Ian Eykamp: CompArch Lab 1

Game of Life

How to Run:

You can clone my repo from https://github.com/ianeyk/olin-cafe-f22.

My work is in the folder named Ian_Eykamp/circuits/life. The current version is set up to be compiled to flash to the FPGA. From the Ian_Eykamp/circuits/life directory, simply run make main.bit.

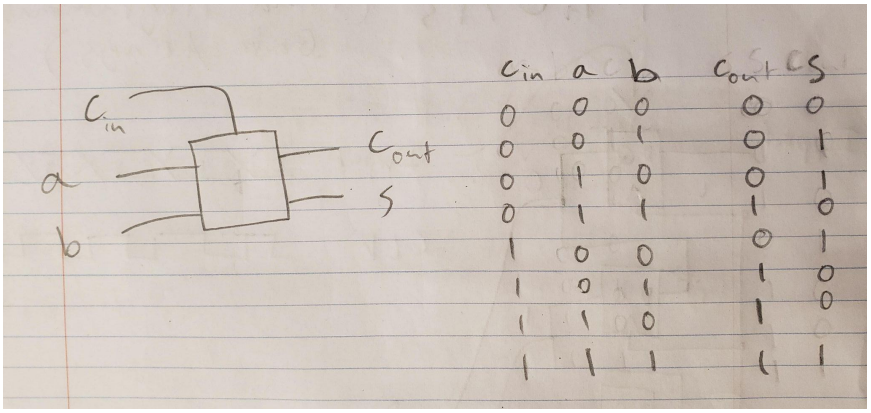
In order to run the simulation tests, you need to change a few lines in three files.

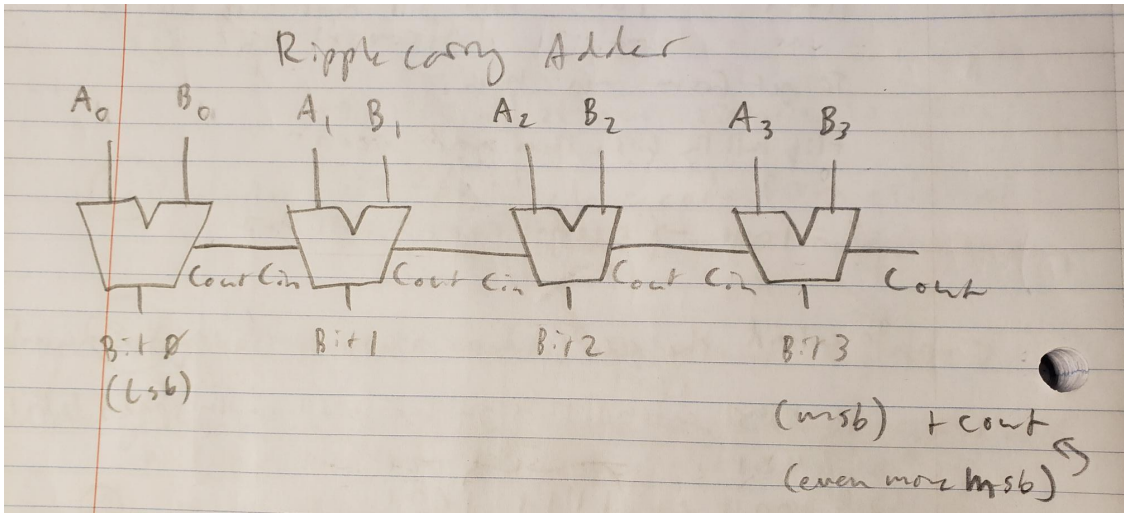
1. In main/main.sv, scroll down to lines 75-76. Comment out the line that says “works in compiler,” and uncomment the line that says “works in simulation.”
2. Do the same thing for lines 10-13 in timer/timer_8tick.sv.
3. Do the same thing for lines 17-18 in timer/timer_1second.sv. This makes the timer actually change on a time scale that you can witness in simulation (instead of requiring 12 million simulation cycles just to see the pattern advance once).
4. Now you can run make test_main to see the pattern play out for ~ 300 clock cycles. You can also run all the tests on the other modules described above (look at the makefile for commands).

