Verilog Snake Game
Digital Logic Design Final Project

Matheus Ottmann
Matthew Yam
Group 11
Professor Meghana Jain

Table of Contents

Original Premise and Changes

The initial idea for the project was to create a version of the classic “Snake Game” utilizing the VGA connection found on the BASYS 3 board. A VGA driver would be used to show the snake on a monitor connected to the board, with the push buttons used to control the movement of the snake. The original plan was for it to be a basic snake game, where an apple would be randomly generated somewhere on the screen where the snake did not reside, and the snake would increase in size.

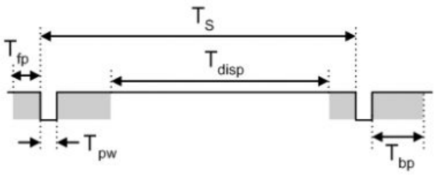
We struggled heavily when creating the VGA driver, causing our initial design to change heavily. We dropped most of the different setting applications and were unable to properly debug our Apple Generation logic, which would involve using linear shift back register to randomly create an x and y location. These locations would then be compared with the head and body location along with the max values for the display time using a twenty-bit comparator, and if the location properly checked all the tests, it would generate a red apple. Due to time constraints, we were unable to properly implement it on our code but were able to finish our VGA driver along with the Snake Movement. However, when the Snake movement was implemented using the VGA, the user-controlled pixel moved erratically, which we deduced to be due to the refresh rates or an issue with the Horizontal Sync. Our final project involved a VGA driver that allowed the user to change the color of the screen based on switches and a semi operational movement simulator that allowed for the user to move a pixel across the screen using the push buttons.

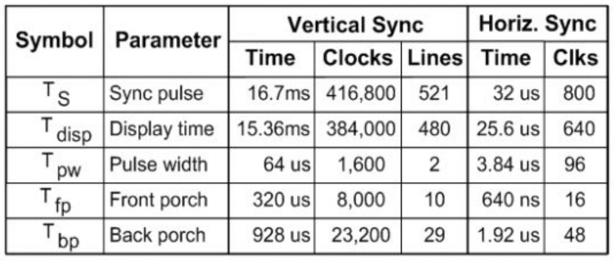
Our initial ambitions with the snake game were unfulfilled, but due to the challenge of the game, our progress with the snake movement and VGA driver should be considered a success.

Module Descriptions

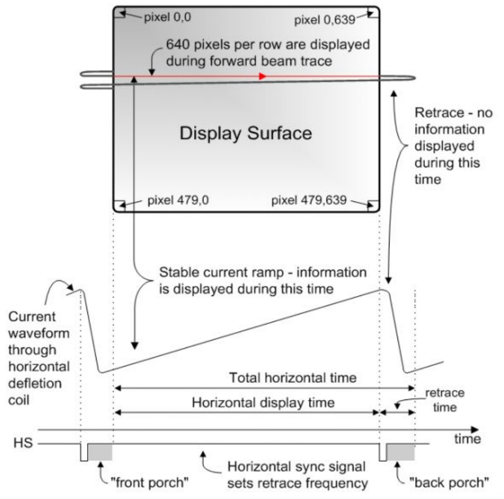
VGA Driver

The VGA driver was built using the detailed reference manuals found on the Digilent Documentation website. According to our final schematic [Schematic 1] of the VGA driver, it consists of a Horizontal and Vertical sync signal using multiple comparators, two counters, two SR latches, and a clock divider.





Symbol	Parameter	Vertical Sync			Horiz. Sync	
		Time	Clocks	Lines	Time	Clks
T_S	Sync pulse	16.7ms	416,800	521	32 us	800
T_{disp}	Display time	15.36ms	384,000	480	25.6 us	640
T_{pw}	Pulse width	64 us	1,600	2	3.84 us	96
T_{fp}	Front porch	320 us	8,000	10	640 ns	16
T_{bp}	Back porch	928 us	23,200	29	1.92 us	48


pixel 0,0
pixel 0,639
640 pixels per row are displayed during forward beam trace
Display Surface
Retrace - no information displayed during this time
pixel 479,0
pixel 479,639
Stable current ramp - information is displayed during this time
Current waveform through horizontal defletion coil
Total horizontal time
Horizontal display time
retrace time
HS
time
"front porch"
Horizontal sync signal sets retrace frequency
"back porch"

Images contain the basic VGA logic along with the required clock cycles for the V and H sync.

