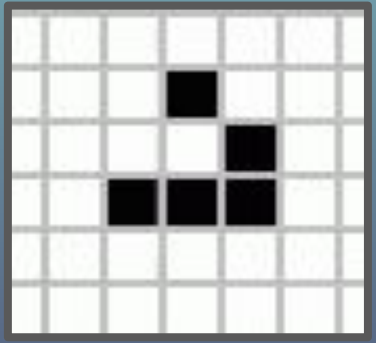
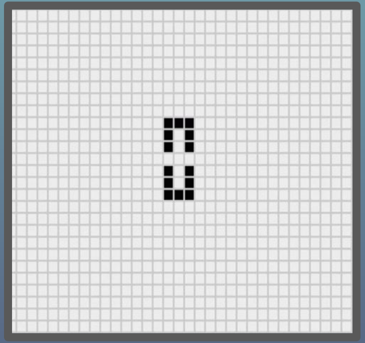
ECE3300 Final
Conway's Game of Life

Group G
Kevin Foyet,
Richie Raymond Wong,
Gerin Fajardo
Tyler Marts

Introduction



Conway's Game of Life, conceptualized in 1970 by mathematician John Horton Conway, is an intriguing zero-player game involving a cellular automaton on a square 2D grid. Each cell within this grid exists in one of two states: alive or dead. The fascinating aspect lies in observing the emergent patterns from the cells' evolution, governed by four fundamental rules.



Rules

1. Any living cell with fewer than two neighbors dies.
2. Any living cell with more than three living neighbors dies.
3. Any living cell with two or three living neighbors may live unchanged.
4. Any deceased cell with exactly three living neighbors can come to life.

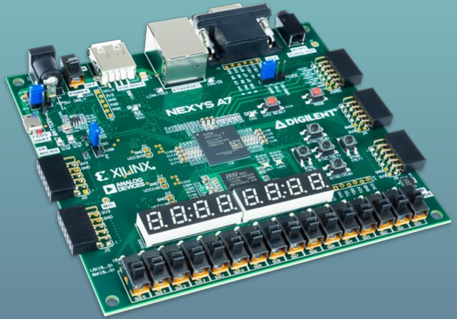
Objective

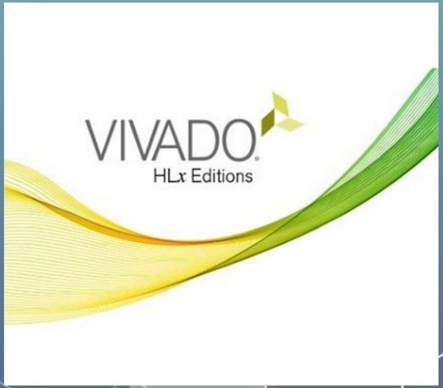
Develop and implement an interactive version of Conway's Game of Life on the Nexys-A7 FPGA board, effectively utilizing its hardware features for an enhanced user experience. This will involve programming eleven out of the sixteen available switches for multifaceted control functions, including an 8-bit speed control, overflow control, enable, and reset capabilities. All onboard buttons will be configured to manipulate the cursor's position and toggle the state of individual cells in the game. Additionally, seven out of the eight seven-segment display digits will be dedicated to showing the game's speed setting and the current generation number. For the visual representation of the game, a 16x16 grid will be displayed using a VGA interface on an external monitor, providing a clear and engaging visual output of the game's progress.

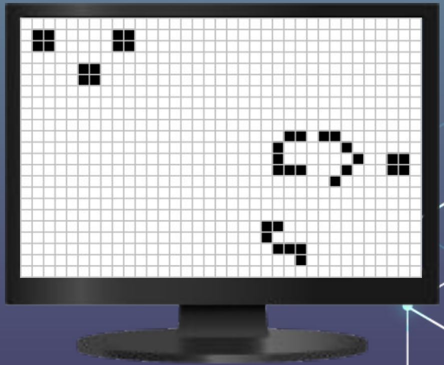
Design and Description

Devices and Tools Used

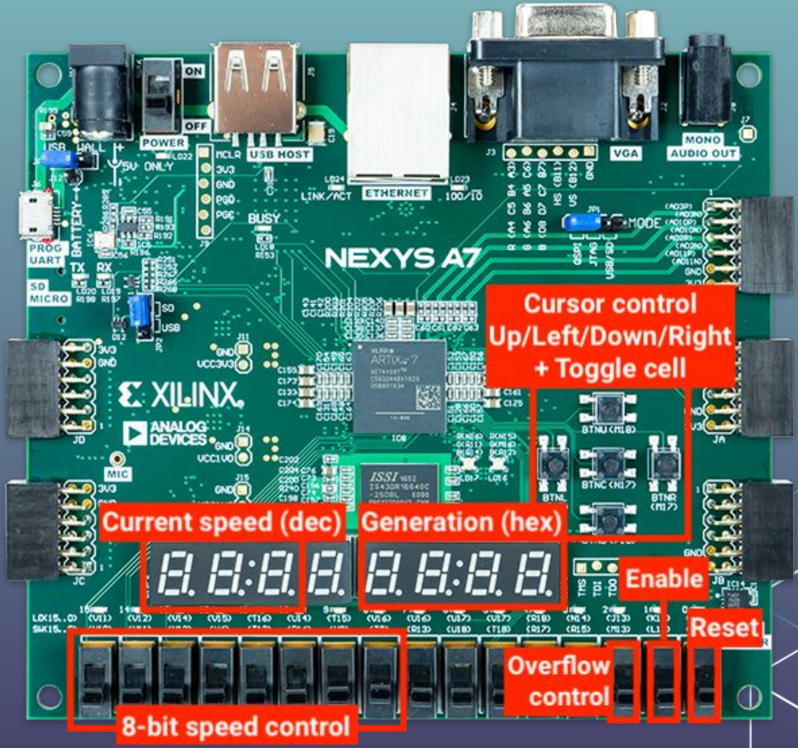
- Nexsys A7 Artix-7 FPGA Board
  - Development & Prototyping
- Xilinx Vivado 2018.1
  - FPGA Design Environment
- VGA-Compatible Monitor
  - Connect to & Display Game



