Laboratory 7

Cellular Automata

7.1 Outcomes and Objectives

The outcome of this lab is to instantiate a 1-dimensional cellular automata using D-Flip Flops and combinational logic. Through this process you will achieve the following learning objectives.

- Basic Memory Element Behavior
- Timing of basic memory element
- Definition of Verilog modules
- Synthesizing a module on the FPGA development board

7.2 1-dimensional cellular automata

A common research theme in intelligent systems is emergent complexity. This is the idea that complex behavior can arise from an interconnected arrangement of simple processing elements following simple rules.

Theory: Cellular Automata

As a testbench for this idea, Stephen Wolfram[1] explored emergent complexity in a 1-dimensional cellular automata (1-DCA) network. A 1-DCA is an array of processing elements, each of which has one of two states (called alive and dead) and is interconnected to the neighbors immediately to its left and right. The next state of each processing element depends on its current state and the current state of the neighbors to its immediate left and right. Let's explore the evolution of a 1-DCA using the setup shown in Figure 7.1, note that cells which are alive are shaded and cells which are dead, white.

Let's examine the next state of the 1-DCA shown in Figure 7.1 using the rule where:

- An alive cell stays alive when exactly 1 of its neighbors is alive, else it dies.
- A dead cell comes alive when exactly 1 of its neighbors is alive, else it stays dead.

[1] Wolfram, S. "Statistical Mechanics of Cellular Automata." Rev. Mod. Phys. 55, 601-644, 1983.

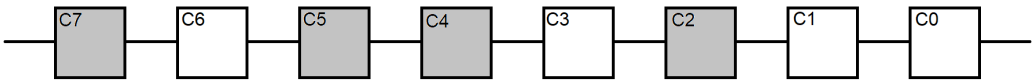


Figure 7.1: A 1-DCA consisting of 8 processing elements. Alive cells are shaded grey, dead white.

Complete the unfilled entries in Table 7.1 using the initial state shown in Figure 7.1 and the rule above. We will imagine that our processing elements are placed on a ring. This implies that the left neighbor of processing element C7 is C0 and the right neighbor of processing element of C0 is C7.

Table 7.1: The next state of a processing element depends on the current state and the number of alive neighbors.

Cell	Current State	# Alive neighbors	Next State
C0	Dead		
C1	Dead		
C2	Alive	0	
C3	Dead		
C4	Alive		Alive
C5	Alive		
C6	Dead		
C7	Alive		

You are going to build a digital system to calculate the next state of a 9-element array of processing elements. In order to do this, you will need a way to easily specify the rule used to determine the next state of a cell because you are going to be able to change it. Before doing this let's agree to associate the value of 1 for an alive cell and illustrate it as a filled black square and associate the value 0 with a dead cell and will illustrate it as an empty square.

We will formalize the next state rule by enumerating the next state of a cell for every configuration of its current state and its neighbor's state. Since a cell and it's two neighbors can each have two states, there are a total of 8 configurations. As an example, I've graphically illustrated the rule used in our previous example, in Figure 7.2. This arrangement shows the 8 configurations arranged so the binary code of the current states goes from 3'b000 at right to 3'b111 at left. The next state for each configuration is shown below the current state configuration.

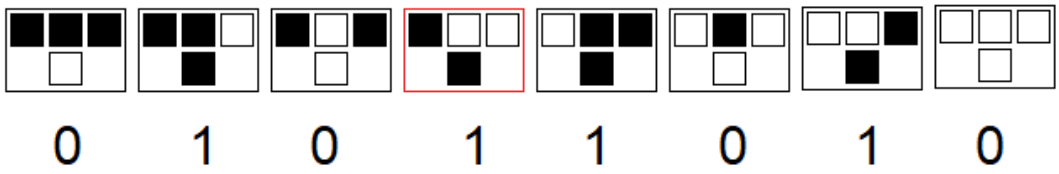


Figure 7.2: The next state of the center cell depends on the current state of a cell and its neighbors.

As an example, let's look at the 4$^{\text{th}}$ from left arrangement, outlined in red. In the top row of this arrangement, the center cell is dead and one of its neighbor's is alive. Note that this configuration has binary code 3'b100. Using the rule used to complete Table 7.1, the next state of this cell is alive, shown black. The next state value is placed below each configuration and the collection forms an 8-bit number 8'b01011010, which when interpreted as a decimal value is equal to 90. We will call the rule used to update the cells in Table 1, rule 90.

