Snake Game Implemented in CMOS Static Logic

Snake_8x8: Aden Briano, Matthew Nutt, Renee Wrysinski

I. Introduction

This is a basic version of the classic Snake game coded in Verilog, synthesized into gate-level logic, and implemented in a physical VLSI layout. The final chip requires a dual-phase clock, reset signal, and directional signals as inputs, and outputs control signals for display multiplexing on an eight-by-eight LED array. For the sake of concise simulations, only one cycle of multiplexing is used (i.e., eight clock cycles). Game updates occur after every multiplexing phase, so if this chip were to be fabricated and run on a multi-megahertz clock, it is imperative to first adjust the number of multiplexing cycles so that the game updates occur at a more reasonable rate.

Dual-phase clocking is used for system synchronization. Two clock signals, *clka* and *clkb*, are patterned such that one signal completes a rising and falling cycle while the other remains low, then vice versa. All flip-flops update on the falling edge of the clocks, but input registers holding intermediate values are updated on the falling edge of *clka*, whereas output registers are updated on the falling edge of *clkb*. This way, timing discipline can be maintained between all the modules of the project.

This project uses the standard cell library from Oklahoma State University for the AMI $0.5\,\mu\mathrm{m}$ process.

GitHub Repository: https://github.com/MatthewNutt5/snake

II. Verilog Modules

A. Controller Module

The Controller Module manages the synchronization between modules for the game. It contains three sub-FSMs:

1) The game FSM, showing whether the game is ready, started, or ended,
2) The direction FSM, keeping track of which direction the snake is moving, and
3) The execution FSM, synchronizing the phases of game updates and display multiplexing.

The inputs to the module are:

- Global clock and restart signals,
- Game update signals from the logic datapath, and
- A one-hot direction signal.

The outputs from the module are:

- Each sub-FSM's current state,
- Game state and direction signals to the logic datapath, and
- Control signals for a multiplexed 8x8 LED display.

Fig. 1 shows state transition diagrams for each sub-FSM in the module. Internally, the module stores each state and iterates through the state transitions according to the module's inputs.

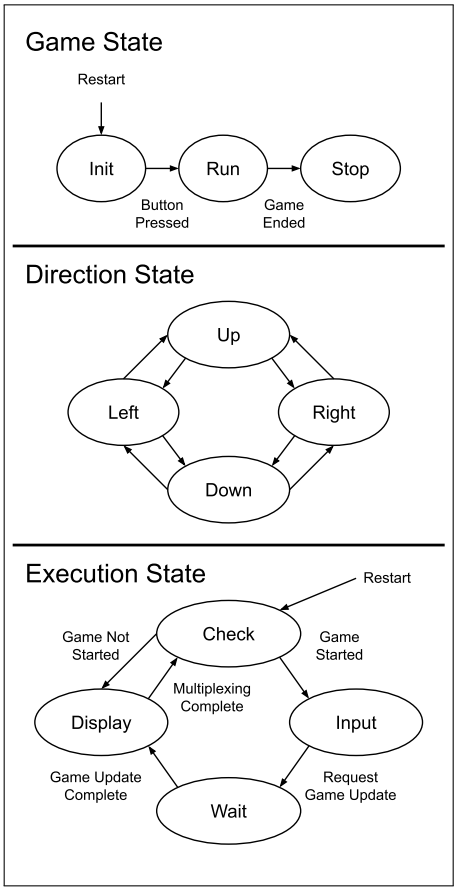

Fig. 1. State transition diagrams of each FSM in the Controller Module.

B. Logic Datapath

The Logic Datapath keeps track of game data, facilitates game updates, and produces signals for displaying the game board.

The inputs to the module are:

- Global clock and restart signals,
- Game state and direction signals from the controller module, and
- The current random number from the PRNG module.

The outputs from the module are:

- Game update signals to the controller module,
- The current state of the game board expressed as an array of LEDs, and

- A signal to the PRNG for requesting a new random number.

The internal module storage includes:

- A 64-long, 6-wide shift register holding the position of each snake body piece,
- 6-bit registers holding the position of the apple and the current snake length, and
- A 64-bit register representing the display array.

The module performs the following calculations each time a game update is requested:

1) Calculate the new head position based on the current position and the directional input.
2) Iterate through the body-location shift register starting at the tail piece, shifting each register and checking for collisions with the new head position.
    a) If a collision is detected, send the controller a signal indicating that the game has ended.
3) Check for a collision between the current apple position and the new head position.
    a) If a collision is detected, iterate the snake's length and request a new apple position.
4) Update the apple position with the current random number (will not change if apple hasn't been eaten).
5) Update the display array with the new data from the game update.

C. PRNG Datapath

The PRNG Datapath manages an LFSR-based random number generator.
The inputs to the module are:

- Global clock and restart signals, and
- The request signal from the logic module.

The sole output is the current random number.

Internally, the module stores the random number in a 6-bit shift register, and each time an update is requested, updates the register with the polynomial $x^6 + x^5 + 1$ in order to generate numbers with the maximum-possible period.

D. Top Module

The Top Module integrates the above modules into one complete unit. It distributes the global inputs to each module, and interlinks the internal signals between the modules.
The inputs to the module are:

- Dual-phase clocks generated by an external oscillator,
- A restart signal provided by an external button, and
- A one-hot direction signal also generated by buttons.

The outputs from the module are:

- The multiplexed display signals, and
- All internal signals between the modules.

The latter is not necessary for the function of the game, but is useful for debugging with testbenches. The structure of the top module, i.e. the connections between the modules, are drawn as a block diagram in Fig. 2.

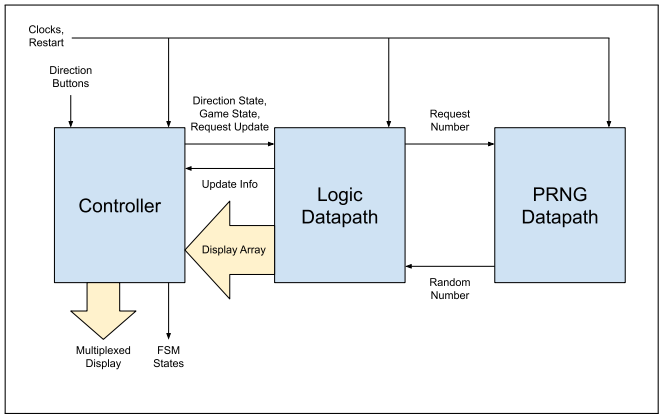

Fig. 2. Block diagram of top module.