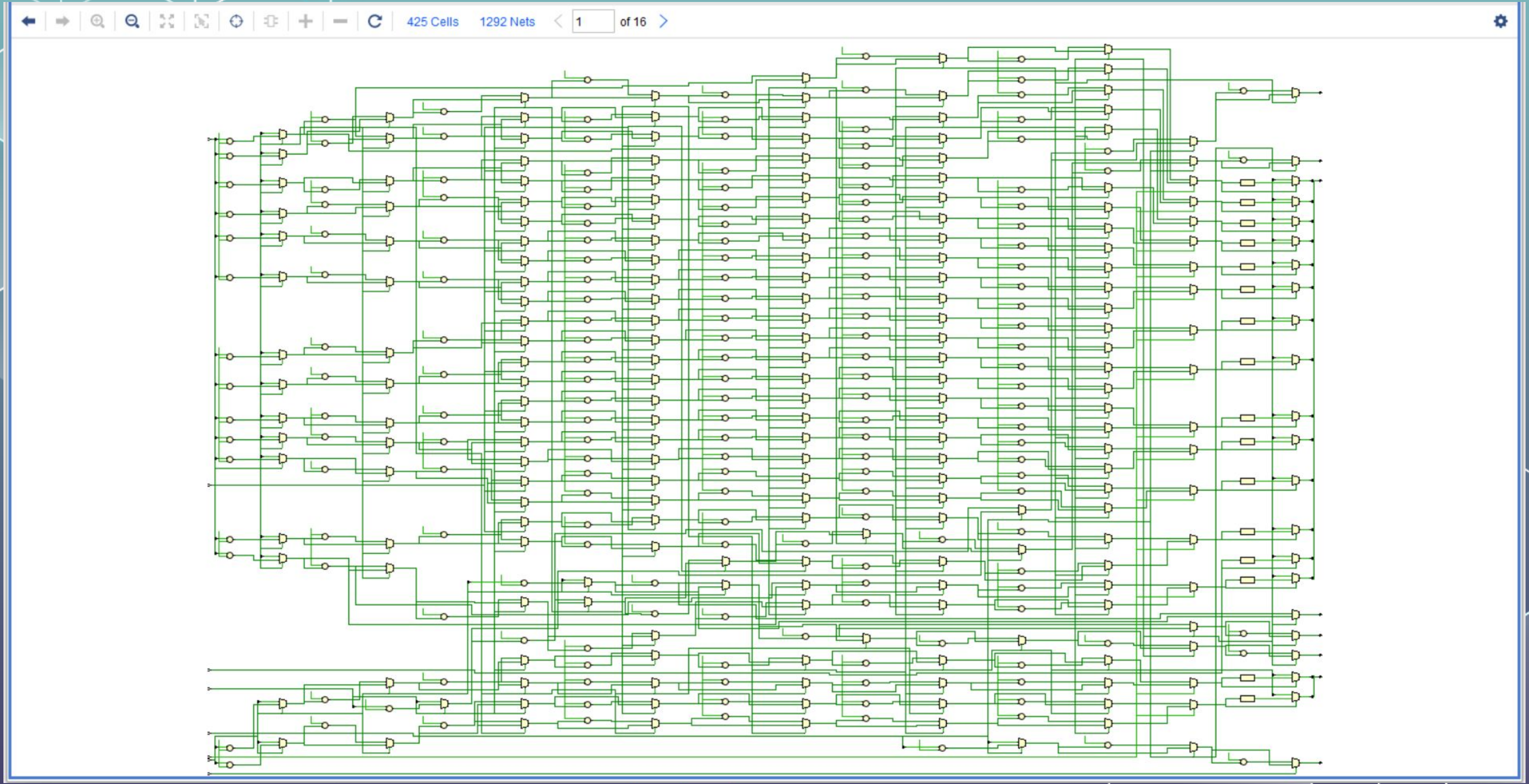




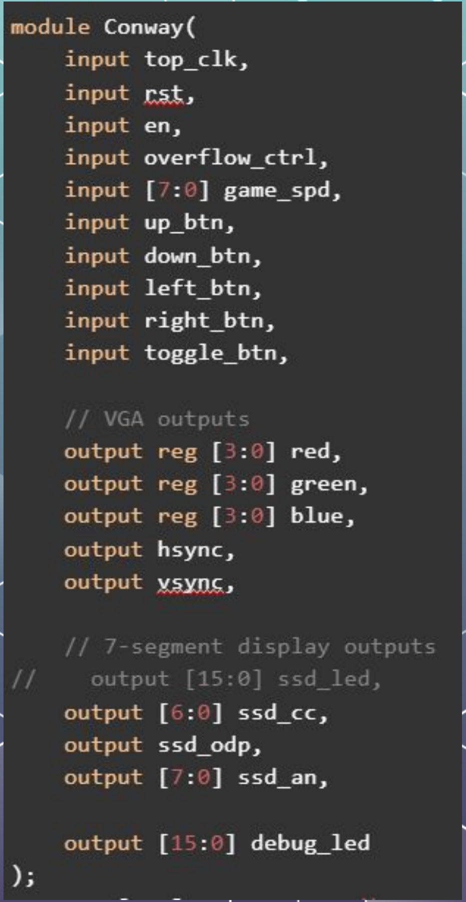
Board I/O Mapping



RTL Schematic



Top Module



```
module Conway(
    input top_clk,
    input rst,
    input en,
    input overflow_ctrl,
    input [7:0] game_spd,
    input up_btn,
    input down_btn,
    input left_btn,
    input right_btn,
    input toggle_btn,

    // VGA outputs
    output reg [3:0] red,
    output reg [3:0] green,
    output reg [3:0] blue,
    output hsync,
    output vsync,

    // 7-segment display outputs
//    output [15:0] ssd_led,
    output [6:0] ssd_cc,
    output ssd_odp,
    output [7:0] ssd_an,

    output [15:0] debug_led
);
```

Module Functionality: The Conway module is designed to simulate and display a cellular automaton, likely Conway's Game of Life, on a VGA-compatible display using an FPGA. It features controls for game speed, grid updates, and cursor movement.