The fact that we are numbering this rule, implies that there may be others. In fact, there are 256 different rule sets. You can create new rules by substituting new combinations of alive and dead states for each next states shown in Figure 7.2. Not all rules generate interesting behavior. For example, rule 0 immediately kills any and all alive cells producing a desert. Stephen Wolfram explored every combination of rules and characterized the resulting patterns into one of four classes depending on their sophistication. Wolfram illustrated this sophistication by arranging successive iterations of the 1-DCA as rows. As an example, if you start with a single alive cell and run Rule 90 over many iterations, drawing iteration below its predecessor, you get the image shown in Figure 7.3. Some of you may recognize this figure as the Sierpiński Triangle[2], an elementary fractal.

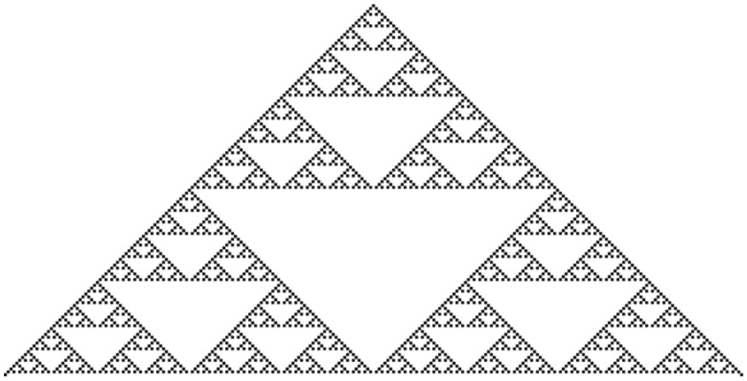


Figure 7.3: The evolution of a single alive cell (shown at top) under Rule 90.

You now have all the information and terminology you need to build a digital system to compute the next state of a 1-DCA, so let's get to it.

Implementation: Cellular Automata

You will use the inputs and outputs shown in Figure 7.4 to realize a 9 cell 1-DCA. I chose 9 cells so that we could have a unique center cell. Each of the 9 **Initial State** slide switches corresponds to one of the 9 cells. In order to set the state of the cells the **loadRun** slide switch must be in the down position. Moving the Initial State up/down will set its corresponding cell to 1/0 respectively when the array is clocked. A cell that is alive will illuminate its associated **CurrentState** LED. After the initial state of the cells is set, you should move the **loadRun** slide switch up and set the 8-bit rule value using the **Rule** slide switches. Every time that the array is clocked, the **CurrentState** LEDs will show the array.

The **reset** button will reset the state of all the cells to 0 – dead. The clock is a bit more complex than you might expect because of switch bouncing. When you press one of the push

[2] https://en.wikipedia.org/wiki/Sierpi%C5%84ski_triangle

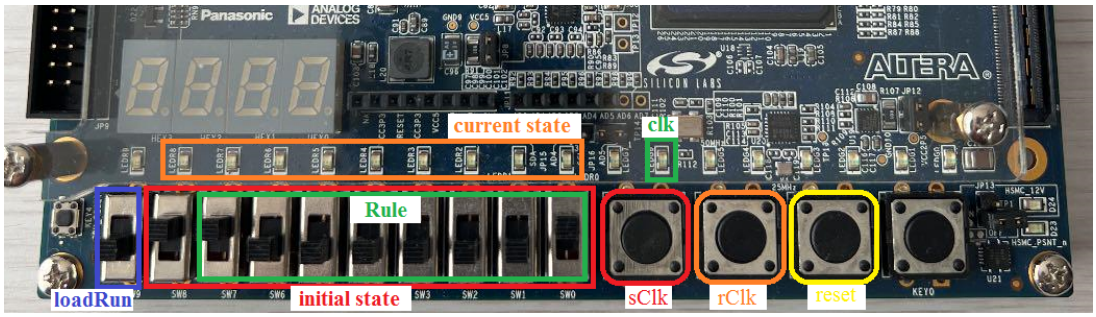
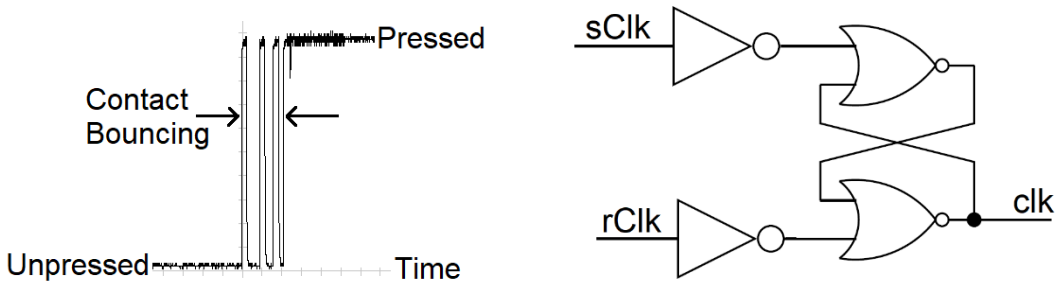