III. Verilog Testbenches (Pre-Synthesis)

A. Controller Module

Before the design was synthesized, the Verilog files were verified with a series of testbenches to ensure they functioned as expected. Each module had its own testbench checking its basic functions, and the testbench for the top module was used to test the behavior of the overall game.

The Questa simulation result of the Controller module testing is shown in Fig. 3. Significant events are labeled; the following are the events that occur at each label:

- t_1: Restart signal goes high, resetting all registers. With no new inputs, the controller repeatedly loops through the multiplexing cycle to display the initial conditions. The game is not yet started (*out_game_state* = 0).
- t_2: A new direction is input.
- t_3: The controller responds to the new direction, updating the direction state and game state to indicate that the game has started (*out_direction_state* = 0, *out_game_state* = 1). In a few cycles, the controller will tell the logic module to update the game, and multiplexing will continue once the logic module has finished.
- t_4: A new direction is input.
- t_5: At the end of a game update, the logic module indicates a collision. The controller updates the game state to the game end condition *out_game_state* = 2, and will send a signal back to the logic module to blink the snake head's position.

B. Logic Module

The Questa simulation result of the Logic module testing is shown in Fig. 4. Note that the random numbers are not actually random; they were fabricated to minimize the length of the simulation. Significant events are labeled; the following are the events that occur at each label:

- t_1: Restart signal goes high, then low. (*out_random_num* = 011101). This is the first location of the apple.
- t_2: Snake begins to move right (*out_direction_state* = 3) towards location of first apple.
- t_3: Snake eats first apple. New random number is requested (*out_request_rand* goes high). New apple spawns

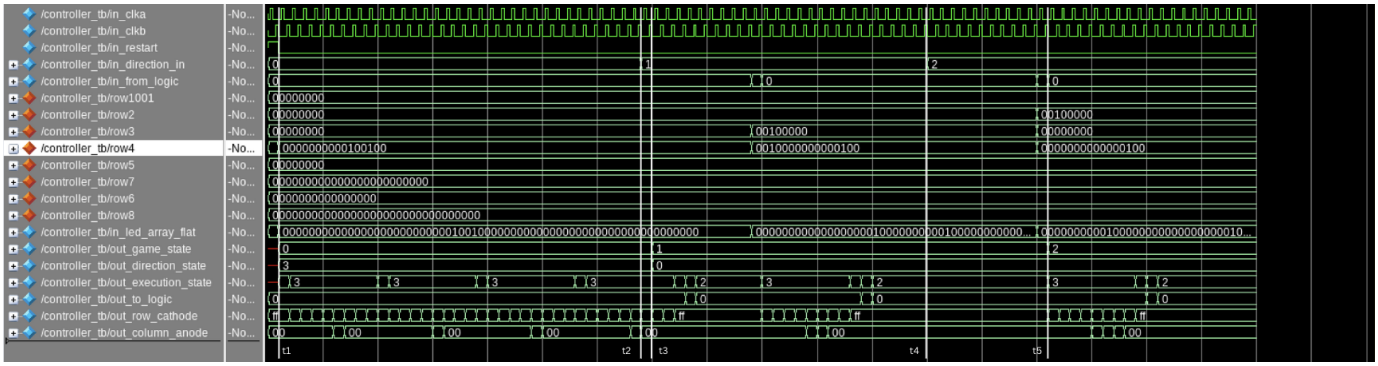

Fig. 3. Pre-synthesis Questa simulation of Controller module.

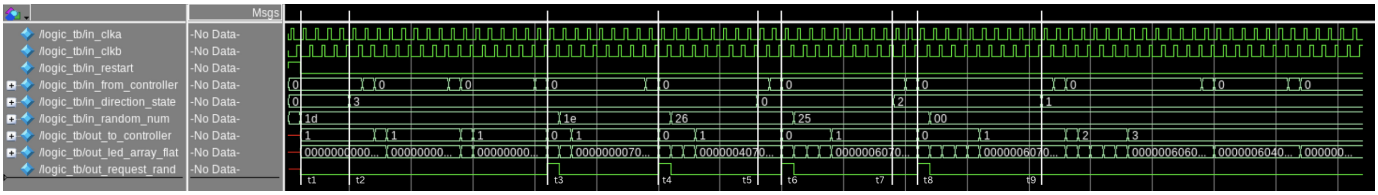

Fig. 4. Pre-synthesis Questa simulation of Logic module.

(*out_random_num* = 101110). Snake continues to move right.

- t_4: Snake eats second apple. New random number is requested (*out_request_rand* goes high). New apple spawns (*out_random_num* = 100110).
- t_5: Snake begins to move up (*out_direction_state* = 0) towards location of third apple.
- t_6: Snake eats third apple. New random number is requested (*out_request_rand* goes high). New apple spawns (*out_random_num* = 100101).
- t_7: Snake begins to move left.
- t_8: Snake eats fourth apple. New random number is requested (*out_request_rand* goes high). New apple spawns (*out_random_num* = 000000). The snake is readyv to kill itself.
- t_9: Snake begins moving down and eventually kills itself and signals game end(*out_to_controller* = 3).

C. PRNG Module

The Questa simulation result of the PRNG module testing is shown in Fig. 5. Significant events are labeled; the following are the events that occur at each label:

- t_1: Restart signal goes low (*out_random_num* = 011101). This is the first location of the apple and the seed for the PRNG.
- t_2: Snake eats first apple. New random number is requested (*in_request_rand* goes high).
- t_3: New apple spawns (*out_random_num* = 101110).
- t_4: Snake eats second apple. New random number is requested (*in_request_rand* goes high).
- t_5: New apple spawns (*out_random_num* = 110111).
- t_6: Snake eats third apple. New random number is requested (*in_request_rand* goes high).
- t_7: New apple spawns (*out_random_num* = 011011).

