Final Report: Cellular Automata on FPGA

James Olsen (jco2127)
Serena Shah-Simpson (ss4354)
Jeff Tao (jat2164)
Robert Viramontes (rsv2111)
Priscilla Wang (pyw2102)

1. Introduction

1.1. Game of Life Rules

The Game of Life is a cellular automaton created by John Horton Conway in 1970. The game is a zero-player game based on some simple mathematical rules. It consists of a two-dimensional grid of cells that can be in one of two states: live or dead. Every cell interacts with 8 neighbors and the state of each cell depends on its neighbors. The rules of the game are:

1. Any live cell with less than two live neighbors dies.
2. Any live cell with two or three live neighbors lives.
3. Any live cell with more than three living neighbors dies.
4. Any dead cell with exactly three live neighbors becomes a live cell.

Based on the initial conditions, the cells form different patterns throughout the game.

1.2. Problem

Our goal was to implement Conway's Game of life in native resolution. Since the Game of Life is very computationally expensive, it is impossible to handle all computations in software quickly enough to be able to display in 1280 x 1024 native resolution without highly parallelized code. Our challenge was to design a hardware accelerator that could handle the huge number of computations that needed to be computed natively.

1.3. Timing

One of the main design challenges of the project was ensuring that we could compute the next state of each cell in the grid faster than the screen updates. The 1280 x 1024 screen uses a 108 MHz clock to update itself at 60 Hz. Thus, the game of life grid must completely update within 1/60 of a second. Using the 108 MHz clock for the calculations, this gives us 1,800,000 clock cycles to perform calculations on the entire grid.

2. Architecture

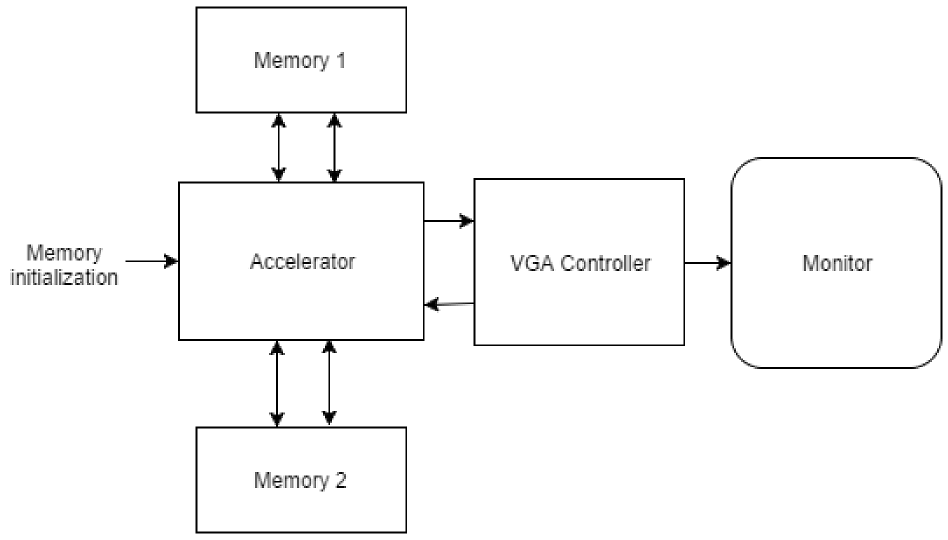
2.1. Overview

Our game of life board is a binary 1280x1024 grid of cells that can either be alive or dead. Given a current state of the board, we need to do two things:

First, using our custom accelerator, we must calculate the next state of the screen. We pass cells from the current state to our accelerator, which calculates the next value of that cell based on the rules of the game of life. The accelerator then saves the new value in a second memory module. This is done iteratively for every cell in the grid. We need two distinct memory modules: one to hold the current state which is being written to the screen, and the other which holds the next state being calculated by the accelerator. These memory modules are composed of on-board 10 Mb Block RAMs that have been combined into a single block using the Quartus Megawizard. To combine perfectly into one block per state, we used 20 bit words, for a total of 65536 words in each combined memory block. Once the computation for the entire grid completes, the accelerator waits to receive a ready signal signifying the current state has finished being written to the screen.

Second, we need to display the entire grid on the monitor using the VGA controller. Using code from lab 3 as a baseline, we fetch the appropriate cell address from the current state of memory based on the current pixel being written to, and write the value of this cell to the screen. If the cell is dead, the pixel is blue. If alive, the pixel is white. Once the end of the screen is reached, a ready signal is sent to the accelerator and the states of the two memory modules are switched.

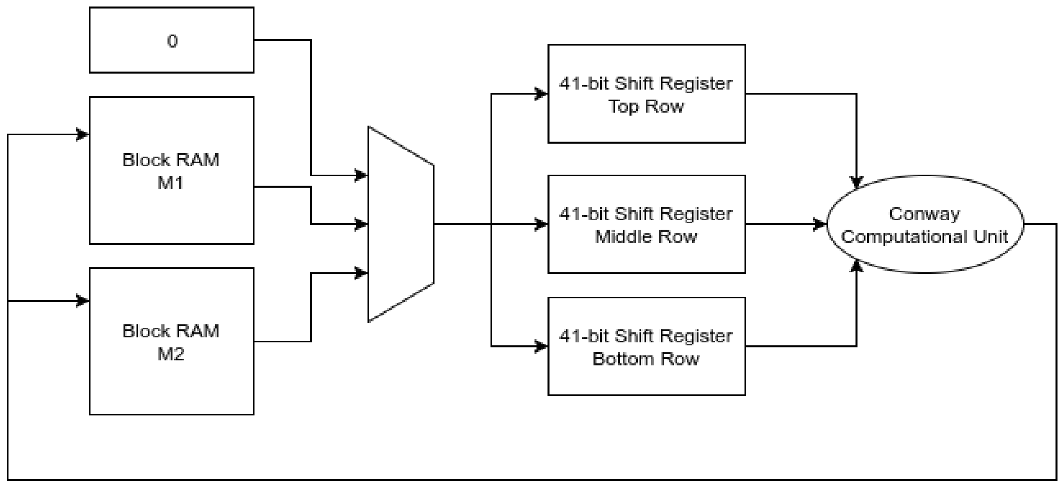
A block diagram depicting a high level view of our system is shown in figure X.



