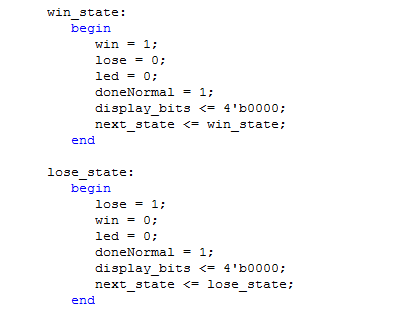
**Figure 5: The show and read states associated with Simon's first round.**

In addition to our 12 combined show and read states, we had three additional states that helped to direct the flow of our game's logic. One such state is the *reset\_state*, which is the state immediately entered when the FPGA is programmed and when the player flicks the reset switch on the board. The logic of the *reset\_state* within the case statement of the FSM is shown in Figure 6. The purpose of the *reset\_state* is to merely direct the *next\_state* to *show\_first\_pattern\_state1*, which launches a new game of Simon beginning from the first round and progresses from there.



**Figure 6: The reset state in the simon\_fsm module.**

We also include a *win\_state* and *lose\_state*, which are self-explanatory but can only be set within specific instances of the other states. For example, within any of the read states, if the game senses a button press but deems it incorrect, the *next\_state* will be immediately directed to the *lose\_state*, regardless of the extent of the player's progression through the game. However, the *win\_state* can only be entered from within the *read\_third\_pattern\_state1* (that is, the state that reads the third pattern from the user from the first time—i.e. the end of the third and final round of the game). The *win\_state* is entered upon a correct button press in this case. The *next\_state* is set to the equivalent state once either the *win\_state* or *lose\_state* has been reached, meaning once the player has won or lost, the game will remain in the respective state until a reset. These states can be viewed in Figure 7. Note that *win* and *lose* (both LEDs) are set appropriately, and *doneNormal* and *display\_bits* are set to default values in order to avoid latches throughout the system. Finally, *next\_state* is left unchanged from the current state.



**Figure 7: The win and lose states in the simon\_fsm module.**

Throughout our state machine, we implemented a few rules governing the illumination of three LEDs on the game board. This greatly aided our debugging process while we were still working on getting a functional state machine that transitioned between the show, read, win, and lose states appropriately. The furthest right LED on the board indicated to the player (and largely us as developers) that the game was actively in one of the read states. This LED indicated that the game was expecting the player to input a button press in order to advance further in the game. The LED second from the right was illuminated whenever the game entered the *lose\_state*. Similarly, the LED third from the right turned on upon the game entering the *win\_state*. The ability to map states to visual components on the board was an efficient way to notice flaws in our game's logic, especially in the case of unanticipated latches or combinatorial loops. It was often the case with our earlier designs that the game's win LED would turn on immediately after showing the pattern via the 7-segment display, despite no button presses from the user. Errors like these in our implementation were easily identified using this debugging process.

Since the LEDs can only be set to high within the appropriate read, lose, or win states, the show pattern states initially sets all LEDs to 0 to avoid latches. Then the pattern is shown on the 7-segment display via *display\_bits*, which uses one-hot-encoding. For example, when *display\_bits* has a value of 1000, the first display lights up. The 7-segment display lights up one panel at a time, so using the bits 0100 we light up the second display. The one-hot encodings of 0010 and 0001 naturally light up the third and fourth displays, respectively.

The read pattern states first set the rightmost LED to 1, indicating to the player that it is their turn to repeat the sequence from the show state. The *doneNormal* variable is set to 0 in the read pattern states since we don’t want the seven segment display to show anything to the user at that time. It’s also set in order to avoid latches. The lose and win LEDs are again set to 0 to avoid latches. The *display\_bits* register is hard-coded again in this read pattern state to match its respective show pattern state. We checked for a match between the player’s button press and the correct display, as represented in *display\_bits*. Shown in Figure 5, if the button press is correct, *next\_state* goes to the beginning of the next round of show pattern states if the player pressed the final button for that round. However, if the sequence is longer than one pattern and more patterns remain, *next\_state* goes to the next read state to read the next button press in the sequence. If no button was pressed during the positive edge of the 1 Hz clock, which governs the state machine, then *next\_state* is simply set to the current read state once again. This gives the player as many seconds as they need to decide on a button to press, rather than automatically going to the *lose\_state*. In order to enter the *lose\_state*, the user needs to explicitly enter a button press that did not match what *display\_bits* was expecting.

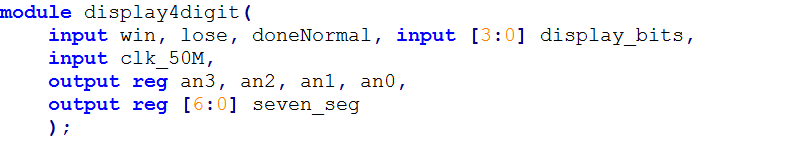
As the system's top module, *simon\_fsm* controlled more than the comprehensive state machine that guided the player through several rounds of a Simon game. This module was also responsible for instantiating the other modules and scaling the 50 MHz *clk\_50M* input to a more usable 1 Hz clock. Other responsibilities include updating several variables that will activate the always blocks within other modules. The *playing* variable is an example—when playing is set, the sound module will play a tone (discussed in Section 7.e). The *display\_bits* variable is another example, and an update of this register will subsequently update the active marker on the 7-segment display. Most importantly, without the *simon\_fsm* module, there would be no bridge between the display, button, and sound modules.

**7.b. Display Module**

*(Author: Dunni Adenuga*)

Display was an important part of this system.The display module interacted mainly with the 7-segment display of our FPGA board. The 7-segment display is a little display at the right-hand corner of the board that outputs an understandable message or feedback to the user. It has four anodes. Each of these anodes can be turned on or off based on preference. These anodes are represented as pins F17, H17, C18 and F15 on the FPGA board and are indicated as such in the simon UCF file. Each anode has eight segments, seven lines and a dot but our game only makes use of the seven lines. These seven lines are represented as pins L18, F18, D17, D16, G14, J17 and H14 on the FPGA board and are represented as such in an array of size 7 in the simon UCF file. Every part of the seven segment display operates on active-low i.e bit 0 turns them on while bit 1 turns them off. The four anodes are represented as *an0*, *an1*, *an2* and *an3* in our main Verilog program while the seven segments are represented in a 7 bit array register *seven\_seg*.

