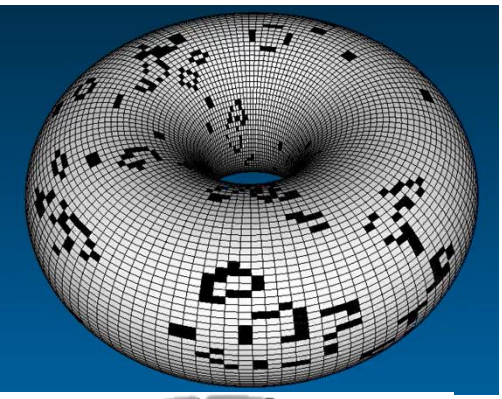
Hardware-Accelerated Conway's Game of Life on FPGA: Torus Topology

Author : Daniel Bernath 324444173

Objective: Modification of the CGOL Non-Wrapped version into the CGOL TORUS version while further optimizing the performance with respect to number of clk cycles and F_{max}.

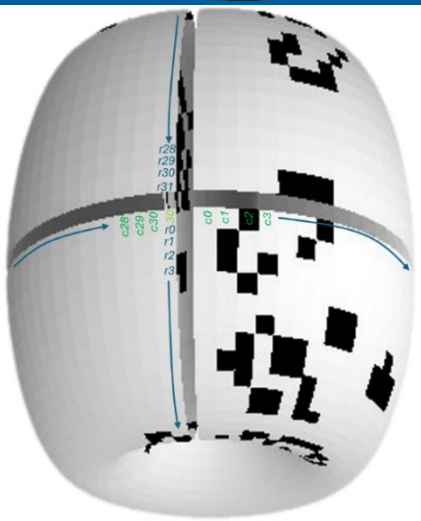


Background

The Torus shape resembles a donut-like shape where the whole surface is continuous such that the previous boundaries are lifted off, meaning that we no longer assume our borders as rows / cols of zeroes.

The implications of such change means the following:

1. Pre-loading 3 rows instead of 2 rows.
2. Computing Row[0] based on Row[N-1] and Row[1]
3. Computing Row[N-1] based on Row[N-2] and Row[0]
4. Processing the current row's edge cases depend on one another.



← In the illustration on the left, we can see an example of how a corner should be computed, providing a helpful guide for our design.
Simplifying the image into words, we can infer that if we're handling the above mentioned implications for the rows, then all we're left to handle is thinking about our current row being processed and make sure that we're computing a 3×3 matrix for the edges rather than a 3×2 matrix as we did for the non-wrapped edition previously.
Therefore, for col[0] we'll be interested in the elements of col[M-1] and col[1] followed by the same logic for col[M-1] where we'll use col[M-2] and col[0]s elements.

Based on our previous implementation, these modifications are relatively simple and intuitive and will be explained in the following pages.

Grid Support

1. **Max Grid Size Relaxation** – The CGOL Grid's dimensions are reduced to a maximum grid size of 64×64 which is our XLRs requirement of grid support.

2. **Multiples of 8** – The grid sizes are guaranteed to be multiples of 8, but not a power-of-2 numbers requiring us to be more cautious with bit shift operations.

3. **Runtime-Configurable** – We're required to make sure to use dynamic variables rather than static variables, enabling the various patterns.

   Note that we're also creating static parameters for the MAX_WIDTH and MAX_HEIGHT corresponding to the maximum columns and rows respectively.

Implementation Walkthrough

The Idea (TL;DR – HW-SW Handshake opt. + FSM Pipeline + STA opt.)

In order to avoid using modulo, I've chosen to create a double buffer to hold the 1st 2 rows and to make the reading more natural and to make the necessary adaptations when writing the data back to the memory. Due to this choice, we'll start with a fake write into $row[0]$ which will be overwritten as the last row once again, which means that our writing order goes from $1,2,\dots,N-1,0$ which is why the double buffer was required. Note that the insertion of the fake write was crucial for timing the FSM pipeline and worked out perfectly with the CGOL TORUS (as well as with the baseline design) while not adding any additional clk cycles and enabling a smooth 2-clk cycle per row processing for any case regardless of the suggested optimizations.

The logic for processing leans on fully optimizing the CGOL RULES away by bit-masking and even altering the rules a little bit to reduce an subtractor. Furthermore, noticing that each element of col[i] is being reused I've opted for a tree adder where I've separated the processing into 2 stages – the 1st one being a vertical sum of each column without caring about the current row's placement – This choice was especially nice because it enable a huge reduction of fanout on the **triple_row_buf_crnt_idx** which was reported through Quartus's reports as the heaviest wire. In the 2nd stage, I'm just adding the correct columns for each col[i] which was going really smoothly with the TORUS's wrap-around structure, the only challenge was how to account for the location of the correct cell which will be later shown with the elegant bit-masking trick.

Throughout the design I will explain the additional optimizations including full HW-SW handshake reduction while going all out on STA performance trying to eliminate the most LEs I can just out of personal curiosity of taking the design to the max on all possible aspects.

Going Further

(TL;DR Improved "0-Row" Detection + XMEM Addr Line Realization)

Due to lack of time I wasn't able to complete the 0-row detection including some improvements such as detecting various combinations of row orders that I already know would provide a row of zeros. Another improvement idea was a 1-row detection which I noticed in the patterns that can occur sometimes and would enable shaving off a couple of clk cycles.

But the Holy Grail of the optimization I'm aiming of continuing on my own is taking advantage of how the data is sent into the memory. After close inspection it is evident that for a 64-bit grid size, the mem_data both for the read and write are holding 4 rows per address line since the XMEM is designed as a 256-bit wide memory, therefore, an idea I was interested in was to make a larger sliding window and processing 2 rows at a time while shifting the addresses accordingly, such a move, if indeed possible as my intuition tells me would significantly break the 1 clk cycle threshold and due to the grid size relaxation and the XMEM's width should be possible to achieve.

Declarations

```
47  /*****************************************************************/
48  /*                          DECLARATIONS                         */
49  /*****************************************************************/
50
51  logic [15:0] grid_xmem_base_addr; // For the base address.
52  logic [31:0] grid_width;          // Number of columns in grid , provided by SW
53  logic [31:0] bytes_per_grid_row;  // Needed to calculate size of memory access
54  logic        start;               // Triggered by SW
55  logic        done;                // Used to inform SW operation is done.
56  logic [31:0] _num_itr_;           // Number of iterations, provided by SW
57  logic        clear_done_on_read;  // Indicates by SW completed read of register allocated to "done"
```

Most of these signals are pretty standard and similar to UDIs suggested names therefore there's no need to explain them.

_num_itr_ - This is a new signal providing the XLR with the information about the number of iterations in order to minimize the HW-SW handshake and only signal done at the last iteration.

```
59  logic [15:0] crnt_rd_addr;        // Current Read Address from grid
60  logic [15:0] crnt_wr_addr;        // Current write address to grid
61
62  logic [1:0][MAX_WIDTH-1:0] double_row_hold_buf;   // Holding Buffer for Rows 0 and 1
63  logic [2:0][MAX_WIDTH-1:0] triple_row_buf;        //  Rolling Buffer for the data rows window
64  logic      [MAX_WIDTH-1:0] updated_row;           // Calculated new row to be updated in grid
65  logic      [MAX_WIDTH-1:0] updated_row_ps;        // Holds the Pre-Sampled result
66
67
68  logic [$clog2(MAX_HEIGHT):0] grid_wr_row_idx;  // Current grid write index - 6 bits wide
69  logic [$clog2(MAX_HEIGHT):0] grid_rd_row_idx;  // Current grid read index - 6 bits wide
70  logic [$clog2(MAX_HEIGHT):0] grid_height;      // Grid Height, provided by SW - 7 bits wide
```

Again, most signals are with the same names, the only change made on them are the bit-width for **grid_wr_row_idx, grid_rd_row_idx & grid_height** which required changing the bit-width from $\$\mathbf{clog2}(\mathbf{MAX\_HEIGHT}) - \mathbf{1}:\mathbf{0} \rightarrow \$\mathbf{clog2}(\mathbf{MAX\_HEIGHT}):\mathbf{0}$ to enable using the max value of the grid dimensions, i.e. the number 64 requiring 7 bits.

double_row_hold_buf – This is another implementation used for reducing the number of clk cycles where the idea is first of all to avoid using modulo operations and etc. – this will be explained later on. Note that the purpose of this buffer is to hold the 1st 2 rows, row[0] & row[1] for the wrap around. Furthermore, while it may seem heavy in terms of DFFs, I'm taking advantage of the grid size relaxation enabling more DFFs so we're far away from reaching more than 1000 DFFs.

```
72  logic is_fake_n;            // Indicator for first fake write
73  logic next_is_last_row;     // Indicate next row will be last (required for border treatment)
74  logic is_last_row;          // Indicate we are at the last row
75  logic is_last_itr;          // Indicate we are at the last itr
76  logic start_next_itr;       // Indicator for resetting the relevant signals for next iteration
77
78  logic dbl_load_idx;     // Index for first 2 reads.
```

is_fake_n – This flag used to handle the first fake write I'm inserting to properly time the pipelined FSM, it's purpose is to simply make sure that during the first write acknowledgement **grid_wr_row_idx** will not increment. The rest will be explained in the relevant part.

next_is_last_row – Indicates when we're about to write the last row.

is_last_row – Last row is being processed and written.

is_last_itr – This flag is used in the DONE state in order to determine whether we're sending the **done** signal or not.

start_next_itr – This signal is used for mimicking the **start** signal required to "reset" the relevant signals for a new iteration.

dbl_load_idx – This simple is used as a trick for decoupling states, reducing logic in the comb. FSM & seq. FSM and also reducing the usage of the buffer indices for reducing fan-out. This will be explained further later on.

```
80    // Holds triple_buf row index (0,1,2)
81    logic [1:0] triple_buf_prev_row_idx;   // index of previous row within the triple row buffer
82    logic [1:0] triple_buf_crnt_row_idx;   // index of current row within the triple row buffer
83    logic [1:0] triple_buf_next_row_idx;   // index of next row within the triple row buffer
```

These are the window indices used for rotating the rows as guided by UDI.

```
85    //~~~~~~~~| CLK CYCLE PERFORMANCE |~~~~~~~~//
86
87
88    logic [MAX_HEIGHT-1:0]       is_zero_row_rd;   // Used for checking if the row we're about to read is 0.
89    logic [MAX_HEIGHT-1:0]       row_is_zero_wr;   // Used for tracking the zero rows written. Will be used for the iteration afterwards.
90    logic                        skip_next_row;    // Indicator for skipping next row
91
92    logic [31:0]                 num_itr_counter;  // Counter for number of iterations, used for sending done signal when = _num_itr_
93
94    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
```

The 1st 3 signals related to zero detection haven't been realized yet sadly due to lack of time, but they're left in there because I'm planning on finishing those once I'm done with the exams, but their idea is really neat as they allow us to track the zero rows we're writing into our memory and from the 2nd iteration and forward we could detect zero rows and skip them really fast. Furthermore, a few more ideas came to mind about 1-row detection which is also a possible combination that may enable us to quickly run through the rows.

num_itr_counter – A simple counter for the iterations.

```
109   enum {
110       GRID_BASE_ADDR_REG_IDX,     // Index of the register holding the base address
111       GRID_WIDTH_REG_IDX,         // Index of the register holding the grid width
112       GRID_HEIGHT_REG_IDX,        // Index of the register holding the grid height
113       START_REG_IDX,              // Index of the register indicating START
114       DONE_REG_IDX,               // Index of the register indicating DONE
115       NUM_ITR_REG_IDX             // Index of the register holding the number of iterations
116   } regs_idx;
```

These are the GPP control's indices where the new addition is **NUM_ITR_REG_IDX**, this is part of the HW-SW handshake optimization which was done in the .c file as well and will be explained later on.