Modules

1. Adder

The lowest level element is a ripple carry adder. As part of Homework 4, I created the 1-bit full adder (adder1.sv) using the sum-of-products method. I also wrote a python script that would generate an n-bit adder by recursively concatenating 1-bit adders together.

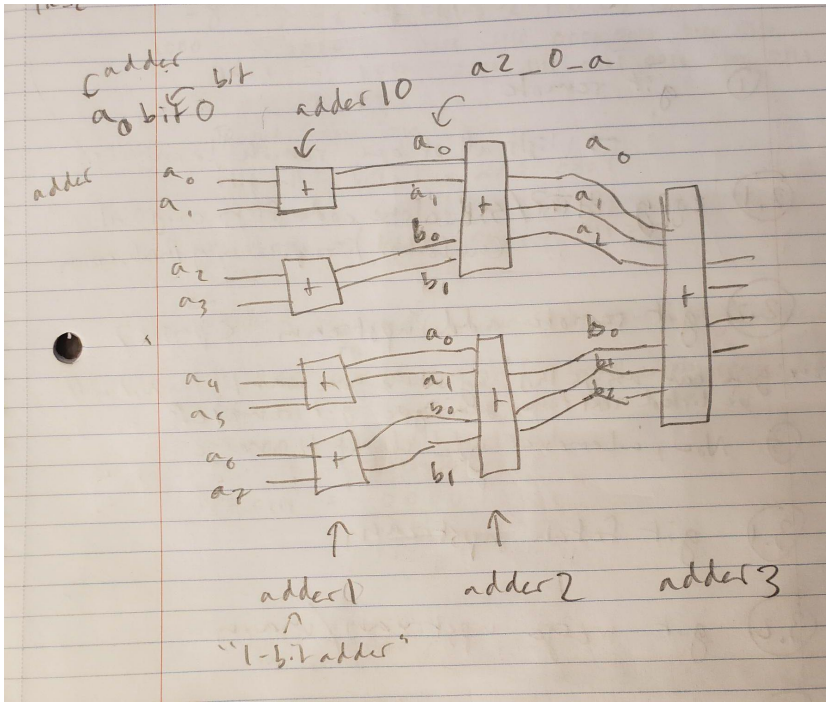




2. Sum

Each cell needs to know how many adjacent cells are alive. Each cell will receive a 1-bit input (1 if alive, 0 if dead) from each adjacent cell (8 inputs in total). It must then add these together to find the number that are alive.

I add these together in three stages, combining each pair of two inputs into a 2-bit number, then each pair of 2-bit numbers into a 3-bit number, and finally the two 3-bit numbers into a single 4-bit number that represents the sum.



3. Comparator

If the cell is alive, it stays alive when it has two or three live neighbors. If the cell is dead, it becomes alive if it has three live neighbors. I used a simple sum-of-products method to create two comparators. Each one takes in a 4-bit number (the output of the Sum module above); the first one returns 1 if the input is 2 or 3, and the second returns 1 if the input is exactly 3. Since these have trivial functionality and not general comparators, I call them “naive comparators.”

```
// Inverters
logic [3:0] a_bar;
always_comb a_bar = ~a; // hopefully this is bitwise

logic is_2;
logic is_3;

// 0010
always_comb is_2 = a_bar[3] & a_bar[2] & a[1] & a_bar[0];
// 0011
always_comb is_3 = a_bar[3] & a_bar[2] & a[1] & a[0];

always_comb out = is_2 | is_3;
```