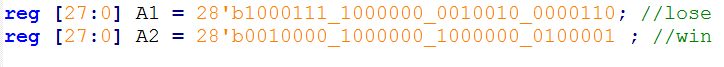
The basis of our Simon game is for the anodes of the seven segment to run through a sequence by displaying the marker “0” on the next anode in the sequence while turning the other anodes off (i.e only one anode is on at a time) and comparing the displayed anode to the user’s button presses. There are also four buttons which are made to correspond with each anode of the display. The buttons are represented as *b0*, *b1*, *b2* and *b3* in accordance with the anodes. Another function the display module performs in our game is to give the user feedback at the end of the game depending on the state (whether win or lose state) the user ends up in as a result of the game logic. The display module was also designed to interact with the simon\_fsm module. The display module is called *display4digit*  in our project and is defined as follows:



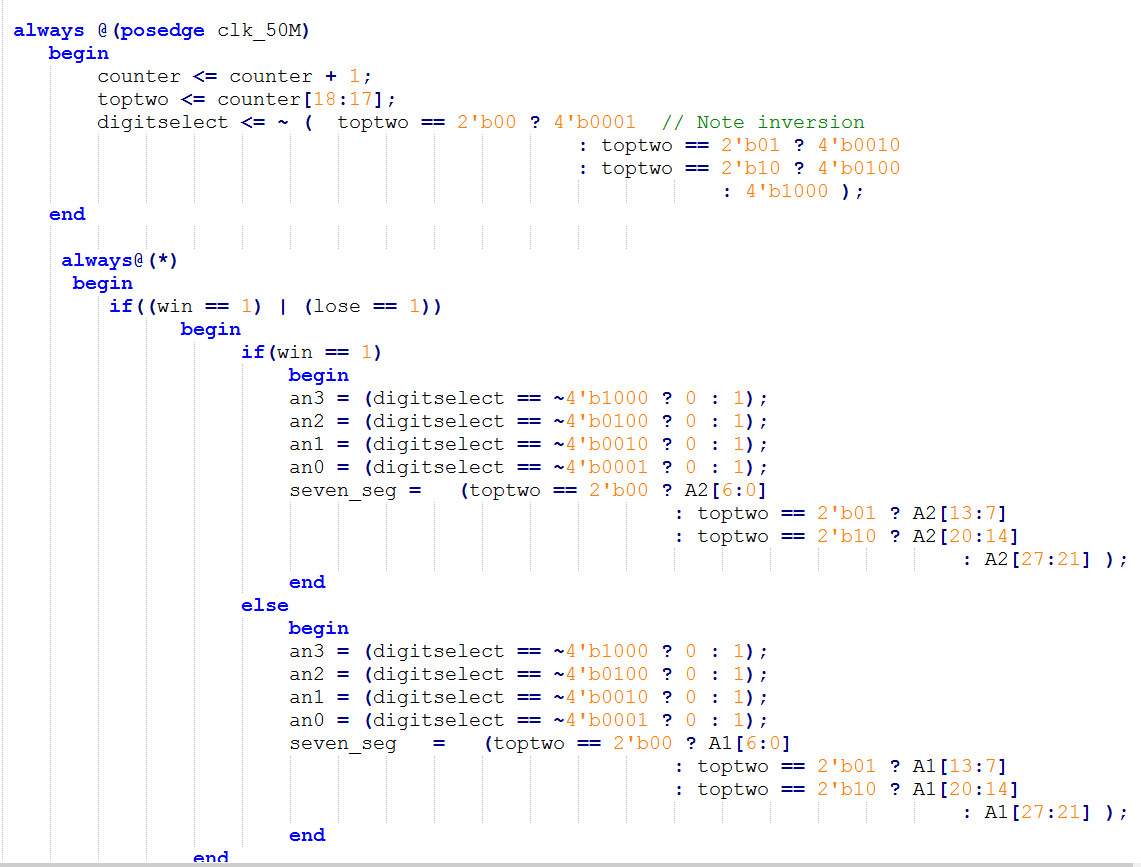
**Figure 8: *display4digit* module: it displays the gOOd or LOSE message**

*display4digit* module takes in input variables *win*, *lose*, *doneNormal*, 4 bit array *display\_bits*, and the FPGA 50 MHz clock. The *win*, *lose* and *doneNormal* variables help the module determine the current state of the system. If *win* is set to 1, “g00d” is displayed on the 7-segment display, if *lose* is set to 1, “L0SE” is displayed on the 7-segment display and if *doneNormal* is set to 0, the 4 bit *display\_bits* array determines which anode on the 7-segment display should display the “0” marker. The *win* variable is only set to 1 in the win state of the *simon\_fsm* module, the *lose* variable is only set to 1 in the *lose\_state* of the FSM, and the *doneNormal* variable is only set to 0 in the show pattern states of the FSM. The outputs of the *display4digit* module (*an3*, *an2*, *an1*, *an0* and *seven\_seg*) are manipulated for these different purposes.

To display “g00d” and “L0SE” a process called “persistence of vision” is used. The anodes can’t display different values at the same time so, we cycle through each anode displaying the required letter fast enough that the human eye thinks they are being displayed at once. We need to decide which seven segments of each anode needs to be turned on to display each letter of “g00d” and “L0SE”. Our decision for the 4 anodes can be combined in a register array. This is shown below in Figure 9.