Host Regs Interface

These are pretty standard and unchanged from the previous non-wrapped design.

```
120  /*****************************************************************/
121  /*                       HOST REGS INTERFACE                     */
122  /*****************************************************************/
123
124  logic [XBOX_NUM_REGS-1:0][31:0] host_regs_data_out_ps; // pre-sampled
125
126  always_comb begin // Update comb logic                                  // default the pre sample regs out
127                                                  host_regs_data_out_ps                = host_regs_data_out;
128    if      (done                                ) host_regs_data_out_ps[DONE_REG_IDX][0] = 1'b1;  // Update the done allocated register
129    else if (host_regs_read_pulse[DONE_REG_IDX]) host_regs_data_out_ps[DONE_REG_IDX][0] = '0;  // clear upon SW reading
130  end
131
132  // Sample host regs output
133  always @(posedge clk, negedge rst_n) begin
134    if(~rst_n) host_regs_data_out <= '0;
135    else       host_regs_data_out <= host_regs_data_out_ps;
136  end
137
138  // Update output host registers
139  always_comb begin
140    host_regs_valid_out              = '0; // Default
141    host_regs_valid_out[DONE_REG_IDX] = host_regs_data_out[DONE_REG_IDX][0];
142  end
```

```
144  // For convenience clone by assign the host regs values
145
146  assign start               = host_regs            [START_REG_IDX          ] &&
147                               host_regs_valid_pulse[START_REG_IDX          ]       ;
148  assign grid_xmem_base_addr = host_regs            [GRID_BASE_ADDR_REG_IDX ][15:0] ;
149  assign grid_width          = host_regs            [GRID_WIDTH_REG_IDX     ]       ;  // assumed to be an integer and not bigger than M
150  assign grid_height         = host_regs            [GRID_HEIGHT_REG_IDX    ][6:0]  ;  // grid_heigh is 7 bits wide, (64)(bin) = 7'b1000
151  assign _num_itr_           = host_regs            [NUM_ITR_REG_IDX        ]       ;
152  assign clear_done_on_read  = host_regs_read_pulse [DONE_REG_IDX           ]       ;
153
154  assign bytes_per_grid_row  = grid_width >> 3; // Dividing by 8
155
```

The addition observed here is assigning **_num_itr_** to hold the information of the total number of iterations required.

The State Machine

Intro – The following state machine is built with 2 major targets in mind, the first one is clk cycle reduction which is according to the assignment's requirements, however, my other important goal was STA performance where I was aiming to test out all of my various tricks and test different hypotheses on critical path, fanout and synthesizer awareness in the design.

Among the different implementations I was heavily aiming on reducing the depth of the different MUX blocks, replacing comparators with bit-masking tricks, inserting single DFFs for Area vs. Performance trade-offs and maximizing the freely available states in order to off-load the READ state which I identified as the critical path after repeating tests.

```
158  /*****************************************************************/
159  /*                          STATE MACHINE                        */
160  /*****************************************************************/
161
162  /*-------------------------*/
163  /*       Definitions       */
164  /*-------------------------*/
165
166  typedef enum logic [6:0] { // 1-Hot Coding for Performance (Sacrificing some Area 3 -> 7 DFFS)
167
168    IDLE      = 7'b0000001,
169    DBL_LOAD  = 7'b0000010,
170    LST_LOAD  = 7'b0000100,
171    WRITE     = 7'b0001000,
172    READ      = 7'b0010000,
173    LAST      = 7'b0100000,
174    DONE      = 7'b1000000
175
176  } state_machine;
177
178  state_machine next_state, state;
179
180  //~~~~~~~~| STATE MASKS |~~~~~~~~~//
181
182  typedef enum int {
183
184    IDLE_IDX      = 0,
185    DBL_LOAD_IDX  = 1,
186    LST_LOAD_IDX  = 2,
187    WRITE_IDX     = 3,
188    READ_IDX      = 4,
189    LAST_IDX      = 5,
190    DONE_IDX      = 6
191
192  } state_index;
193
194  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
```

Starting with the definitions for the SM, choosing 1-Hot Coding combined with bit-masking was the elite choice after comparing it with various other options such as leaving **state** & **next_state** with a 3-bit binary coding and another attempt of bit-masking the 3-bit binary coding to reduce all the if-else condition bit-widths. Sacrificing 4 DFFs was definitely worth it.

```
197   /*--------------------------*/
198   /*       Combinatorial      */
199   /*--------------------------*/
200
201   always_comb begin
202
203     // Defaults
204
205     next_state                    = state;
206     mem_intf_write.mem_size_bytes = bytes_per_grid_row[5:0];
207     mem_intf_read.mem_size_bytes  = bytes_per_grid_row[5:0];
208     mem_intf_write.mem_data       = updated_row;
209     mem_intf_write.mem_start_addr = crnt_wr_addr;
210     mem_intf_read.mem_start_addr  = crnt_rd_addr;
211     mem_intf_read.mem_req         = '0;
212     mem_intf_write.mem_req        = '0;
213     done                          = 1'b0;
```

Stepping into the comb. block we're starting with the default values which are pretty standard and ensure that there's no accidental latches whatsoever.

```
215   case (state) // State Machine case
216
217     IDLE: begin
218
219       if (start) begin
220         mem_intf_read.mem_req = 1'b1;
221         next_state            = DBL_LOAD;
222       end
223
224     end
```

IDLE – The 1st case state is a simple clean IDLE, where I'm raising a read request immediately in order to make the data available in the next clk cycle while moving on to **DBL_LOAD**.

```
226     DBL_LOAD: begin // Used for Reading First 2 Rows into tripl
227
228       if (mem_intf_read.mem_valid) begin
229         mem_intf_read.mem_req = 1'b1; // Using the flag to safe
230         next_state = state_machine'(DBL_LOAD << dbl_load_idx);
231       end
232
233     end
```

DBL_LOAD is a unique state with 2 purposes – which will be seen in the in sequential block but the main purpose which is also evident here was off-loading the **READ** state which had a very long critical path. As we can see, we're raising another read request and I'm using a neat trick with my ENUMs, the **dbl_load_idx** variable, and the 1-hot encoding to avoid using an adder and a MUX at the cost of 1 DFF. Definitely worth the trade both performance-wise and area-wise.

```
235     LST_LOAD: begin // Used for Reading 3rd
236
237       if (mem_intf_read.mem_valid) begin
238         mem_intf_write.mem_req  = 1'b1; //
239         mem_intf_read.mem_req   = 1'b0; //
240         next_state              = WRITE;
241       end
242
243     end
```

LST_LOAD – This state is also a unique state for the 3rd and final row needed to be read from the XMEM before beginning the processing. Note that while it could've been merged with the READ state, decoupling it was the better approach as it enabled me to further reduce the critical path from the READ state and another small trick in the sequential block which will be seen later for maximizing the performance within a single clock cycle.

```
245       WRITE: begin
246
247         if (mem_intf_write.mem_ack) begin
248           mem_intf_write.mem_req  = 1'b0;
249
250           if (grid_rd_row_idx[6:4] == grid_height[6:4]) begin
251             mem_intf_read.mem_req = 1'b0;
252             next_state = LAST;
253           end else begin
254             mem_intf_read.mem_req = 1'b1;
255             next_state = READ;
256           end
257         end
258       end
```

WRITE – One important thing that's crucial to note here is that the **WRITE & READ** names aren't exact due to the timing of the pipelining where I began with inserting an initial fake write into row[0] (which will be overwritten) followed by true writings going from 1,2, … , $M - 1$,0.

Optimization – Note the truncated if condition, this is used after realizing that the possible row sizes are either 16,32,48,64 which allowed me to create a simple condition for exiting the **READ** − **WRITE** ping-pong mechanism of the FSM pipeline based on the 3 MSBs of these 2 signals, moving into **LAST** state once we're done reading row[N-1]. Note that rows 0,1 are available within my internal double buffer which will be re-assigned into **triple_row_buf**, further decoupling the critical path from **READ** state. The only reason this choice is 100% worth it (after testing with \ without this design choice) is due to solely focusing on performance therefore reducing the critical path and the fanout on some signals was the main goal here.

```
READ: begin
  if (mem_intf_read.mem_valid) begin
    mem_intf_write.mem_req  = 1'b1;
    mem_intf_read.mem_req   = 1'b0;
    next_state              = WRITE;
  end
end
```

Now it is pretty clear (I hope) why the **READ** state looks so clean and minimalistic.

```
LAST: begin
  mem_intf_write.mem_req        = 1'b1;
  if (is_last_row)  next_state  = DONE; //
  else              next_state  = WRITE;
end
```

LAST – This state is being used 3 times due to the pipeline timing considerations whereas the 1st entry is for loading row[0] into the next row for processing row[N-1]. The 2nd entry is for loading row[1] into the next row for processing row[0] which is overwriting the fake write.

The 3rd entry will be clearer through the seq. Block as its main purpose is to enable 1 more clk cycle for loading the row and finishing the writing of row[0] and also to increment the number of iterations by 1 before completely finishing the iteration.

```
274       DONE: begin
275         if (is_last_itr) begin
276           done = 1'b1;
277           if (clear_done_on_read)
278             next_state            = IDLE;
279
280         end else begin // Start Next Iteration
281             mem_intf_read.mem_req = 1'b1;
282             next_state            = DBL_LOAD;
283         end
284       end
```

DONE – This state gained a double purpose once the SW-HW HS opt. Was realized, it's added purpose was to determine whether we're at the last iteration or not, and if not, then we're mimicking all the assignments done for **start** = **1'b1**.

