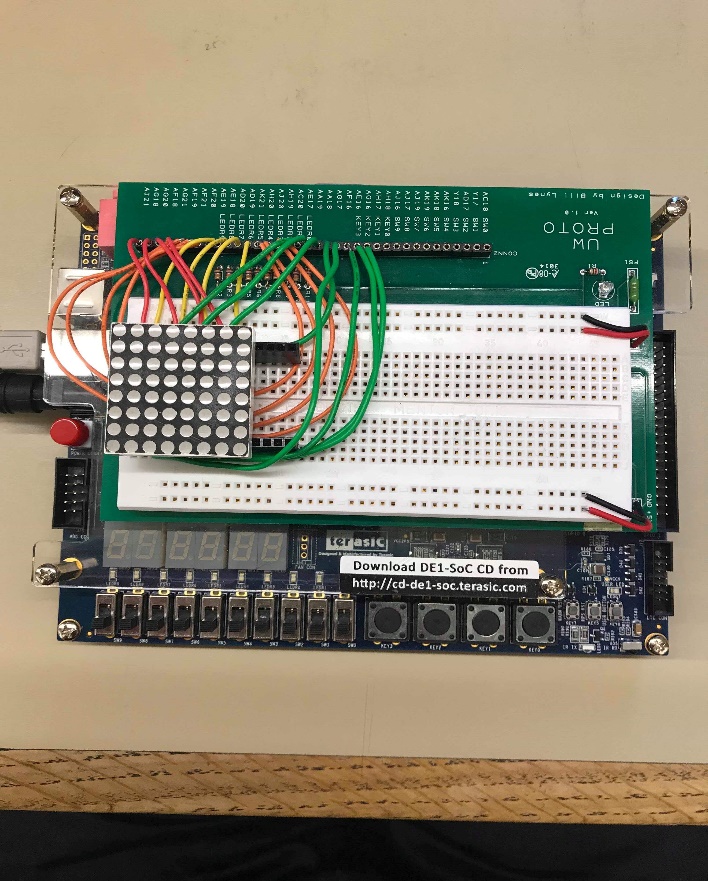
Seungjae moon

EE271 Final Project: Frogger

**Procedure**

I approached the Frogger design problem by creating a top-level block diagram of the modules I would need based on the given project description and the 1981 KONAMI Frogger Game.

|  |
| --- |
| KEYS  Clock (counter)  metastabilityFree  userInput  directions  patterns  startRowFrog  startFrog  frog  cars  scoreCalculator  checkFrog  \*reset information back to userInput, frogs, and cars  seg7  \*led\_matrix\_driver gets updated corresponding to the contents in the green and red array |



LED Board Setup on the FPGA

The top-level gets the input from the KEYS and passes it through the metastabilityFree and userInput modules to equate the button press to the 4-bit move variable, where KEY3 through 0 corresponds to left, up, down, right, respectively. Then the information about the user input is passed into the different frog modules to store the position of the frog based on the green\_array. The frog has places where it behaves differently. The index [3][0] of the green\_array is where the frog starts, so the light must be ON at reset (behaves like center light from Tug-of-War). This state machine is stored in the frog module. The other indices of the green\_array is OFF to begin with (behaves like the normal light from Tug-of-War). However, startRowFrog uses a different logic because the bottom row should not do anything when the user presses the ‘down’ key.

On the other hand, there is logic for the cars stored in the red\_array. There is a preset patterns and directions for the cars shown below. Then, the red lights move to the given direction at each clock cycle. Overall, each 128 indices of the green and red array have their own modules, which is simplified by the ‘generate’ syntax. The wrap around of the green\_array and red\_array on the leftmost and rightmost columns are dealt with using modulus where [i+1]%8 and [i+7]%8 in the generate for loops allows for the wrap around.