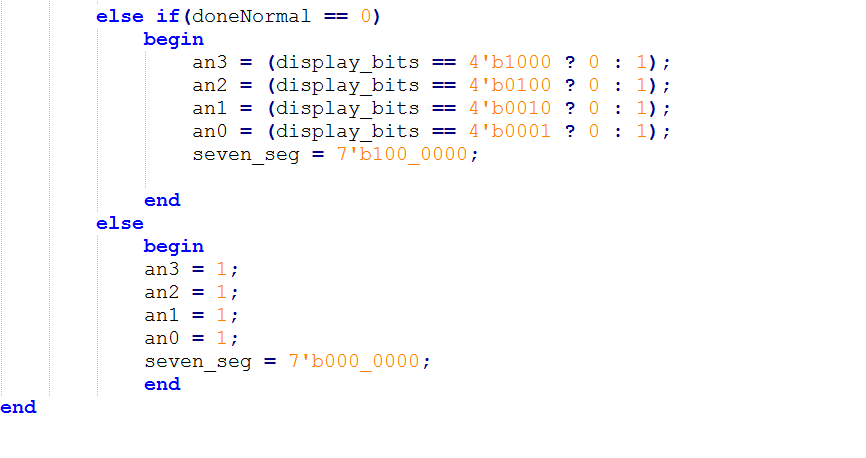


**Figure 9: Seven segments representation for the 4 anodes**



**Figure 10: Block of code to cycle between Anodes of 7-segment display**

In the sequential logic block of code above an 18 bit variable counter is used to create this cycle between anodes. Variable *toptwo* checks the first two most significant bits to determine what anode would be on at a particular time and what’s displayed on it. It takes about 20ns for the most significant bit to turn 1. The combinatorial logic block beneath it helps the module decide the state the system is currently. If in the win or lose state, the “persistence of vision” logic runs else if we’re in the show\_pattern states, marker “0” (as signified by the segments turned on in seven\_seg) is displayed on the specific location indicated by display\_bits as shown below:



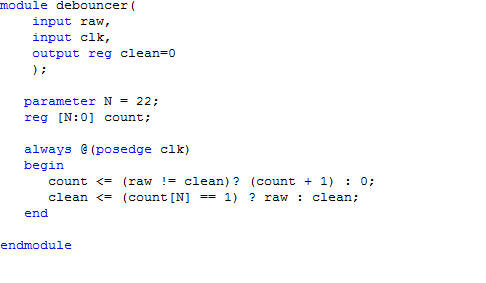
**Figure 11: Block of code that represents the display logic when the system is in the show\_pattern states.**

Otherwise, the display is turned off by setting all the anodes to 1 as shown in Figure 11.

**7.c Debounce Buttons**

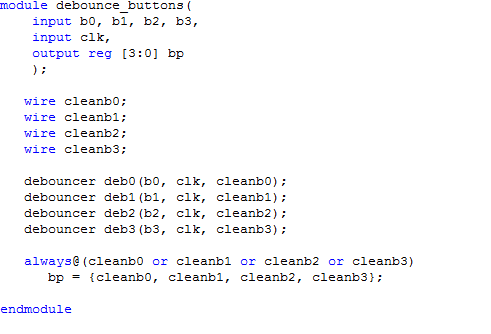
*(Authors: Bobby Cao and Andrew Capuano)*

The first step to debouncing the buttons of the FPGA board correctly was to debounce a single button. As seen below in Figure 12, we used code from our work in lab in order to debounce a single button correctly. We had a counter that we would increment if the signal was not clean. When the signal stabilized and came through as clean, we would set *count* to 0, which in turn sets *clean* to itself. This validates that the button has been debounced.



**Figure 12: The debouncer module.**

Our next task was to debounce 4 buttons, as the FPGA board has 4 buttons that we used in conjunction with the 7-segment display in order to play the game. Now, we inputted 4 buttons (*b0-b3*) and our *clk*, and outputted a 4 bit (one hot encoded) register called *bp* (for button press). We used 4 wires to carry the output of 4 debouncers that we called in order to take the initial button press, debounce it, and then wire it into the *simon\_fsm* top module. As seen below in Figure 13, we created a module named *debounce\_buttons*, which utilizes the debouncer to debounce each of the 4 buttons. Then, we used one always @ block to see which button had been pressed, and store that value in the variable *bp*. This is then wired into the next module, *validate\_btns*, which is essential to the gameplay.

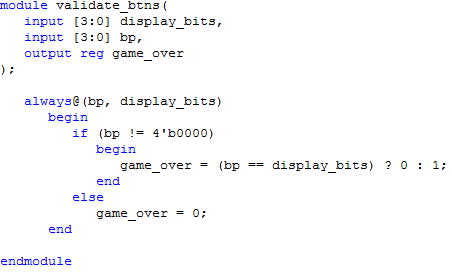


**Figure 13: The debounce\_buttons module.**

**7.d. Validate Buttons**

*(Authors: Bobby Cao and Andrew Capuano)*

The other button-related module that we implemented can be seen below in Figure 14. This module checks to see whether the inputted clean button press (from the *debounce\_buttons* module) is equal to the button that corresponds to the sequence of the game. We use logic to check to see whether *bp == display\_bits*, where the latter is a variable that holds the value of the sequence that should be inputted to the FPGA board. As seen in the code, if *bp* is equal to *display\_bits*, the user pressed the correct button, so *game\_over* is set to 0 (the user is still alive and playing). Otherwise, *game\_over* is set to 1 and the user enters the *lose\_state*, in which the game ends and the display shows “LOSE” to the user. A simple module, *validate\_btns* gets the job of checking whether a button pressed was the correct one and successfully implements one essential function of the game that makes it exactly like the original game.

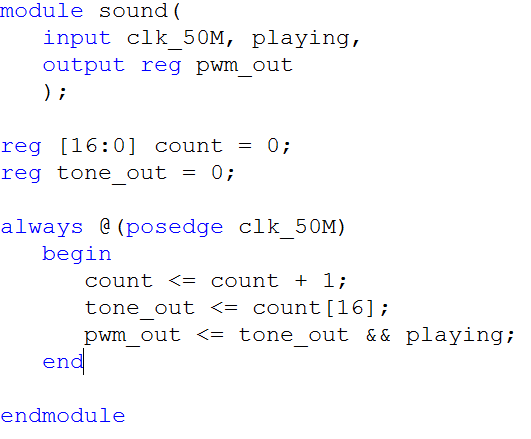


**Figure 14: The validate\_btns module.**

7.e. Sound Module   
 *(Author: Brooke Bullek*)

Sound was the last module we added to our system. It was important not only because we needed some feedback mechanism to alert the user to the timing of their button presses, but also because it was an opportunity to make use of external components (a small amp and a speaker).

