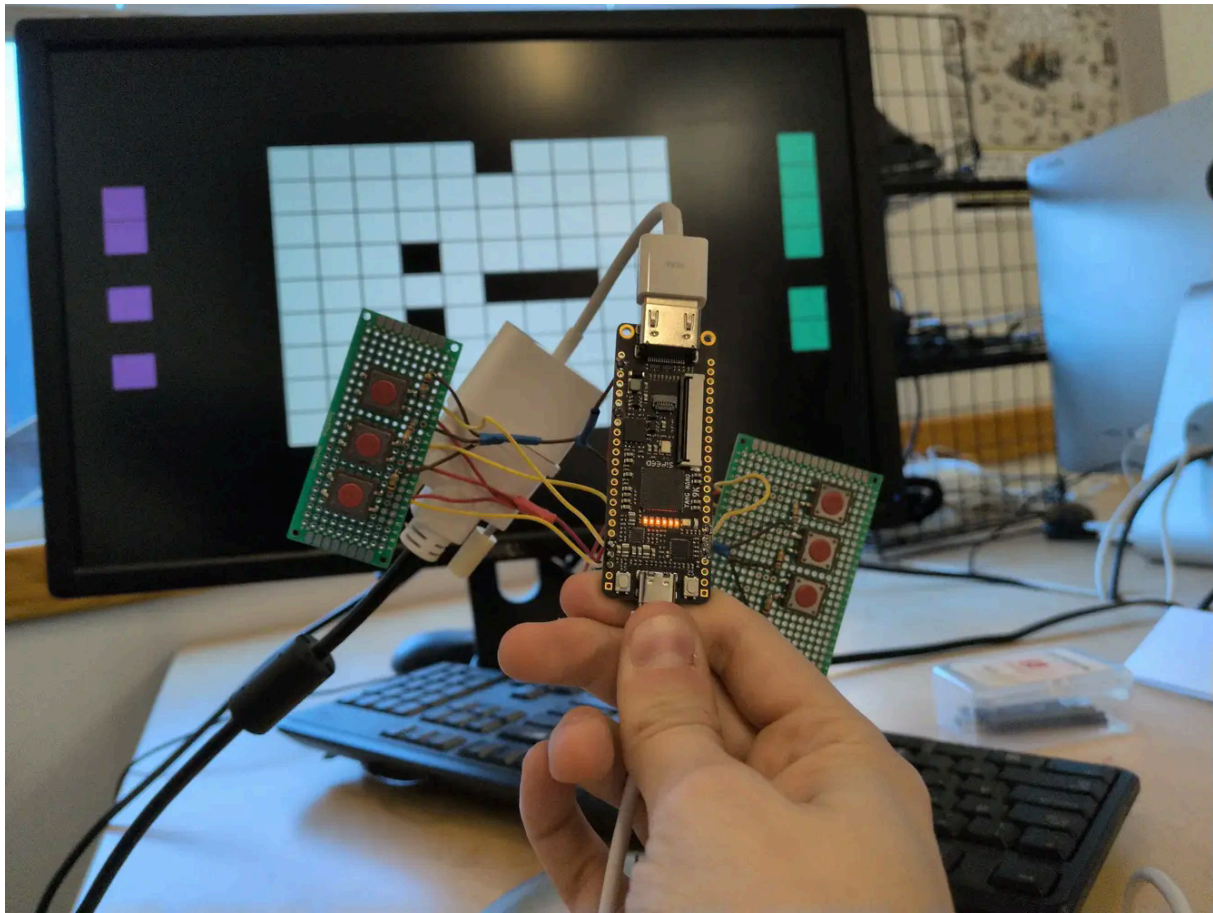


A Hardware Implementation of a Competitive Adaptation of John Conway's *Game of Life*



Background

As the final project for CSCI 395: Advanced Computer Architecture, I implemented a multi-player and competitive version of John Conway's Game of Life on hardware with an FPGA.

Conway's Game of Life (hereafter GoL) is traditionally a 'zero-player game' in which, after someone sets up the board, the game plays itself. The board is a (theoretically infinite, but practically finite) 2-dimensional grid of cells. Each cell is either alive or dead, and at each time step, each cell might change state. If a dead cell is surrounded by exactly three living cells, it, too, becomes alive. If a living cell is surrounded by two or three cells, it remains alive, and, otherwise, it becomes dead, of "overpopulation" or "isolation" [1]. Depending on how the initial game board is set up (i.e., which cells are initially alive and which are initially dead), it turns out to be possible to perform any deterministic computation.