Lastly, the checkFrog does a bitwise OR with all 64-bits of the frog (green\_array) and the cars (red\_array) to see if any of them are ON at the same index, which corresponds to a collision. This crashed output which triggers a reset. Similar is true for the survive where the ‘up’ key and the entire top row is checked to output a survived bit. The score incrementation and the seg7 is changed slightly, so the max number of win is now 9 (4-bit), compared to 7 (3-bit) before.

Frog FSM Diagram

|  |
| --- |
|  |

Initial Car Pattern and Direction

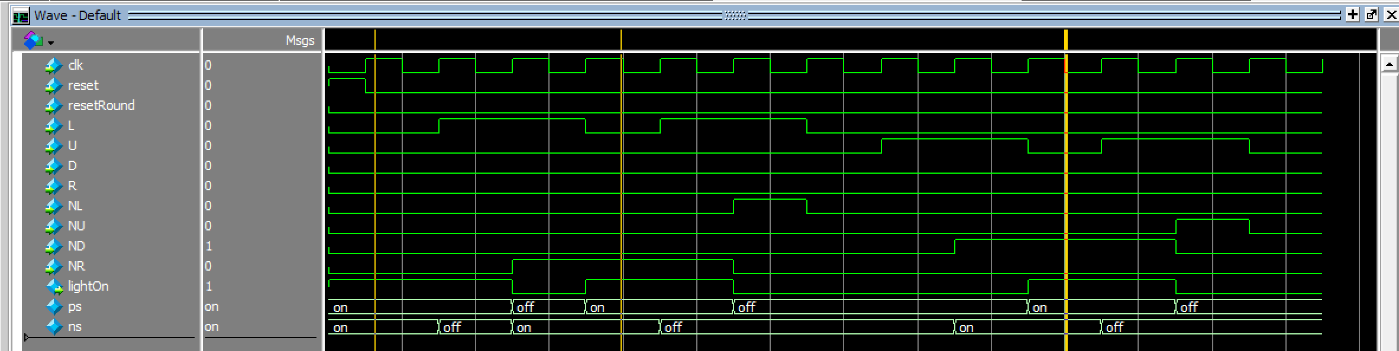
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| LEFT | ● | ● |  |  | ● |  |  | ● |
| RIGHT |  | ● | ● |  |  | ● | ● |  |
| RIGHT | ● | ● | ● | ● |  |  | ● |  |
| EMPTY |  |  |  |  |  |  |  |  |
| LEFT |  | ● |  |  | ● |  | ● | ● |
| RIGHT |  | ● | ● |  |  |  | ● |  |
| LEFT | ● | ● |  |  |  | ● |  |  |
| EMPTY |  |  |  |  | ● |  |  |  |