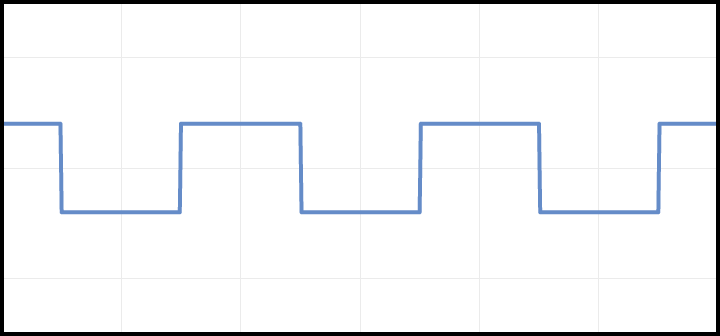
Our implementation of the sound module in Verilog is shown in Figure 15. This module has two one-bit inputs: *clk\_50M* and *playing*. The *clk\_50M* input is a 50 MHz clock passed via the calling *simon\_fsm* module, which in turn reads it as an input from the FPGA board from the UCF file. The playing input is simply used as a boolean flag that is also passed and set in the *simon\_fsm* calling module.



**Figure 15: The body of our sound module.**

*playing* is set whenever a correct button press is registered, and there are two conditions that must be met in order for this to happen. First, the state machine must be in one of the read states, which are described in the *simon\_fsm* module's section. Second, the *validate\_btns* module must have output the most recent button press, represented as a four-bit one-hot encoding of the debounced button input. When this one-hot encoding's debounced button press variable matches the hard-coded pattern inside of its respective read state, *playing* is set to 1. *playing* is set to 0 initially inside of the state machine, independent of state, in order for the read states to overwrite it if necessary and prevent latches.

Our output is *pwm\_out* (where PWM stands for Pulse Width Modulation, which is a technique for acquiring an analog signal by digital means). Using a digital system, it's simple to use PWM and create a square wave, observed in Figure 16:



**Figure 16: A square wave.**

A classic square wave is easy to represent using a digital binary signal; the amplitude of the wave is simply 1 or 0, where 1 represents a positive amplitude and 0 represents no amplitude. A square wave was better suited to our digital implementation than, for example, the curvature of a sine wave. The frequency of these shifts between a 1 and 0 in this audio signal is an important attribute of the wave. If the frequency is too low or too high, it will be imperceptible to the human ear. We selected a 440 Hz frequency for our output tone (represented by the flipping of pwm\_out's bit on an appropriate time scale). We needed to modify the 50 MHz clock (the *clk\_50M* input) in order to scale it down to a 440 Hz clock.

In order to create this square wave, we effectively needed to ignore *n* clock cycles from the 50 MHz clock. The 50 MHz clock is several orders of magnitude faster than a 440 Hz signal, so we would only want to record every 50 MHz / 440 Hz = 50,000,000 / 440 or approximately 113636th clock cycle. This was done by determining the minimum number of bits that could contain this number; 2^17 (which can store a number up to 131072) satisfied this. Any number of bits higher than 17 would have resulted in a lower frequency, and a number lower than 17 would have ignored fewer clock cycles and thus yielded a higher frequency. However, 440 Hz sits comfortably in the audible range of 20 Hz to 20 kHz, and a 440 Hz square wave is a very recognizable tone.

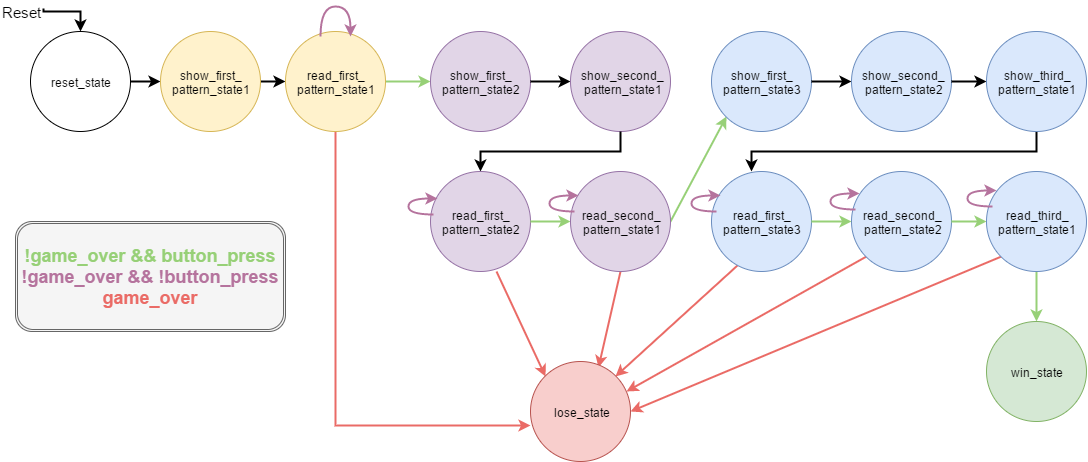
After determining the proper width of the new register used to scale the clock input to 440 Hz, which was a variable named *count*, we were able to use it to help generate a signal. Each time the 50 MHz clock changed (50,000,000 times per second, or every 20 nanoseconds), the count register would be incremented. If the most significant bit of this *count* register was 1, it was interpreted as a positive edge of the artificial 440 Hz clock. Any other series of bits, so long as the most significant *count* bit was 0, was interpreted as a negative edge. A register named *tone\_out* was simply set to the most significant bit of the *count* register.

When the 440 Hz had a positive edge, and when the *playing* input given from the outer *simon\_fsm* module was 1, both conditions were met to set the *pwm\_out* output to 1 (as seen in the line *pwm\_out <= tone\_out && playing*).

The logic in this simple module works as expected; a tone is output (i.e. *pwm\_out* is set) in accordance with this new 440 Hz clock whenever the game decides a tone should play (as interpreted by the playing input).

Although he was not part of our team, it's important to mention the help provided by class member Daniel Vasquez in implementing sound successfully. He provided guidance on how to locate the pin for the audio output of the Nexys2 board, as well as a summary of PWM.

