CSE 369 Lab 8

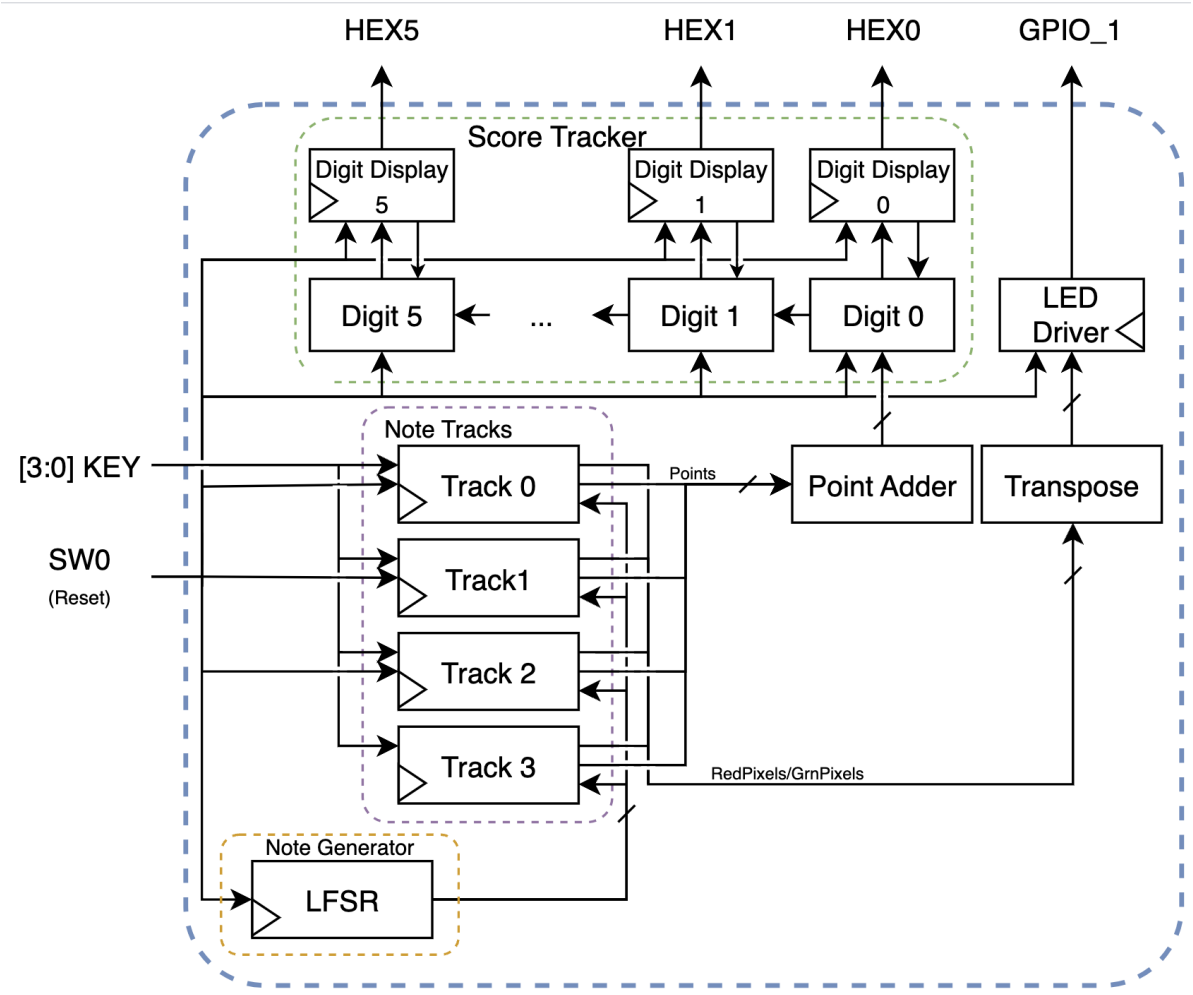
Final Project: DDR

Isaac Wu
2360957
December 7, 2024

1. Instructions

Once programmed, flip SW0 to reset the game to guarantee correct behavior. Then, on the LED Board, red notes will 'fall' from the top of the board to the bottom. There is a row of orange lights, which represent the hit zone. Your note inputs are the KEY buttons and your goal is to press the corresponding button when a note overlaps with the hit zone row.
If you press the input at the correct time, the hit zone light will turn green, and will be red otherwise. If you hit at the right time, then two points will be added to your score. If you are one row early, then you will get one point. If you are more than two rows early or just press an input when there is no note present, then two points will be deducted from your score. You can hold down a button to repeatedly press each clock cycle if there is a large column of notes in one track.