It may, however, be considered disappointing that the game has no players. And, indeed, there are a variety of implementations of multi-player games of life, e.g., [2], [3]. The GoL I implement here is inspired by the single-player [4]. There are two players in control of three squares on opposite edges of a board. By pushing physical buttons (pictured above), each player is able to control whether or not there is life in each of their three controlled squares. A player scores a point when life would spawn in a cell on their opponent's side of the

board (excluding, of course, the three squares they control). The goal is to score as many points as possible, with no upper bounds other than integer overflow and the boredom of the players. Due to hardware limitations, the board is a finite 10×9 cells and wraps around such that the neighbors of the cells on the top are the cells on the bottom, forming a cylinder.

The particular hardware limitations here deserve some further attention. The game runs on an FPGA (Field Programmable Gate Array), requiring that, unlike in a general-purpose computer, the hardware is specified ahead-of-time. Where a general-purpose computer contains a CPU (Central Processing Unit) capable of running arbitrary instructions, FPGAs may be programmed to emulate any pattern of logic gates. That is, the FPGA programmer spells out how digital electric signals should flow through the chip, and a lot of software and hardware works in tandem to program the FPGA chip to actually have this behavior. More detail on this process from a programmer's point of view is introduced as necessary, but the upshot for now is that an FPGA gives us a way to physically build a digital circuit that, when connected to some buttons and a usual HDMI monitor, acts like a small gaming console capable (exclusively!) of running this multiplayer version of The Game of Life.

Implementation

Logic & Circuitry

Since we're working on an FPGA, the goal is to write a digital circuit that can represent the game state; that being the board and the scores of each player. We begin by describing the implementation of the board. In software, one might think of the board as a 2-dimensional array of boolean values indicating whether a cell is alive (`true`) or dead (`false`):

```
bool board[HEIGHT][WIDTH];
```

As it happens, in Verilog, (the language we use to describe hardware, unique in how it attempts to mimic C/C++ syntax), we actually do represent the board with something that looks like a 2-dimensional array:

```
logic board[HEIGHT][WIDTH];
```

Semantically, however, this isn't a traditional two-dimensional array. The most obvious difference is that instead of storing `bool`s, we use `logics`. C and derived languages have the abstraction of fixed-width integers to work with, and they can use this to arrive at booleans (0 is false, every other value is true), but in Verilog, we're designing circuits that read from a wire. And when we read from a wire, the SystemVerilog spec mandates that we read one of four values: (1) a high value, represented by a 1, (2) a low value, represented by a 0, (3) an unknown value, represented by an x, or (4) a high impedance reading, represented by a z.¹

In a further difference from software, these `logic` values need not be adjacent blocks in any 'static memory', but (if they aren't optimized away) will take up some physical space

¹Verilog users not accustomed to SystemVerilog may note that `logic` is exactly equivalent to `reg`, and neither of them imply the synthesis of a hardware register. Note also that there are other types that are only two-valued; e.g., this is how `int` (2-state) is distinguished from `integer` (4-state).

somewhere in the FPGA IC (Integrated Circuit; i.e., the chip).² There are a number of ways this might turn out, but quite often they occupy so-called *D flip flops*. These receive a continuously-oscillating clock current which alternates between low and high, and at the moment the clock signals turns high (the *positive edge* of the signal), the flip-flop reads in an input and stores it, continuously outputting that value until the next positive edge when it adopts a new value.³ The game to play now to implement the game logic is to figure out what signals to feed the D flip flop. It turns out that, while involved, this isn't actually difficult. If a given cell is alive at t_n , it is alive at t_{n+1} if and only if the number of living neighboring cells is two or three. If the given cell is dead, it becomes alive exactly when the number of living neighboring cells is 3. The only question, then, is how one can count the number of neighboring cells. We'll take it for granted here that we can add numbers and check for equality in hardware (this comes down to the more fundamental operations) and say that we'll add the 8 values given to us in each neighboring cell. All told, each cell has a circuit that looks something like Figure 1.⁴

Completing the game logic is almost as easy as duplicating this circuit a bunch of times. Some care needs to be taken to ensure that we always look at the correct neighbors, and we need to make sure that we never spawn anything on the "goal" squares at each end of the board. If such a square has exactly three neighbors, then, we need to increment the point count of the opposite player.