As we can see, we're initiating another early read request and moving on to **DBL_LOAD** once again.

```
// last logic comb assignment.
assign next_is_last_row     = (grid_wr_row_idx + 2 == grid_height);
assign is_last_row          = (grid_wr_row_idx + 1 == grid_height);

assign is_last_itr          = ((|_num_itr_) & (num_itr_counter == _num_itr_));
```

Finally, 3 simple logic comb. Assignments where **is_last_itr** is the addition where we check if we're at the last iteration and we're safe-guarding this continuous assign statement with a simple bit-wise OR to ensure the signal remains low prior to receiving valid data through the GPP.

```
298  /*----------------------------*/
299  /*         Sequentials        */
300  /*----------------------------*/
301
302  //~~~~~~~~~~~~~~~~~~~~~~~~~| FSM - ITR Counter |~~~~~~~~~~~~~~~~~~~~~~~~~~//
303
304  always @(posedge clk or negedge rst_n) begin // Incrementing the counter in the 1st time we
305    if (!rst_n) num_itr_counter <= '0;
306    else if (state[LAST_IDX] && next_is_last_row) num_itr_counter <= num_itr_counter + 1'b1;
307  end
```

Moving on to the sequentials which was really challenging at first. The main objectives was to strive for readability, correct structure, minimization of unnecessary conditions, signal grouping, and trying to reduce every little LE I could to see how far I can go, the biggest win here was the ability to completely shred-off and change the Sequential block once I've made a logical grouping of the signals, Confirming what we've been taught in DVD and also in the exercises of this course as the smartest approach.

The first seq. Block was a simple iteration counter designed to increment the iteration count before moving on to **DONE** where we decide if a new iteration will be started or a **done** signal sent to the SW.

```
309  //~~~~~~~~~~~~~~~~~~~~~~~~~| FSM - State Seq |~~~~~~~~~~~~~~~~~~~~~~~~~~//
310
311  always @(posedge clk or negedge rst_n) begin
312    if (!rst_n) state <= IDLE;
313    else        state <= next_state;
314  end
```

A simple SM block also used to further improve readability.

```
316  //~~~~~~~~~~~~~~~~~~~~~~~~| FSM - Windows-Be-Sliding |~~~~~~~~~~~~~~~~~~~~~~~~//
317
318  always @(posedge clk or negedge rst_n) begin
319
320    if (!rst_n) begin
321
322      dbl_load_idx              <= '0;
323      double_row_hold_buf       <= '0;
324      triple_row_buf            <= '0;
325
326      start_next_itr            <= '0;
327
328    end else if (start || start_next_itr) begin
329
330      dbl_load_idx              <= '0;
331      double_row_hold_buf       <= '0;
332      triple_row_buf            <= '0;
333
334      start_next_itr            <= '0;
```

This is the heart of the seq. Block, starting with the simple reset & **start\start_next_itr** signals used to make sure everything is flushed out properly between iterations.

```
336     end else begin
337
338       if (state[DBL_LOAD_IDX]) begin // dbl_load_idx is used as decoupling the critical path tha
339         triple_row_buf     [dbl_load_idx] <= mem_intf_read.mem_data[MAX_WIDTH_BYTES-1:0];
340         double_row_hold_buf[dbl_load_idx] <= mem_intf_read.mem_data[MAX_WIDTH_BYTES-1:0]; // Sto
341         dbl_load_idx <= !dbl_load_idx; // Swap dbl_load_idx from 0 -> 1 and back to 0.
342       end else if (state[LST_LOAD_IDX]) // 3rd Read of Row[2]
343         triple_row_buf[2] <= mem_intf_read.mem_data[MAX_WIDTH_BYTES-1:0]; // Direct 2 for a sing
344
345       else if (state[READ_IDX])
346         triple_row_buf[triple_buf_next_row_idx] <= mem_intf_read.mem_data[MAX_WIDTH_BYTES-1:0];
347
348       else if (state[LAST_IDX]) begin // Load the stored rows, Row[N-1] is stored in Prev Row !
349         if (is_last_row)
350           start_next_itr <= 1'b1; // Asserting the flag for next iteration
351         else if (next_is_last_row) // Next Row is Row(1) used for calculating Row[0]
352           triple_row_buf[triple_buf_next_row_idx] <= double_row_hold_buf[1];
353         else              // For computing Row(N-1), next_row is Row[0]
354           triple_row_buf[triple_buf_next_row_idx] <= double_row_hold_buf[0];
355       end
356     end
357   end
```

Moving into the 2nd part of this seq. Block, one can see the **if** conditions with the bit-masking technique allowing the usage of a single-bit entry enabling the most reduced form of MUX I could create, while focusing on making the **select** bit-width the smallest possible.

DBL_LOAD – Knowing that the 1st 2 rows are loaded into $idx = 0,1$ I'm reusing **dbl_load_idx** instead of the designated window indices to reduce the fanout and with a simple NOT gate I'm flipping the value of **dbl_load_idx** so it matches the intended behavior perfectly.

LST_LOAD – Again, taking advantage of the prior knowledge that the 3rd row will be read into $idx = 2$, the best approach is to explicitly assign it, further reducing the fanout on the index signals.

READ – This is the final reduced form after completely taking apart the critical path and letting it be as simple as possible.

LAST – This state has a nested MUX with depth $= 2$ as it's corresponding to those 3 different stages explained before.

```
359  //~~~~~~~~~~~~~~~~~~~~~| FSM - READ Addr & Counter |~~~~~~~~~~~~~~~~~~~~~//
360
361  wire        next_is_last_rd = (grid_rd_row_idx + 1 == grid_height);
362  wire [15:0] next_rd_addr    = (next_is_last_rd) ? grid_xmem_base_addr : crnt_rd_addr + bytes_per_grid_row[15:0];
363
364  always @(posedge clk or negedge rst_n) begin
365    if (!rst_n) begin
366      crnt_rd_addr            <= '0;
367      grid_rd_row_idx         <= '0;
368    end else begin // None of these signal are mutually exclusing - Independent if statements for each.
369      if (start || start_next_itr)
370        grid_rd_row_idx       <= '0; // Set to '0 on purpose, used for 3 first LOADs.
371
372      if (mem_intf_read.mem_valid)
373        grid_rd_row_idx       <= grid_rd_row_idx + 7'h1;
374
375      if (mem_intf_read.mem_req)
376        crnt_rd_addr          <= next_rd_addr;
377    end
378  end
```

The next group contains all the **READ** signals including some useful flags computed with direct usage of **wire** because it's more concise this way, so it's complimenting the goal for readability.

Note – Parallel if statements & **Mutual Exclusivity** :

One of the more interesting and crucial things to pay attention to is **Mutual Exclusivity** and the corresponding **if – else** statements. After countless of attempts for finding out the trade-offs between if-else ifs and parallel ifs and how the synthesizer reacts to them, I can conclude that **parallel ifs** are crucial when signals are non-mutual exclusive and usually the synthesizer (and Quartus in particular) can heavily optimize if-else if conditions when signals are **Mutual Exclusive**, therefore this a special case where it wasn't possible, but in the other blocks, the minimum depth for the muxes was prioritized and if-else if statements which were the best in performance out of the other attempts I tried.

Other than that, this is a pretty simple FSM block as well with the only highlight that in the **wire** assignment I'm making sure to reset **crnt_rd_addr** we're done reading and I've completely decoupled the logic out except for **grid_rd_row_idx**.

```
380   //~~~~~~~~~~~~~~~~~~~~| FSM - WRITE Addr & Counter |~~~~~~~~~~~~~~~~~~~~//
381
382   wire [15:0] next_wr_addr = (is_last_row || next_is_last_row) ? grid_xmem_base_addr : (crnt_wr_addr + bytes_per_grid_row[15:0]);
383   // Consider removing the ternary if it can be done - TODO
384   wire row_is_zero_wr_ps   = is_fake_n ? (~(|updated_row_ps[MAX_WIDTH_BYTES-1:0])) : '0; // Bit-Wise OR Followed by NOT
385
```

```
386   always @(posedge clk or negedge rst_n) begin
387     if (!rst_n) begin
388       crnt_wr_addr                  <= '0;
389       grid_wr_row_idx               <= '0;
390       is_fake_n                     <= '0;
391
392       updated_row                   <= '0;
393
394       triple_buf_prev_row_idx       <= '0;
395       triple_buf_crnt_row_idx       <= '0;
396       triple_buf_next_row_idx       <= '0;
397
398       row_is_zero_wr                <= '0;
399     end else if (start || start_next_itr) begin
400       grid_wr_row_idx               <= '0;
401       is_fake_n                     <= '0;
402
403       updated_row                   <= '0;
404
405       triple_buf_prev_row_idx       <= 2'b00;
406       triple_buf_crnt_row_idx       <= 2'b01;
407       triple_buf_next_row_idx       <= 2'b10;
```

Next comes up the **WRITE & Counter** related signals where I was able to speed up the design by a whopping 7 MHZ after applying **Consolidation** of small signals from 2-3 different always blocks to reduce critical paths.

← Note that among the **WRITE** and **Counter** signals I've also moved the window index pointers here because this was part of the signal grouping under the same condition which was a smarter choice for performance over a slightly less structured signal organization.

```
408     end else begin
409       if (mem_intf_write.mem_req) begin
410         crnt_wr_addr                  <= next_wr_addr;
411         row_is_zero_wr[grid_wr_row_idx] <= row_is_zero_wr_ps;
412       end
413
414       else if (mem_intf_write.mem_ack) begin // !state[DBL_LOAD_IDX] use
415         grid_wr_row_idx               <= grid_wr_row_idx + is_fake_n;
416         is_fake_n                     <= 1'b1;
417
418         updated_row                   <= updated_row_ps;
419
420         triple_buf_prev_row_idx       <= triple_buf_crnt_row_idx; // r
421         triple_buf_crnt_row_idx       <= triple_buf_next_row_idx;
422         triple_buf_next_row_idx       <= triple_buf_prev_row_idx;
423       end
424     end
425   end
```

Here's the core part of this FSM block, the highlight of this part is mainly the usage of **is_fake_n** which TRUE for $1'b0$ (same idea like **rst_n**) but it is also used to increment **grid_wr_row_idx** enabling me to avoid MUXes and unwanted comparators for making sure the wr counter doesn't increment unwantedly.