4. Cell

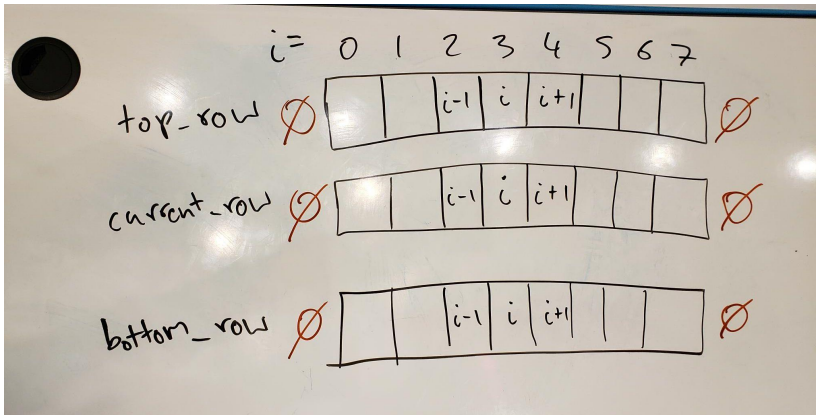
The Cell module implements the basic rules described above. It receives its current state (1 for alive, 0 for dead), receives a bus of 8 inputs representing the states of its 8 neighbors, and

outputs its next state. If its current state is alive, it uses the 2-or-3 comparator on the sum of its neighbors; otherwise, it uses the 3-comparator on the sum of its neighbors.

5. Cell Row

The next task is to arrange the inputs and outputs of 64 cell modules such that each one is assigned the correct neighbors. I did this in two stages: first with a module named cell_row, representing 8 cells, and later with cell_array, representing 8 cell rows.

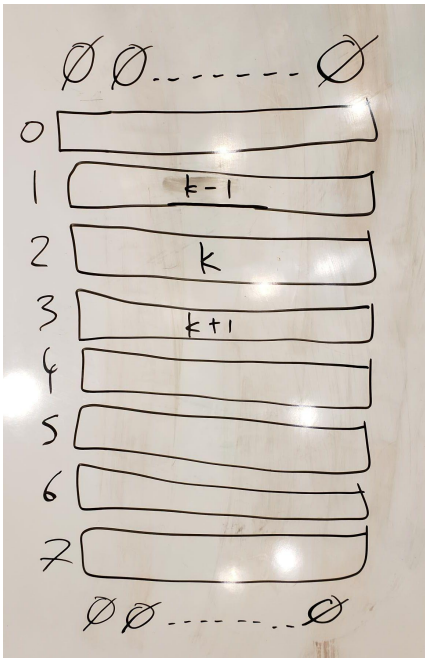
A cell row takes three inputs: the current state of all the cells in its row; the state of all cells in the row above it; and the state of all cells in the row below it. It essentially initializes 8 cell modules, assigning neighbors to each one in the geometrically correct way. For example, a cell with index 0 < i < 7 would have as neighbors cells i-1, i, and i+1 from the top row; i-1 and i+1 from the current row (excluding the cell i); and cells i-1, i, and i+1 from the bottom row. For the first and last cells in the row, three of the neighbors are hardwired to 0, because they would be off the playing grid.



6. Cell Array

Finally, 8 cell rows are constructed into an 8x8 cell array. Each cell row with index 0 < k < 7 is passed in the row with index k-1 as the top row; k as the current row; and k+1 as the bottom row. The first and last rows receive zeroes in place of their top and bottom rows, respectively.

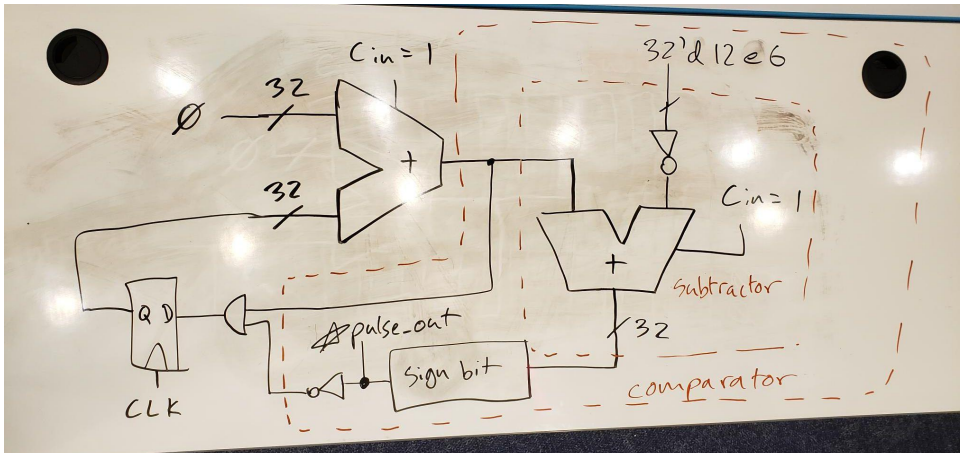
The cell array takes in a bus of size 64 representing its current state and returns a bus of size 64 representing its next state. So far everything has been implemented in combinational logic, meaning it responds instantaneously to changes in the input state. The main module contains flip-flops that will increment the state of the game every second by assigning current_state <= next_state at the appropriate time.