Figure 7.4: The input and output of the 1D cellular automata.

buttons on the Cyclone V GX board, the signal generated may not transition smoothly from logic 0 to logic 1. Instead, it may do something like that shown in Figure 7.5. In this figure, a single press of the button created four transitions from logic 0 to logic 1. This happens because the metal contact attached to the round plastic button physically bounced off the metal contacts attached to the body of the button. This is more prone to happen if you quickly and sharply jab at the button.



A B

Figure 7.5: A) Switch bouncing makes generating a single clock edge problematic. B) The organization of the SR latch to run the clock requires inverters in the sClk and rClk inputs and an SR latch to help remove signal bouncing.

However, even the most casual of presses may generate switch bouncing, so you are going to design a digital circuit (an SR latch) to prevent this from happening. When you press the **sClk** button, the clock signal will be set. When you press the **rClk** button, the clk signal will be reset. If the **sClk** or bounces, the clock line will not bounce because this switch bounce will only cause the clk line to be repeatedly be set when it already set. Likewise, with the **rClk** signal repeatedly resetting the clk signal if **rClk** bounces. Finally, in order for you to know the state of the clk, its logic level is display on the **clk** LED.

Since the buttons driving the rClk and sClk signals are logic 0 when pressed, you will want to invert these two signals before feeding them into the SR latch as shown in Figure 7.5B. Thus, to toggle the clock signal you will press/release the sClk button to set clk to 1. Then press/release the rClk button to clear the clk to 0. The back to sClk, etc...

7.3 System Architecture

The system architecture shown in Figure 7.6 shows the overall organization of the cellular automata. Slide switches [0-7] are used as initial state when the loadRun slide switch is set to 0. When the loadRun slide switch is set to 1, slide switches [0-7] are used as the evolution rule. The clock is generated by the SR latch whose inputs are sClk and rClk. The state of the cells are displayed on the 9 red LEDs. The processing elements forming the array are called singleCell and discussed in the next section.

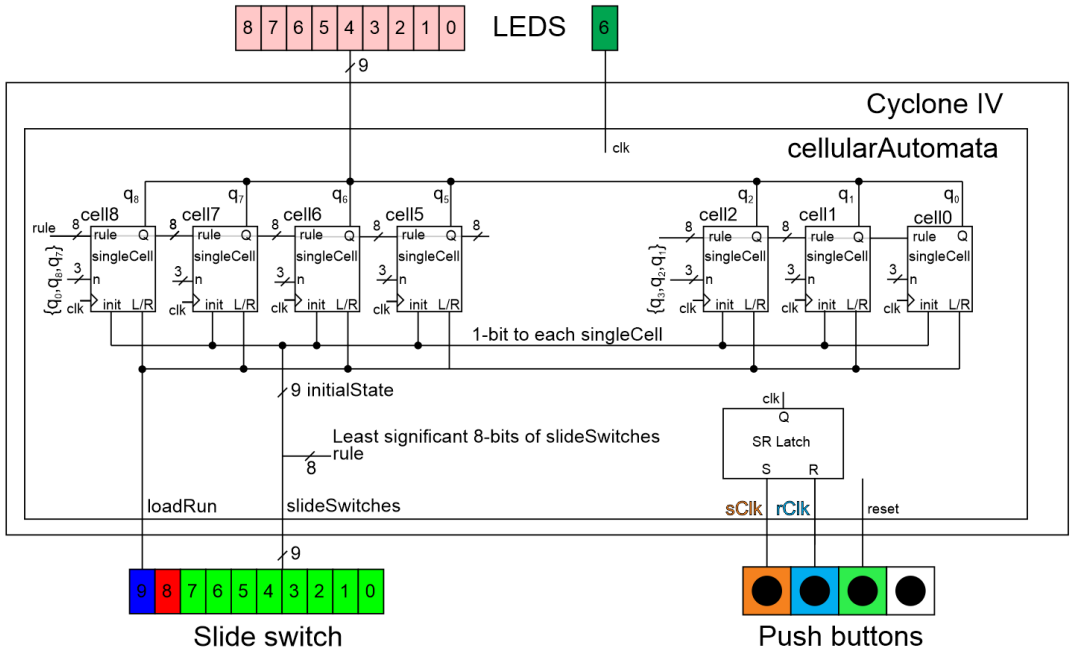


Figure 7.6: The system architecture of the cellularAutomata. Due to tight spacing, only the n inputs to cell8 and cell2 are shown.