CGOL TORUS PROCESSING – The Heart Of The Design

```
35  /*****************************************************************************/
36  /*                                THE MASK                                   */
37  /*****************************************************************************/
38
39  //~~~~~~~~~| THE CGOL MASKS |~~~~~~~~~~//
40
41  localparam M_CGOL = 20'b0000_0000_0010_1100_0000; // Mega Mask
42
43  //localparam M_CGOL = 18'b00_0000_0000_1110_0000; // The Semi-Mega Mask
```

Starting with the bit-mask itself, I have created a concatenated signal which is going into this static variable enabling the reduction of all the comparators muxes and even a subtractor into a hardwired ultra-fast LUT which as really improved the design, I left the previous version (namely **The Semi-Mega Mask**) for an example of new ideas that kept coming up for reducing the design over & over again.

```
429 /*****************************************************************************/
430 /*                          TORUS CGOL PROCESSING                            */
431 /*****************************************************************************/
432
433 // tree adder implementation to calculate updated row to be written back ro grid - less
434
435 wire [6:0] grid_width_opt = grid_width[6:0]; // By-Passing the annoying synthesis warni
436
437 logic [1:0] sum_col [0:MAX_WIDTH-1]; // a 2-bit 1-D array for the 1st stage of vertical
438
439 always_comb begin // Stage 1 - Vertical Sum across columns
440   for (int i = 0; i < MAX_WIDTH; i++) begin
441     if (i < grid_width_opt) begin
442       sum_col[i] = triple_row_buf[0][i] + triple_row_buf[1][i] + triple_row_buf[2][i];
443     end else begin
444       sum_col[i] = 2'b00; // bit masking
445     end
446   end
447 end
```

Another huge optimization enabling me to reach $F_{max} \approx 81[MHz]$ was **grid_width_opt** after noticing that I'm using a 32-bit wide signal within the condition statements for a number requiring only 7-bits. The reason for concatenating it indirectly was due to a weird non-justified bug coming from the dummy wrapper when trying to change the bit-width of **grid_width** leaving me with no choice but a huge win anyways.

Stage 1 – Vertical Sum Across Columns – This is the 1st stage where I'm opting for fan-out reduction followed reducing the logic depth and bit-masking the unused elements.

Stage 2 - Reusing Pre-Calculated Sums for each cell

```
449  always_comb begin // Stage 2 - Reusing Pre-Calculated Sums for each cell
450    for (int i = 0; i < MAX_WIDTH; i++) begin
451
452      logic [3:0] sum_neighs; // Local Variable for summing relevant cols (Initialized to 0)
453
454      if (i < grid_width_opt) begin
455        logic i_cell;
456        i_cell = triple_row_buf[triple_buf_crnt_row_idx][i]; // Cell i in Current Row
457
458        if (i == 1'b0) sum_neighs = sum_col[grid_width_opt-1] + sum_col[i] + sum_col[i+1];
459        else if (i == grid_width_opt - 1 || (i == MAX_WIDTH - 1)) sum_neighs = sum_col[i-1] + sum_col[i] + sum_col[0];
460        else sum_neighs = sum_col[i-1] + sum_col[i] + sum_col[i+1];
461
462        updated_row_ps[i] = M_CGOL[{sum_neighs, i_cell}];
463
464      end else begin
465
466        updated_row_ps[i] = 1'b0; // Masking
467        sum_neighs        = 4'd0; // Default value for sum_neighs
468
469      end
470    end
471  end
```

This is where another huge advantage comes into play, we're reusing the vertical sum results in **sum_col** while accounting for the wrap-around in a pretty intuitive way and once ***sum_neighs*** is computed we're concatenating it with the correct **i_cell** which holds the true state of the current cell.

Important – CGOL Rules "Modified" – Here comes a relatively small but cool optimization, instead of subtracting the value of **i_cell** from **sum_neighs** we can slightly alter the rules matching the idea of counting 9 neighbors instead of 8 which fits well into this structure, the rule remains the same and nothing is really changed.

How is it done? – We're laying out all of the possible combinations for the results when $\mathbf{i_{cell}} = \mathbf{1}$

Which is either $\mathbf{sum\_neighs} = \mathbf{4'0100}, \mathbf{or\ 4'0011}$, doing the same for $\mathbf{i_{cell}} = \mathbf{0}$ we know that we'll get 1 if $\mathbf{sum\_neighs} = \mathbf{4'0011}$ therefore all of the possible combinations are:

$$5'01001 = 2^0 + 2^3 = 9$$

$$5'00111 = 2^0 + 2^1 + 2^2 = 7$$

$$5'00110 = 2^2 + 2^1 = 6$$

We can see that instead of 2 or 3, we're looking for 3 or 4 if the cell is alive because if it's alive it means that it's counted in, but if the cell is dead then we're using the usual REBIRTH rule, explaining why the MASK looks like this :

```
35  /*****************************************************************************/
36  /*                                THE MASK                                   */
37  /*****************************************************************************/
38
39  //~~~~~~~~| THE CGOL MASKS |~~~~~~~~~//
40
41  localparam M_CGOL = 20'b0000_0000_0010_1100_0000; // Mega Mask
```

Finally, the choice for the 20-bit width was for choosing the largest possible result we're required to return 0 for, leaving no room for mistakes and errors.

C Software – Quick overview of the Adjustments

```
enum {
    GRID_BASE_ADDR_REG_IDX,
    GRID_WIDTH_REG_IDX,
    GRID_HEIGHT_REG_IDX,
    START_REG_IDX,
    DONE_REG_IDX,
    NUM_ITR_REG_IDX
};

#define GRID_BASE_ADDR_REG ((volatile unsigned int *) (XBOX_REGS_BASE_ADDR + (4*GRID_BASE_ADDR_REG_IDX)))
#define GRID_WIDTH_REG     ((volatile unsigned int *) (XBOX_REGS_BASE_ADDR + (4*GRID_WIDTH_REG_IDX)))
#define GRID_HEIGHT_REG    ((volatile unsigned int *) (XBOX_REGS_BASE_ADDR + (4*GRID_HEIGHT_REG_IDX)))
#define START_REG          ((volatile unsigned int *) (XBOX_REGS_BASE_ADDR + (4*START_REG_IDX)))
#define DONE_REG           ((volatile unsigned int *) (XBOX_REGS_BASE_ADDR + (4*DONE_REG_IDX)))
#define NUM_ITR_REG        ((volatile unsigned int *) (XBOX_REGS_BASE_ADDR + (4*NUM_ITR_REG_IDX)))
```

Starting with the GPP control signals, here's the **NUM_ITR_REG** control signal I added for providing the HW with the information about the number of iterations.

```
void cgol_xlr_init(cgol_conf_t* cgol_conf_p) {


  *GRID_BASE_ADDR_REG = XSPACE_BASE_ADDR;
  *GRID_HEIGHT_REG    = cgol_conf_p->height;
  *GRID_WIDTH_REG     = cgol_conf_p->width;
  *NUM_ITR_REG        = _NUM_ITR_;
}
```

Followed by a simple modification of the XLR initialization function.

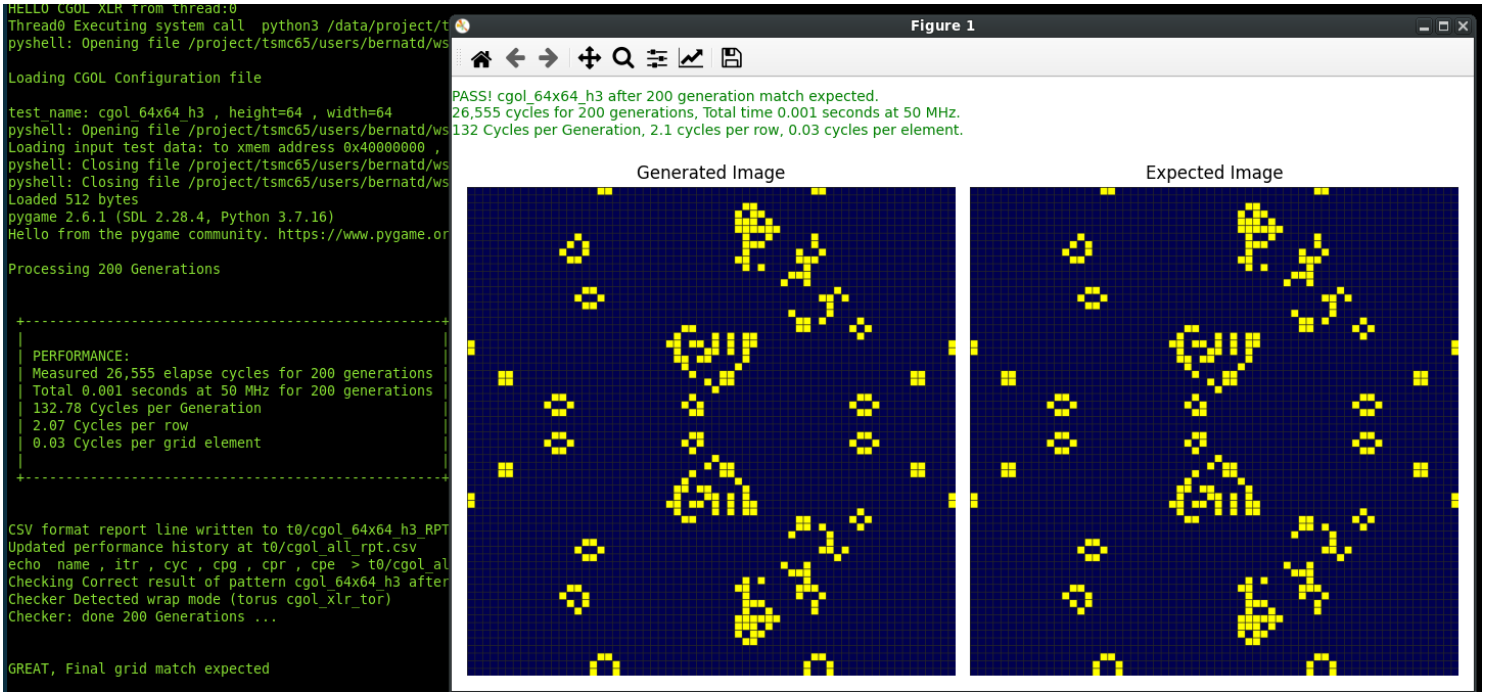
```
    cgol_xlr_init(&cgol_conf);

    mesure_init();

    *START_REG = 1; // Trigger Start
    while(!*DONE_REG){}

    mesure_itr();

    display_grid(_NUM_ITR_,1); // Always cal

    animate_terminate();  // Always call, wi

    report_mesure(0,1); // is_mt8=0 , is_xlr
```

Finishing strong with removing the for loop and the function call for **update_grid** and making it inline instead.

Note – **mesure_itr**() isn't necessarily used for a single iteration and after going over the function & **report_mesure**() carefully it was pretty straightforward that we can use it this way and achieve the optimized SW-HW handshake.

Performance Results

Getting to the interesting part, I'm starting with a simple 200 itr simulation for a 64 × 64 pattern of "h3" :



As we can see, we were already able to achieve an awesome (& correct) result of 2.07 clk cycles per row.