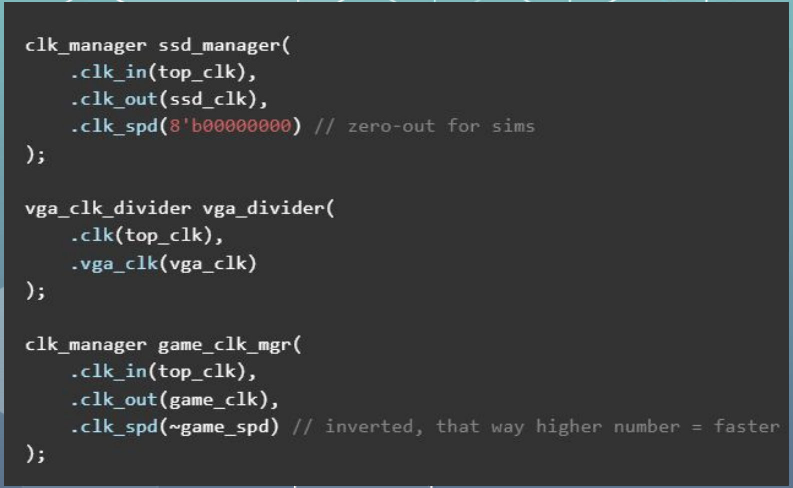
Inputs and Outputs:

- Inputs: Clock signal (top_clk), reset (rst), enable (en), overflow control, game speed, and buttons for cursor movement and toggling cell states.
- Outputs: VGA display signals (RGB, hsync, vsync), 7-segment display outputs for showing speed and generation count, and a debug LED array.

Top Module

Key Components:

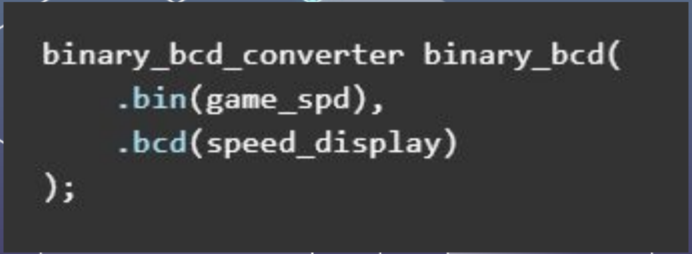
- Clock Managers: Separate clock signals generated for SSD, VGA, and game logic.
- VGA Generator: Handles the synchronization and timing for the VGA display.
- SSD Driver: Manages the 7-segment display output.
- Binary to BCD Converter: Converts binary game speed to BCD for display.



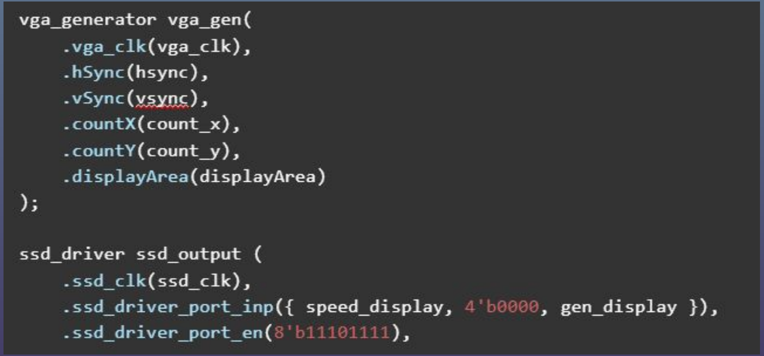
```
clk_manager ssd_manager(
    .clk_in(top_clk),
    .clk_out(ssd_clk),
    .clk_spd(8'b00000000) // zero-out for sims
);

vga_clk_divider vga_divider(
    .clk(top_clk),
    .vga_clk(vga_clk)
);

clk_manager game_clk_mgr(
    .clk_in(top_clk),
    .clk_out(game_clk),
    .clk_spd(~game_spd) // inverted, that way higher number = faster
);
```



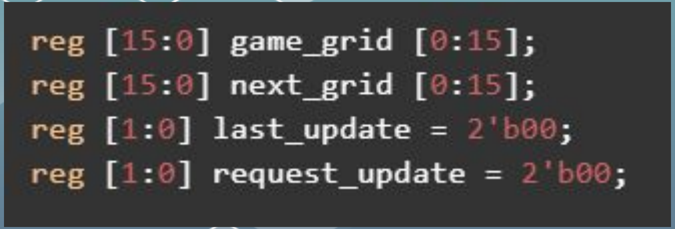
```
binary_bcd_converter binary_bcd(
    .bin(game_spd),
    .bcd(speed_display)
);
```



```
vga_generator vga_gen(
    .vga_clk(vga_clk),
    .hSync(hsync),
    .vSync(vsync),
    .countX(count_x),
    .countY(count_y),
    .displayArea(displayArea)
);

ssd_driver ssd_output (
    .ssd_clk(ssd_clk),
    .ssd_driver_port_inp({ speed_display, 4'b0000, gen_display }),
    .ssd_driver_port_en(8'b11101111),
```