2. Top-Level Block Diagram



This is my Top-Level Block Diagram for the DDR module. This module takes in 5 inputs, KEYs 0-3 as input presses, and SW0 as a Reset switch. I chose not to synchronize the KEY inputs because it would introduce too much delay between when KEYs were pressed and when they were registered in the system. I also chose not to use an Edge Detector to make it easier for Users to hit multiple notes in a row.

I split each column of notes into their own separate Track module. Each track gets its own KEY as user input, and a 'note' input that instructs the module to create a new 'note' at the start of its range. The Track modules keep its state as if they were handling rows for simpler handling. At each rising clock edge, the track will shift its contents leftward. The track will keep track of when the input is registered and will reward or deduct points according to how accurate the input was to the note's position.

I used the exact same 9-Bit LFSR from Lab 7 as a randomizer. I took two bits from its resulting status each cycle and &'d them together and passed it into each Track's note input. Track 0: [6] & [3], Track 1: [2] & [5], Track 2: [4] & [8], Track 3: [7] & [1].

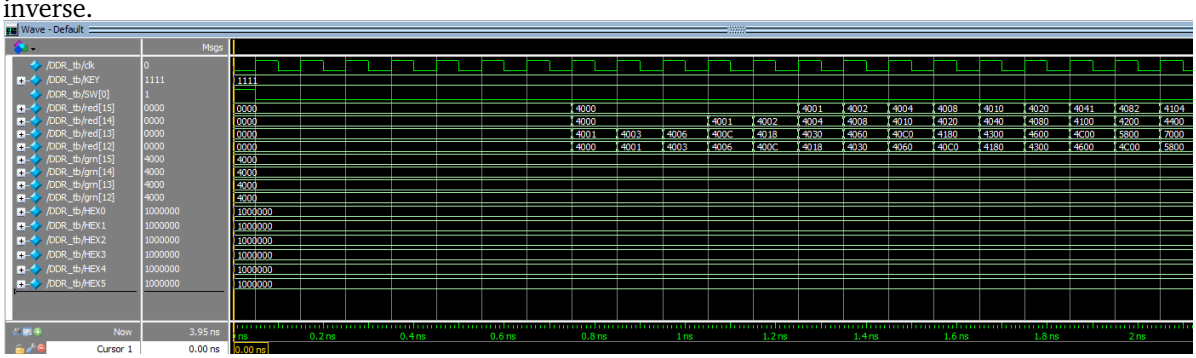
The light statuses from each Track module is combined together and then passed through a Transpose module to convert the rows of lights the Tracks keep track of into columns of notes to display. The resulting light matrices are then passed to the provided LED Driver (along with the reset input), which controls the connected LED Board through the GPIO_1 output.

The points from each track is passed to a Point Adder module, which calculates the total amount of points earned/lost this cycle. This total is passed into a chain of Digit modules.