1. **Final State Transition Diagram**

Our final state transition diagram expanded upon our system's V0 by incorporating three different rounds of Simon. Our design, while somewhat repetitive with respect to the multiple show pattern and read pattern states, allowed us to quickly finish a working multi-round prototype that showcased the main features of the game. This implementation freed up more time to add other functionality (such as sound) to improve the user's experience with our Simon project. The final diagram is shown below in Figure 17.  
  


**Figure 17: The final state transition diagram of our system.**

As shown, the logic of our completed FSM tied together the display module (handled by the show pattern states) and button module (read pattern states), and relied upon an output from the *validate\_btns* module called *game\_over* to direct the next state to the appropriate location. The directional arrows marked in red indicate an immediate transition to the system's *lose\_state* when the *game\_over* flag has been set. In the alternative case where *game\_over* has not been set to true, the game's logic depends on the presence of a button press.

In other words, when the user has pressed a button (but *game\_over* has not been set), it is known that this button press was correct. This causes the *next\_state* to advance further into the game, possibly even into the *win\_state* if all rounds have been completed. These transitions are marked as green arrows.

However, if no button press was detected (as given by a one-hot encoding of four zeroes as output from the *debounce\_buttons* module), Simon will remain in the same read pattern state, listening again for a button press during the next clock cycle. These transitions are marked as purple arrows.

The arrows marked in black in Figure 17 are unconditional transitions; the next state will be set after a 1 second delay in accordance with the modified 1 Hz clock. These unconditional transitions are predominantly limited to the show pattern states, which require no input from the user. During these show pattern states, it is the user's job to merely observe the pattern one step at a time as it flashes across the array of four displays on the board.

1. **Challenges**

We overcame a number of challenges during our endeavor. Our initial idea was having a design where we could intuitively add more rounds and then add a random module to add random rounds. However, this proved to be really difficult so we scrapped that design and we all created a new design that made the addition of rounds not as intuitive but our final product works.   
 Then we had a large number of problems with Verilog warnings. We had a large amount of “*FF/latches*” (See Appendix A), and we solved it by completing our cases and made sure each if statements had the proper else statement. Also we got rid of the *curr\_round* variable all together and it got right of most of our latches. For example, latches caused the win led to on before the user finished all the rounds. This took a few days to get sorted.   
 Then, there were problems with combinatorial loop warnings (See Appendix A) which took a couple days to figure out. We just had to be consistent with our variable placement. Then we had to make sure that any variables on the right side of any assignments is completely distinct from any variables the left side of any assignments.  
 Timing button presses was also an issue because we initially would have the player press the button and hold it for about 1 second. Of course, that is difficult to keep consistent and made the game somewhat frustrating to play. We added a sound module to output a 440 hertz tone that indicated to the player how long to press the button.   
 In turn, when we had to address the challenge of timing button presses we had to address the challenge of adding sound. Adding sound was a bit difficult but it took a good amount of google searches and forums to implement the sound with our clock.

1. Future Work

Future work refers to features we would love to implement in our Simon game if we had more time. We started this project with a lot ambitions but had to streamline some of them due to availability of time, knowledge and materials. Some of these features include:

* An improved FSM design: ECEG 240 is a class based on digital design. We learned methods of simplifying circuits. Electrical designs should be simple and efficient. We tried to implement this in our simon game finite state machine design by using a variable “current\_round” to keep track of the level of the game compared to the hard coded number of rounds. Unfortunately, Xilinx IDE displayed warnings such as “latches” and combinatorial loops. We solved the latches issue but we did not have enough time to tackle the combinatorial loops issue. So, we settled for a repetitive structure depending on the number of rounds. For example, a game with three rounds has 3 similar “show\_pattern” and “read\_pattern” states but with different display bits. If we had more time or in the future, we would like to implement a more efficient simon game finite machine with a working “curr\_round”. Once we’re able to implement a current round, it makes it easy to tag a user-friendly feature of a scoreboard. The scoreboard would keep track of the user score by assigning marks depending on the time it takes the user to replicate the game pattern and the highest level the user is able to attain.
* VGA module: Our Simon game is essentially about the user. It’s a user-centric game. A feature we believe would have made the user engaged for a longer period of time is a larger display. Our current version of the game makes uses of the seven-segment display, a little display at the right corner of the FPGA board. This little display may be uncomfortable for our users with short-sighted vision. Also, playing this game in its current state would require an uncomfortable downward position of the head. Implementing a larger display would require a more comfortable forward position since the likely component attached to the VGA module would be a computer monitor. Manipulating a computer monitor would be relatable task for the user. Also, a larger display would give more than 4 possible marker positions for the user unlike the 4 anodes currently used in our game. Also, a larger display gives the chance to display “WIN :) !!!” or “LOSE :(” instead of the bland “g00d” or “L0SE” currently displayed after the user completes the game. A larger display means we can display the remaining number of buttons the user has to press to the move to the next stage. It means we can keep the user engaged by intermittently showing their current level and scoreboard.
* Random Module: The original Simon game is a game that shows patterns either through colours or sounds. These patterns are created as new sequences every round by adding a new term to the previous sequence. Presently, our game simulates this addition of new terms by hard-coding the sequence in the display\_bits variable. It also stays in the “show\_pattern state” to display every term in the sequence for that level before moving to the “read\_pattern state” to check the user input. This hard coded display\_bits variable makes the game only novel the first time it’s played by the user. The game would only play with this sequence that could be easily mastered by the user so, that they win every time. Then, the user becomes quickly bored of the game. A random module would ensure that the user is not guaranteed the same sequence every time they play the game. This would make it easy to allow more rounds. In Fact, it could be a user dependent feature. Creating this module would require additional verilog knowledge.
* Harder Mode: What’s a game without challenges ? Solving challenges and moving up a ladder of difficulty is what keeps a game player addicted to a game. Our current game only contains 3 rounds i.e the highest number of terms a user has to remember to win a game is 3. 3 is a really small number and does not really challenge the memory capacity of a user. A feature we would have loved to include in our game is the possibility for a user to choose the level of difficulty. This would mean extending the number of terms in a sequence to more than 3. If it’s possible, we would like to make the highest number of rounds be a number in the thousands. The user would be informed of the highest possible level of difficulty on the welcome slide and they would then choose. If the user can reach their choice they would be declared a winner. Aside the number of rounds, another possible harder round is a game with the switches and LEDs on the FPGA board. Our game currently uses the buttons and seven segment display. We would like the LEDs synchronized to debounced switches for the same game logic. Then, the LEDs could display different colours for different switches. The user would then have the choice to play the game either with the seven segment display and buttons or switches and LEDs.
* Different sounds for different button presses: Another way a user can register the position of a marker is by sound. If each anode on the display corresponds to sound of a particular frequency and the button synchronized to that anode makes the same sound when pressed, the game can be played entirely on sound. This opens our game to a more diverse audience and makes our game more engaging and interesting for the player. Currently, our game only makes the same sound at each button press with no distinguishing characteristic and no sound when the wrong button is pressed in a sequence. Instead, an additional feature we would have loved to include if we had more time is a losing sound when the user presses the wrong button in a sequence. Also, a victory song should be played after the user completes a whole sequence of their chosen difficulty. As the user transitions between each level of difficulty, a sound should be played to alert the user. These additions would simulate the original Simon Says game and elaborate on the original designs.
* Timer Module: The timer module is a module we would have loved to integrate into our design. In our Simon Game, the simon\_fsm module has a read\_pattern\_state module that waits indefinitely until a button is pressed. When the simon\_fsm reaches the read\_pattern\_state, the logic checks to see if a valid button was pressed. If no button was pressed, it cycles back to the same read\_pattern state and checks again. Implementing a timer module would reduce the length of this cycle. If the user does not press the right buttons in a certain amount of time, the player immediately goes to the lose\_state. This implementation would make the game more challenging especially in more difficult rounds.