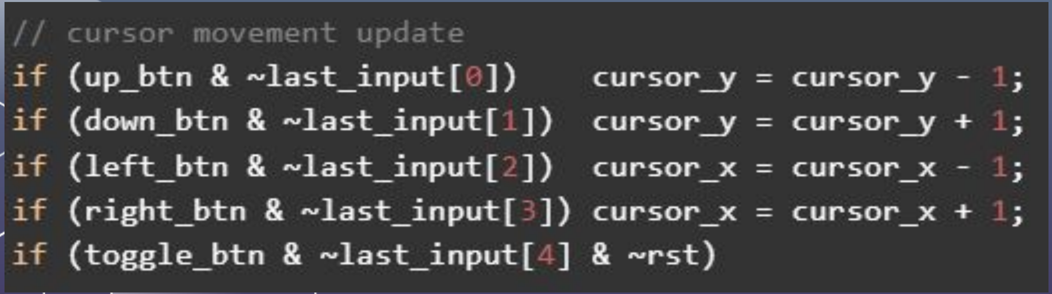
Top Module



```
reg [15:0] game_grid [0:15];
reg [15:0] next_grid [0:15];
reg [1:0] last_update = 2'b00;
reg [1:0] request_update = 2'b00;
```

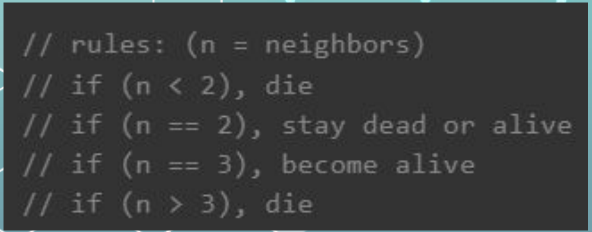
Game Logic:

- Grid Management: Uses a two-dimensional grid to represent the game state, updating it based on neighbor cell states.
- Cellular Automaton Rules: Implements rules for cell survival, death, and birth based on the number of neighboring cells.
- Cursor Control: Allows movement of a cursor on the grid using input buttons and toggles cell states.

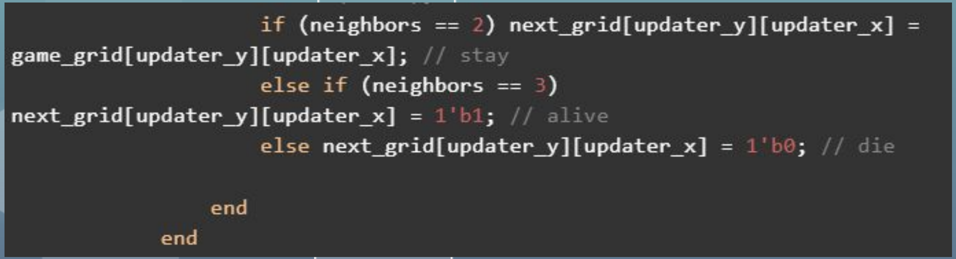


```
// cursor movement update
if (up_btn & ~last_input[0])    cursor_y = cursor_y - 1;
if (down_btn & ~last_input[1])  cursor_y = cursor_y + 1;
if (left_btn & ~last_input[2])  cursor_x = cursor_x - 1;
if (right_btn & ~last_input[3]) cursor_x = cursor_x + 1;
if (toggle_btn & ~last_input[4] & ~rst)
```

Top Module

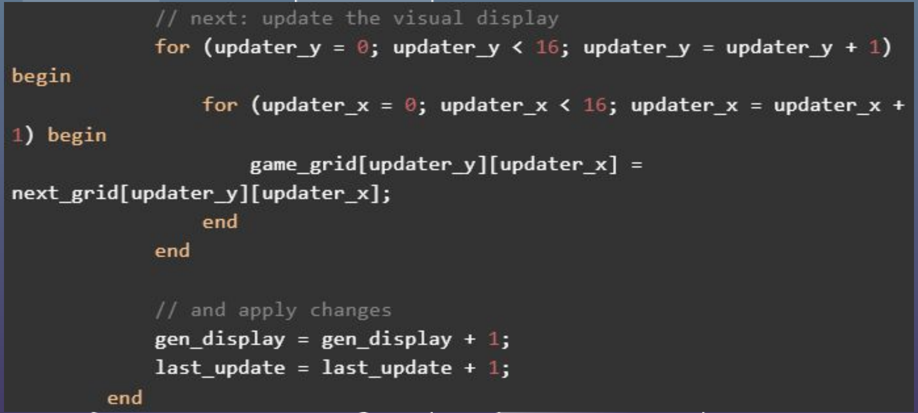


```
// rules: (n = neighbors)
// if (n < 2), die
// if (n == 2), stay dead or alive
// if (n == 3), become alive
// if (n > 3), die
```



```
                    if (neighbors == 2) next_grid[updater_y][updater_x] =
game_grid[updater_y][updater_x]; // stay
                    else if (neighbors == 3)
next_grid[updater_y][updater_x] = 1'b1; // alive
                    else next_grid[updater_y][updater_x] = 1'b0; // die

                end
            end
```



```
            // next: update the visual display
            for (updater_y = 0; updater_y < 16; updater_y = updater_y + 1)
begin
                for (updater_x = 0; updater_x < 16; updater_x = updater_x +
1) begin
                    game_grid[updater_y][updater_x] =
next_grid[updater_y][updater_x];
                end
            end

            // and apply changes
            gen_display = gen_display + 1;
            last_update = last_update + 1;
        end
```

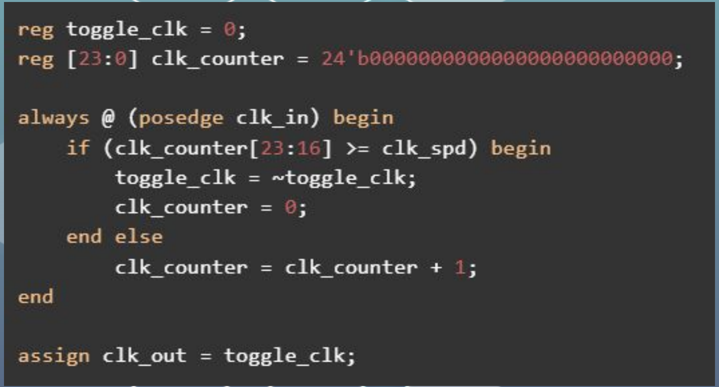
Display Logic:

- VGA Output: Determines the color of each pixel based on the game grid state and cursor position.
- Edge Highlighting: Highlights the edges of cells for better visibility.
- Cursor Highlighting: Differentiates the cursor location with unique color coding.