D. Top Module

The Questa simulation result of the Top module testing is shown in Fig. 6. Significant events are labeled; the following are the events that occur at each label:

- t_0: Restart signal goes high, resetting all registers. Directional input starts the game (*out_game_state* = 1). Snake begins to move right (*out_direction_state* = 3) towards location of first apple.
- t_1: Snake eats first apple. New random number is requested (*out_request_rand* goes high), and next apple appears at location determined by random number . Snake is now 2 pixels long.
- t_2: Snake begins to move up (*out_direction_state* = 0).
- t_3: Snake begins to move right.
- t_4: Snake eats next apple. New random number is requested, and next apple appears at location determined by random number. Snake is now 3 pixels long.
- t_5: Snake begins to move up.
- t_6: Snake begins to move right.
- t_7: Snake eats next apple. New random number is requested, and next apple appears at location determined by random number. Snake is now 4 pixels long.
- t_8: Snake begins to move down (*out_direction_state* = 1).
- t_9: Snake begins to move left (*out_direction_state* = 2).
- t_{10}: Snake eats next apple. New random number is requested, and next apple appears at location determined by random number. Snake is now 5 pixels long.
- t_{11}: Snake begins to move down.
- t_{12}: Snake begins to move right.
- t_{13}: Snake begins to move up towards its own tail.
- t_{14}: Snake collides with its tail, and game ends (*out_game_state* = 2).

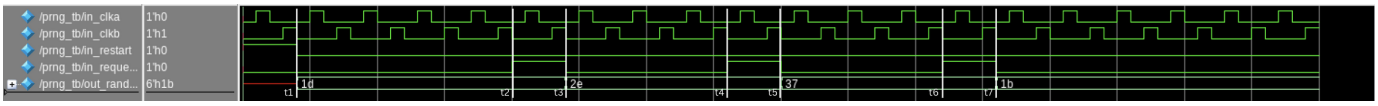

Fig. 5. Pre-synthesis Questa simulation of PRNG module.

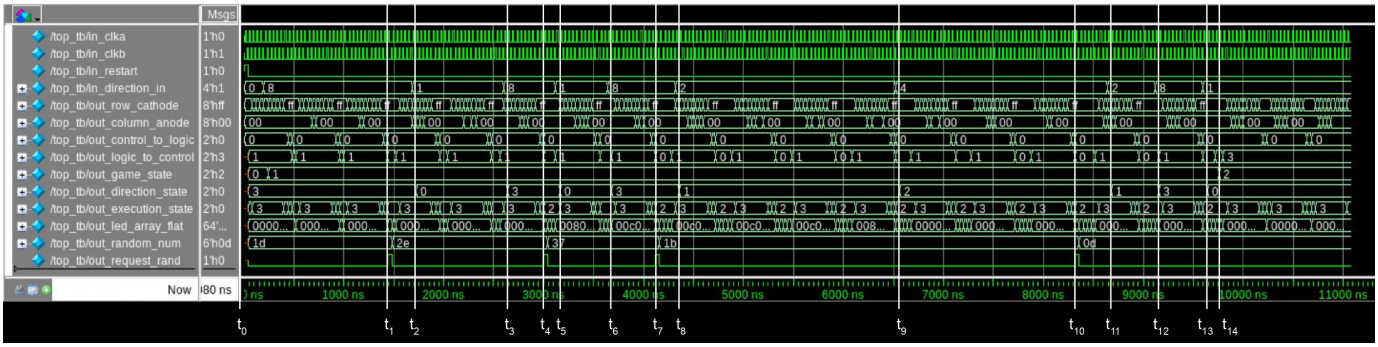

Fig. 6. Pre-synthesis Questa simulation of Top module.

IV. Synthesis

The Verilog description was synthesized with Design Compiler, set to a target frequency of 20 MHz. This produced a design with a total of 3435 cells and an estimated power consumption of 3.8514 mW.

V. Verilog Testbenches (Post-Synthesis)

A. Controller Module

After synthesis by Design Compiler was performed, the Controller module was once again simulated in Questa with the testbench described in Subsection III-A. Figure 19 (contained in XI. Appendix) shows the annotated simulation results. The simulation results were the same as the Questa simulation performed before synthesis, confirming that no functional issues were created by synthesis.

B. Logic Module

After synthesis by Design Compiler was performed, the Logic module was once again simulated in Questa with the testbench described in Subsection III-B. Figure 20 (contained in XI. Appendix) shows the annotated simulation results. The simulation results were the same as the Questa simulation performed before synthesis, confirming that no functional issues were created by synthesis.

C. PRNG Module

After synthesis by Design Compiler was performed, the PRNG module was once again simulated in Questa with the testbench described in Subsection III-C. Figure 21 (contained in XI. Appendix) shows the annotated simulation results. The simulation results were the same as the Questa simulation performed before synthesis, confirming that no functional issues were created by synthesis.

D. Top Module

After synthesis by Design Compiler was performed, the Top module was once again simulated in Questa with the testbench described in Subsection III-D. Figure 7 shows the annotated simulation results. The simulation results were the same as the Questa simulation performed before synthesis, confirming that no functional issues were created by synthesis.

VI. Chip Layout

Following post-synthesis verification, the design was ready to be placed and routed through Innovus. However, the initial core size generated by the standard density setting of 0.6 was too large to fit within the desired 64-pin padframe housing, as can be seen in Fig. 8.

```
Root cell box:
          width x height  (   llx,  lly  ), (   urx,  ury  )  area (units^2)

microns:  1481.400 x 1452.000  ( 4.200,  0.000), ( 1485.600,  1452.000)  2150993.000
lambda:   4938.00 x 4840.00  ( 14.00,  0.00 ), ( 4952.00,  4840.00)  23899920.00
internal:   9876 x 9680    (    28,  0    ), (  9904,  9680 )  95599680
%
```

Fig. 8. Original size of core as described by Magic.

To decrease the size of the design, the density target in the configuration file, which can be seen in Fig. 9, was altered to 0.75. This increases the density of placed cells and the routing between them.

```
# Floorplan the design with a 1.0 square shape and 0.8 density
# Leave 21 micron space around core for power and ground rings
# If errors occur, such as "ENCSP-2002" or "ENCSP-2020" and the layout does not
# complete, then reduce density from 0.8 in floorPlan to 0.6.
floorPlan -r 1.0 0.75 21 21 21 21
```

Fig. 9. Density target in configuration file.