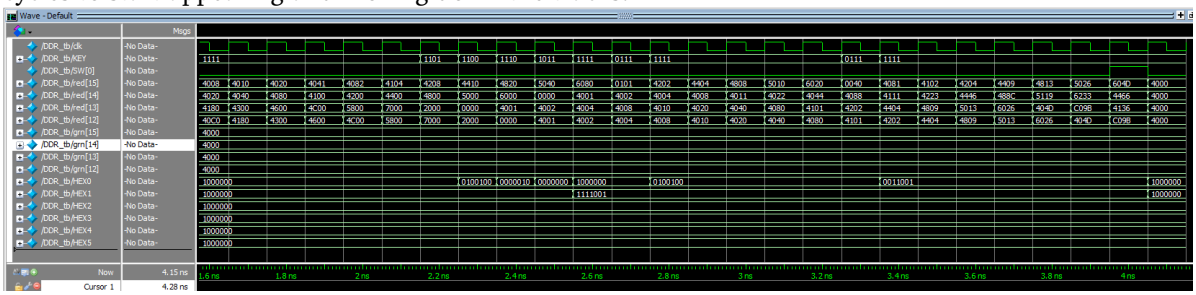
Each Digit module takes in a current state representing the digit that is currently being displayed (coming from the Digit Display module) and an input to add to the current total. This module is responsible for handling all of the overflow/underflow that occurs when the result is greater than 10 or negative. The resulting state is given to a matching Digit Display module. These modules simply keep track of their current state and outputs to a HEX display, which shows its state as a number between 0-9. The carry outputs from the Digit modules are passed down the chain as their inputs.

3. Top-Level Simulation

Red & Green rows are shown in hex here, and KEYs are 1s by default and the input handling takes the inverse.



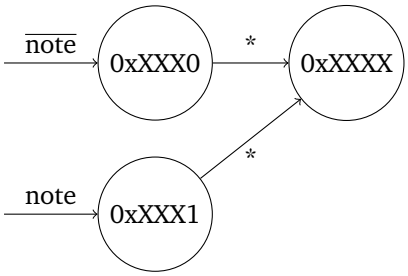
This is the behavior of the top-level module right after a reset input at the beginning. Notes take a few cycles to start appearing and moving down the tracks.



Here, KEYs are pressed on time when notes are at the hit zone, updating the scores displayed on the HEXs.

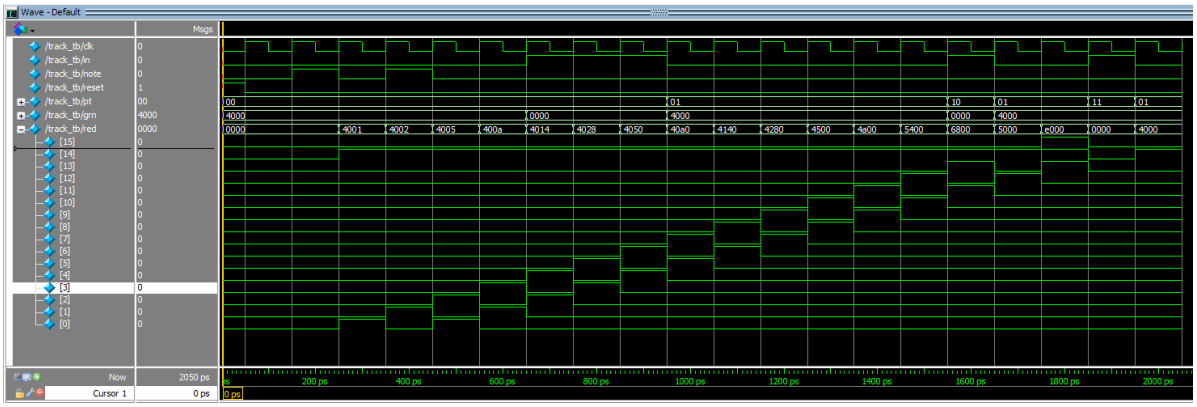
4. Track Module

4.1. Finite State Machine



I am not entirely sure what the correct method is for making this FSM. If I were to draw it accurately it would have 65,536 states and way too many transitions. The Track module uses a [15:0] buffer that adds a note if it is input at the end and shifts its data to the left after each clock cycle.

