Game of Life

Annie Chu and Manu Patil

14 October 2021

1 Introduction

Conway's game of Life is an example of a cellular automaton. In this lab, we explore developing a simulation and fpga hardware implementation using SystemVerilog. In this report we focus on the high level logic and design of keys sections of combinational logic. Full code can be viewed alongside this submission.

2 Procedure

2.1 Half-Adder

We started by creating a half adder module. Figure 1 shows the implementation with a XOR gate and an AND gate.

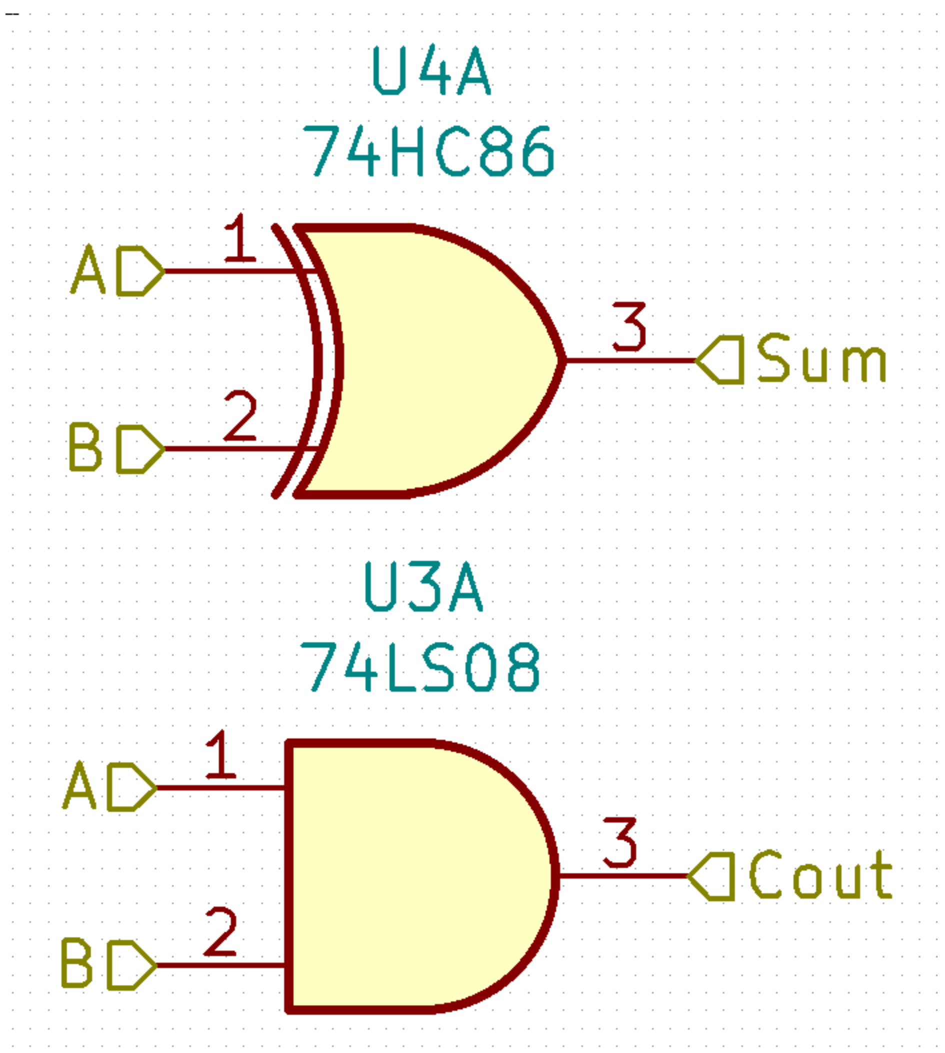

Figure 1: Half Adder

2.2 Assumptions

Next, we considered how to use this module to evaluate whether a cell should be alive or not in the next stage of Conway's Game of Life. We initially started out by trying to make a adder. However, we realized in this process that we ultimately didn;t need to differentiate all sums. For example if a sum exceeded 4, we could immediately return that the cell will die in the next iteration. Using logical deductions as the one above, we came to the Figure 2 to handle combinational logic for our logic cell.