2.2. Parallel Computation

As previously explained, we need to accelerate computation so that the next state can be calculated faster than the screen is updated. To achieve this speedup, our accelerator calculates the next state values for entire 20 bit words in parallel. The cells in each 20 bit word are horizontally adjacent on the grid. Due to the rules of Conway's Game of Life, to calculate the next state values of a 20 bit word, the accelerator must be given that word and all the cells around it. Thus the 20 bit words directly above and below the current word must be passed to the accelerator. These three words are denoted the *top, middle,* and *bottom* words. In addition, the rightmost cell of the three words to the left and the leftmost cell of the three words to the right must also be passed into the accelerator.

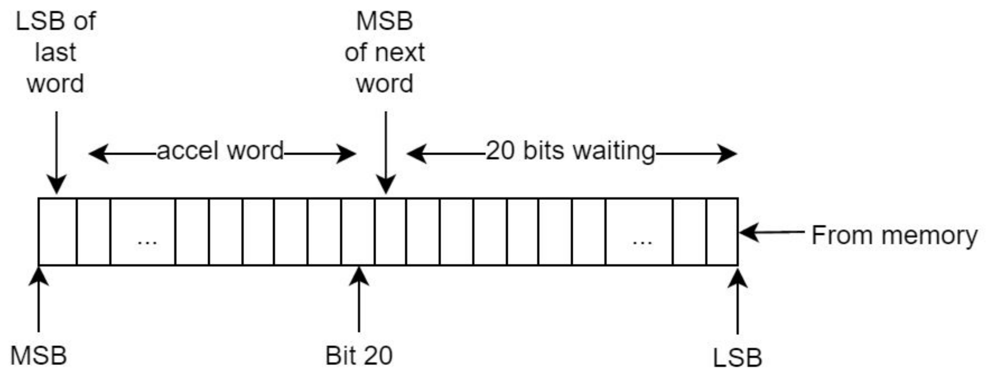
2.3. Memory Pipeline


0
Block RAM M1
Block RAM M2
41-bit Shift Register Top Row
41-bit Shift Register Middle Row
41-bit Shift Register Bottom Row
Conway Computational Unit

2.3.1. T & T+1 RAMs

We keep track of two states of the grid, labeled as T and T+1. The memory module that is labeled T contains the values of the current grid. The module labeled T+1 is always the grid where values for the next state are being written to. Both grids are the same shape and size. So that we are not needing to constantly write the next state memory to current state memory when the next state has finished computing, we instead toggle the T and T+1 labels between the two RAMs. This way, when the next frame memory has been updated, we switch directions and begin to read from this memory and write to the original current frame.
VGA controller itself does not directly access either of these RAMs. Instead, Conway accelerator is constantly reading in data from the address requested from the VGA controller from each grid and deciding which set of data should be sent to the controller according to the current direction (e.g. if we're currently reading from RAM 1 and writing to RAM 2, the VGA controller is fed values from RAM 1, since it is the one that currently has stable values).

2.3.2. Shift Registers



Each row that's currently being used for computation is stored in a shift register. We keep three shift registers (top, middle, bottom). Each shift register is 41 bytes long. The word currently being examined, along with the cells directly above and below it, are in bits 39 - 20. Since Game of Life calculations depend on all 8 neighbors surrounding a bit, we need the left and right columns and precede/follow the current word. Thus, we have bit 40 and bit 19. Bit 40 is the least significant bit of the word that preceded the current one. Bit 19 is the most significant bit of the next word.

When we are finished performing computations on the current word, and when the next word for the current row becomes available from memory, we set the shift enable of the appropriate shift register high and shift in 20 bits (the next word). In doing so, the previous word shifts out except for 1 bit, which remains as the left column for the next word. The awaiting word, which had been in bits 19 - 0 of the shift register, shifts up to bits 39 - 20. The new word shifts in as bits 19 - 0.

2.3.3. States

In order to determine which register(s) to shift, compute addresses for memory access and write, and when to enable writes, we break our memory logic into four states related to Conway's Game of Life. This is not implemented as a formal state machine, but the states allow for a clearer reasoning of what needs to happen. Note that these computations are implemented as non-blocking assignments, so the actual change in value occurs on the rising edge of the *next* clock cycle.

The states are as follows: TOP, which will act to shift the top row of data into the appropriate shift register, MID which will shift the middle row in, BOT which will shift the bottom row in, and EOR which syncs up the reads and writes at the end of a row.

Memory access addresses are also determined in these states. Due to a two-cycle fetch delay, TOP calculates the address for the bottom row, MID for the top row, and BOT for the middle row.

The memory write address is determined in TOP as 2 words before the current address being read. This is because the current address being read while *in* top is for the middle row two words ahead, with the delay introduced by the two shifts necessary to get data into the compute unit.

Write is enabled by TOP, at the same time that the write address is computed. It is disabled by MID (and remains disabled in BOT). This allows the result to be locked by memory and written just before the new data begins to be shifted into the computational unit.

The above describes a kind of "steady state" for moving through the computation. We now must deal with edge cases, namely: the vertical edges of the screen (at the beginning and end of the row) and the horizontal edges of the screen (at the top and bottom of the screen).