Quartus Synthesis Check

```
ip-10-70-128-168:bernatd:/data/project/tsmc65/users/bernatd/ws/DDP25/DDP-Projects/CGOL/hw/xlrs/cgol_xlr_tor>qsyn_xlr cgol_xlr_tor -all
Checking Synthesis ...
Done Synthesis ...


Total FPGA Logic Elements:       2,468
Total FPGA memory registers:       707
Total FPGA memory bits:              0
Info: Quartus Prime Analysis & Synthesis was successful. 0  errors, 0 non-justified warnings

NOTICE! Your design include some possible justified warnings as:
"No output dependent on input pin"
"input pin(s) that do not drive logic"
Please Carefully review the report and make sure the specific reported unused inputs are justified


For more details see qsyn_output_files/cgol_xlr_tor.map.rpt


Checking Fit ...


Info: Quartus Prime Fitter was successful. 0  errors, 0 non-justified warnings

For more details see qsyn_output_files/cgol_xlr_tor.fit.rpt


Checking Timing (STA) ...


Info: Quartus Prime Timing Analyzer was successful. 0  errors, 0 non-justified warnings

Max FPGA Frequency 78.88 MHz


For more details see qsyn_output_files/cgol_xlr_tor.sta.rpt


Generating qsyn submission archive: qsyn_cgol_xlr_tor_bernatd_180725_2155.tgz
```

As it is seen, the synthesis came our clean, with a relatively small drop of 2.5 [MHz] after inserting the optimization for SW-HW Handshake.

```
Total FPGA Logic Elements:       2,362
Total FPGA memory registers:       642
Total FPGA memory bits:              0
Info: Quartus Prime Analysis & Synthesis was successful. 0  errors, 0 non-justified warnings

NOTICE! Your design include some possible justified warnings as:
"No output dependent on input pin"
"input pin(s) that do not drive logic"
Please Carefully review the report and make sure the specific reported unused inputs are justified


For more details see qsyn_output_files/cgol_xlr_tor.map.rpt


Checking Fit ...


Info: Quartus Prime Fitter was successful. 0  errors, 0 non-justified warnings

For more details see qsyn_output_files/cgol_xlr_tor.fit.rpt


Checking Timing (STA) ...


Info: Quartus Prime Timing Analyzer was successful. 0  errors, 0 non-justified warnings

Max FPGA Frequency 81.54 MHz


For more details see qsyn_output_files/cgol_xlr_tor.sta.rpt


Generating qsyn submission archive: qsyn_cgol_xlr_tor_bernatd_180725_2200.tgz
```

The Best F_{max} per clk cycle:

I'm attaching the synthesizers results prior to the SW-HW Handshake for comparing the trade-offs between the highest performing design I was able to achieve before opting for further reduced clk cycles, this design is the full-pipelined FSM without SW-HW Handshake where I achieved:

$$F_{max} = 81.54[MHz]$$

This designs performance was ~4.2 clk cycles per row which was also impressive considering that no reduction was made on the SW at this point.

We can see that the integration of the SW-HW opting cost me around ~60 DFFs and ~100 LEs and a relatively small loss of ~2.6 [MHz] (although I'm aware of the FPGA limiting to ~50[MHz])

Bitstream File Creation & Final Synthesis Check

```
ip-10-70-128-168:bernatd:/data/project/tsmc65/users/bernatd/ws/DDP25/DDP-Projects/CGOL/hw/gen_fpga>comp_fpga cgol_xlr_tor -all
Synthesis mapping...
Done Synthesis ...


Total FPGA Logic Elements (out of about 50K available):   23,948
Total FPGA memory registers:     2,074
Total FPGA memory bits (out of about 1.6 Mbit available): 1,327,704
Info: Quartus Prime Analysis & Synthesis was successful. 0  errors, 0 non-justified warnings

For more details see output_files/k5_xbox_rc3.map.rpt


Checking Fit ...


Info: Quartus Prime Fitter was successful. 0  errors, 0 non-justified warnings

For more details see output_files/k5_xbox_rc3.fit.rpt


Checking Timing (STA) ...


Info: Quartus Prime Timing Analyzer was successful. 0  errors, 0 non-justified warnings

Max FPGA Frequency 59.46 MHz
```

Here's the final synthesis check while creating the bitstream file, we can see that the design is clean, and everything is tight as intended.

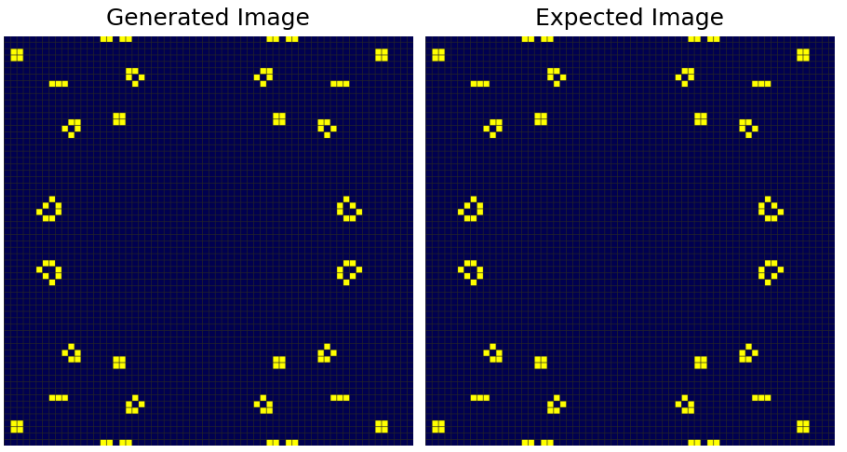
FPGA Simulation Results

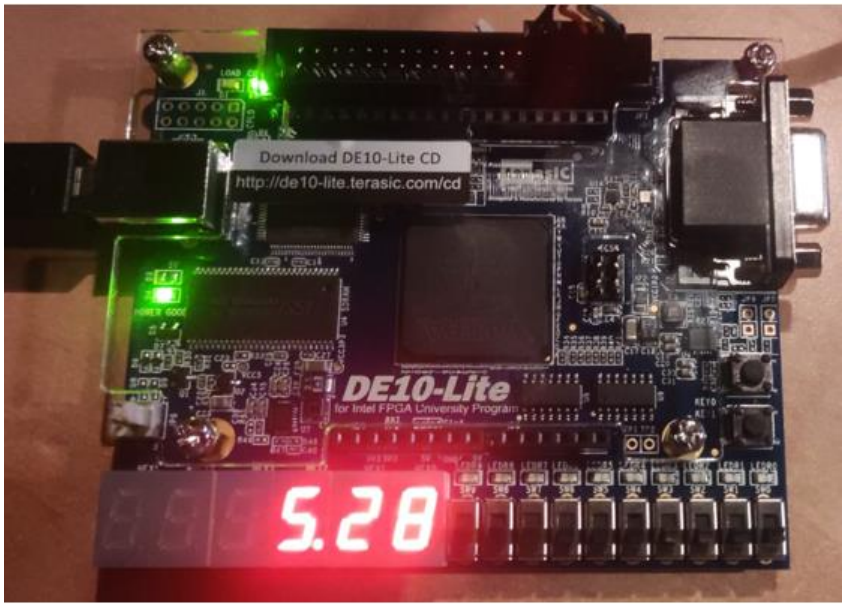
This section presents the execution of all CGOL patterns using the 7-segment display to capture accurate runtime measurements on the FPGA.

cgol_64 × 64_h1

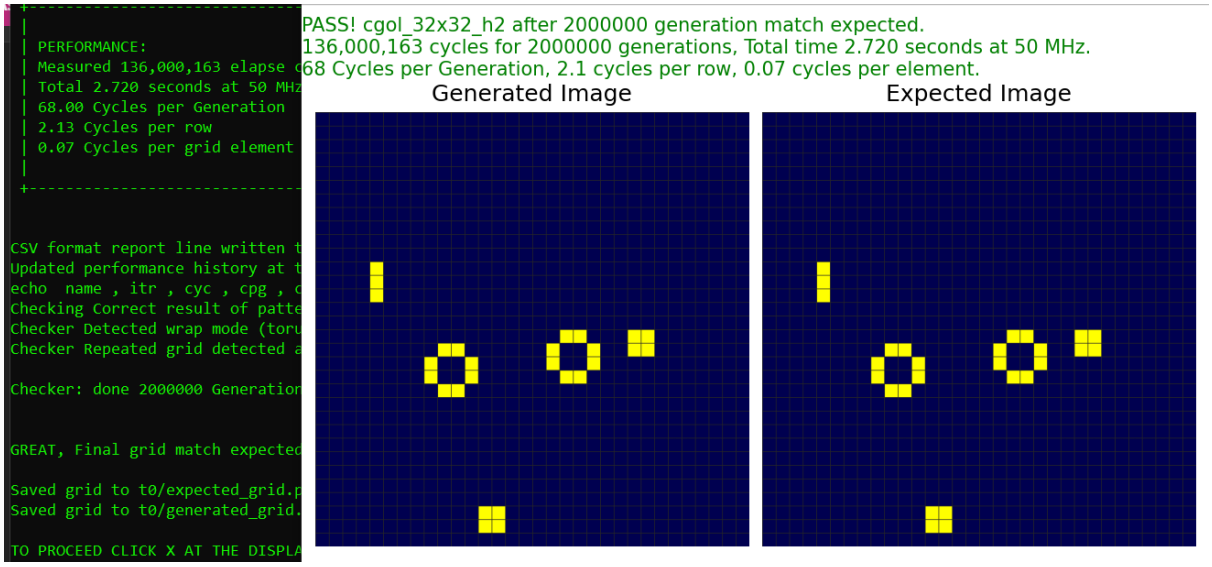
```
+---------------------------------
|
| PERFORMANCE:
| Measured 264,000,163 elapse cycles
| Total 5.280 seconds at 50 MHz for
| 132.00 Cycles per Generation
| 2.06 Cycles per row
| 0.03 Cycles per grid element
|
+---------------------------------


CSV format report line written to t0/
Updated performance history at t0/cgo
echo  name , itr , cyc , cpg , cpr ,
Checking Correct result of pattern cg
Checker Detected wrap mode (torus cgo
Checker Repeated grid detected at gen

Checker: done 2000000 Generations ...


GREAT, Final grid match expected

Saved grid to t0/expected_grid.png
Saved grid to t0/generated_grid.png

TO PROCEED CLICK X AT THE DISPLAYED W
```