2.3 Conway Logic

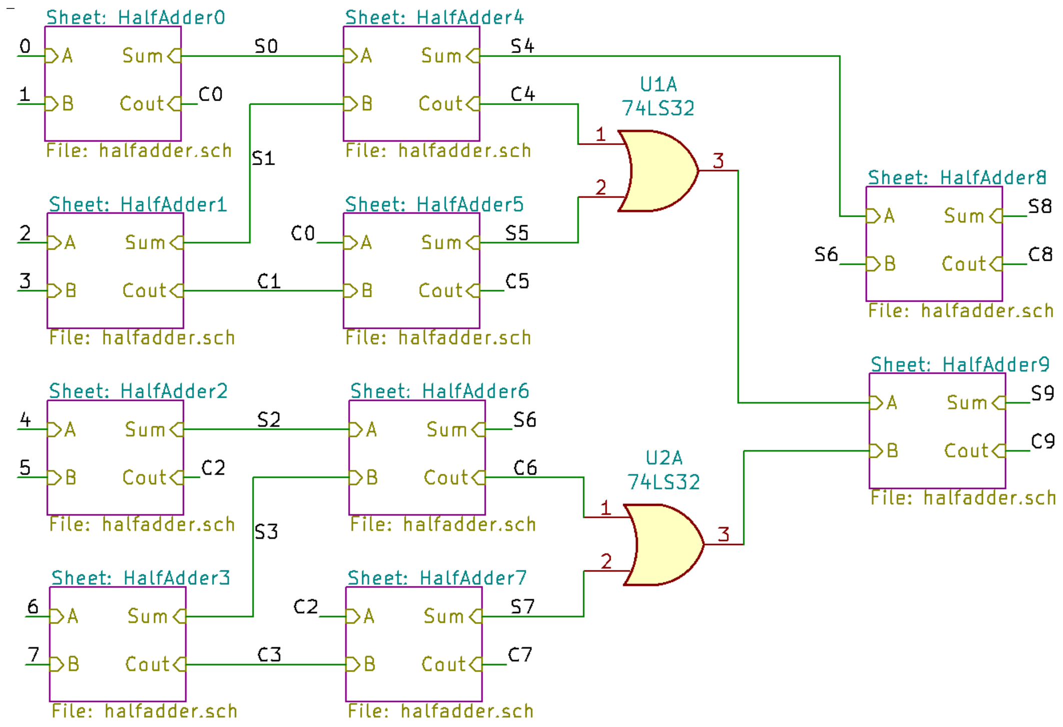

Figure 2: Conway Logic

Another simplifying assumption we were able to apply is that C4 and S4 will never be high simultaneously. This is why we can get away with OR-ing those signals before feeding them into the final set of half-adders. Using this schematic, we were able to apply the following combinational logic.

```
`default_nettype none

module conway_logic(in, prev_state, next_state);
  input wire [7:0] in;
  input wire prev_state;
  output logic next_state;

  logic [9:0] s;
  logic [9:0] c;
  logic [1:0] intermediate;
  logic [9:0] cases;

    half_adder ADD0(in[0], in[1], s[0], c[0]);
    half_adder ADD1(in[2], in[3], s[1], c[1]);
    half_adder ADD2(in[4], in[5], s[2], c[2]);
    half_adder ADD3(in[6], in[7], s[3], c[3]);

    half_adder ADD4(s[0], s[1], s[4],  c[4]);
    half_adder ADD5(c[0], c[1], s[5],  c[5]);
    half_adder ADD6(s[2], s[3], s[6],  c[6]);
    half_adder ADD7(c[2], c[3], s[7],  c[7]);

    half_adder ADD8(s[4], s[6], s[8],  c[8]);
    half_adder ADD9(intermediate[0], intermediate[1], s[9],  c[9]);

  always_comb begin
    intermediate[0] = c[4] | s[5];
    intermediate[1] = c[6] | s[7];
    cases[0] = ~(c[5] | c[7] | c[9]);
    cases[1] = ~(s[8] & ~s[9]);
    cases[2] = ~(c[8] & s[9]);
    cases[3] = c[8] & ~s[9] & prev_state;
    cases[4] = ~s[8]  & ~c[8] & s[9] & prev_state;
    cases[5] = s[8] & s[9];

    next_state = cases[0] & cases[1] & cases[2] & (cases[3] | cases[4] | cases[5]);
  end
endmodule
```

We would like to draw attention to the cases logic presented at the end of the combinational block. Here we and together the death cases and then or the living cases. The idea here is that the death cases take precedence and that if any of them is true then, we are certainly death. As a fail safe, we say that if all the accounted for death cases are false and then if a life condition is true, then and only then do we return living as the result of the logic block. This is needed particularly because in case where either where C5 or C7 are high, we stop caring about what the rest of the signals are as the sum is already greater than 4.

2.4 Conway Cell

From here the remainder of the implementation is rather typical. The file conway_cell calls the logic defined above and updates the stored value in the flip-flop on the games positive clock edge.

2.5 Led Array Driver

Finally, in led_array_driver, we applied a generic formula to control the leds appropriately.

```
  always_comb begin : blockName
    cols = x_decoded[N-1:0];

    rows[0] = ~| (cells[0*N+N-1: 0*(N)] & cols[N-1:0]);
    rows[1] = ~| (cells[1*N+N-1: 1*(N)] & cols[N-1:0]);
    rows[2] = ~| (cells[2*N+N-1: 2*(N)] & cols[N-1:0]);
    rows[3] = ~| (cells[3*N+N-1: 3*(N)] & cols[N-1:0]);
    rows[4] = ~| (cells[4*N+N-1: 4*(N)] & cols[N-1:0]);
    rows[5] = ~| (cells[5*N+N-1: 5*(N)] & cols[N-1:0]);
    rows[6] = ~| (cells[6*N+N-1: 6*(N)] & cols[N-1:0]);
    rows[7] = ~| (cells[7*N+N-1: 7*(N)] & cols[N-1:0]);
  end
```

Here we only turn on columns high when the clock says to via the decoder. Next, we only set rows low (the led on state) if both for a cells in the row, both the column is currently active and that particular cell is active in the incoming cells pattern.

3 Results

Here are links to some of our successful runs:

Glider

Blinker

4 Conclusion + Extension

Overall this lab was a bit of a stressful experience. Still, however, we learned a lot. We really enjoyed being just thrown at the tools, however, it would have been nice to have seen more Verilog examples before jumping into the lab. I think a lot of our issues and roadblocks were syntactical and just spent fighting Vivado setup. One extension could be to consider a larger grid. We've seen some really large Game of life simulation blocks. I think it could be fun to see how extensible our current design is and at which points we would start running up against other limits.