Update Mechanism:

- Generation Update: Increments the game generation and updates the grid state at each game clock pulse, if enabled.
- Debugging: Uses an LED array for debugging and visual feedback of the generation count.

Clock Manager Module



```
reg toggle_clk = 0;
reg [23:0] clk_counter = 24'b000000000000000000000000;

always @ (posedge clk_in) begin
    if (clk_counter[23:16] >= clk_spd) begin
        toggle_clk = ~toggle_clk;
        clk_counter = 0;
    end else
        clk_counter = clk_counter + 1;
end

assign clk_out = toggle_clk;
```

Module Purpose: The clk_manager module is designed to manage and generate a clock signal with a variable frequency.

Inputs and Output:

- Input:
  - clk_in: The incoming base clock signal.
  - clk_spd: An 8-bit value that determines the speed of the output clock.
- Output:
  - clk_out: The managed output clock signal.

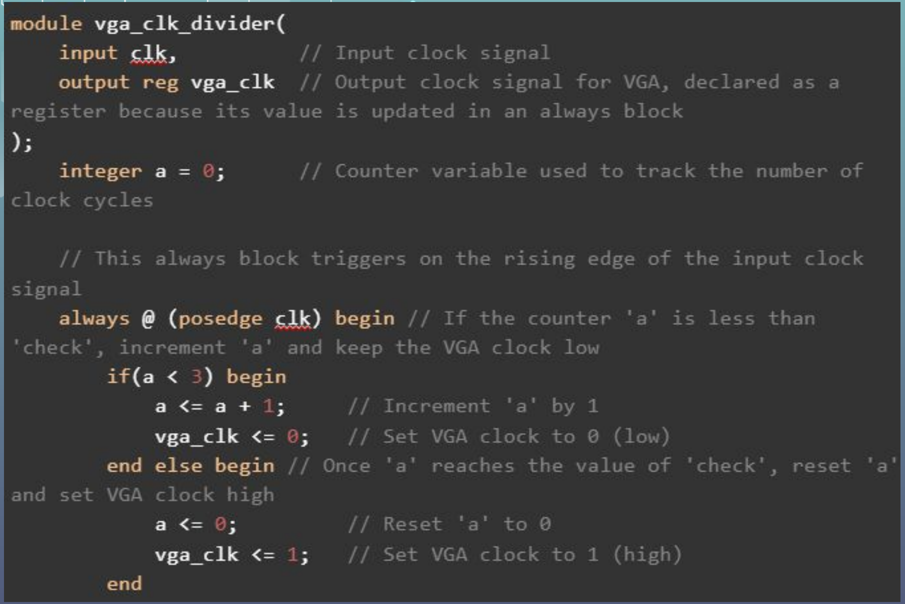
Key Components:

- toggle_clk: A binary register used to toggle the output clock signal.
- clk_counter: A 24-bit counter that tracks the number of clk_in pulses.

Clock Generation Logic:

- The module operates on the rising edge (posedge) of the clk_in signal.
- The clk_counter increments with each pulse of clk_in.
- When the most significant 8 bits of clk_counter equal or exceed the value of clk_spd, the toggle_clk signal is flipped, and clk_counter is reset to zero. This creates the output clock pulse.
- If the clk_counter is less than clk_spd, it simply increments without affecting the toggle_clk.

VGA_CLK_Divider Module



```
module vga_clk_divider(
    input clk,          // Input clock signal
    output reg vga_clk  // Output clock signal for VGA, declared as a
register because its value is updated in an always block
);
    integer a = 0;      // Counter variable used to track the number of
clock cycles

    // This always block triggers on the rising edge of the input clock
signal
    always @ (posedge clk) begin // If the counter 'a' is less than
'check', increment 'a' and keep the VGA clock low
        if(a < 3) begin
            a <= a + 1;      // Increment 'a' by 1
            vga_clk <= 0;    // Set VGA clock to 0 (low)
        end else begin // Once 'a' reaches the value of 'check', reset 'a'
and set VGA clock high
            a <= 0;          // Reset 'a' to 0
            vga_clk <= 1;    // Set VGA clock to 1 (high)
        end
```

Module Function: The vga_clk_divider module generates a clock signal specifically for driving a VGA display from a base clock signal. It divides the frequency of the input clock to produce an appropriate clock signal for VGA timing requirements.