The finished place-and-route from Innovus can be seen in Fig. 10. There are no shorts or overlaps between cells, which are issues that might arise when increasing the target density to values too close to 1.

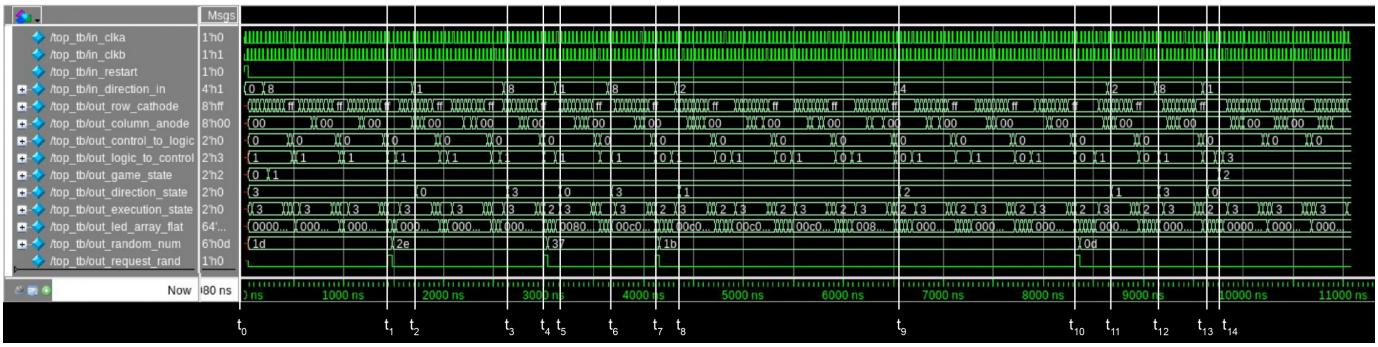

Fig. 7. Post-synthesis Questa simulation of Top module.

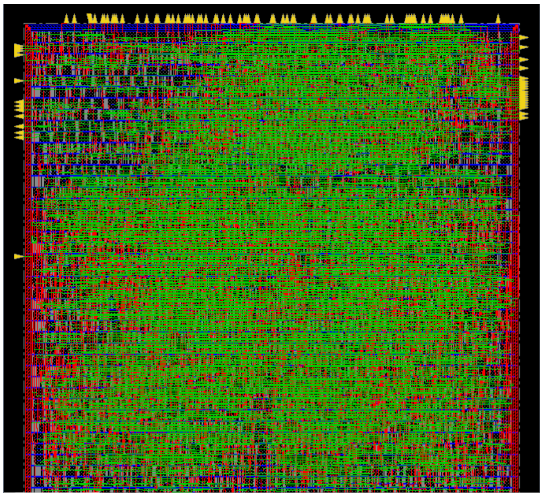

Fig. 10. Innovus place-and-route of core.

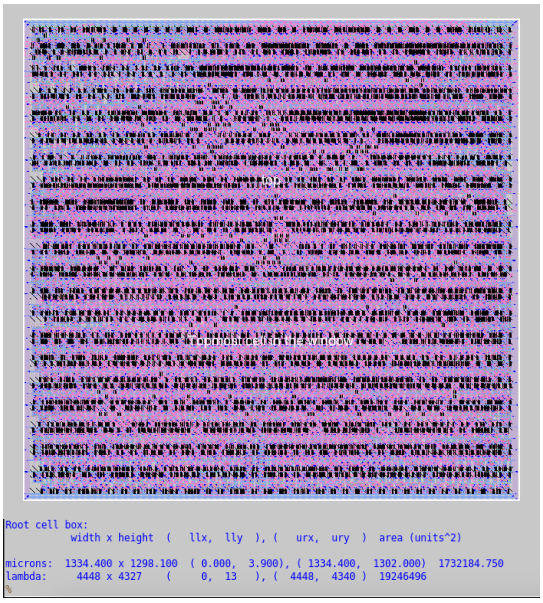

Fig. 11. Finalized core in Magic.

Finally, the layout file created by Innovus was ready to be read into Magic. The finalized core and size can be seen in Fig. 11. The core is now ready for testing, which is covered in the following section.

VII. Core Testbench

After chip layout, Magic generated a behavioral file for simulating the design. A testbench file for use in IRSIM was created from the Verilog testbench with the help of a custom Python script that converts text from a Verilog testbench to the format of an IRSIM test vector file, so the exact same sequence of events was tested. Fig. 12 shows the annotated IRSIM result of the simulation of the behavior of the core when provided with the inputs described in the core testbench. The labels correspond to the same events described in Section III-D.

VIII. Padframe Integration

After successfully importing the core into Magic and testing to ensure all functionality is as expected, the core was fully prepared for injection into a custom 64-pin padframe. It is important to note the exact configuration of the bidirectional pad that was used in the construction of the 64-pin padframe, which can be seen in Fig. 13.

The pad seen in Fig. 13 demonstrates the behavior seen in Fig. 14.

In other words, when OEN is HIGH, DO is written to PAD at the top of the bidirectional pad, and when OEN is low, PAD is written to DI. Also in the 64-pin padframe are VDD and GND pads, whose purpose is to take external VDD or GND and provide it to the core VDD and GND rails, as well as set all OEN flags accordingly. The VDD pad can be seen in Fig. 15. The GND pad is constructed the same.

As mentioned above, inputs to the top module are in the form of a two-phase clocking discipline, *clka* and *clkb*, four buttons for the direction input from the user, and one button for the restart input from the user. The only outputs from the top module are 8-row cathode and 8-column anode signals, which conduct a typical LED multiplexing outline. After inserting the core into the 64-pin padframe, which can comfortably house a core up to 1400x1400 microns, wiring all relevant pads to the inputs and outputs of the core accordingly, and setting all unused pads to input pads, the complete design follows.
The integrated core with pin mapping can be seen in Fig. 17.

I/O Pin and Signal Description

- *clkb* is connected to input pin 31.