1. **Innovation**

As we planned out the game that we wanted to produce, we thought of different aspects of the game to implement that would make the game sleek and user friendly. This was a crucial aspect of our project because we wanted to create a game that was very similar to the hit Simon game that was released in 1978, while also being usable on an FPGA board. We had to brainstorm for some time as to how we could make this possible, and the work we did with sound, control of the seven segment display and adjustment of the clock show that our final product had many innovative aspects to it.

First, as seen in section **??**, we worked to synchronize a button press with a sound outputted by the board. Whenever the user pressed a button that was validated as correct by the module validate\_btns versus the display\_bits that was on the 7 segment display, a tone would play. We thought that this was a very important addition to the game because it gave the user real time feedback as to their advancement in repeating the sequence shown to them, as well as knowing that the game was actually registering their button press. Sometimes, the FPGA board would not register the users press if they did not press hard enough or for long enough, so this addition made that very clear to a user. Before we added this feature, it was hard to gauge where the user was in the game. We could hardly tell how we were doing until an LED flashed which told the user whether they were in game, had won, or lost. This was the first step in upgrading the quality of our game, and more was to follow.

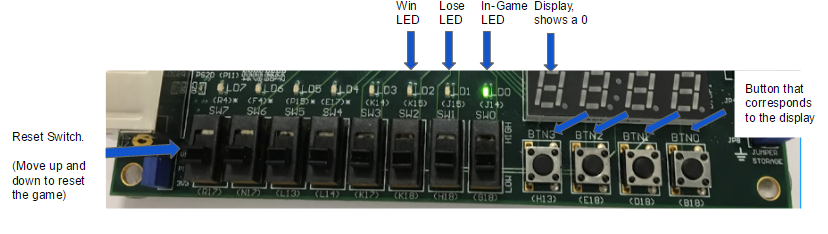
Two additions to the 7 segment display gave the user even more feedback in conjunction with synchronization of sound with button presses. The first was the movement of a marker “0” across the 4 sections of our display in order to show the user what sequence they currently had to repeat. This works in agreement with the clock, which has been optimized and will be explained momentarily. It is important for the user to see this sequence because it is essential to gameplay being possible. We believe that we made this work in a very intuitive manner and with a marker that worked best with the 7 segment display that we could work with. Next, we added concluding remarks to the 7 segment display that outputted the user’s game status - “gOOd” or “LOSE”. The former is displayed if the user has correctly completed all 3 stages of the sequence that the simon\_fsm had outputted to the display, while the latter plays if the user makes an incorrect button press at any time. This feature combined with the sound of each button press makes the game even more usable. Also, it made testing the functionality of our game easy, as we knew what the hard coded sequence would be, when we should hear a sound (which would tell us that our button press was being registered by the FPGA board), and when the message should appear on the 7 segment display, as well as what it should say. Each form of feedback helped us assess how our game was progressing, and helped us figure out when it was finally polished and was complete.

The final element to our program that was extremely innovative with the adjustment of the clock speed. As seen in section 4, our whole system works with a clock that is universal. This means that some of our optimizations, like the marker flashing across the display, and the sounds that the user hears when they press a button were synched to the same clock. The issue was that we were working with a 50 Mhz clock, and this meant that the marker would fly across the screen and would make memorizing a longer sequence extremely hard. Also, the sound that would come with a button press would not be audible by the human ear, as it does not fall into the acceptable range of frequencies that humans can hear. As explained in section **??**, we worked in Verilog to slow our clock to a frequency of about 440 Hz. This was the sweet spot of values that made the whole system work in harmony, it was just the right speed for a user to follow the marker across the display at a fair speed, and it was just the right frequency to make the sound of each button press audible. This was one of our proudest accomplishments as it raised the game’s usability through the roof.