4.2. Simulation

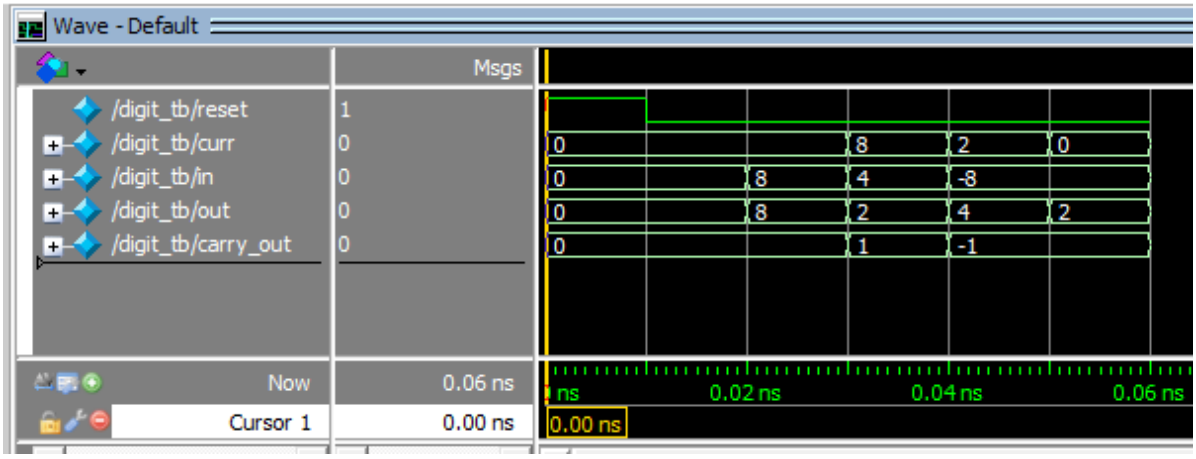


I reset the module at the beginning. Then, I input two note inputs to create two new notes. You can see them appear on row 0 of the red matrix one clock cycle afterwards. The red and grn matrices are displayed in this diagram as a collection of rows represented in hex. These notes move up one row after each clock cycle. Some time after, I enter an erroneous user input to simulate miss behavior. This results in the green LEDs in the hit zone turning off to show red lights in that row and a Miss encoding in the point output (00). When no input is being pressed, the point output changes to Idle (01). After some more time, an input is pressed one row too early (as seen, row 14 is the hit zone and at this cycle, the note is on row 13), resulting in Early points (10). Then, the input is pressed right when the note is on row 14. This results in the red light in the hit zone row turning off to display green. The points output also changes to a Hit (11). Notice that on the next clock cycle, the hit note does not continue on to row 15.

5. Point Adder Module

In this test bench, I demonstrate the proper decoding from the input to their correlated points. As a reminder, 00 means a Miss (-2 points), 01 is Early (+1 points), 11 is a Hit (+2 points), and 10 is Idle (0 points). This module just takes each input in pts and adds them together. The total output is displayed as a signed integer.