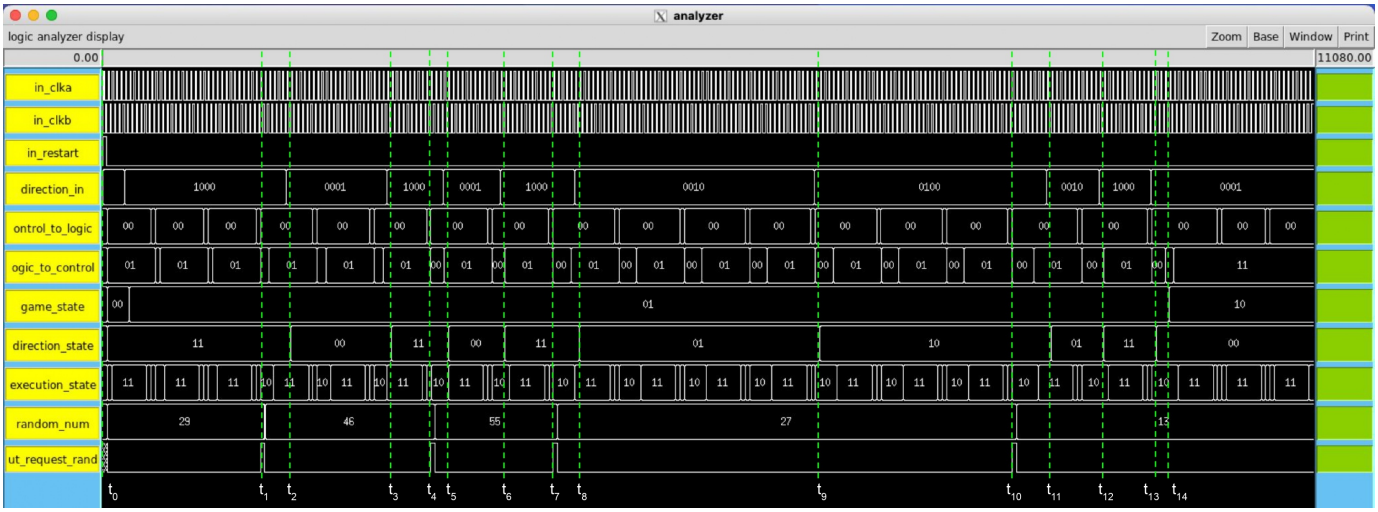

Fig. 12. IRSIM simulation of the core design.

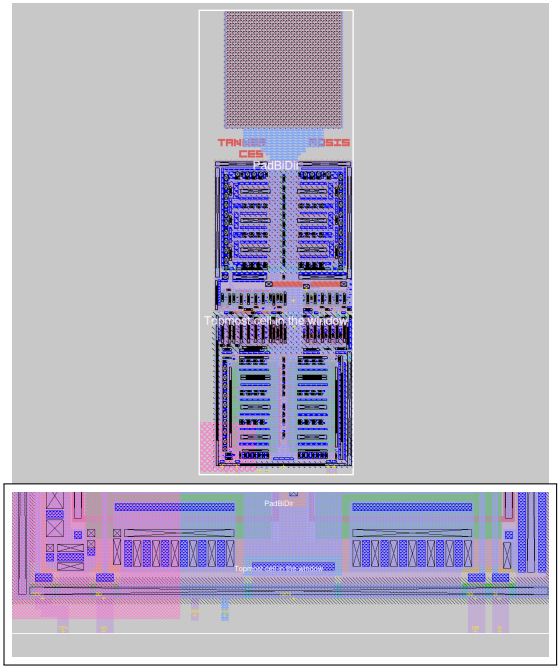

Fig. 13. Bidirectional Pad provided by OSU.

OEN	PAD	DI	DO
0	1	1	X
0	0	0	X
1	1	X	1
1	0	X	0

Fig. 14. Bidirectional Pad Behavior.

- The restart signal is connected to input pin 27.
- The UP direction signal is connected to input pin 25.
- The DOWN direction signal is connected to input pin 24.
- The LEFT direction signal is connected to input pin 23.
- The RIGHT direction signal is connected to input pin 22.
- *clka* is connected to input pin 21.
- *row_cathode7* is connected to output pin 20.

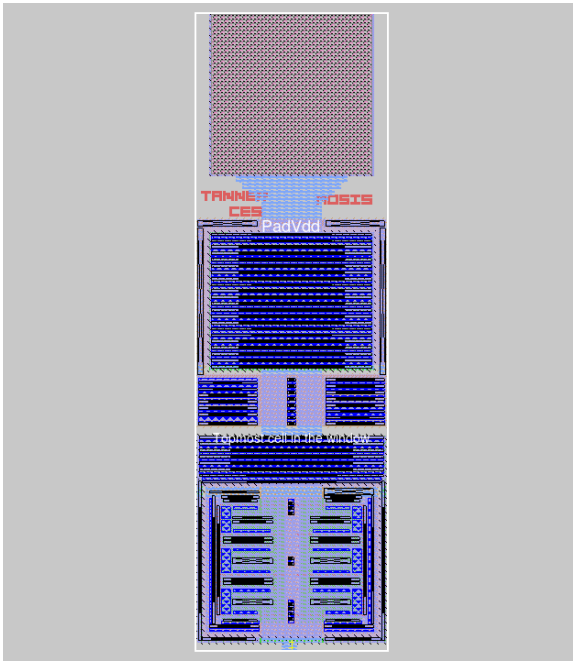

Fig. 15. Vdd pad provided by OSU.

- *row_cathode6* is connected to output pin 19.
- *row_cathode3* is connected to output pin 18.
- *row_cathode5* is connected to output pin 17.
- *row_cathode1* is connected to output pin 16.
- *row_cathode2* is connected to output pin 14.
- *row_cathode4* is connected to output pin 13.
- *row_cathode0* is connected to output pin 12.
- *column_anode7* is connected to output pin 11.
- *column_anode6* is connected to output pin 10.
- *column_anode1* is connected to output pin 9.
- *column_anode0* is connected to output pin 7.
- *column_anode5* is connected to output pin 6.
- *column_anode3* is connected to output pin 5.
- *column_anode2* is connected to output pin 4.