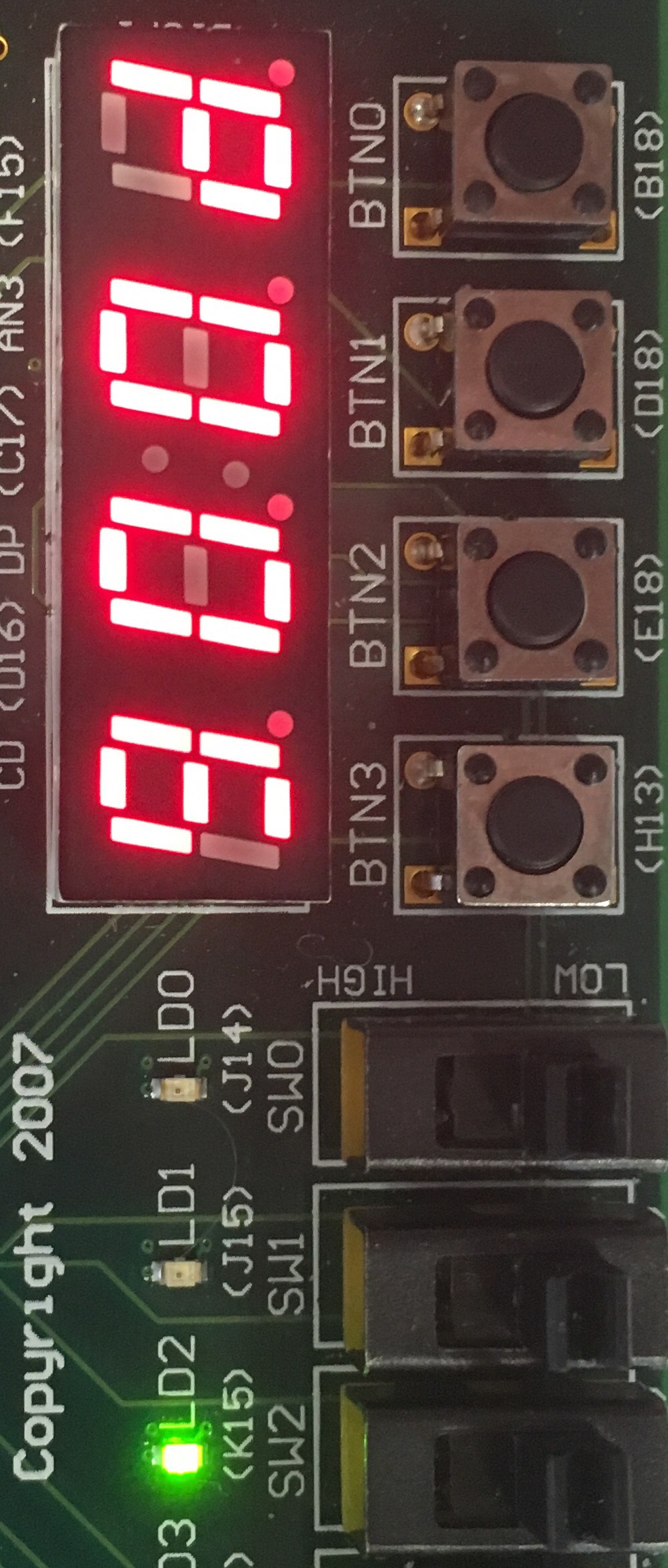
1. **FPGA Board Overview**



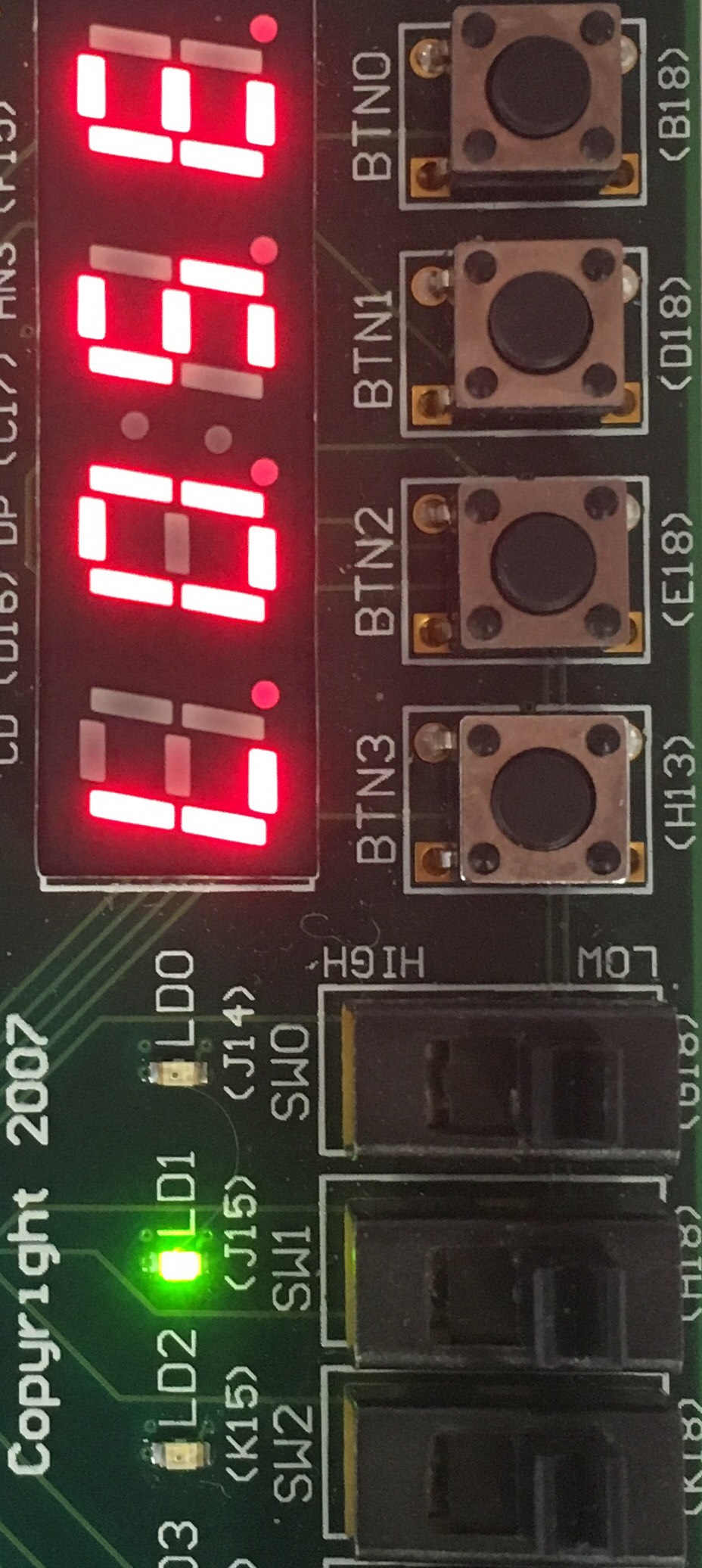
**Figure 18: FPGA Board with Sound Module attached.**