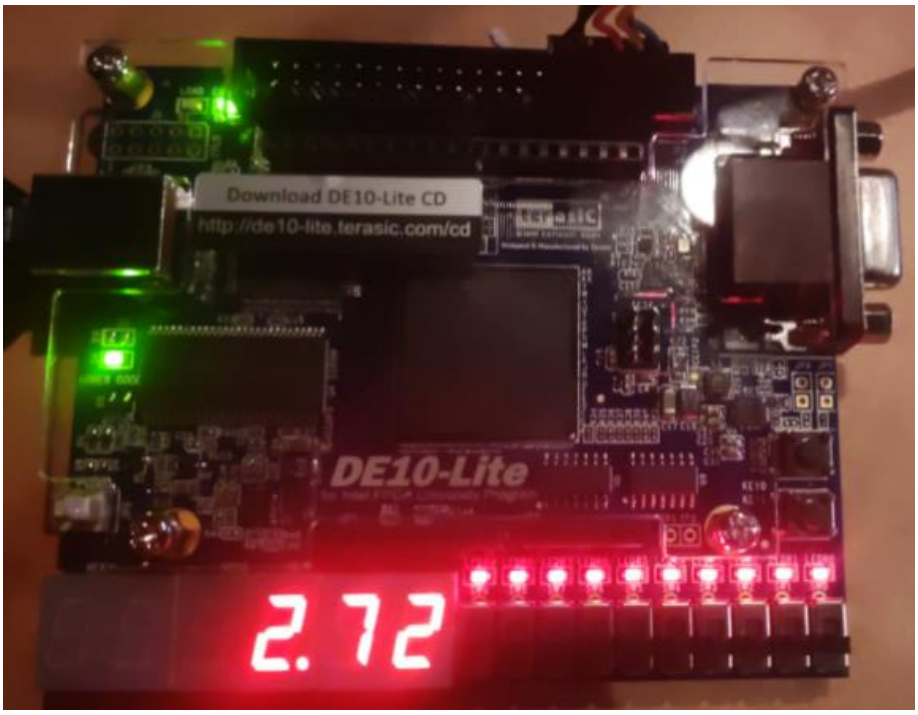
```
PASS! cgol_64x64_h1 after 2000000 generation match expected.
264,000,163 cycles for 2000000 generations, Total time 5.280 seconds at 50 MHz.
132 Cycles per Generation, 2.1 cycles per row, 0.03 cycles per element.
```





cgol_32 × 32_h2

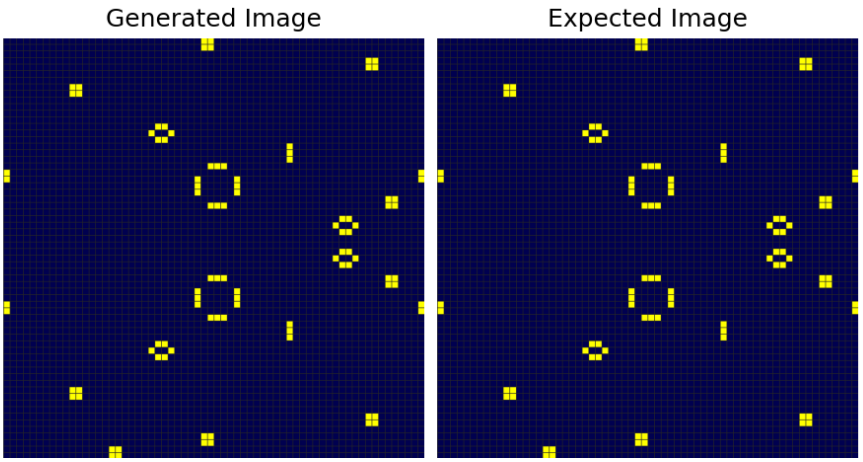

PERFORMANCE:
Measured 136,000,163 elapse c
Total 2.720 seconds at 50 MHz
68.00 Cycles per Generation
2.13 Cycles per row
0.07 Cycles per grid element
CSV format report line written t
Updated performance history at t
echo name , itr , cyc , cpg , c
Checking Correct result of patte
Checker Detected wrap mode (toru
Checker Repeated grid detected a
Checker: done 2000000 Generation
GREAT, Final grid match expected
Saved grid to t0/expected_grid.p
Saved grid to t0/generated_grid.
TO PROCEED CLICK X AT THE DISPLA
PASS! cgol_32x32_h2 after 2000000 generation match expected.
136,000,163 cycles for 2000000 generations, Total time 2.720 seconds at 50 MHz.
68 Cycles per Generation, 2.1 cycles per row, 0.07 cycles per element.
Generated Image
Expected Image

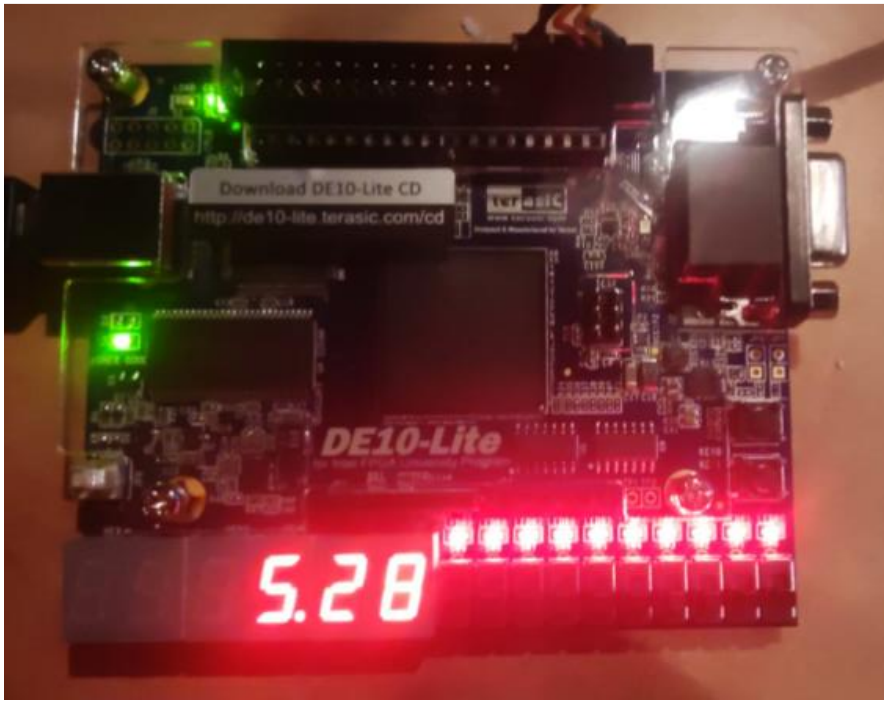

Download DE10-Lite CD
http://de10-lite.terasic.com/cd
DE10-Lite
2.72

cgol_64 × 64_h3

```
 |
 | PERFORMANCE:
 | Measured 264,000,163 elapse cy
 | Total 5.280 seconds at 50 MHz
 | 132.00 Cycles per Generation
 | 2.06 Cycles per row
 | 0.03 Cycles per grid element
 |
 +------------------------------


CSV format report line written to
Updated performance history at t0
echo  name , itr , cyc , cpg , cp
Checking Correct result of patter
Checker Detected wrap mode (torus
Checker Repeated grid detected at

Checker: done 2000000 Generations


GREAT, Final grid match expected

Saved grid to t0/expected_grid.pn
```

PASS! cgol_64x64_h3 after 2000000 generation match expected.
264,000,163 cycles for 2000000 generations, Total time 5.280 seconds at 50 MHz.
132 Cycles per Generation, 2.1 cycles per row, 0.03 cycles per element.

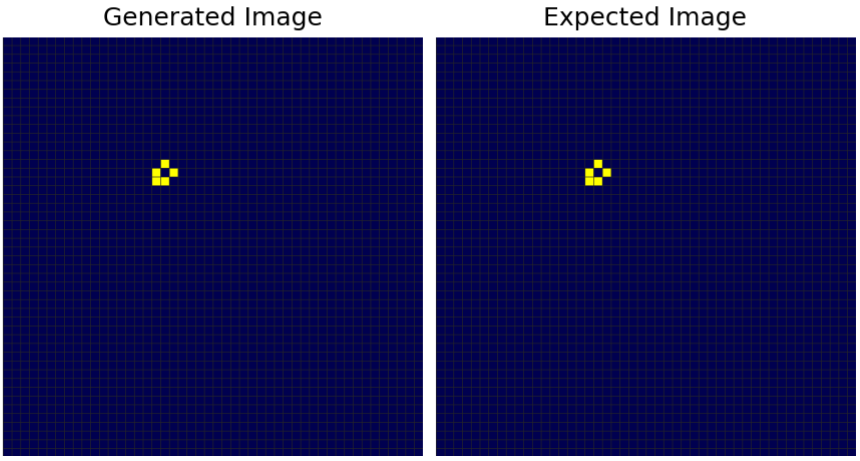


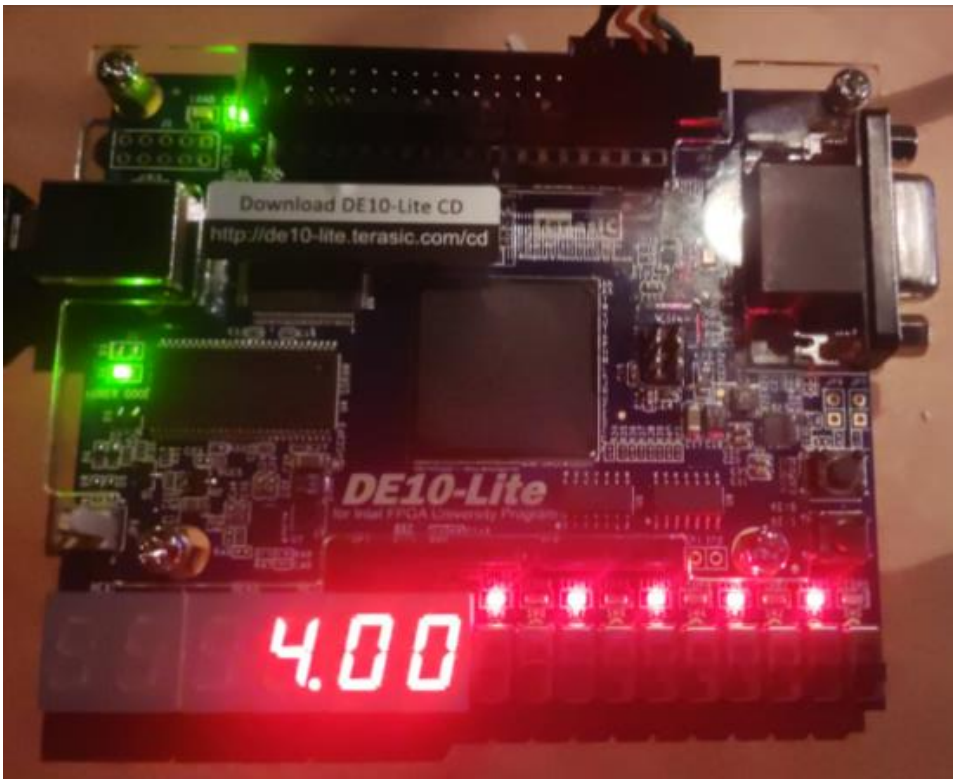


cgol_48 × 48_h4

```
| PERFORMANCE:
| Measured 200,000,163 elapse c
| Total 4.000 seconds at 50 MHz
| 100.00 Cycles per Generation
| 2.08 Cycles per row
| 0.04 Cycles per grid element

CSV format report line written t
Updated performance history at t
echo  name , itr , cyc , cpg , c
Checking Correct result of patte
Checker Detected wrap mode (toru
Checker Repeated grid detected a

Checker: done 2000000 Generation

GREAT, Final grid match expected

Saved grid to t0/expected_grid.p
```

PASS! cgol_48x48_h4 after 2000000 generation match expected.
200,000,163 cycles for 2000000 generations, Total time 4.000 seconds at 50 MHz.
100 Cycles per Generation, 2.1 cycles per row, 0.04 cycles per element.