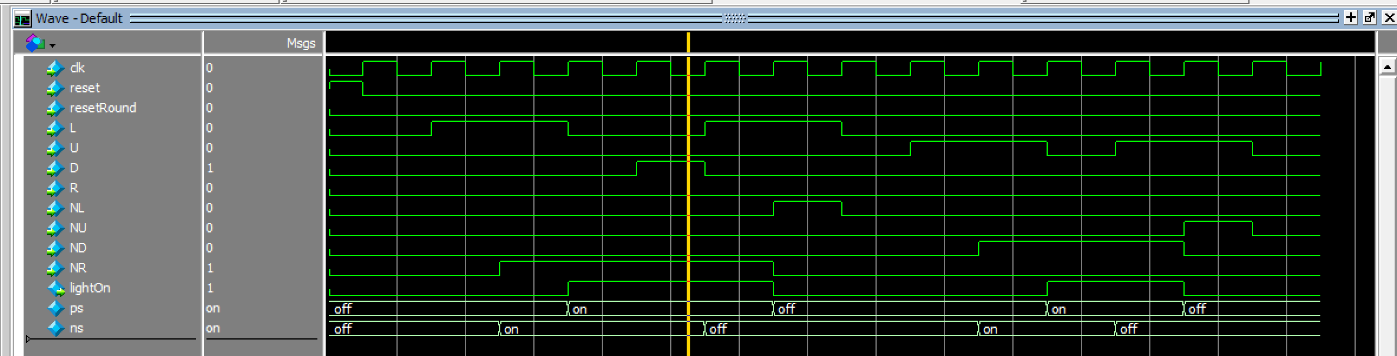
**Results**

Each module was tested on ModelSim using the corresponding testbench module. For metastabilityFree.sv and the userInput.sv, the code and the ModelSim are identical as the Tug-of-War. Note that like before, when the game is completed by dying or completing maximum of 9 rounds, the user is disallowed to move by using the needReset logic.

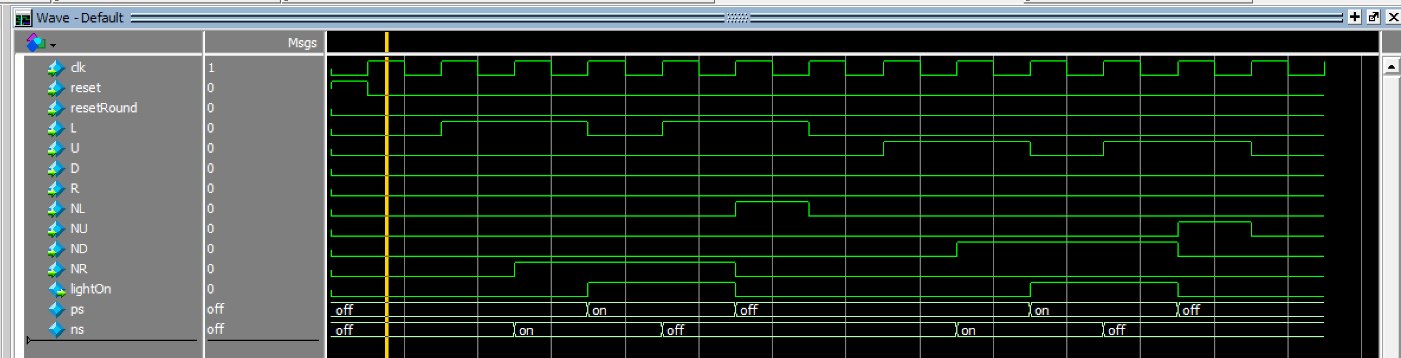
The different type of frog.sv is also similar to the center and normal lights, except now it has information about U, D, NU, ND (up, down, next up, next down). The leftmost cursor on the startFrog ModelSim shows that light is ON from the start as expected. Also, the next two cursors show that the light is ON when L and NR is ON and when the U is ON and ND is ON during the clock cycle before. The light is OFF everywhere else when the adjacent light ON and the opposite press do not happen simultaneously. The cursor on the startRowFrog ModelSim shows that the down key press does not do anything. Lastly, the cursor on the regular frog.sv shows that it is identical to the startFrog other than its reset state, which the frog.sv starts with its light turned OFF at resest.