With that sorted, the next task is to make it interactive. The buttons are all wired with a resistor in a pulled-up configuration, meaning that each button has a dedicated GPIO pin that we can read a voltage from. This reading is high when the button is pressed and low when the button is released. To get that into the game, we put a wire between each button's GPIO pin and the cell we want it to control. I had really hoped that this would be more complicated so that there would be another cool circuit diagram here, but making the buttons stateful, debouncing them, &c just made them worse to use. The only difficulties here were due to my relative inexperience with soldering.

²Everything here pertains to synthesis; there are constructs in non-synthesizable (System)Verilog for, e.g., dynamic arrays that occupy contiguous memory as expected.

³Sometimes D flip-flops aren't like this. There may be extra set and reset signals that override behavior, and we may read a new value on some other subset of the clock cycle.

⁴An observant reader will notice that this is a really bad circuit, including gates that compare an input value with 1 instead of simply... letting the value go through. The initial stages of yosys output just appear to look like this, and unfortunately there's no middle-ground between this and a more heavily-optimized but unintuitive netlist.

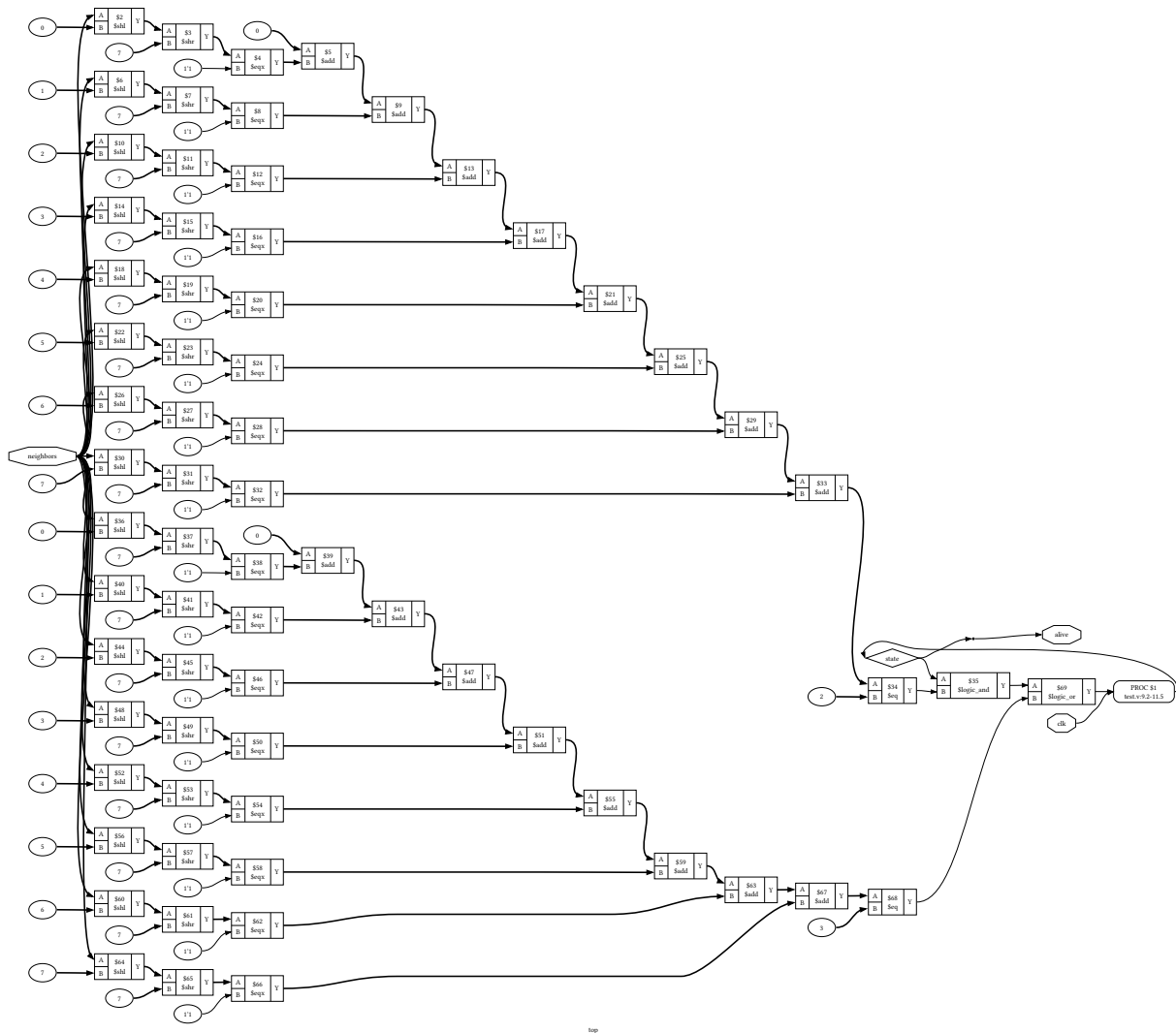


Figure 1: The yosys-generated netlist of the above-described circuit.

HDMI

The FPGA where the game logic and input occurs is a completely different place than the display on which we see the results of the computation. A display itself is a very complex mix of digital and analog circuitry, which, thanks to some standardization, we don't need to concern ourselves with. Instead, the only thing that matters is the HDMI (High-Definition Media Interface) cable by which a video source (in this case, the FPGA) can communicate with a monitor. An HDMI cable consists of several wires and terminates in 19 pins that we can apply charges to and read charge from in the FPGA. These pins are arranged as in Figure 2:

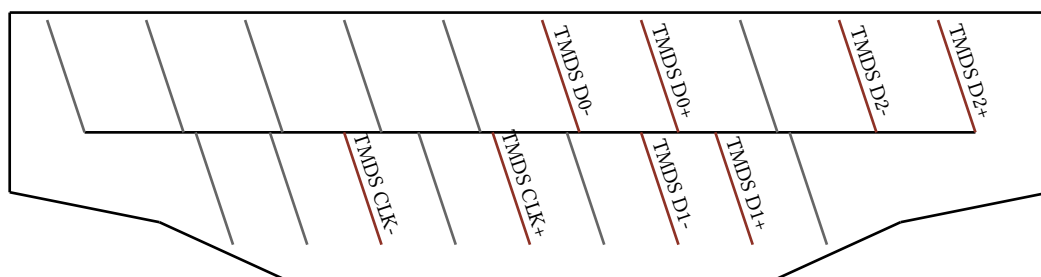


Figure 2: A logical diagram of the pins in an HDMI connector.