Since we encounter the end of the row regularly, we implement a state to handle this. As mentioned above, the memory writes are 2 cycles behind the memory reads. In this state, we essentially stall to allow the final two words to be written (they have already been read from memory and put into shift registers). We also use this state to handle the behavior that the edge of the screen should be viewed as dead pixels, so we select the zeros from the mux to be shifted in next. This per-row stalling was necessary to prevent us from overwriting valid data with zeros, but has the added benefit of syncing memory writes and reads for every row with only a 3 cycle stall overhead.

The other edge case is when we are trying to calculate the top or bottom row of the screen. We implement the same behavior of treating anything off-screen as a dead cell. We use an out-of-bounds variable to keep track of when were are in these states to ignore memory data and instead shift zeros in where appropriate.

These states are embedded in a larger if/else based on a direction variable, which indicates which memory is being used as t and which as t+1, in order to address the correct memory.

These states represent when the accelerator is “running”. When a full grid is computed, a frame_complete signal is generated which then halts the system until a ready_signal is received from the VGA controller indicating that can begin computing the next screen.

2.4. VGA Driver

For the VGA driver, we modified the VGA driver developed in Lab 3 to support displaying at the native resolution of the displays in the lab, 1280x1024 pixels with a 60Hz refresh rate. This required a pixel clock of 108 MHz[1], which is generated using the system clock of 50 MHz (selected by the CLKSEL pins and available through Qsys) and inputting this into a PLL generated through the Megawizard in Qsys, which outputs a clock of 108MHz. The front porch, back porch, and sync timing details are updated for proper syncing with the monitor. These values can be calculated or are readily available online[1].

The VGA controller implements an internal 40-bit (2 word) buffer to contain the words to draw to screen. The VGA controller needs to hold two words because there can be no delay between drawing one word and the next: the pixel clock cannot be ‘paused’ to wait for a memory request to be filled (or a shift register to shift, etc.). We ensure that we

[1] https://eewiki.net/pages/viewpage.action?pageId=15925278

constantly have a stream of data available to the VGA controller by requesting the replacement for a word as soon as it finishes drawing to screen. A typical workflow will look something like:

1. Begin drawing word x from bottom 20 bits, word y is already requested.
2. Word y becomes available while drawing word x, place in top 20 bits.
3. Finish drawing word x, request word z, immediately begin drawing word y.

The VGA controller requests the data at a specific address from the accelerator module. The module decides which memory to serve the request from, which should be the T memory (same that it is reading from). The transaction takes place through an Avalon Memory Mapped interface, with the VGA controller serving as the master and the accelerator as the slave.

This VGA controller raises a ready signal when it has finished drawing the current frame, letting the Conway Accelerator block know that the memory block it was reading from is now ready to be written to for the next state. This is to prevent the accelerator from out-pacing the screen refresh and drawing multiple states during one screen refresh.

3. Interface

In order to be able to generate different patterns, we have to reprogram the board with different .sof files which contain different memory initialization files. We have written some Python scripts which help to automate this process. The first is mif_from_bits.py, which converts an input text document of 1s and 0s (example included in code file) to a mif (memory initialization file) that can be loaded as an initial state for the board. We then use the swap_mem.py script to look for mif files and allow the user to decide which data to use as mem_init.mif, which is the file that the compilation points to. Then, in Quartus we use Processing>Update Memory Initialization File to refresh the data and then Processing>Start>Start Assembler to generate a .sof file with the new memory. These can be saved and renamed in the output_files directory. The ui.py script will search for .sof files in this directory and then allow the user to select which it will program.

4. Results

By using this accelerator, we easily complete the computation for the next state before the screen finishes updating. Using the three shift registers for the top, middle, and bottom rows, each 20 bit word requires three clock cycles to be loaded and calculated by the accelerator. Four cycles are also lost at the end of each row to satisfy our end of row logic. Since there are 64 words per row, this means the accelerator computes a single row in 3*64 + 4 = 196 cycles. The entire screen can then be computed in 196*1024 = 200,704 cycles. In other words, the max refresh rate that our accelerator could support would be about 555 Hz, which is nearly ten times faster than the required 60 Hz.

5. Conclusion

In conclusion, we count this as a very successful project. Although we did not have sufficient time to create a slick user interface, we met our main goal of creating a hardware accelerator that computes Conway's Game of Life in native resolution, and we learned a lot in the process!

Appendix A: File Manifest

SystemVerilog Component Files
conway_accel.sv: Accelerator and memory pipeline
conway_multiple.sv: Performs a word-wide computation for Conway's Game of Life rules
conway_single.sv: Performs one cell computation for Conway's Game of Life rules
mux20.sv: 20-wide, 3-input mux
SoCKit_top.v: top file for board, unchanged from Lab 3
t_memory.v: RAM module generated by Megawizard function
shift_reg_41.sv: 41 bit shift register used in data pipeline.
VGA_LED.sv: top file for VGA controller, interfaces with the accelerator
VGA_LED_Emulator.sv: implements VGA scanning, small memory buffer, and color logic

Testbench Files
conway_accel_tb.sv: Testbench for simulating accelerator.
vga_tb.sv: Testbench for top-level VGA component. Useful for examining Conway/VGA interface.
vga_led_emu_tb.sv: Simulation of VGA logic component and screen drawing.

Qsys Components
conway_accel_hw.tcl: Hardware description for Qsys for the Conway accelerator module.
vga_led_hw.tcl: Hardware description for Qsys for the VGA module.

Quartus Files
lab3.qpf: Quartus project file.
lab3.qsys: the Qsys configuration File
SoCKit_Top.qsf: Quartus settings file

Python Scripts
ui.py: User-friendly interface for programming already-compiled files.
mif_from_bits.py: Generates MIF files from easier-to-write initialization files of 1's and 0's.
swap_mem.py: Switches out the contents of the MIF file being used to initialize the FPGA without having to recompile the whole project.