Clock divider:

A clock divider is necessary as the clock cycle on the board is a standard 100MHz, while the clock cycle required for a 640 by 480 display is 25 MHz pixel clock for the proper 60hz refresh rate. Schematic 4 shows the Clock divider being properly implemented using one of our Lecture designs for subtractors. We needed to use two flip flops in order to divide the clock from 100 to 25 MHz.

Counters:

The twenty-bit counters, found in schematic 2, were used because the max value for the Vertical sync pulses was 416,800, which when shown in binary contains twenty bits. A twenty-bit comparator can be found inside the counter, as the pulse for the Vertical and Horizontal syncs each have distinct max values that must not be exceeded. The code can be found in Module Code 3 of the appendix.

Comparators:

Comparators are also used for the reset/set detect in for the sync signals. The comparators also must have a set and reset state where the HSync must be set when the Hcounter detects a zero and reset when the comparator detects 1600 clock cycles. For the horizontal counter, the counter will set when the counter detects zero clock cycles and resets when it detects 96 clock cycles. This can be found in Code 3 in the appendix which shows the .v code for the 20 bit counter.

Synchronizer:

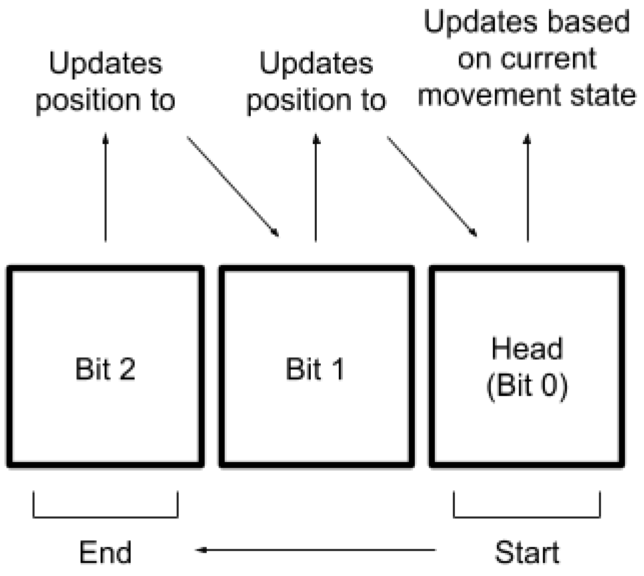
A synchronizer is used for the Set and Reset detect signals, as they don't follow the clock cycle. As shown in schematic 1, an SR latch was used as a synchronizer for the set/reset of the horizontal and vertical sync.

Snake Movement

The movement of the snake was separated into two distinct components: an FSM (snakeMoveState) that determines the current movement direction, and an ALU (snakeMoveLogic) that calculates the snake's position for each clock cycle. The two components, when used in conjunction with the VGA driver, is supposed to draw the snake on a connected display. [Module Code 2]

snakeMoveState:

In a classic game of snake, when a directional button is pressed, the snake will constantly travel in that direction until another button is pressed or the snake collides with a wall. To do this, an FSM was needed that would output a constant binary number based on the state is was currently in. Five states were detailed, Up, Down, Left, Right, and No Move; transitions between the states were done based on button presses. When started, the game would start in the default 'No Move' state. Our implementation of this FSM is slightly inefficient since there are five states, a three-bit output is required but we used a four-bit output. This is fixed by snakeMoveLogic, which converts the four-bit signal into a three-bit one.



snakeMoveLogic:

When a snake moves, the length of its body follows its head but not necessarily the direction it's presently traveling. This means that the head of our snake should be the only part that updates based on the output of snakeMoveState, the rest of the body should update its position based on the part that moved before it. Since the head follows different logic from the rest of the body, we use an ALU to differentiate the two logic evaluators. The opcode of the ALU is driven by an input called Bit Number. This number tells the logic what part of the body it's currently working on updating and it is sourced from a counter [Module Code 3]; at every rising edge of the clock cycle, the counter will increment until it reaches a max. The max is based on another counter that keeps track of the player's current score; that score is incremented whenever the snake consumes an apple. For our testing, we kept the Bit Number a constant, 20-bit, binary zero (Which means only the head was present).

Since each body part is being updated based on a previous state, four D-Flip Flops are utilized, two to store the previous body X and Y positions and two to store the previous X and Y head positions. The D-Flip Flop for the previous head position uses an enable connected to the Bit Number to ensure that its position only updates when the logic is working on Bit Number 0, or the head.

When the logic is updating the head, another ALU is used to determine which direction to move the head. The four-bit number from snakeMoveState is converted into a three-bit number and used as the opcode. When traveling up, the ALU subtracts one from the previous Y head position; it adds one when traveling down. The same occurs for traveling left and right, the logic just updates the previous X head position.