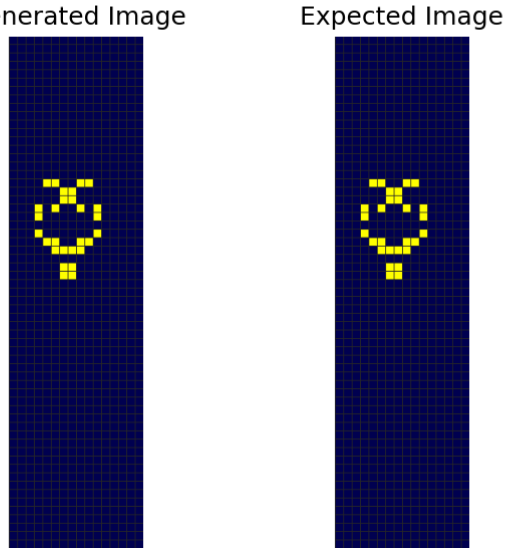


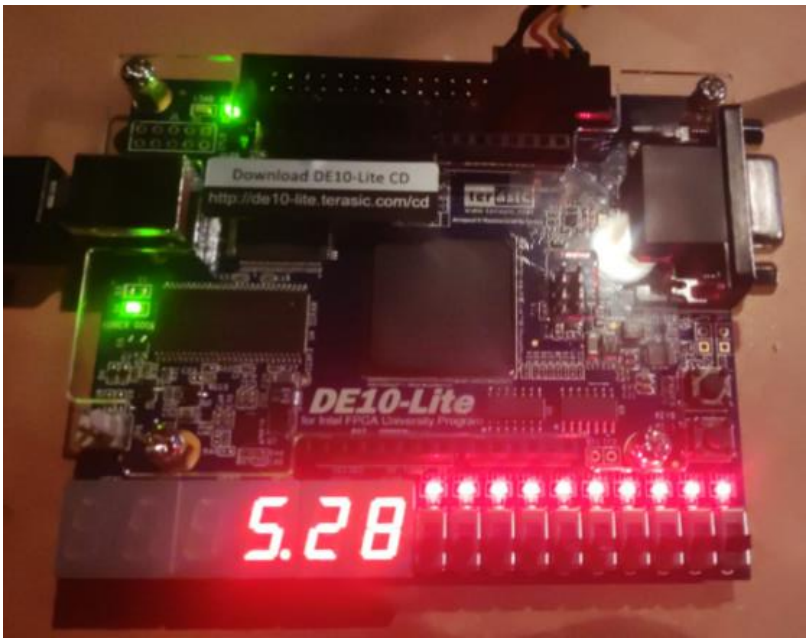


cgol_64 × 16_h5

```
+------------------------------
|
| PERFORMANCE:
| Measured 264,000,163 elapse cy
| Total 5.280 seconds at 50 MHz
| 132.00 Cycles per Generation
| 2.06 Cycles per row
| 0.13 Cycles per grid element
|
+------------------------------


CSV format report line written to
Updated performance history at t0/
echo  name , itr , cyc , cpg , cp
Checking Correct result of patter
Checker Detected wrap mode (torus
Checker Repeated grid detected at

Checker: done 2000000 Generations


GREAT, Final grid match expected

Saved grid to t0/expected_grid.pn
Saved grid to t0/generated_grid.p

TO PROCEED CLICK X AT THE DISPLAY
```

PASS! cgol_64x16_h5 after 2000000 generation match expected.
264,000,163 cycles for 2000000 generations, Total time 5.280 seconds at 50 MHz.
132 Cycles per Generation, 2.1 cycles per row, 0.13 cycles per element.



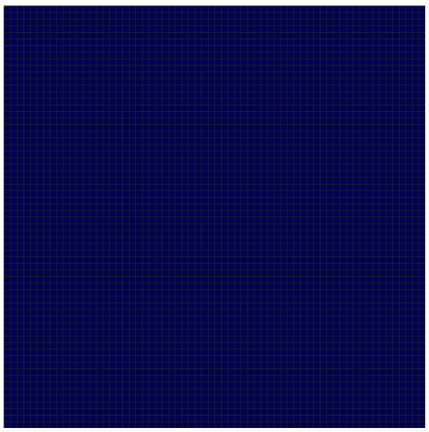


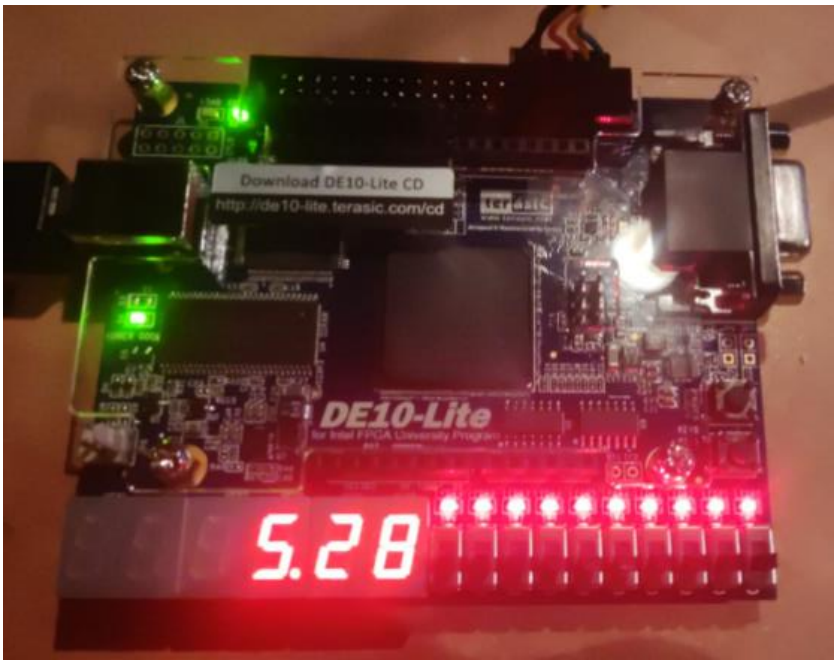
cgol_64 × 64_h6

```
+-----------------------------
|
| PERFORMANCE:
| Measured 264,000,163 elapse c
| Total 5.280 seconds at 50 MHz
| 132.00 Cycles per Generation
| 2.06 Cycles per row
| 0.03 Cycles per grid element
|
+-----------------------------


CSV format report line written t
Updated performance history at t
echo  name , itr , cyc , cpg , c
Checking Correct result of patte
Checker Detected wrap mode (toru
Checker Repeated grid detected a

Checker: done 2000000 Generation


GREAT, Final grid match expected

Saved grid to t0/expected_grid.p
```

```
PASS! cgol_64x64_h6 after 2000000 generation match expected.
264,000,163 cycles for 2000000 generations, Total time 5.280 seconds at 50 MHz.
132 Cycles per Generation, 2.1 cycles per row, 0.03 cycles per element.
```

Generated Image | Expected Image



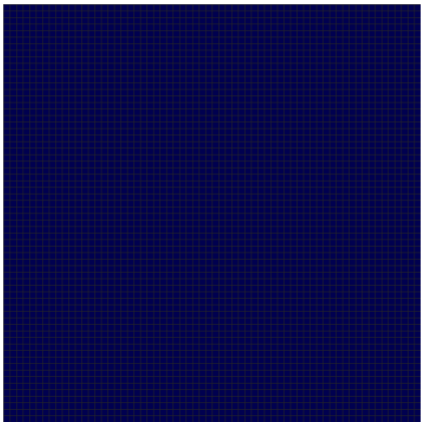


cgol_64 × 64_h7

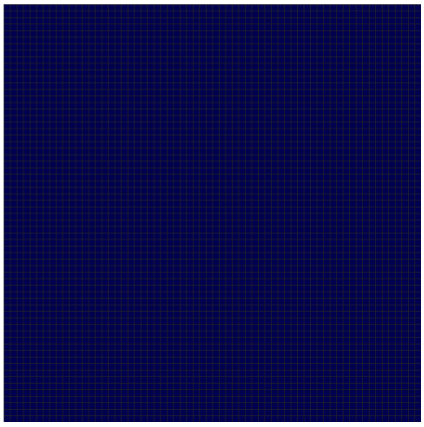
```
 +-----------------------------
 |
 | PERFORMANCE:
 | Measured 264,000,163 elapse c
 | Total 5.280 seconds at 50 MHz
 | 132.00 Cycles per Generation
 | 2.06 Cycles per row
 | 0.03 Cycles per grid element
 |
 +-----------------------------


CSV format report line written to
Updated performance history at t
echo  name , itr , cyc , cpg , cp
Checking Correct result of patter
Checker Detected wrap mode (torus
Checker Repeated grid detected at

Checker: done 2000000 Generations


GREAT, Final grid match expected

Saved grid to t0/expected_grid.pr
```

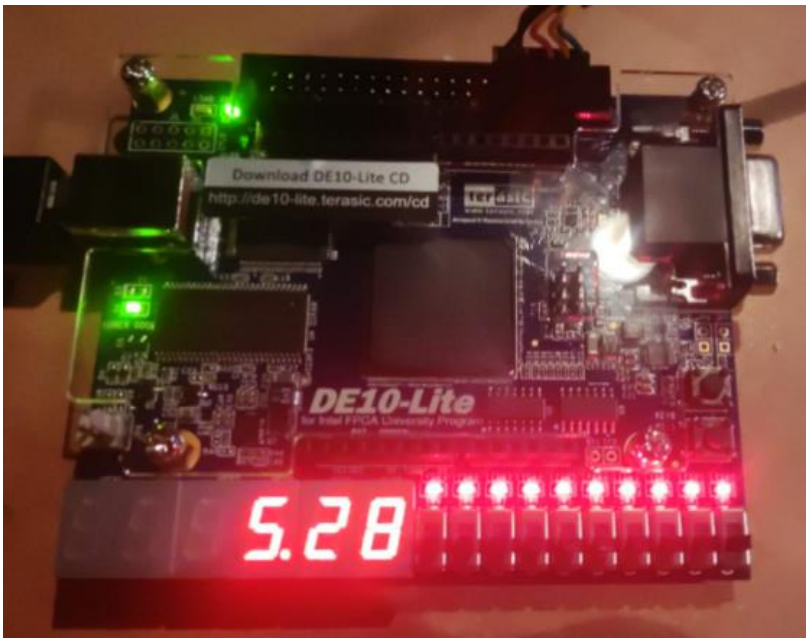
```
PASS! cgol_64x64_h7 after 2000000 generation match expected.
264,000,163 cycles for 2000000 generations, Total time 5.280 seconds at 50 MHz.
132 Cycles per Generation, 2.1 cycles per row, 0.03 cycles per element.
```

