Austin Jenkins

Mitchell Everett

CEC 287 Final Project: Simon

For our project we decided to make a version of the classic game Simon. The computer, Simon, creates a pattern of lights, which the user must replicate in the correct order. For each correct run through the pattern, Simon will add another light to the end of the pattern and present it again. Play continues until the user inputs an incorrect pattern or fails to input the pattern fast enough.

Our design is in the form of a Finite State Machine. The states are as follows:

* Add a random light to the pattern – State A
* Display the pattern – State B
* Accept and check user input – Actually consists of two states – States C and F
* Failed State – Flashes lights to denote failure, Also consists of two states – States D and E

The sequence is stored in the variable s. S is a standard logic vector of 1001 length. It stores each piece of the pattern as a 4bit number (e.g. the first light is “1000”, second “0100” and so on). The current size of the pattern is represented by q, which is incremented by four each time a new light is added to the pattern.

In state A, the currently generated random light is added to the pattern, in order to do this we increment q by four and assign the value of the next light to s(q downto q-3).

State B increments through the pattern and displays value through the red LEDs. The variable u is set to 3 before entering B, and is incremented until u = q, signifying that all piece of the pattern have been displayed.

States C and F accept user input. We used two states to cause a delay, giving the user time to input. This state also makes use of u to increment through the pattern, once u = q, the entire pattern has been input correctly. Simon then increments the current score by one. It then checks to see if the current score is higher than the high score. If it is, the current score replaces the high score and is stored into the appropriate variable.

States D and E are only entered when the user inputs incorrectly in state C. If the user is too slow the input is treated as all zeros, which is always incorrect and thus has the same effect as an incorrect entry. These states flash red lights in a pattern to tell the user that they have lost

Our design makes use of the four keys on the DE2-115 board, as well as several lights, switches and all HEX displays. The four keys are for the user input. Four red LEDs are used to display the computer generated pattern, and show the user’s input as it is entered. The red LEDs are turned on when Simon displays the pattern, or when the user is inputting their pattern. The user can only turn the lights on when they are in the input states of the FSM.

One green LED is on when the computer is generating and displaying values, and off when awaiting input. After each button is pressed the light turns green to indicate that the answer was correct and turns off to await the next bit of input.

Three switches are used to determine the difficulty level. No switches signifying level 1, with the three switches representing the other difficultly levels, prioritizing the most challenging difficulty. Each difficulty level has its own clock speed, harder difficulty means a faster clock, and thus faster input is required. Each difficulty saves its own high score value, which is retained when difficulty is switched. The difficulty can only be changed while the user is not currently playing.

The leftmost set of HEX displays is used to display the high score of the current difficulty. The next set is used to display the current score. The method used to display scores on the HEX displays is from an earlier project to convert a 4bit number to hex. So, when the right HEX would display A, we set it to zero and increment the left display. The final set of four displays shows the current difficulty level. While not currently playing, the user can cycle through the difficulties and see the high score for each.

We make use of three flag variables which we feel we should mention, as their purposes may not be immediately apparent. Aflag checks to see if this is the first run through the code, because q (the variable that keeps track of the array size) must be incremented each time through, except for the first time. Bflag is used to make the code loop through B twice before moving on to the input stage, in order to display the value and then reset the display to zeros. Without the flag, the lights display consecutively, so if the same light appears back to back in the sequence it will display for two cycles, but is hard for the user to distinguish. Cflag is used to determine whether the FSM is currently in state C, if the FSM is in state C, the user is inputting, and their input will affect the red LEDs. If Cflag is off, the user can have no impact on the red LEDs.

Randomly producing each step of the pattern was tricky and required us to borrow code from online.[[1]](#endnote-1) The code creates a pseudo random binary number by use of several shifters and XOR gates. Due to unknown reasons, the random number generator appears to break after generating a string of all zeros, to fix this, we generate more bits than necessary and change an irrelevant value to ‘1’. We then take the last two bits of the randomly generated number and use a decoder to create a one high value. This process runs on a much faster clock than the main game and thus produces enough pseudo random values in a normal clock cycle to be random enough for our purposes.

The project turned out very nicely, and we are very happy with the way it works as well as the way the game plays. The only feature we thought of that didn’t go into the final design is multiplayer. After having much more trouble with the overall design than anticipated, multiplayer turned out to be a much bigger project than expected, and we unfortunately ran out of time to implement the idea. Overall the project was really fun to work on, and we’re really glad is turned out the way it did.

1. http://vhdlguru.blogspot.com/2010/03/random-number-generator-in-vhdl.html# [↑](#endnote-ref-1)