Memory Initialization Files
gun.txt: A text file with a glider gun described in 1s and 0s, to be converted with mif_from_bits.
gun.mif: The resultant mif file, can be immediately loaded into the memory.

random_init.mif: Memory initialization file with randomly generated values.

Appenxix B: Code Listing

conway_accel.sv

```
// Management surrounding the Conway accelerator including buffers
module Conway_Accel(
 input clk, reset,
 // VGA controller will access memory exclusively on B ports, through this
module.
 input ready_sig,
 input  [15:0] address_b,
 output [19:0] q_b,
 output wait_request,
 output logic halp //testing...

);

  // for memories, data_b should be unused (never writing through the data
ports)
  // clocks can *probably* be the same? implement buffering in the VGA
controller?
  //
  assign wait_request = 0;

  logic [15:0] address_a_1, address_a_2;
  logic [19:0] data_b, q_a_1, q_a_2;
  logic [0:0] wren_a_1, wren_a_2, wren_b;
  wire [19:0] q_b_1, q_b_2;

  assign data_b = 20'd0;
  assign wren_b = 1'd0; // never write over B port
  wire [19:0] result;

  tmemory m1(
                             .address_a(address_a_1),
                             .address_b(address_b),
                             .data_a(result),
                             .data_b(data_b),
                             .q_a(q_a_1),
                             .q_b(q_b_1),
                             .wren_a(wren_a_1),
                             .wren_b(wren_b),
                             .clock_a(clk), .clock_b(clk)
```

```
                              );

tmemory m2 (
                              .address_a(address_a_2),
                              .address_b(address_b),
                              .data_a(result),
                              .data_b(data_b),
                              .q_a(q_a_2),
                              .q_b(q_b_2),
                              .wren_a(wren_a_2),
                              .wren_b(wren_b),
                              .clock_a(clk), .clock_b(clk)
                              );

  logic [19:0] zeros = 20'd0;
  wire [19:0] dout;
  logic [1:0] sel;
  assign halp = q_b_2[0];

  // mux chooses zeros or which memory to shift in
  mux20 memmux (.din_0(zeros), .din_1(q_a_1), .din_2(q_a_2), .sel(sel),
.dout(dout));
  logic direction; //when 0, m1 is t, when 1, m2 is t


  // reads from memory will go into here.
  // necessary so that the shift registers are able to read from
  // both m1 and m2 (can't directly wire them up to different memories
  // so the different memories will go here first.)
  // logic [19:0] memory_buffer;
  logic oob;

  // wires connecting shift output to conway input
  wire [21:0] top_out;
  wire [21:0] middle_out;
  wire [21:0] bottom_out;

  // control logic for the shift registers
  logic shift_enable_t, shift_enable_m, shift_enable_b, clear;

  // instantiate shift register
  shift_buffer top    (.din(dout), .dout(top_out),
.shift_enable(shift_enable_t), .clear(clear), .clk(clk));
```

```
  shift_buffer middle (.din(dout), .dout(middle_out),
.shift_enable(shift_enable_m), .clear(clear), .clk(clk));
  shift_buffer bottom (.din(dout), .dout(bottom_out),
.shift_enable(shift_enable_b), .clear(clear), .clk(clk));

// deal with requests from the VGA controller

assign q_b = (direction) ? q_b_2:q_b_1;

// instatiate conway module, wire together.


  Conway_Multiple cm (.top_row(top_out),
                                          .middle_row(middle_out),
                                          .bottom_row(bottom_out),
                                          .result(result),
                                          .clk(clk));




enum logic [1:0] {TOP, MID, BOT, EOR} state;
logic [0:0] frame_complete;
reg [5:0] word_count;
logic [2:0] eorstall;

always_ff @(posedge clk or posedge reset) begin

      if (reset) begin
        address_a_1 <= 16'd0;
        address_a_2 <= 16'd0;
        shift_enable_b <= 1'd0;
        shift_enable_m <= 1'd0;
        shift_enable_t <= 1'd0;
        clear <= 1;
        word_count <= 6'd0;
        state <= TOP;
        oob <= 1;
        wren_a_1 <= 0;
        wren_a_2 <= 0;
        frame_complete <= 0;
        direction <= 0;
        eorstall <= 3'd0;
        end
      else if (frame_complete && ready_sig) begin
        address_a_1 <= 16'd0;
        address_a_2 <= 16'd0;
        shift_enable_b <= 1'd0;
        shift_enable_m <= 1'd0;
        shift_enable_t <= 1'd0;
        clear <= 1;
        word_count <= 6'd0;
        state <= TOP;
        oob <= 1;
        wren_a_1 <= 0;
        wren_a_2 <= 0;
        frame_complete <= 0;
        if (direction == 0)
            direction <= 1;
        else
            direction <= 0;
        end

      else if (frame_complete && ~ready_sig) begin end

      else if (direction == 0) begin
      clear <= 0;
        case (state)
             TOP : begin
                         if (address_a_1 < 16'd64 && oob == 1)
                              sel = 2'b00; // dead cell buffer at top
                         else begin
                       sel = 2'b01;
                         end
                        // memory address computation
                         if (oob == 1 && address_a_1 > 16'd65471)
                              address_a_1 <= address_a_1; // Doesn't matter,
won't be checked. do not overflow the variable.
                         else
                           address_a_1 <= address_a_1 + 16'd64; //address
for BOT

                         if (word_count == 6'd0 && oob == 1)
               wren_a_2 <= 0; // if looking at first word, EOR zeros still
in accelerator, invalid output.
```

```
                        else if (word_count  < 6'd2)
                             wren_a_2 <= 0;

                        else wren_a_2 <= 1;
                        if (address_a_1 > 1)
                          address_a_2 <= address_a_1 - 16'd2;  // will
write to the MID address in t+1 grid

                        shift_enable_m <= 0;
                        shift_enable_b <= 0;
                        shift_enable_t <= 1;
                        state <= MID;
                        end

            MID : begin
                        sel = 2'b01;
                        shift_enable_t <= 0;
                        shift_enable_m <= 1;

         // Compute address for TOP
                 if (oob == 1)
                             address_a_1 = address_a_1 - 16'd64 + 16'd1; //
address for TOP; go back one row, move forward one word
                 else begin
                          address_a_1 <= address_a_1 - 16'd128 + 16'd1; //
address for TOP; go back two rows, move forward one word
                 end

         state <= BOT;
                 if (wren_a_2 == 1)
                    wren_a_2 <= 0;
                 end

            BOT : begin
                        shift_enable_m <= 0;
                        shift_enable_b <= 1;

         // compute address for mid
         if ((oob == 1 && address_a_1 < 16'd64))
            address_a_1 <= address_a_1; // address for mid
                 else if ((oob == 1) && (address_a_1 < 16'd65))
                       address_a_1 <= 0;
         else if (word_count == 6'd63)
```

```
                        address_a_1 <= address_a_1; // address for top to
account for extra cycle by EOR
                  else
            address_a_1 <= address_a_1+16'd64; // address for mid


                  if (oob == 1 && address_a_1 < 16'd128) begin
                              sel = 2'b01;
                         end
                  else if (oob == 1 && address_a_1 > 16'd65471) begin