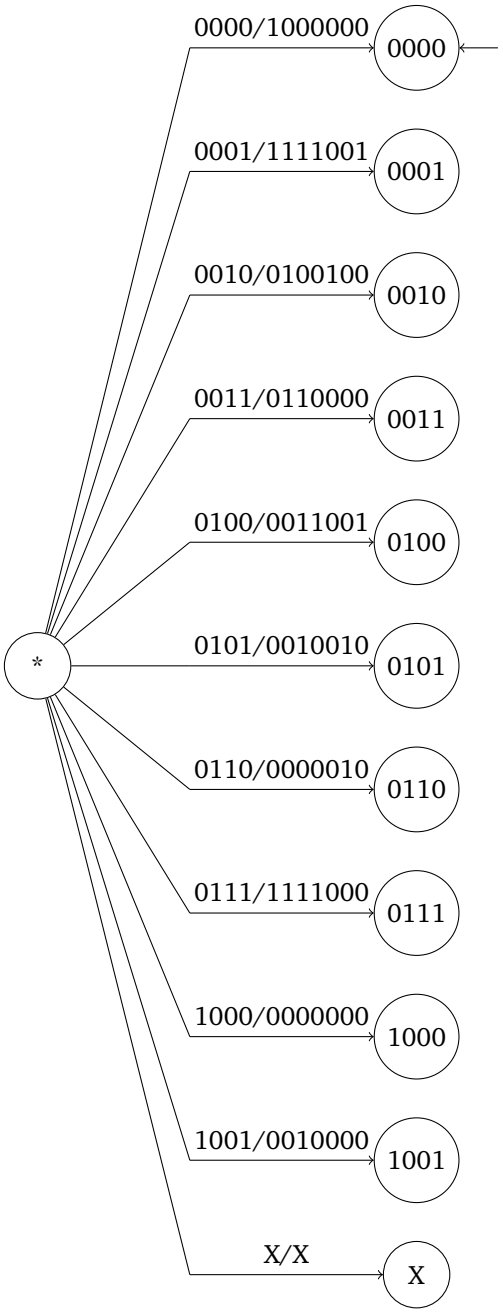
6. Digit Module



This module takes the current state, adds the input, and outputs the corresponding digit and any carry outputs. The current state and output state is displayed as an unsigned integer (since the digit state is between 0 and 9). The input is displayed as a signed decimal (because the point total can be as low as -8 and as high as 8). The carry is also displayed as signed. This test bench demonstrates when the current state is 0, overflow, and underflow.

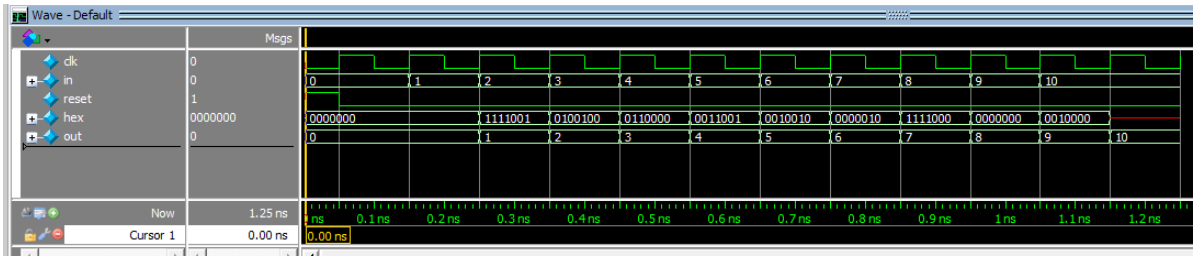
7. Digit Display Module

7.1. Finite State Machine



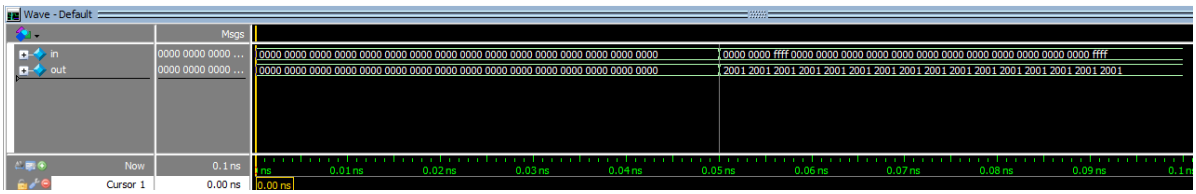
The * state represents any state. In this module, no matter the present state, the next state is dependent only on its input. The output is the HEX display for that digit, which is defined for digits 0000 to 1001, and is undefined for digits 1010 to 1111.

7.2. Simulation



This simple test bench first resets the module, then increments the input state to 10. One clock cycle after the input changes, the HEX output switches to display the state and its state is output accordingly.

8. Transpose Module



The matrices in this diagram are displayed in hexadecimal. In this test bench, I set two rows of the input module, 0 and 13, to all 1's. The output shows the equivalent transposed matrix, with columns 0 and 13 set to all 1's, represented by each row in the output being 0x2001.

9. Testing

My method for testing my project was to test each module individually before combining them together. For example, the Digit modules I created had to be re-made three times to organize how the base 10 overflow was being handled, the HEX display output was being updated, and what module was keeping track of the internal state of the digit. Then, testing the carry output chaining into the other digit modules had to be tested as a group, and only after that could I be confident my Digit module was sufficient enough to use in my top-level module.

10. Misc.

How many hours (estimated) it took to complete this lab in total, including reading, planning, designing, coding, debugging, and testing.

It took around 26 hours in total to complete this lab.