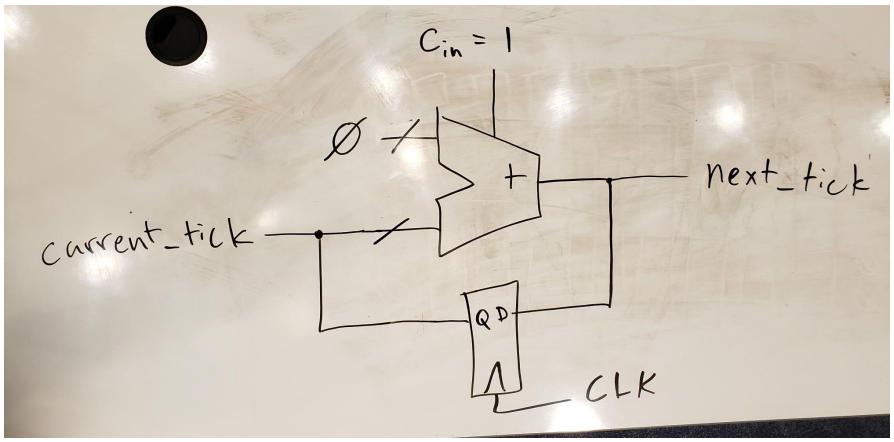
7. Timers

The main module must receive a rising-edge signal telling it when to increment the state. It would be silly to increment the state on every clock edge, as it would be impossible to see the game if it iterated through 12 million states a second.

I implemented a 1-second timer which uses a 32-bit number to keep track of the number of clock cycles. This module contains a 32-bit adder which adds one to the current count. Every clock cycle, the output from the adder is assigned to the current count using a flip-flop. There is another adder which adds the bit-wise inverse of the number 12 million to the current count (essentially, this is a subtraction). When the sign bit of the result becomes 1, it means the result is negative, so the count has exceeded 12 million. That's when I output a high value and reset the timer.



I also implemented an 8-tick counter that acts in mostly the same way (conveniently, I was able to implement it with a 3-bit adder where the carry bit is ignored—once it overflows at t=8, it wraps back around to t=0 automatically. This timer outputs a 3-bit number from 0 to 7; it is used to control which column of the LED matrix is currently being displayed.



8. Main

The main module does the job of incrementing and displaying the state of the cell array. It has a flip-flop which assigns the next state of the cell array to the current state every time it sees a rising edge from the 1-second timer (which occurs every one second). Additionally, on every clock edge, it cycles between each of the columns to display. It has a 3-8 decoder which takes in the 3-bit number from the 8-tick counter and drives one of the FPGA's pins corresponding to one of the 8 LED matrix columns high. Meanwhile, I use sum-of-products combinational logic to drive the FPGA pins corresponding to the rows of the LED matrix low. Since the clock cycles happen so quickly, this creates a persistence of vision effect where it appears that all columns of the matrix are lit up at once. In fact, changing the column every clock cycle might be too fast; it is possible that the speed is responsible for the flickering/faded nature of some of the LEDs. I even took a 'super-slow-motion' video with my phone, and the column cycling is not visible at all.