Due to backwards compatibility with DVI, getting a display signal from a source to a sink only requires using the pins labeled and in red: The unlabeled gray pins can be used to alter how the data is displayed or to send additional data, such as audio or Ethernet. As such, it's perfectly safe to ignore them if we're only planning on sending a basic video signal. This video signal is encoded in three channels of TMDS (Transition Minimized Differential Signaling) pairs. TMDS wires come in pairs: For each signal (labeled 0, 1, 2, or CLK), there are a pair of positive and negative wires. The positive wire always contains the voltage we mean to send and the negative wire contains the inverse of that voltage. When + is high, - is low, and vice versa. This provides resistance to EMI (Electro-Magnetic Interference) over the length of the wire. If a voltage in the + wire is disturbed, so is the voltage in the - wire, and the differential between them remains static. (A reader familiar with AV equipment may recognize this as the same technique that provides increased stability of audio signals over XLR cables as compared to TRS or TRRS.) If we apply these voltages quickly enough while cycling the clock (this being necessary so that the receiver knows when to sample voltages), we're able to send 3 bits at a time (one per data channel D0, D1, and D2) through the wire.

But what bits to send? All we're really trying to do is to display a whole lot of colors in quick succession. If we can agree with the monitor we're plugged into on when to send a color to be displayed at which pixel, all we need to do is figure out how to send a color. That's not actually too difficult: The additive primary colors being red, green, and blue, all we need to do is send three 8-bit values that represent the intensity of the red, green, and blue light we'd like. Recall that we have exactly three data channels, and we can send blue through the 0th, green through the 1st, and red through the 2nd. Of course, none of this is useful unless we can agree with the display on which pixels to color at which point in time. A naive solution may be to arbitrarily designate the first pixel as the top left, the last pixel as the bottom right, and then to proceed in row-major order as though we were writing English words to a page instead of colors to a screen. This might work, but has a flaw in that it doesn't communicate enough information to the monitor. If we just plug an HDMI cable into a monitor and it starts seeing fluctuating voltages, how will it be able to tell where the first pixel is? How will it be able to tell how long a row is or how tall a column is?