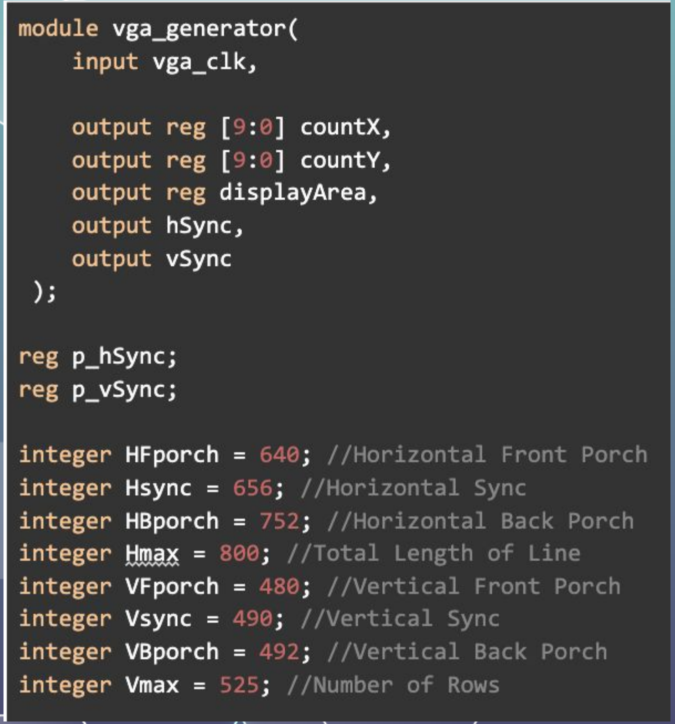
Inputs and Output:

- **Input:**
  - clk: The base clock signal input.
- **Output:**
  - vga_clk: The generated clock signal for the VGA display.

Clock Division Logic:

- The module is triggered on the rising edge (posedge) of the clk signal.
- If the counter a is less than 3, it increments a and keeps the VGA clock low (vga_clk <= 0).
- Once a reaches 3, it resets a to 0 and sets the VGA clock high (vga_clk <= 1).

VGA_Generator Module



```
module vga_generator(
    input vga_clk,

    output reg [9:0] countX,
    output reg [9:0] countY,
    output reg displayArea,
    output hSync,
    output vSync
 );

reg p_hSync;
reg p_vSync;

integer HFporch = 640; //Horizontal Front Porch
integer Hsync = 656; //Horizontal Sync
integer HBporch = 752; //Horizontal Back Porch
integer Hmax = 800; //Total Length of Line
integer VFporch = 480; //Vertical Front Porch
integer Vsync = 490; //Vertical Sync
integer VBporch = 492; //Vertical Back Porch
integer Vmax = 525; //Number of Rows
```

Module Purpose: The VGA_Generator module is designed to create VGA signal timing for display.

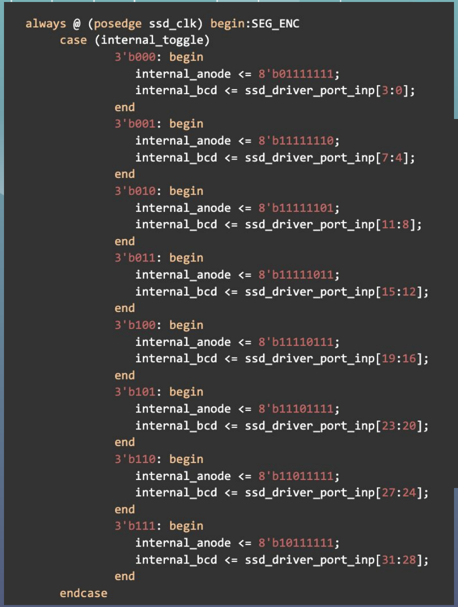
Inputs and Output:

- Input:
    - Vga_clk: Clock Signal
- Output:
    - countX: Horizontal Counter
    - countY: Vertical Counter
    - displayArea: Active Display Indicator
    - hSync: Horizontal Sync Signal
    - vSync: Vertical Sync Signal

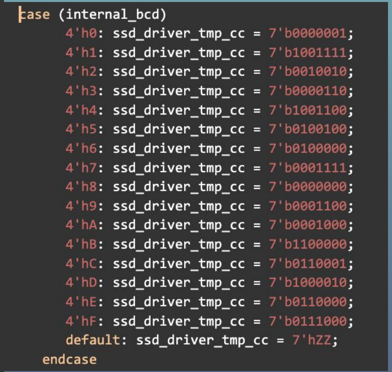
VGA_Generator Logic:

- Counter logic tracks horizontal (countX) and vertical (countY) positions
- `countX` resets at Hmax; `countY` resets at Vmax after countX reaches Hmax
- Determines displayArea based on position comparisons
- Generates sync signals (hSync and vSync) from timing parameters and counters
- Synchronized to the positive edge of vga_clk

SSD_Driver Module



```
always @ (posedge ssd_clk) begin:SEG_ENC
     case (internal_toggle)
          3'b000: begin
             internal_anode <= 8'b01111111;
             internal_bcd <= ssd_driver_port_inp[3:0];
          end
          3'b001: begin
             internal_anode <= 8'b11111110;
             internal_bcd <= ssd_driver_port_inp[7:4];
          end
          3'b010: begin
             internal_anode <= 8'b11111101;
             internal_bcd <= ssd_driver_port_inp[11:8];
          end
          3'b011: begin
             internal_anode <= 8'b11111011;
             internal_bcd <= ssd_driver_port_inp[15:12];
          end
          3'b100: begin
             internal_anode <= 8'b11110111;
             internal_bcd <= ssd_driver_port_inp[19:16];
          end
          3'b101: begin
             internal_anode <= 8'b11101111;
             internal_bcd <= ssd_driver_port_inp[23:20];
          end
          3'b110: begin
             internal_anode <= 8'b11011111;
             internal_bcd <= ssd_driver_port_inp[27:24];
          end
          3'b111: begin
             internal_anode <= 8'b10111111;
             internal_bcd <= ssd_driver_port_inp[31:28];
          end
     endcase
```



```
case (internal_bcd)
       4'h0: ssd_driver_tmp_cc = 7'b0000001;
       4'h1: ssd_driver_tmp_cc = 7'b1001111;
       4'h2: ssd_driver_tmp_cc = 7'b0010010;
       4'h3: ssd_driver_tmp_cc = 7'b0000110;
       4'h4: ssd_driver_tmp_cc = 7'b1001100;
       4'h5: ssd_driver_tmp_cc = 7'b0100100;
       4'h6: ssd_driver_tmp_cc = 7'b0100000;
       4'h7: ssd_driver_tmp_cc = 7'b0001111;
       4'h8: ssd_driver_tmp_cc = 7'b0000000;
       4'h9: ssd_driver_tmp_cc = 7'b0001100;
       4'hA: ssd_driver_tmp_cc = 7'b0001000;
       4'hB: ssd_driver_tmp_cc = 7'b1100000;
       4'hC: ssd_driver_tmp_cc = 7'b0110001;
       4'hD: ssd_driver_tmp_cc = 7'b1000010;
       4'hE: ssd_driver_tmp_cc = 7'b0110000;
       4'hF: ssd_driver_tmp_cc = 7'b0111000;
       default: ssd_driver_tmp_cc = 7'hZZ;
    endcase
```