startFrog.sv



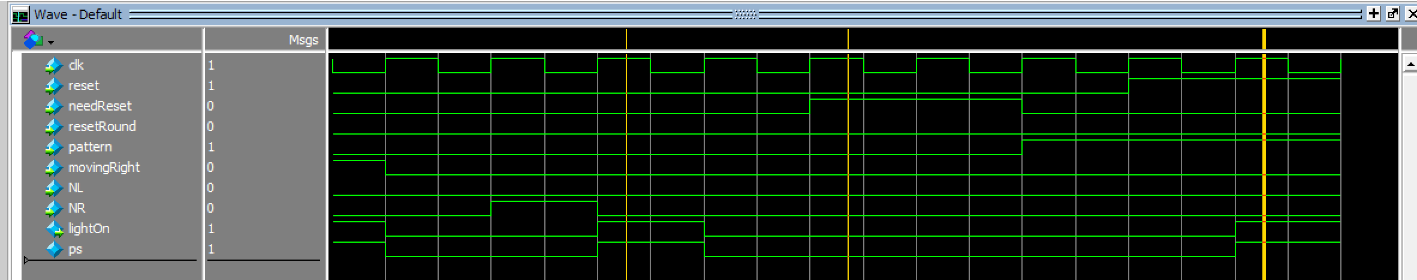
startRowFrog.sv

frog.sv



The leftmost cursor in the cars.sv shows that when the movingRight is 0 (thus moving left) and the NR is ON the clock cycle before, the light is ON in the present state, ps. The next cursor shows that when the cars.sv is in needReset due to the game being over, nothing happens. The last cursor shows that when the default pattern is 1 at that position, the light turns ON at reset.

cars.sv



The checkCrashed module was logically the most involving. Trying to compare each bit to see if any of the green\_array bits and red\_array bits are both 1’s at the same index was difficult, but it was solved using the advanced Verilog features. The middle cursor on the ModelSim below shows that when the frog has an overlapping 1 at any of the bits, in this case row 8, the crashed is true. This behavior is verified at a different row, row 2 by the rightmost cursor. The leftmost cursor shows that the frog survived if the frog has survived if the frog is on row 8, which is the last row, AND the user has pressed up. I also have an additional condition for survived where it must have not crashed either at the last row, for safety.

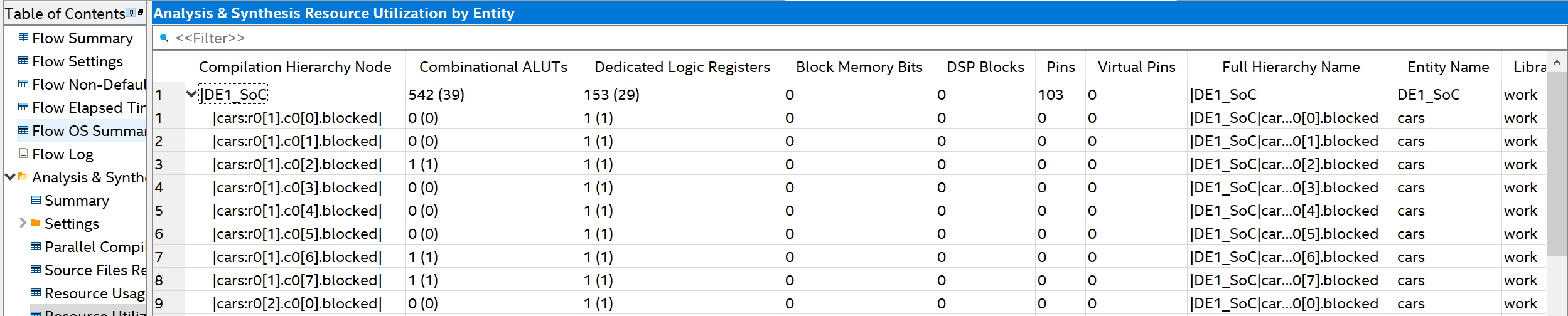
checkCrashed.sv



The ModelSim for the scoreCalculator and the seg7 code was nearly identical to the Tug-of-War, and the ModelSim for the DE1\_SoC was not required due to the 64-bit arrays being not so easy to read on the simulation.

The size of my design was computed in terms of #FPGA logic and DFF resources from the “Resource Utilization by Entity” page (with the subtraction of the clock divider):

542 + 153 – (5 + 5) = 685.



**.**

**.**

**.**



**.**

**.**

**.**

Overall, the finished project accurately modeled Frogger by having the correct LED matrix behavior for all input combinations that includes the key, input, reset switch, and other logics.

**Problems Faced and Feedback**

There were no major problems and feedback other than that getting used to the generate statement was difficult. Now that I know how to utilize generate and other advanced Verilog tools, it will help me immensely in my future SystemVerilog courses. Otherwise, the project was very engaging because students could make their own design choices.