When the logic is updating the body, the previous X and Y values stored in the D-Flip Flop are being passed as the X and Y values for the current bit.

These X and Y values are fed to a comparator. This compares the X and Y values to the hCount and vCount values from the VGA driver. If the values match, that means the VGA driver is currently drawing at those coordinates and all three RGB pins of the VGA port are driven high simultaneously, drawing that part of the body on-screen.

Debugging Process

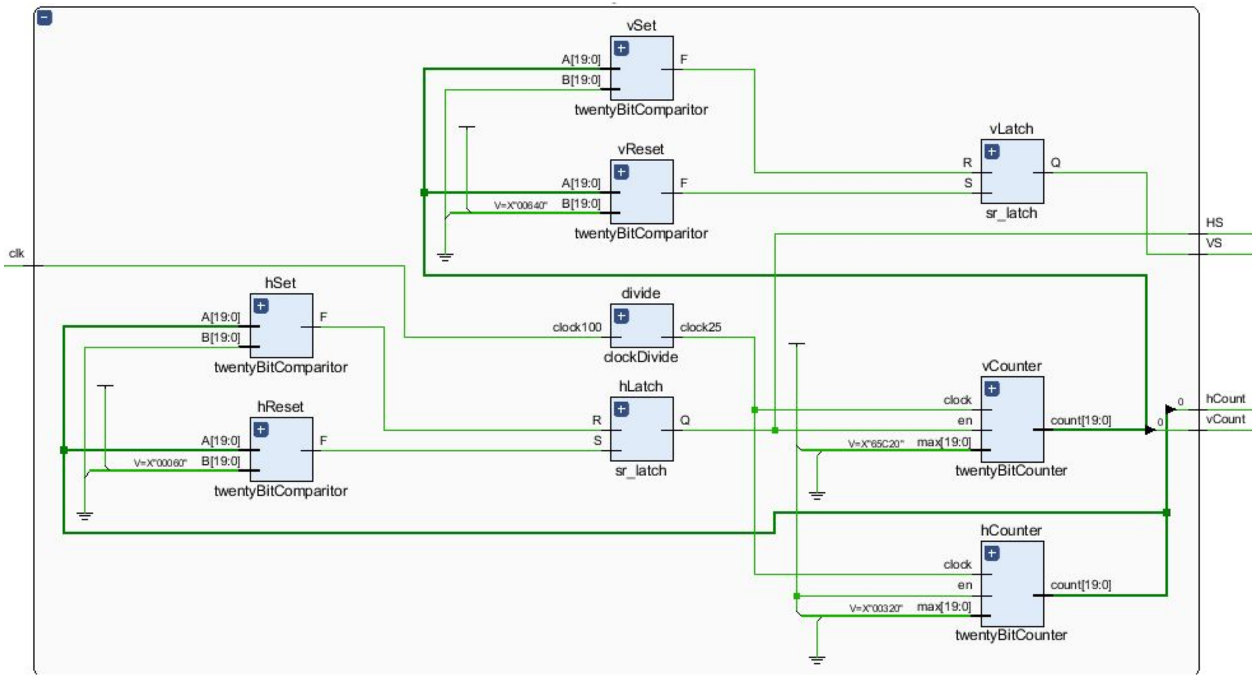
Debugging process for the VGA board involved testing each individual module to see if it did its intended function. For the counter, we first tested the adder using switches that would output the values on the LED’s. Then followed by testing the Comparator, and finally the Counter with all its components using the LEDs found on the board along with the switches. When testing the full VGA board, we assigned switches for the four Red, Green, and Blue bits, where each color was assigned to one switch, and any time one was flipped it would cause the four outputs for the chosen color to go high, making the screen that color. The first code found in the appendix for the VGA driver shows how we implemented these switches.. We also needed to take into account the retrace time for the horizontal and vertical sync of the colors. Based on the diagrams found in the documentation, the retrace time includes the pulse width along with the back porch of the cycle. So we added another comparator that would detect a set signal when the counter would match the length for the Pulse width added with the back porch and reset when the count cycles matched the Pulse width and back porch along with the display time. These set and reset values would then be synchronized using sr latches and then connected using an AND gate that will output the color.

Debugging process for the Snake movement logic involved first creating an FSM diagram for the possible movements of the head, using a chart and Karnaugh Maps. Once we hardcoded the Snake movement, we used the LED’s to test a code where each light was assigned to one snake state, and if the top push button was pressed, the top movement assigned LED would turn on, if the left push button then the left movement assigned LED would turn on and so on.