                              sel = 2'b00; // dead cell outline at bottom
                  end
                  else begin
                           sel = 2'b01;
                  end

                  word_count <= word_count + 6'd1;

                  if(word_count == 6'd63) begin// at end of row
                           state <= EOR;
                  end
                  else
                           state <= TOP;
                  end

            EOR : begin
                        if (eorstall == 3'd3) begin
                              if (address_a_1 > 16'd65407) // in the
second-to-last row
                                    oob <= 1;

                              if (oob == 1 && address_a_1 < 16'd129)
                                oob <= 0;
                              else if (oob == 1) begin
                                    frame_complete <= 1;
                              end
                              address_a_1 <= address_a_1 + 16'd64;
                              word_count <= 6'd0;

                              wren_a_2 <= 0;
                              state <= TOP;
                              eorstall <= 0;
                        end
                        else begin
                              eorstall <= eorstall+1;
                              state <= EOR;
                              if (eorstall == 3'd0) begin
                                    shift_enable_b <= 0;
                                    address_a_2 <= address_a_2 + 16'd1;
                                    wren_a_2 <= 1;
                              end
                              else if (eorstall == 3'd1) begin
                                    sel = 2'b00;  // when to deassert write
enable so this isn't the next thing written?
                                    shift_enable_b <= 1;
                                    shift_enable_m <= 1;
                                    shift_enable_t <= 1;
                                    wren_a_2 <= 0;
                              end
                              else if (eorstall == 3'd2) begin
                                    address_a_2 <= address_a_2 + 16'd1;
                                    wren_a_2 <= 1;
                              end
                        end
                  end
        endcase

      end

      else if (direction == 1) begin
      clear <= 0;
        case (state)
             TOP : begin
                         if (address_a_2 < 16'd64 && oob == 1)
                              sel = 2'b00; // dead cell buffer at top
                         else begin
                       sel = 2'b10;
                         end
                        // memory address computation
                         if (oob == 1 && address_a_2 > 16'd65471)
                              address_a_2 <= address_a_2; // Doesn't matter,
won't be checked. do not overflow the variable.
                         else
                           address_a_2 <= address_a_2 + 16'd64; //address
for BOT
```

```
                        if (word_count == 6'd0 && oob == 1)
               wren_a_1 <= 0; // if looking at first word, EOR zeros still
in accelerator, invalid output.

                        else if (word_count  < 6'd2)
                             wren_a_1 <= 0;

                        else wren_a_1 <= 1;
                        if (address_a_2 > 1)
                          address_a_1 <= address_a_2 - 16'd2;  // will
write to the MID address in t+1 grid

                        shift_enable_m <= 0;
                        shift_enable_b <= 0;
                        shift_enable_t <= 1;
                        state <= MID;
                        end

            MID : begin
                        sel = 2'b10;
                        shift_enable_t <= 0;
                        shift_enable_m <= 1;

         // Compute address for TOP
                 if (oob == 1)
                             address_a_2 = address_a_2 - 16'd64 + 16'd1; //
address for TOP; go back one row, move forward one word
                 else begin
                          address_a_2 <= address_a_2 - 16'd128 + 16'd1; //
address for TOP; go back two rows, move forward one word
                 end

         state <= BOT;
                 if (wren_a_1 == 1)
                   wren_a_1 <= 0;
                 end

            BOT : begin
                        shift_enable_m <= 0;
                        shift_enable_b <= 1;
```

```
          // compute address for mid
          if ((oob == 1 && address_a_2 < 16'd64))
             address_a_2 <= address_a_2; // address for mid
                  else if ((oob == 1) && (address_a_2 < 16'd65))
                        address_a_2 <= 0;
          else if (word_count == 6'd63)
                        address_a_2 <= address_a_2; // address for top to
account for extra cycle by EOR
                  else
             address_a_2 <= address_a_2+16'd64; // address for mid


                  if (oob == 1 && address_a_2 < 16'd128) begin
                              sel = 2'b10;
                         end
                  else if (oob == 1 && address_a_2 > 16'd65471) begin
                              sel = 2'b00; // dead cell outline at bottom
                  end
                  else begin
                           sel = 2'b10;
                  end

                  word_count <= word_count + 6'd1;

                  if(word_count == 6'd63) begin// at end of row
                           state <= EOR;
                  end
                  else
                           state <= TOP;
                  end

             EOR : begin
                        if (eorstall == 3'd3) begin
                              if (address_a_2 > 16'd65407) // in the
second-to-last row
                                    oob <= 1;

                              if (oob == 1 && address_a_2 < 16'd129)
                                oob <= 0;
                              else if (oob == 1) begin
                                    frame_complete <= 1;
                              end
                              address_a_2 <= address_a_2 + 16'd64;
                              word_count <= 6'd0;

                              wren_a_1 <= 0;
                              state <= TOP;
                              eorstall <= 0;
                        end
                        else begin
                              eorstall <= eorstall+1;
                              state <= EOR;
                              if (eorstall == 3'd0) begin
                                    shift_enable_b <= 0;
                                    address_a_1 <= address_a_1 + 16'd1;
                                    wren_a_1 <= 1;
                              end
                              else if (eorstall == 3'd1) begin
                                    sel = 2'b00;  // when to deassert write
enable so this isn't the next thing written?
                                    shift_enable_b <= 1;
                                    shift_enable_m <= 1;
                                    shift_enable_t <= 1;
                                    wren_a_1 <= 0;
                              end
                              else if (eorstall == 3'd2) begin
                                    address_a_1 <= address_a_1 + 16'd1;
                                    wren_a_1 <= 1;
                              end
                        end
                  end

        endcase
      end
end

endmodule
```