The Verilog code for the cellularAutomata has been partially provided to you. You will need to complete the missing pieces. For this module:

- Use the cellularAutomata.v file provided in the Canvas folder as the starting point.
- Make a vector for the *rule* and *initialState* from the *slideSwitches* input vector.
- Make a 3-bit vector input for each singleCell by appending its current state to the two neighbors current state using the "{}" operators. See the singleCell module for more information.
- Connect the ends of the CA together
  - Make cell 8 have cell 0 as its "left" neighbor in Figure 7.6
  - Make cell 0 have cell 8 as its "right" neighbor in Figure 7.6
- Use cross-couples NORs and a pair of inverters to realize the SR-latch. This means that you should have two lines of Verilog Code for the SR-latch, both starting with "assign".
- You are encouraged to use the generate statement to instantiate singleCell 1-7. Due the ring architecture, you will need to instantiate cells 0 and 7 individually. For an example of the generate statement, look at the adderSubtractor provided to you in a previous lab.

7.4 Module: singleCell

The significant design problem in today's lab comes in this section, building the singleCell module. The module interface for the singleCell module in Figure 7.6 used some shorthand for the single names due to the space constraints. The internal organization and module interface for the singleCell module is shown in Figure 7.7. For example, the output currentState in this figure was called Q in Figure 7.6. You should be able to decipher the rest of the signal abbreviations used in Figure 7.6.

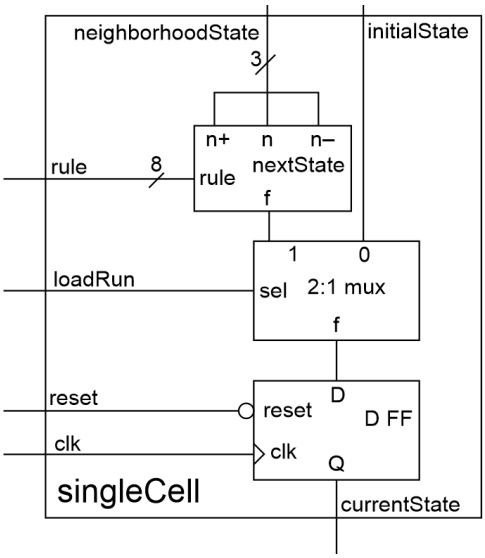


Figure 7.7: The architecture and module interface for the singleCell module.

The most complex portion of logic in the singleCell module is the box labeled "nextState". This circuit has 11-bits of input. While it may seem a bit daunting at first, the Verilog code to realize this function is pretty straightforward when you have the right perspective. Table 7.2 lists all the combination of state for a cell and its 2 neighbors in the left column (n+ is the state of the neighbor to the left, n the state of the cell itself, and n- the state to the right). These 8 combinations are the case values in the always/case statement for the nextState logic. You need to know the output for each of these cases. As an example, complete the nextState column in Table 7.2 for Rule 90 using the information in Figure 7.2. In the "Rule bit" column in Table 7.2, generalize the nextState column to the bit values in the 8-bit rule vector that is passed into the singleCell module.

Table 7.2: The input/output relationship for the nextState functionality in Figure 7.7.

{n+, n, n-}	nextState	Rule bit
3'b000		
3'b001		
3'b010		rule[2]
3'b011		

{n+, n, n-}	nextState	Rule bit
3'b100	1	
3'b101		
3'b110		
3'b111		

For the singleCell module, I want you to:

- Use the singleCell.v file provided in the Canvas folder as the starting point.
- Use an always/case statement for the nextState logic
- Use the module definitions for:
  - Use the genericMux2x1 from a previous lab.
  - Use the dffNegEdge module provided in the Canvas lab folder for this lab.
- Provide meaningful names to the wires in the module.
- Properly tab-indent your code
  - Single level for wire declarations
  - Single level for component instantiations
  - Two levels for case statement
  - Three levels for case values

7.5 Testbench

Run the testbench for the cellularAutomata module provided on Canvas. Produce a timing diagram with the following characteristics. Zoom to fill the available horizontal space with the waveform. Color inputs green and outputs red. Order the traces from top to bottom as

signal	radix	tracel color
reset	default	Blue
rule	unsigned	Blue
sButton	default	Green
rButton	default	Green
clk	default	Yellow
currentLifeState	hex	Red

Do not use the signals from the testbench, but rather the signals from inside the cellularAutomata module. You can do this in ModelSim, by expanding the cellularAutomata_tb instance in the left ModelSim and selecting "uut". Since uut is an instance of the cellularAutomata module, all the signals accessible in the cellularAutomata module are shown in the center Object. You can add signals using a drag-and-drop operation. Likewise, you can reorder the signals by dragging them. Your completed timing diagram should look something like Figure 7.8.