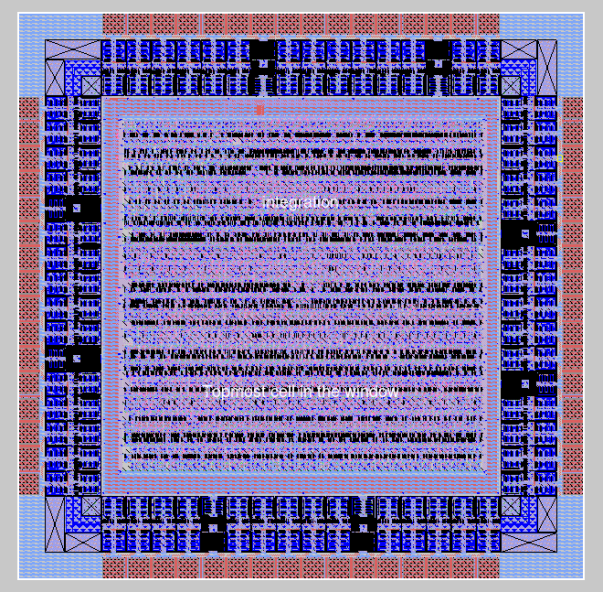

Fig. 16. Fully integrated core with padframe.

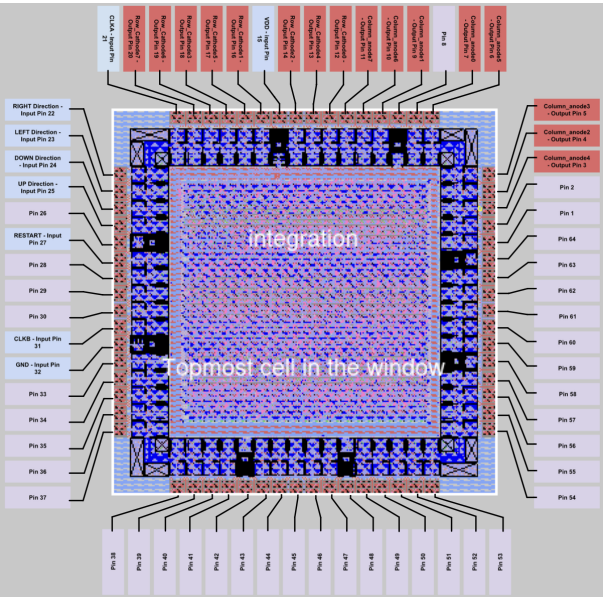

Fig. 17. 64-Pin PadFrame with pinmap (unused pins are configured as inputs).

- *column_anode4* is connected to output pin 3.
- GND is connected to input pin 32.
- VDD is connected to input pin 15.

To verify the integration of the core into the 64-pin padframe, testing of all functionality, similar to what was done before, is necessary. This will be the subject of the next and final section.

IX. Padframe Testbench

When testing the design, the functionality under test was the following:

- Change in direction upon input from the user,
- Successful growth of snake's body after consuming an apple, and
- Snake death after collision with the body.

Since the design had already been tested during core testing (see VII. Core Testbench), all that was necessary was to replicate the testbench written earlier for the integrated core. The main difference, however, is that in testing the integrated core, internal signals were not connected to the padframe, and thus only look at the specified inputs and outputs in the IRSIM simulations. The testbench only considers inputs available to the user and outputs necessary to conduct the LED array visible to the user. $1385\,\text{ns}$ of the beginning and end of the IRSIM simulation can be seen in Fig. 22 (contained in XI. Appendix).

The entire IRSIM simulation is not provided; however, enough is shown to provide the reader with a clear understanding of how the signal waveforms correspond to a physical manifestation of the game state on the LED array. In the first photo, a RIGHT direction input can be viewed. In the second image, the 3rd bit, or row, of *row_cathode*, which follows a one-cold encoding, is lit on the 3rd and 5th columns. These lights represent the snake and the apple. As the display continues to iterate through the rows, note that no other LEDs are on. In the next display cycle, in the 3rd row, the 4th and 5th columns are now lit, reflecting the right direction input from the user. In the 3rd photo, the user inputs UP, which will be reflected in the next display cycle. In the 3rd row, the 4th and 5th columns are lit. This corresponds to the body of the snake, whose head is now in the position of the previous apple after a successful consumption. Also, in the 5th row, the 6th column is lit. This LED corresponds to the new spawn location of the apple that was just consumed. The entire journey of the snake can be seen in Fig. 23 (contained in XI. Appendix).

The IRSIM outputs successfully demonstrate all of the functionality desired in the snake game. The simulation accurately reflects the movement of the snake, apple consumption, apple spawning, and death upon collision with the body.

Part	Quantity
Snake ASIC	1
Green LED	64
Pushbutton	5
3.3V Battery	1

Fig. 18. Required parts for testing chip.

X. Testing Strategy

In order to test a fabricated ASIC of this design, it would be necessary to connect it to a multiplexed 8x8 LED grid, 5 push buttons, and a $3.3\,\text{V}$ battery (or other $3.3\,\text{V}$ power supply) as shown in Fig. 24 (contained in XI. Appendix). The parts required for this design are listed in Fig. 18.

XI. Appendix

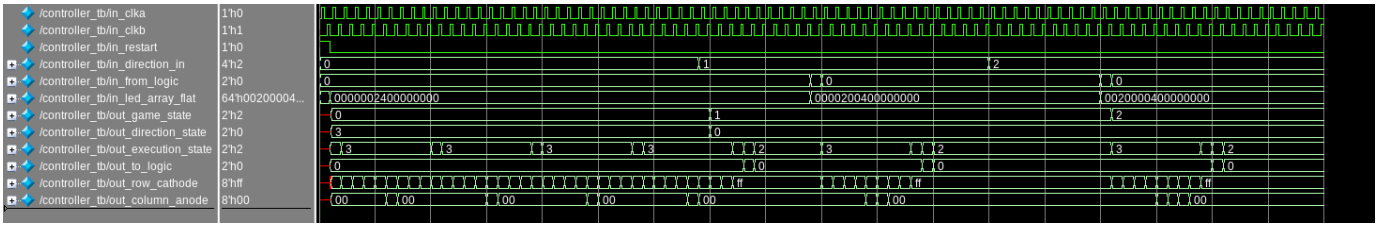

Fig. 19. Post-synthesis Questa simulation of Controller module.