**Figure 19: Close up of the FPGA Board with functionalities of the game indicated**



**Figure 20: Close up of the 7-segment display display the *gOOd* message and the win led on showing the user they won.**



**Figure 21: Close up of the 7-segment display display the *LOSE* message and the lose led on showing the user that they lost.**

Above are pictures of the board and a close-up of the board with all the functionalities of the game included. The game utilizes three LEDs as indicated in Figure 19. The *Win* LED (K15) will turn on after the player has correctly repeated the sequence through the three rounds. Along with that, the 7-segment display will display the word “*gOOd*” meaning good or win. Displaying *WIN* was not possible on this display because there was no way to output a W or N with the display’s lack of a slanted segment. Also displaying *GOOD* was later scrapped because it looked like *6000*. The message “*gOOd*” was the best way to display a win message as an intuitive way to let the user know that they won rather than just have an LED go off (shown in Figure 20). The *Lose* LED (J15) indicates to the player that they have lost. It turns on when the player presses the incorrect button thus failing to repeat the sequence. Along with that the 7-segment display will display the word “*LOSE*” (shown in Figure 21). This was much easier to display because of the straight segments.

The display is used for many state of the game. In Figure 19 the displayed is used to display the sequence for the user to repeat. It does this by displaying a 0 and then going to the next display and displaying the next hardcoded 0 until the sequence ends. The display is also used to display *gOOd* or *LOSE*.

The buttons are used to repeat the sequence. So the buttons correspond to their respective display. For example, button H13 corresponds to display F17. Then the game utilizes a reset switch which resets the game to the beginning. At any stage of the game, when the player flicks the switch up and down, the game starts displaying the first sequence and then continuing on.

1. **Lessons Learned**

Throughout the process of planning, designing, and implementing our system, we gained valuable experience in the following areas: gaining familiarity with Verilog, solidifying electrical engineering concepts taken out of the classroom, working collaboratively and constructively as part of a group, and applying engineering principles to a real-world project.

First, by the end of the project, the members of our group had gone from having a shaky understanding of writing Verilog code to becoming comfortable with the language. It was at times difficult to forego prior years of experience with coding in imperative programming languages such as Python and Java, as trying to tackle Verilog in the same way led us time and time again to errors and warnings that were irresolvable without some level of design overhaul. Throughout weeks of "jumping into the deep end" so to speak with respect to designing Verilog modules from scratch and wiring components together, we were able to better reconcile the implicit yet extremely important differences between other programming languages and hardware description languages (HDLs) such as Verilog. Creating our system in Verilog was not about learning a new set of syntax as much as it was learning a new style of thinking. Not only did we learn how to construct an always block or how to resolve latches in our code, but we also learned the right way to approach a new programming paradigm that translated directly to logic gates on a physical device. For this reason, the approaches and design patterns we learned here will likely translate to future courses and employment prospects, even if Verilog isn't explicitly the language we are asked to work with.

Second, our approach to analyzing the constraints of the project and coming up with a feasible design solidified the principles taught in our Digital System Design course. We applied knowledge of finite state machines and their design through the use of state transition diagrams once we began to think about the implementation of our project. We also needed to consider the "three -Y's" of managing complexity: hierarchy, modularity, and regularity. The division of our system into many modules, each of which served a concrete purpose, satisfied the hierarchy and modularity characteristics. Many of our modules, perhaps with the exception of our specialized simon\_fsm module, also satisfied regularity; their inputs and outputs are general enough to be useful in other projects that need similar functionality with the buttons and display on any FPGA board. The careful and deliberate selection of the widths of registers by taking into account the upper limits of the range of binary numbers was yet another aspect that proved helpful during the scope of our project.

Third, juggling multiple schedules and following through on frequent communication was instrumental to the success of this project. From observing the collective hours logged, it is clear that this was not a task that could have been accomplished by a single member in the allotted time. Two different working styles were utilized throughout the project; sometimes, everyone would meet at once and share the burden of one aspect of the project, because having multiple pairs of eyes proved helpful. Other times, we delegated responsibilities on an increasingly individual basis and were able to work independently yet collaboratively to produce a cohesive system with different modules. We learned to alternate between both of these working styles in order to maximize productivity and minimize frustration with our project, and this will be a vital skill for future success in the classroom and workplace.

Fourth, the adoption of an ad-hoc Scrum methodology and commitment to the iterative improvement of a system with an ambitious scope allowed us to refine our ability to tackle problems within an engineering discipline. We learned to compromise certain features in the name of quickly rolling out a working prototype, because in these cases a critical design flaw would often be caught almost immediately and get resolved before continuing deeper into the project. At the same time, the importance of keeping our design in check in order to allow the addition of more functionality in a later sprint is something we learned toward the end of the project. Finally, we learned to be comfortable with the idea of scrapping old code that would otherwise bottleneck our progress toward a future version of the system (as was the case with our original FSM design). Each of these lessons will serve us especially well in our year-long senior design projects.

Although we didn't accomplish everything we had set out to do in our proposal, we are proud of our effort as well as the final result. Any redesign or future attempt at this or a similar project could only be improved as a result of the aforementioned skills we sharpened throughout the duration of this project.

1. **Conclusion**

All in all, we found that this project was the ultimate learning experience. Not only did we create a functional implementation of Simon that is playable compared to its real-life counterpart, we also learned a lot more the HDL, Verilog. We had to build a project from the ground up and it took a good amount of preparation, sprint meetings and teamwork to make this accomplishable. We also became really comfortable with the FPGA board through persistence and error corrections.

By completing this project, we learned a lot about implementation of a digital system with Verilog. At the same time, this project encompassed a good amount of the course from previous weeks. Completing this project gave us the utmost confidence in Verilog that we never had before, as we all came from backgrounds with little to experience using an HDL. From a feedback standpoint, this was a great way for us to learn Verilog; to put everything we had learned into one comprehensive project was an awesome learning experience. We were able to take everything we learned through the semester and utilize it in the creation of our complete final project. Ultimately, we crafted a prototype that each team member was proud of and gained a lot of experience using hardware to implement a digital system.

1. **Appendices**

**Appendix A: Verilog Warnings**