Module Purpose: Generates control signals for a 7-segment display

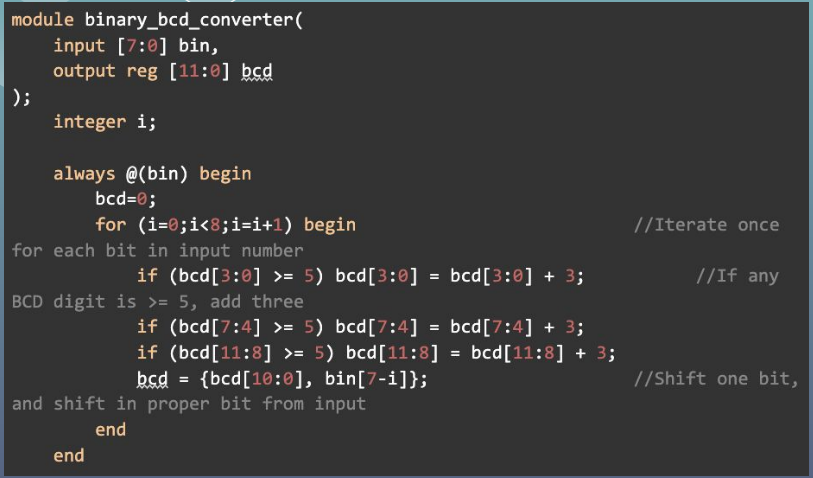
Inputs and Output:

- Input:
  - ssd_clk: Clock signal
  - ssd_driver_port_inpL 32-bit Input Port
  - ssd_driver_port_en: Enable Signals
  - Ssd_driver_port_idp: Input Data Signal
- Output:
  - ssd_driver_port_led: 16-bit LED output
  - ssd_driver_port_cc: Common Cathode Output
  - ssd_driver_port_odp: Output Data Sginal
  - Ssd_driver_port_an: Output for SSD Anodes

SSD_Driver Logic:

- Segment Encoding (`SEG_ENC`)
  - Sequential logic synchronized with ssd_clk
  - Controls internal_anode & internal_bcd based on internal_toggle

Binary_BCD_Converter Module



```verilog
module binary_bcd_converter(
    input [7:0] bin,
    output reg [11:0] bcd
);
    integer i;

    always @(bin) begin
        bcd=0;
        for (i=0;i<8;i=i+1) begin                          //Iterate once
for each bit in input number
            if (bcd[3:0] >= 5) bcd[3:0] = bcd[3:0] + 3;          //If any
BCD digit is >= 5, add three
            if (bcd[7:4] >= 5) bcd[7:4] = bcd[7:4] + 3;
            if (bcd[11:8] >= 5) bcd[11:8] = bcd[11:8] + 3;
            bcd = {bcd[10:0], bin[7-i]};                      //Shift one bit,
and shift in proper bit from input
        end
    end
```

Module Purpose: Converts an 8-bit binary input into a 12-bit BCD (Binary-Coded Decimal) representation

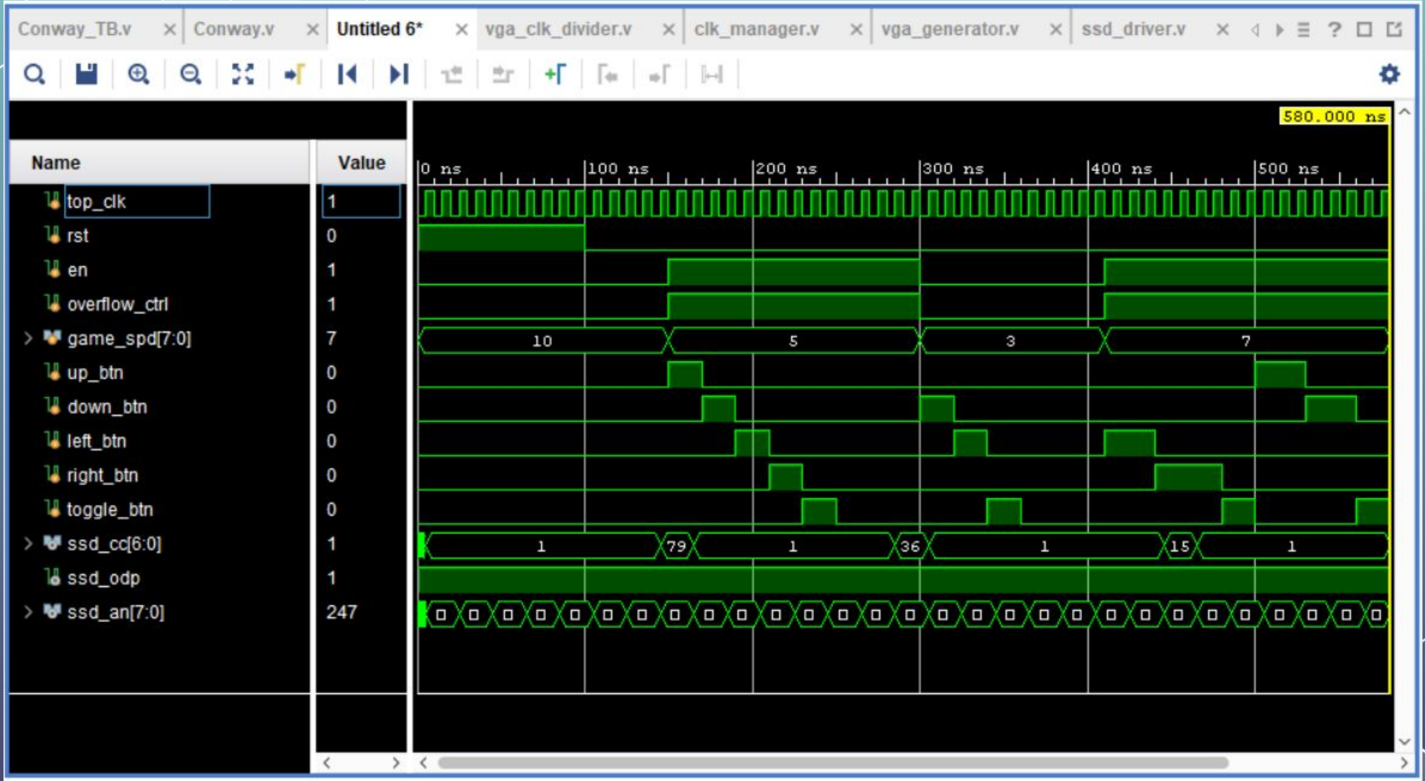
Inputs and Output:

- Input:
  - Bin: 8-bit binary input
- Output:
  - Bcd: 12-bit BCD output

Binary_BCD_Converter Logic:

- Utilizes an `always @(bin)` block to continuously evaluate changes in the bin input
- Resets `bcd` to 0 at the start of each evaluation cycle
- Processes each bit of the `bin` input in a loop
- Checks each group of four bits in `bcd` for values >= 5
- If >= 5, adds 3 to the corresponding BCD digit
- Shifts bits in `bcd` and incorporates the next bit from `bin` to form the BCD output

Simulation Waveform



Name	Value
top_clk	1
rst	0
en	1
overflow_ctrl	1
game_spd[7:0]	7
up_btn	0
down_btn	0
left_btn	0
right_btn	0
toggle_btn	0
ssd_cc[6:0]	1
ssd_odp	1
ssd_an[7:0]	247

Synthesis Report

0.218 W

Total Power

2401

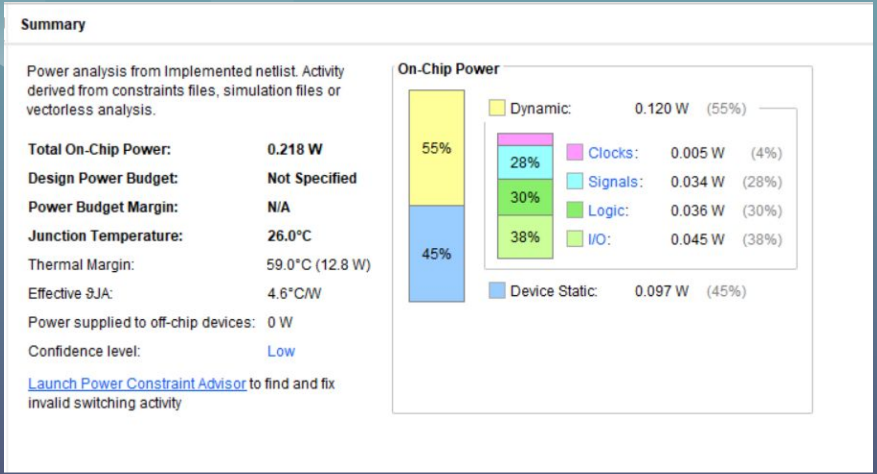
LUTs

422

FFs

Note:
LUTs = Lookup Tables
FFs = Flip-Flops

On-Chip Power Report



Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.218 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26.0°C**
Thermal Margin:	59.0°C (12.8 W)
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

🟨	Dynamic:	0.120 W	(55%)
🟪	Clocks:	0.005 W	(4%)
🟦	Signals:	0.034 W	(28%)
🟩	Logic:	0.036 W	(30%)
🟩	I/O:	0.045 W	(38%)
🟦	Device Static:	0.097 W	(45%)

0.218 W
Total On-Chip Power

0.120 W
Dynamic Power

0.097 W
Device Static Power

0.005 W
Clocks

0.034 W
Signals

0.036 W
Logic

0.045 W
I/O

Video Demonstration

ECE3300.02 - Digital Circuit Design
Dr. Mohamed Aly // Cal Poly Pomona
Dario Fajardo, Kevin Foyet,
Tyler Marts, and Raymond Wong
064 0000

References

“Binary to BCD and BCD to Binary.” *RealDigital*, www.realdigital.org/doc/6dae6583570fd816d1d675b93578203d.

“Conway’s Game of Life.” *Wikipedia*, Wikimedia Foundation, 30 Nov. 2023, en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

“Nexys A7-100T Master Constraints File.” *GitHub*, Digilent, github.com/Digilent/digilent-xdc/blob/master/Nexys-A7-100T-Master.xdc.

Krishnajith S S. “Snake Game on FPGA in Verilog.” *PDF*, slideshare, 2017, www.slideshare.net/sskrishnajith/snake-game-on-fpga-in-verilog.

Larson, Scott. “VGA Controller (VHDL).” *DigiKey*, 17 Mar. 2021, forum.digikey.com/t/vga-controller-vhdl/12794.

slidesgo