conway_multiple.sv:

```
parameter word_len = 20;

module Conway_Multiple(
  input wire [21:0] top_row, middle_row, bottom_row, // 22 columns in
  input logic clk,
  output [19:0] result// 20 columns out
);


// Generate and wire together 20
// Conway single modules.
      generate
            genvar i;

            for (i=0; i<word_len; i = i+1) begin:accelerator
                  Conway_Cell c (.top_row(top_row[i+2:i]),

.middle_row(middle_row[i+2:i]),

.bottom_row(bottom_row[i+2:i]),
                                                  .next_state(result[i]));
            end
      endgenerate
endmodule
```

<u>conway_single.sv:</u>

```
/*
 * Computes a single cell of Conway's Game of Life.
 * Takes 9 bits of input, and computes whether the
 * the center bit is alive or dead based on the rules
 * of Conway's Game of Life.
 *
 */

module Conway_Cell(
  input wire [2:0] top_row, middle_row, bottom_row,
  output reg next_state
);

  logic [1:0] sum_t, sum_m, sum_b;
  logic [3:0] sum;


    assign sum_t = top_row[2] + top_row[1] + top_row[0];
    assign sum_m = middle_row[2] + middle_row[1] + middle_row[0];
    assign sum_b = bottom_row[2] + bottom_row[1] + bottom_row[0];

    assign sum = sum_t + sum_m + sum_b;

  // Determine if a cell is dead(0) or alive(1)
  // based on how many 1s are around it and
  // its own state.

  always @ (sum)
    if ( middle_row[1] == 1) begin
      if ( sum == 4'd3 || sum == 4'd4)
        next_state = 1;
      else
        next_state = 0;
    end
    else begin
      if ( sum == 4'd3)
        next_state = 1;
      else
        next_state = 0;
    end

  endmodule
```

<u>mux20.sv:</u>

```
/*
 * Basic 20-bit wide mux
 * with 3 inputs.
 */


module mux20 (
input wire [19:0] din_0,
input wire [19:0] din_1,
input wire [19:0] din_2,
input wire [1:0] sel,
output reg [19:0] dout);

  always_comb begin
```

```
    case (sel)
      2'b00: dout = din_0;
      2'b01: dout = din_1;
      2'b10: dout = din_2;
      2'b11: dout = din_0;
    endcase
  end

endmodule
```

shift_reg_41.sv:

```
/*
 * Shift register that holds 41 bits.
 * 22 of those are availble on the output.
 * A shift moves in 20 bits of data.
 * Available with a synchronous clear.
 *
 * Built with guidance from:
 *
https://courses.cs.washington.edu/courses/cse467/05wi/pdfs/lectures/07-Sequ
entialVerilog.pdf
 */

module shift_buffer(input logic clk, clear,
input logic shift_enable, // do shift when enabled, else hold
input [19:0] din, // 20 bits in
output [21:0] dout // 22 bits out
);

logic [40:0] sregisters;
assign dout = sregisters[40:19];

      always @(posedge clk) begin
            if (clear)
                  sregisters = 41'd0;

            else if (shift_enable) begin
                  sregisters[19:0] <= din;

                  sregisters[40:20] <= sregisters[20:0];
            end else
                  sregisters[40:19] <= dout;
      end

endmodule
```

VGA_LED.sv:

```
/* jco2127 jat2164 */

/*
 * Avalon memory-mapped peripheral for the VGA LED Emulator
 *
 * Stephen A. Edwards
 * Columbia University
 */

/*
 *
 * Adapted for use with Conway Accelerator project.
 * Adds interface for Avalon Mapped-Memory for data
 * transactions with the accelerator. Serves as the
 * master (defined in Qsys).
 *
 */

module VGA_LED(input logic        clk, clkmem,
             input logic       reset,
             input logic [31:0]  writedata,
             input logic       write,
                   output logic read1,
             input             chipselect,
                   input wait_request,
             input logic [2:0]  address,
                   input logic [19:0] q_b,
                   input logic halp, // for testing...

                   output logic ready_sig,
                   output logic [15:0] address_b,
             output logic [7:0] VGA_R, VGA_G, VGA_B,
             output logic      VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
             output logic      VGA_SYNC_n);


      VGA_LED_Emulator led_emulator (.clk108(clk), .reset(reset),
.VGA_R(VGA_R),

.VGA_G(VGA_G), .VGA_B(VGA_B),

.VGA_CLK(VGA_CLK), .VGA_HS(VGA_HS), .VGA_VS(VGA_VS),

.VGA_BLANK_n(VGA_BLANK_n), .VGA_SYNC_n(VGA_SYNC_n),
                                                                  .q_b(q_b),
.address(address_b), .ready_sig(ready_sig));
      logic halpflag;
            assign halpflag = !halp;

            endmodule
```