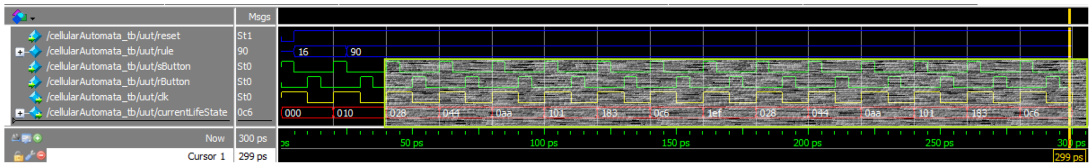


Figure 7.8: Partial timing diagram for the cellular automata.

7.6 Pin-Assignment and Synthesis

Use the image of the FPGA Board in Figure 7.1 and the information in the C5G User Guide to determine the FPGA pins associated with the input and output devices used by the cellular automata module.

Table 7.3: Pin Assignment for the 1-D calllular automata.

slide[9]	AE19		Clk LEDG6	
slide[8]			led [8] LEDR8	
slide[7]			led [7]	K8
slide[6]			led [6]	
slide[5]			led [5]	
slide[4]			led [4]	
slide[3]			led [3]	
slide[2]			led [2]	
slide[1]			led [1]	
slide[0]			led [0] LEDR0	

sClk	Key[3]	
rClk	Key[2]	Y15
reset	Key[1]	

Complete the pin-assignment in Quartus, compile your design and download to the FGPA development boards. If you are having difficulty getting your circuit to work correctly, please refer to Section 7.8 for some useful debugging tips.

Once you get your design working, demonstrate it to a member of the lab team.

7.7 Turn in

You may work in teams of at most two. Make a record of your response to the items below and turn them in a single copy as your team's solution on Canvas using the instructions posted there. Include the names of both team members at the top of your solutions. Use complete English sentences to introduce what each of the following listed items (below) is and how it was derived. In addition to this submission, you will be expected to demonstrate your circuit at the beginning of your lab section next week.

Cellular Automata Module

- Complete Table 7.1.
- Link Verilog code for the body of the cellularAutomata module (courier 8-point font single spaced), leave out header comments.

singleCell Module

- Complete Table 7.2.
- Link Verilog code for the body of the singleCell module (courier 8-point font single spaced), leave out header comments.

Testbench

- Complete testbench and timing diagram from Section 7.5.

Pin-Assignment and Synthesis

- Complete the pin assignment in table 7.3.
- Demonstrate your completed circuit to a lab team member.

7.8 Debugging Tips

To provide you with some examples to run on the Altera boards, the following table illustrates the evolution of the CA with different initial conditions and different rules.

- In the left most column contains several different rows
  - The row labeled "Rule #" is the rule that you will enter on the slide switches. The binary code of the rule is provided to make setting your dip switches easier.
  - The row labeled "start" is the initial load value stored in the CA. A value of "x" means that the initial load value doesn't matter (don't care).
  - The rows labeled "iteration" count successive positive edges generated by the clock.
- The column labeled "$q_8q_7q_6q_5$" is the output from the 4 most significant bits of the CA
- The column labeled "q_4" is the output from the middle bit of the CA
- The column labeled "$q_3q_2q_1q_0$" is the output from the 4 least significant bits of the CA

Rule 0: **0000 0000**	$q_8q_7q_6q_5$	q_4	$q_3q_2q_1q_0$
Start:	XXXX	X	XXXX
Final:	0000	0	0000
Rule 255: **1111 1111**			
Start:	XXXX	X	XXXX
Final:	1111	1	1111
Rule 90: **0101 1010**			
Start:	0000	1	0000
1st Iteration:	0001	0	1000
2nd Iteration:	0010	0	0100
3rd Iteration:	0101	0	1010
4th Iteration:	1000	0	0001
5th Iteration	1100	0	0011
6th Iteration	0110	0	0110

7th Iteration	1111	0	1111
Rule 90: **0101 1010**			
Start:	1011	0	1101
1st Iteration:	1011	0	1101
Final:	1011	0	1101
Rule 254: **1111 1110**		1111	1110
Start:	1000	0	0000
1st Iteration:	1100	0	0001
2nd Iteration:	1110	0	0011
3rd Iteration:	1111	0	0111
Final:	1111	1	1111