DVI (with which HDMI is backwards-compatible) solves this with a three-part solution. First, we pretend the screen is bigger than it is, and we only draw in the middle. Second, we expand all the 8 bit sequences to 10 bit sequences, and, third, we send special 10 bit control sequences around the edges of the frame, before the first pixel, at the end of each row, &c. It's important here that the set of 10 bit symbols has more members than the set of 8 bit symbols. We can encode all the possible values for a red, green, or blue component of a color, and still have space for some extra control sequences. Figure 3 shows us how we can use these codes. After we complete a row, the HSYNC (horizontal sync) signal follows a brief interval of silence, and the same occurs vertically for VSYNC (vertical sync). Outside of the *active drawing area*, in the so-called *blanking interval*, we have four possible control codes to send: Either we're sending an HSYNC, a VSYNC, both, or (in the "porches" where this is the case) neither of the two.⁵

⁵The exact timing of these signals is beyond scope here. The interested reader can look into the source code linked later on or search the Internet for "Coordinated Video Timings."

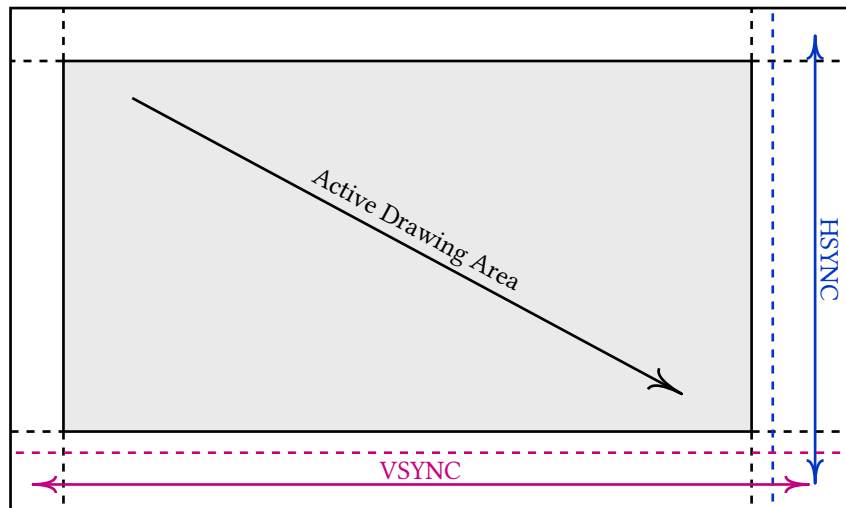
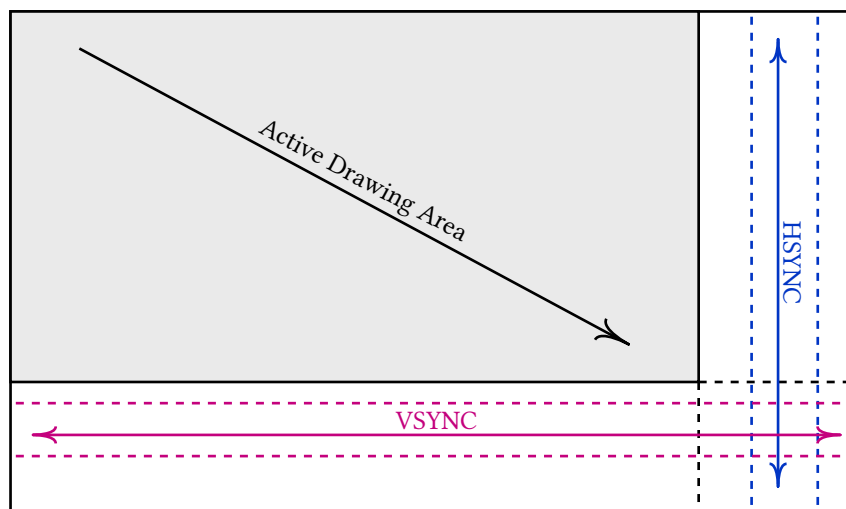


Figure 3: HSYNC and VSYNC signals.