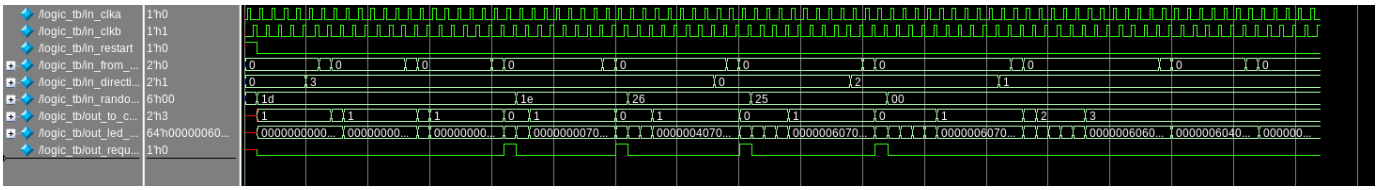

Fig. 20. Post-synthesis Questa simulation of Logic module.

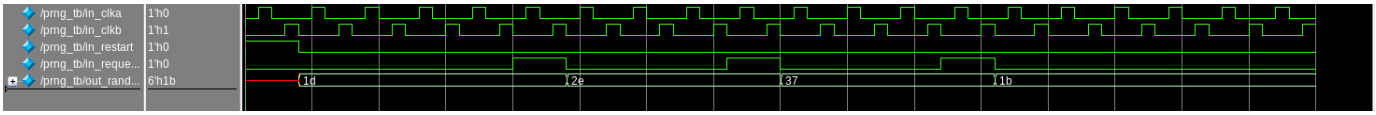

Fig. 21. Post-synthesis Questa simulation of PRNG module.

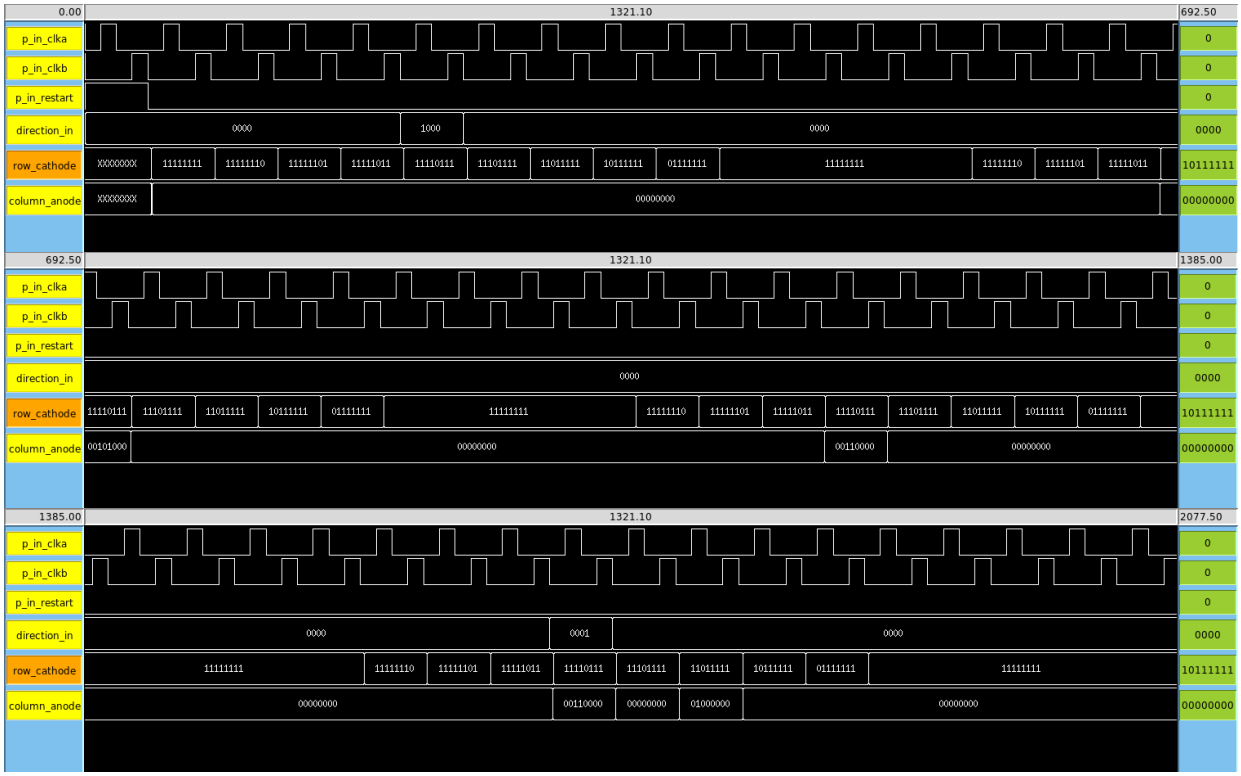

Fig. 22. IRSIM simulation of the design, from the padframe.