VGA_LED_Emulator.sv:

```
/*
 * Seven-segment LED emulator
 *
 * Stephen A. Edwards, Columbia University
 */

/*
 * Adapted for use with Conway Accelerator.
 * Able to draw a 1280x1024 screen. Computes
 * pixel color based on alive or dead cells
 * received from the Conway module.
 */

module VGA_LED_Emulator(
 input logic         clk108, reset,
 input logic [19:0] q_b,

 output logic ready_sig,
 output logic [15:0]address,
 output logic [7:0] VGA_R, VGA_G, VGA_B,
 output logic        VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 1280 x 1024 VGA timing for a 108 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0             1279       1599 0
 *             _______________              ________
 * ___________|    Video      |____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *       _______________________      _____________
 * |____|       VGA_HS          |____|
 */

 // values determined with help from:
 // https://eewiki.net/pages/viewpage.action?pageId=15925278

   // Parameters for hcount
   parameter HACTIVE      = 11'd 1280,
             HFRONT_PORCH = 11'd 48,
             HSYNC        = 11'd 112,
             HBACK_PORCH  = 11'd 248,
             HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH;
// 1688

   // Parameters for vcount
   parameter VACTIVE      = 11'd 1024,
             VFRONT_PORCH = 11'd 1,
             VSYNC        = 11'd 3,
             VBACK_PORCH  = 11'd 38,
             VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH;
// 1066

   // Horizontal counter
      logic [10:0]             hcount;
   logic                    endOfLine;

      // Vertical counter
   logic [10:0]          vcount;
   logic                    endOfField;

   always_ff @(posedge clk108 or posedge reset)
```

```
  if (reset)          hcount <= 0;
  else if (endOfLine) hcount <= 0;
  else            hcount <= hcount + 11'd 1;

assign endOfLine = hcount == HTOTAL - 1;

always_ff @(posedge clk108 or posedge reset)
  if (reset)          vcount <= 0;
  else if (endOfLine)
    if (endOfField)   vcount <= 0;
    else              vcount <= vcount + 11'd 1;

assign endOfField = vcount == VTOTAL - 1;

// Horizontal sync: from 0x520 to 0x5DF (0x57F)
// original 101 0010 0000 to 101 1101 1111
   // new      101 0011 0000 to 101 1010 0000
   // should go high again on 101 1010 0001

   always_ff @(posedge clk108) begin
         if (hcount >= 11'b10100110000 && hcount <= 11'b10110100000)
               VGA_HS <= 1;
         else
               VGA_HS <= 0;



         if (vcount >= 11'b10000000001 && vcount <= 11'b10000000100)
               VGA_VS <= 1;
         else
               VGA_VS <= 0;

     // VGA_BLANK_n turns off screen when not drawing pixels
     if (hcount < HACTIVE && vcount < VACTIVE )
               VGA_BLANK_n <= 1;
         else
               VGA_BLANK_n <= 0;


   end
   assign VGA_SYNC_n = 0; // For adding sync to video signals; not used for
VGA

   // Horizontal active: 0 to 1279     Vertical active: 0 to 479
   // 101 0000 0000  1280           01 1110 0000  480
   // 110 0011 1111  1599           10 0000 1100  524


   /* VGA_CLK is 108 MHz
    *             __    __    __
    * clk108    __|  |__|  |__|
    *
    */
   assign VGA_CLK = clk108; // 108 MHz clock: pixel latched on rising edge

      logic [39:0] buffer;
      enum logic [1:0] {START, LT, RT} state;
      logic [5:0] pixel_count;


      always_ff @(posedge clk108 or posedge reset) begin

            // reset or end of frame
            if(reset) begin
                  address <= 16'd0;
                  pixel_count <= 6'd0;
                  state <= START;
                  buffer <= 40'd0;
                  ready_sig <= 1'd0;
            end
            // Just drew the last pixel
    else if(address == 16'd65535 && pixel_count == 6'd18) begin
                  address <= 16'd0;
                  pixel_count <= pixel_count + 6'd1;
                  ready_sig <= 1;
            end
    // In any other pixel on screen
            else if (hcount < 11'd1280 && vcount< 11'd1024) begin
                  case(state)
                        START : begin // Extra set up after a reset
                              buffer[19:0] <= q_b; // first word into
register
                              address <= address + 16'd1;
                              state <= RT;
                        end
                        LT : begin // drawing left(39:20), writing right
                              if(pixel_count == 6'd18) begin
                                    buffer[19:0] <= q_b;
                                    address <= address + 16'd1;
                              end
                              ready_sig <= 0;
                              if (pixel_count == 6'd19)
                                    state <= RT;
                        end
                        RT: begin // drawing right(19:0), writing left
                              if (pixel_count == 6'd18) begin
                                    buffer[39:20] <= q_b;
                                    address <= address + 16'd1;
                              end
                              if (pixel_count == 6'd19)
                                    state <= LT;
                        end
                  endcase
                  if (pixel_count == 6'd19)
                        pixel_count <= 6'd0; // reset at end of word
                  else
                        pixel_count <= pixel_count + 6'd1;

            end
      end

      always_comb begin
            {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h88, 8'h88}; // Not Black
            case(state)
                        LT: begin
                              if (buffer[6'd39 - pixel_count])
                                    {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff,
8'hff}; // White
                        end
                        RT: begin
                              if (buffer[6'd19 - pixel_count])
                                    {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff,
8'hff}; // White
                        end
            endcase
   end
```

```
endmodule // VGA_LED_Emulator
```