Only two more steps remain before we can start designing hardware that can put colors to a display, and the first of this is to assign a coordinate system. A naive (and entirely acceptable!) solution would be to take a look at Figure 3 and say that the top left corner is $(0, 0)$ and the bottom right corner is $(\text{WIDTH} - 1, \text{HEIGHT} - 1)$. Since the display doesn't actually care where we start signaling, however, we can select $(0, 0)$ to be the start of the active drawing area. From the perspective of our hardware, it looks like we've shifted HSYNC and VSYNC around to look like Figure 4, but, from the perspective of any monitor we're interfacing with, nothing's changed.

Figure 4: HSYNC and VSYNC signals offset such that $(0, 0)$ begins the active drawing area.

Our last step is to agree with the monitor on what 10-bit symbols mean what: In particular, we need four 10-bit signals for each combination of VSYNC and HSYNC, on or off respectively, and a method for taking an 8-bit-valued color intensity and turning it into 10-bits. For physical reasons, we want to keep the wire we're transmitting on *DC balanced*, meaning that we want a roughly equivalent number of 1's and 0's on the line. There are actually a variety of encodings that satisfy this fairly common property,⁶ but the HDMI spec mandates that we use *TMDs* (Transition-Minimized Differential Signaling). (Recall this

⁶The interested reader can look for more information on "Manchester Encoding" and "8b/10b."

initialism from the diagram of HDMI pins.) For transmitting control codes, TMDS amounts only to a lookup (see Table 1). For arbitrary bits, we keep track of a running bias, counting roughly how many more 1's or 0's we've transmitted. We can then use this information to change how we manipulate each byte as we send it. Due to time constraints, the procedure is not discussed here. See <https://tmds.tali.network/> for an interactive demonstration, and the source code of that website or [5] for the algorithm.

C1 (VSYNC)	C0 (HSYNC)	Encoding
0	0	0b0010101011
0	1	0b1101010100
1	0	0b0010101010
1	1	0b1101010101

Table 1: TMDS-specified control code encodings.

This is finally enough to display pixels on the screen. In combination with the previous sections, it's enough to make the game! Unfortunately, none of the techniques here are relevant to making a fun game, and it also doesn't help that I have no idea how to consistently score a point without first scoring one on myself. An exercise for the reader, perhaps. :)

Issues, Bugs, and Potential Future Work

What follows is a laundry list of things I didn't do and would have liked to fix or add:

- Sometimes the display that the FPGA is connected to just goes momentarily blank. It doesn't lose signal or anything, it's just entirely blank. I'm not sure why this one happens. I suspect (baselessly) that the FPGA's PLL isn't 100% accurate, and the clock we have is slightly out-of-spec some of the time.
- Sometimes the FPGA screws up the logic responsible for assigning colors and there's this awesome primary color mess (Figure 5) that smears across the display. I suspect (hope) that this is because the TMDS logic can't keep up with the speed of the pixel clock. Pipelining the encoder could possibly assist here if this is actually the issue, though it would of course introduce timing issues everywhere else that would need to be fixed.
- The player scores are displayed as binary counters instead of as decimal values. I'm half willing to defend this as an artistic decision, but really I just ran out of time to do font rendering.
- More input devices would have been fun! As much as I like dramatic red buttons, it would be nice to drive this over an RS232 serial port, UART, or even a USB keyboard.
- The reset button doesn't work. For some reason, adding anything that depends on `rst` into the circuit causes the HDMI output to completely break (i.e., no signal is detected from the monitor).
- Due to my soldering inabilities, two of the buttons are tied together; i.e., you can't press one of them without the other registering as pressed.

Acknowledgements

- Jay Ewing (Reed College machine shop supervisor) provided some materials for the physical assembly.
- Tristan Figueroa-Reid described a version of this competitive adaptation of Conway's Game of Life to me.
- Doran Penner read through a draft and provided feedback on the writing.
- Ula Shipman provided the picture in Figure 5.
- Alex White assisted several times when the software involved (primarily Yosys and Verilator) behaved unexpectedly.