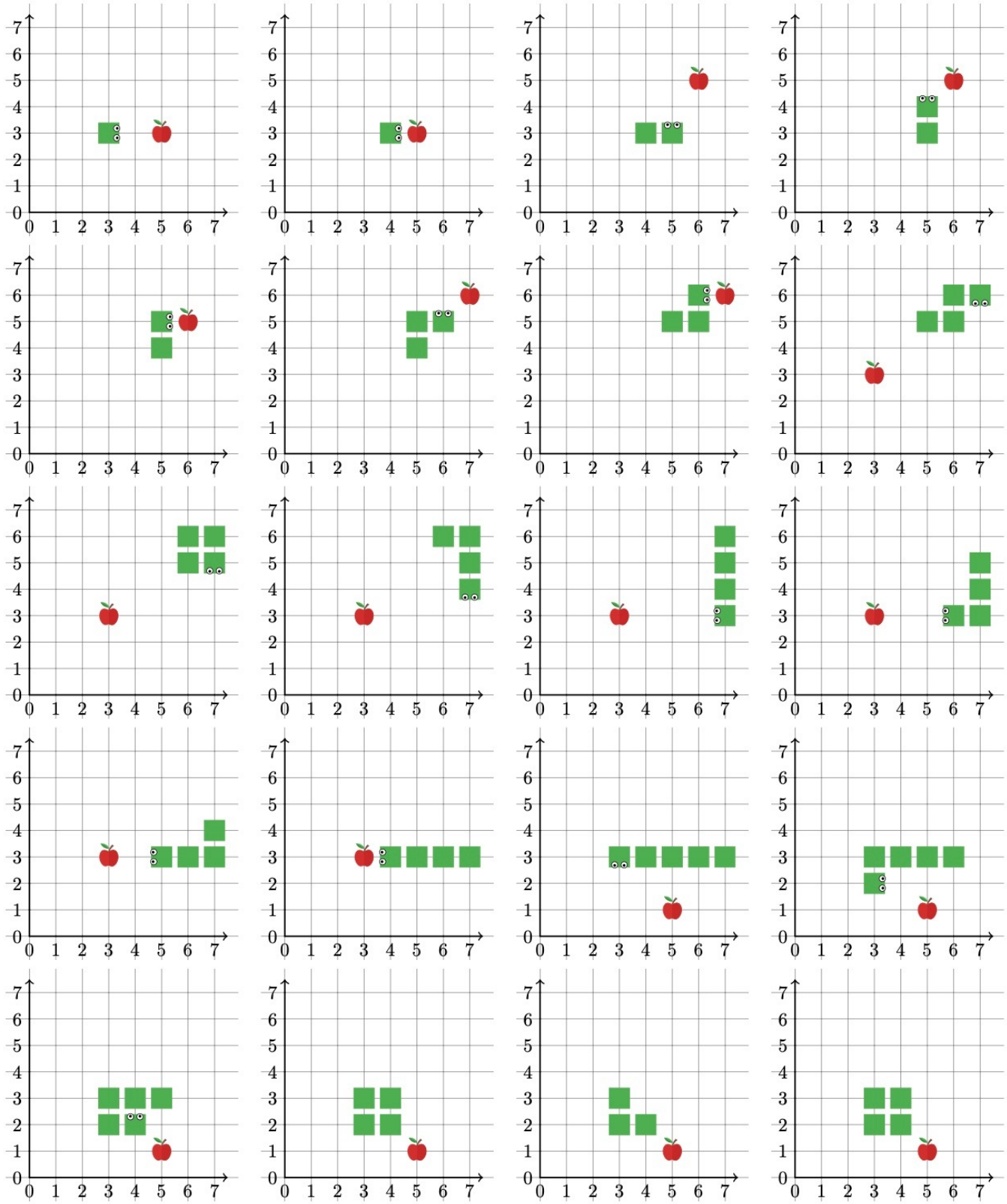

Fig. 23. Visualization of IRSIM Output as it would appear on an 8x8 LED array.

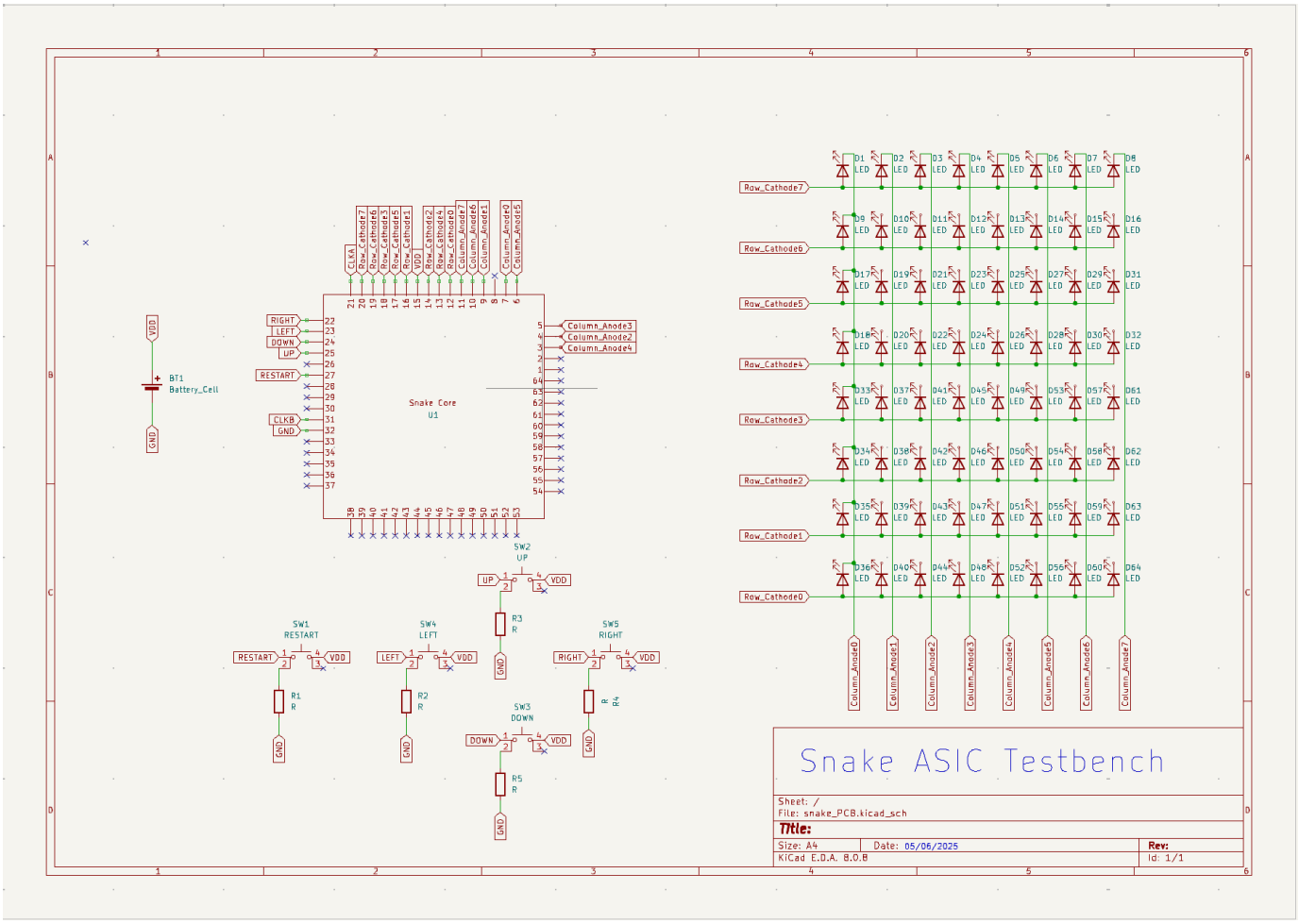

Fig. 24. Schematic of circuit required to test Snake ASIC.