Testbench Files

conway_accel_tb.sv:

```
module conway_accel_tb (
);

logic clk, reset;
logic halp, ready_sig;
logic [15:0] address_b;
logic wait_request;
logic [19:0] q_b;

assign address_b = 0;
assign ready_sig = 0;

Conway_Accel test(.*);

initial begin
clk = 0;
reset = 1;
#2
reset = 0;
end


always
  #5 clk = ~clk;


endmodule
```

vga_tb.sv:

```
module vga_tb (
);

logic clk, clkmem, reset, ready_sig, write, read1, chipselect,
wait_request;
logic [15:0] address_b;
logic [19:0] q_b;
logic [2:0] address;
logic [31:0] writedata;

logic [7:0] VGA_R, VGA_G, VGA_B;
logic          VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n;

VGA_LED test(.*);

assign q_b = 20'b10110011100011110000;

initial begin
clk = 0;
reset = 1;
#3
reset = 0;
end


always
  #5 clk = ~clk;

endmodule
```

vga_led_emu_tb.sv:

```
module vga_emu_tb (
);

logic clk108, reset, ready_sig;
logic [15:0] address;
logic [19:0] q_b;

logic [7:0] VGA_R, VGA_G, VGA_B;
logic          VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n;

VGA_LED_Emulator test(.*);

assign q_b = 20'b10110011100011110000;

initial begin
clk108 = 0;
reset = 1;
#3
reset = 0;
```

```
end


always
  #5 clk108 = ~clk108;

endmodule
```

Python Scripts

ui.py:

```
'''
 Script that searches for and displays already-generated
 .sof files. User then selects which file to load and
 the quartus programmer is called.

 Should be placed in project folder and Quartus told to compile
 to ./output_files
'''
#!/usr/bin/env python

import os

def main():
  print "Welcome to our lame tui\n"

  sof_files = []

  for f in os.listdir("./output_files/"):
      if f.endswith(".sof"):
          sof_files.append(f)
          print len(sof_files), ")",  f[:-4]

  choice = raw_input("# of sof file to program: ")
  choice = int(choice) - 1
  fname = sof_files[choice]
  callname = "quartus_pgm --mode=JTAG -o \'P;./output_files/%s\'"%fname

  # print callname
  stream = os.popen(callname)
  stream.close()

main()
```

mif_from_bits.py:

```
# -*- coding: utf-8 -*-
"""
From some game of life init file, generate mif

"""

import sys

HEADER = """WIDTH=20;
DEPTH=65536;

ADDRESS_RADIX=UNS;
DATA_RADIX=UNS;
CONTENT BEGIN
"""

FOOTER = """END;"""

def print_grid(grid):
    for line in grid:
        print ''.join(['.' if c == 0 else '%' for c in line])

def data_at(addr, grid):
    # For some bit grid
    # Assuming 64 word lines
    row_num = addr / 64
    start_index = (addr % 64) * 20
    if row_num >= len(grid) or start_index >= len(grid[0]):
        return 0
    else:
        bits = grid[row_num][start_index:start_index + 20]
        if len(bits) < 20:
            bits += [0] * (20 - len(bits))
        value = int(''.join([str(c) for c in bits]), 2)
        # print addr, row_num, start_index, value
        return value

def verify_schematic(mif_file):
    with open(mif_file, "r") as f:
        # Throw away the header
        for x in range(5):
            f.readline()
        grid = []
        for i in range(1024):
            grid.append([])
        rightmost_column = 0
        bottommost_row = 0
        for line in f:
            line = line.split(":")
            addr = int(line[0].strip())
            value = int((line[1].strip())[:-1])

            row = addr / 64
            bits = [int(d) for d in bin(value)[2:]]
            # Pad to 20 bits
            if len(bits) != 20:
                bits = ([0] * (20 - len(bits))) + bits
            grid[row] += bits

            if value != 0:
                if row > bottommost_row:
                    bottommost_row = row
                if (addr % 64) > rightmost_column:
                    rightmost_column = addr % 64

        # Trim grid to only the relevant region
        grid = grid[:bottommost_row + 1 ]
        right_index = rightmost_column * 20 + 20
        grid = [line[:right_index] for line in grid]

        print_grid(grid)

def main():
    if len(sys.argv) != 2:
        print "Usage: python mif_from_bits.py input_file"

    output_path = sys.argv[1] + ".mif"
    initial = []

    with open(sys.argv[1], "r") as bitmap:
        for line in bitmap:
            this_line = []
            for c in line.strip():
                this_line.append(int(c))
            initial.append(this_line)

    print_grid(initial)

    print "Rows:", len(initial)
    print "Cols:", len(initial[0])

    raw_input("Writing the above to "
        + sys.argv[1] + ".mif\n"
        + "Press Enter to continue or Ctrl+C to abort")

    with open(output_path, "w") as target:
        target.write(HEADER)
        for i in range(65536):
            to_write = data_at(i, initial)
            target.write("%d : %d;\n" % (i, to_write))
        target.write(FOOTER)

    verify_schematic(output_path)


if __name__ == "__main__":
    main()
```

swap_mem.py:

```
#!/usr/bin/env python

import os
import shutil

OUTPUT_FILE = "mem_init.mif"
mif_files = []

print "MIF Files:"
for f in os.listdir("."):
    if f.endswith(".mif") and not f == OUTPUT_FILE:
        mif_files.append(f)
        print len(mif_files), ") ", f[:-4]

choice = raw_input("Enter # of file to program:")
choice = int(choice) - 1

chosen_file = mif_files[choice]
```

```
os.remove(OUTPUT_FILE)
shutil.copyfile(chosen_file, OUTPUT_FILE)
```