Generated Image



Expected Image



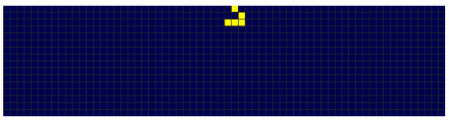


cgol_16 × 64_h8

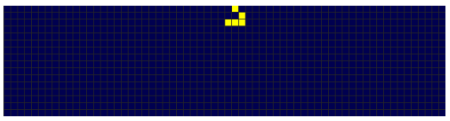
```
+-----------------------------
|
| PERFORMANCE:
| Measured 72,000,163 elapse cy
| Total 1.440 seconds at 50 MHz
| 36.00 Cycles per Generation
| 2.25 Cycles per row
| 0.04 Cycles per grid element
|
+-----------------------------


CSV format report line written t
Updated performance history at t
```

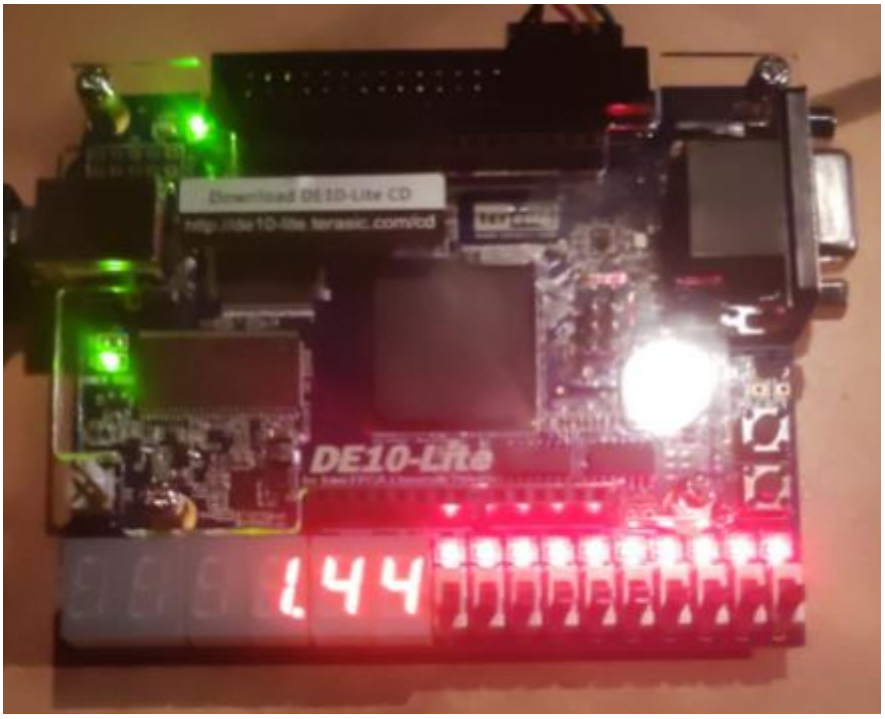
```
PASS! cgol_16x64_h8 after 2000000 generation match expected.
72,000,163 cycles for 2000000 generations, Total time 1.440 seconds at 50 MHz.
36 Cycles per Generation, 2.3 cycles per row, 0.04 cycles per element.
```

Generated Image

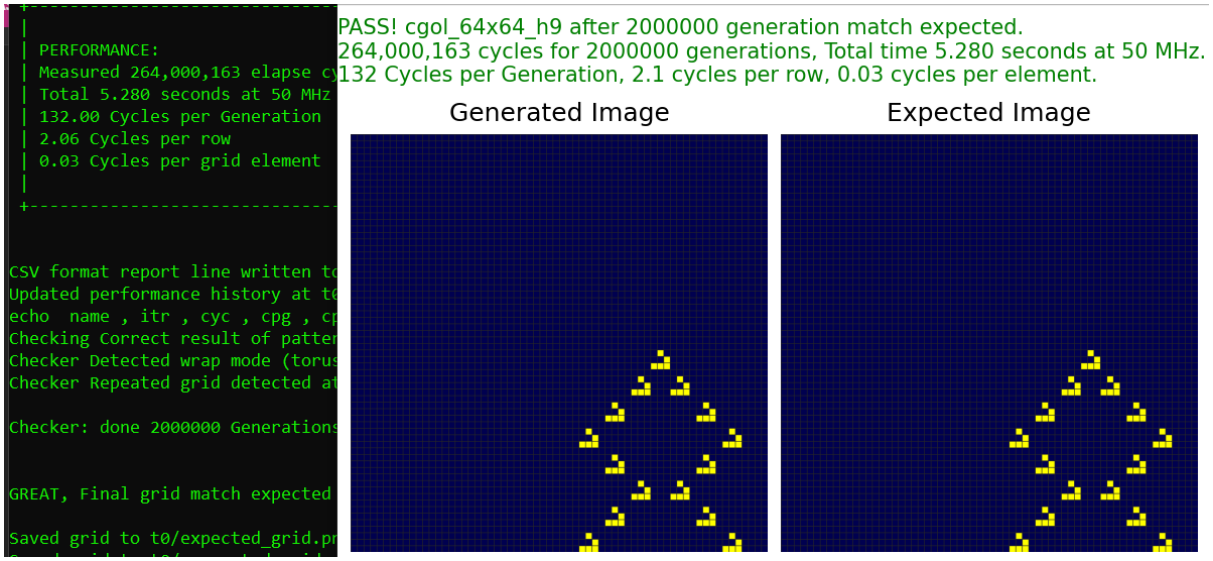


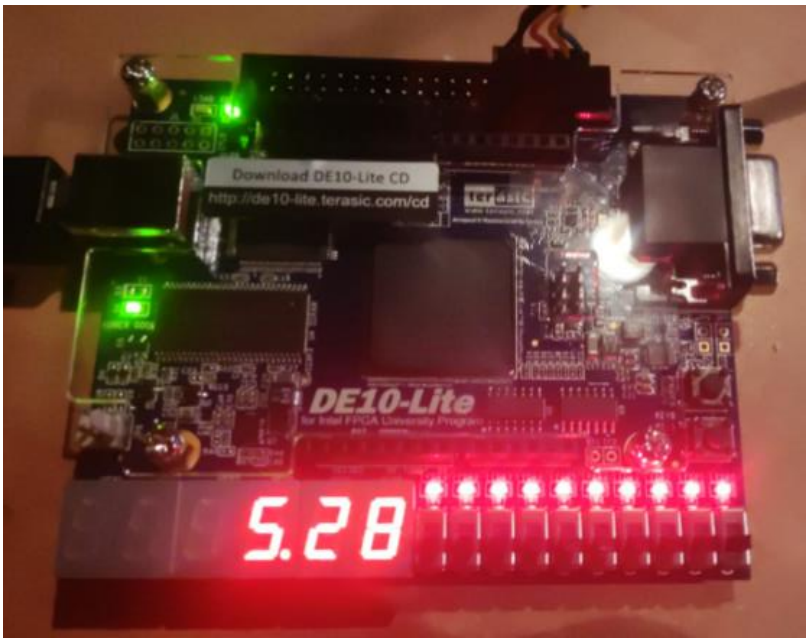
Expected Image





cgol_64 × 64_h9


PERFORMANCE:
Measured 264,000,163 elapse c
Total 5.280 seconds at 50 MHz
132.00 Cycles per Generation
2.06 Cycles per row
0.03 Cycles per grid element
CSV format report line written t
Updated performance history at t
echo name , itr , cyc , cpg , c
Checking Correct result of patte
Checker Detected wrap mode (torus
Checker Repeated grid detected a
Checker: done 2000000 Generations
GREAT, Final grid match expected
Saved grid to t0/expected_grid.p
PASS! cgol_64x64_h9 after 2000000 generation match expected.
264,000,163 cycles for 2000000 generations, Total time 5.280 seconds at 50 MHz.
132 Cycles per Generation, 2.1 cycles per row, 0.03 cycles per element.
Generated Image
Expected Image


Download DE10-Lite CD
DE10-Lite
5.28

Final FPGA Execution Results – CGOL Torus XLR Benchmarks Across Grid Patterns

Pattern Name	FPGA Time [s]	Total Cycles	Cycles/Gen	Cycles/Row	Cycles/Element
cgol_64 × 64_h1	5.280	264,000,163	132.0	2.06	0.03
cgol_32 × 32_h2	2.720	136,000,163	68.00	2.13	0.07
cgol_64 × 64_h3	5.280	264,000,163	132.0	2.06	0.03
cgol_48 × 48_h4	4.000	200,000,163	100.0	2.08	0.04
cgol_64 × 16_h5	5.280	264,000,163	132.0	2.06	0.13
cgol_64 × 64_h6	5.280	264,000,163	132.0	2.06	0.03
cgol_64 × 64_h7	5.280	264,000,163	132.0	2.06	0.03
cgol_16 × 64_h8	1.440	72,000,163	36.00	2.25	0.04
cgol_64 × 64_h9	5.280	264,000,163	132.0	2.06	0.03

Table 1: Execution time and performance breakdown of CGOL Torus XLR design across various input patterns. All tests verified at 50 MHz over 2,000,000 iterations.

Summary

This comprehensive documentation of our work around the CGOL TORUS wasn't just done for a grade — it was a means to bring out everything we had. To explore and push the design to its limits through endless tryouts and deep research on how synthesizers behave, and how they can be manipulated through explicitness, precise engineering, and countless hours of testing — **not only to validate correctness, but to build deep intuition**.
We wanted to *feel* the hardware behind every line of code. To see how each gate responds, how timing evolves, and how true control can only come through relentless iteration.
From evaluating design tricks and trade-offs, to tackling the most challenging aspects of chip design we could inspect — this project became our playground and our proving ground.

While the design demonstrates high performance with additional optimization potential explored during development, the current implementation achieves 2.06 cycles per row through systematic optimization of the pipelined FSM, timing closure to ~80 MHz, and architectural refinements