Bibliography

- [1] Martin Gardner, "Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life"." Scientific American, Inc, 1970. doi: 10.1038/scientificamerican1070-120.
- [2] "Conway's Multiplayer Game of Life." Accessed: Dec. 15, 2025. [Online]. Available: <http://lifecompetes.com/>
- [3] "LifeWiki:Life links - LifeWiki." Accessed: Dec. 15, 2025. [Online]. Available: https://conwaylife.com/wiki/LifeWiki:Life_links#Multiplayer_Versions

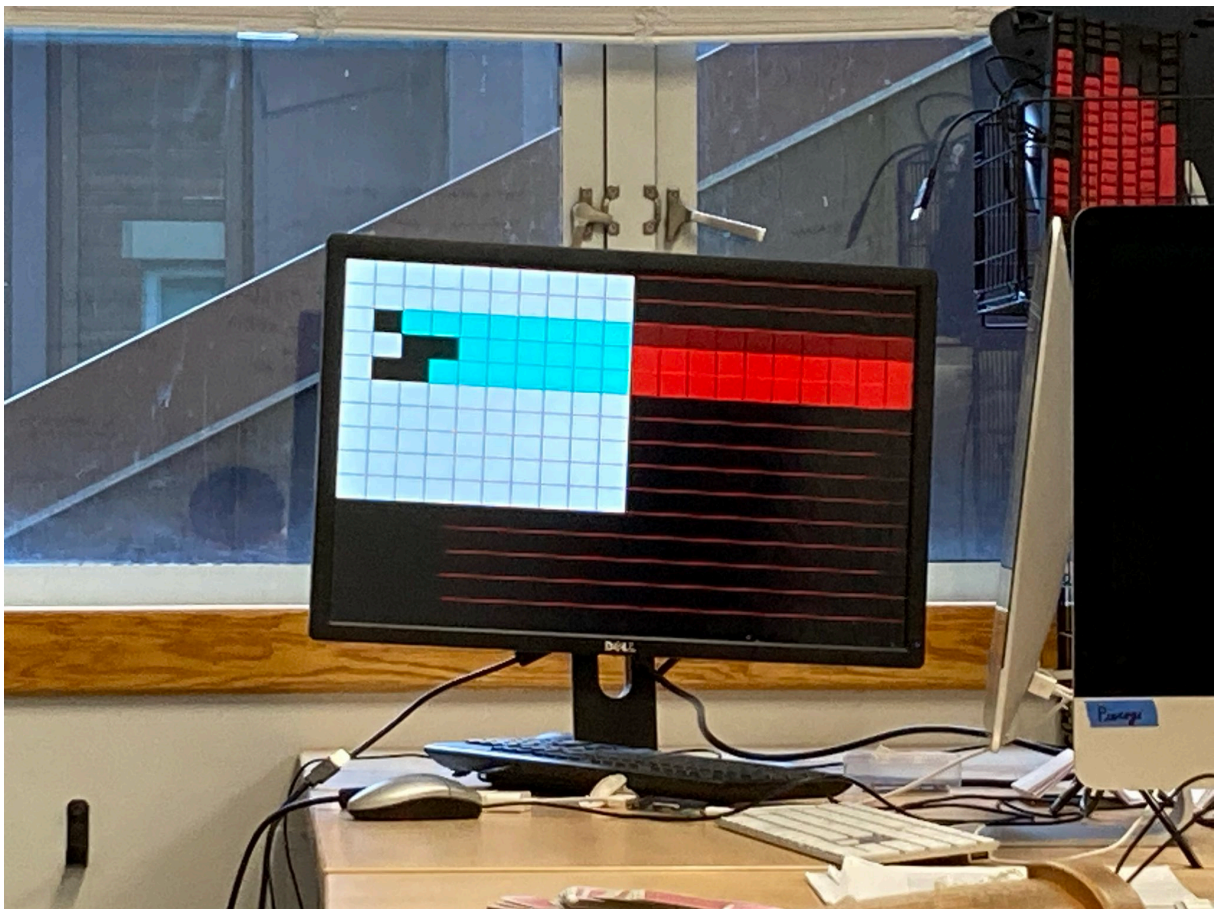


Figure 5: Everything broke just a little.

- [4] T. Figueroa-Reid, “tristan-f-r/pad-life.” Accessed: Dec. 15, 2025. [Online]. Available: <https://github.com/tristan-f-r/pad-life>
- [5] “Digital Video Interface.” Digital Display Working Group Promoters, Apr. 1999.