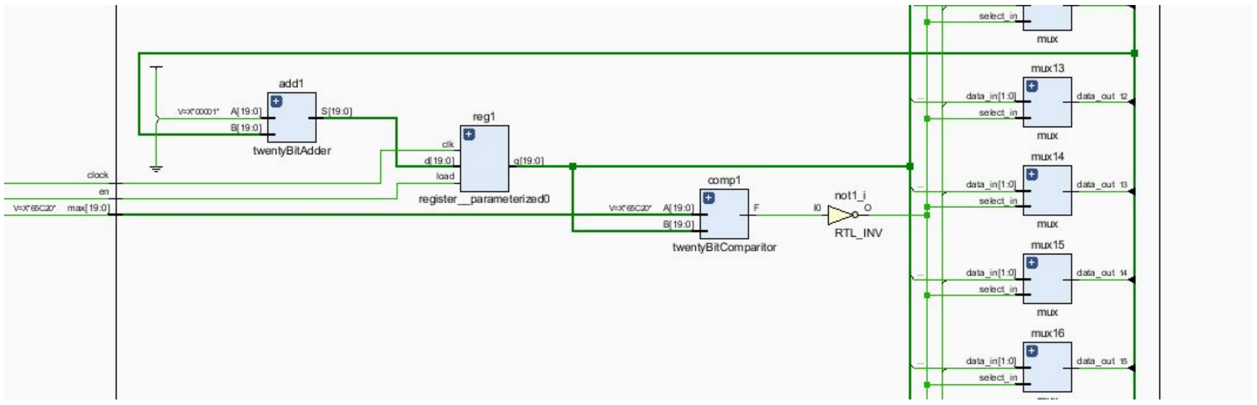
Appendix

Schematics

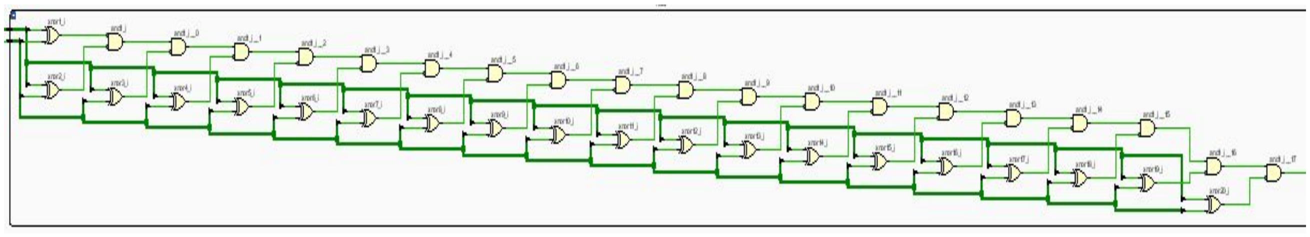
VGA Driver [1]



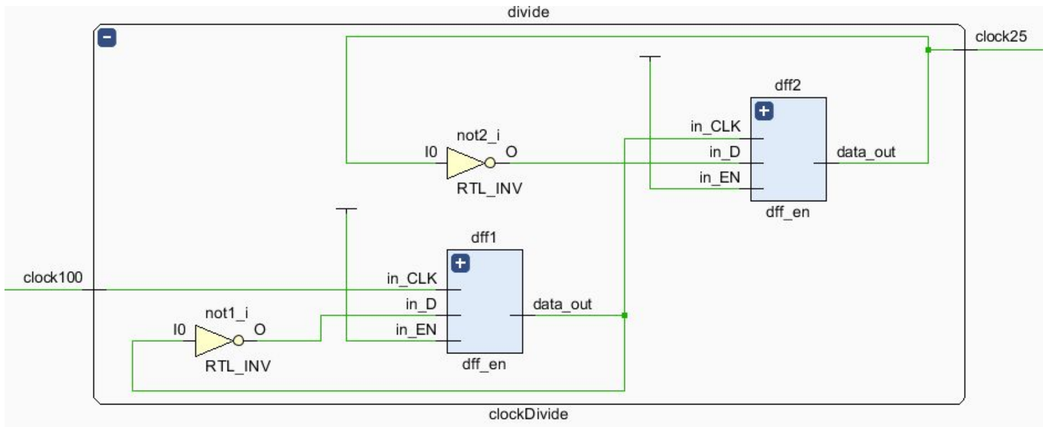
20 Bit Counter [2]



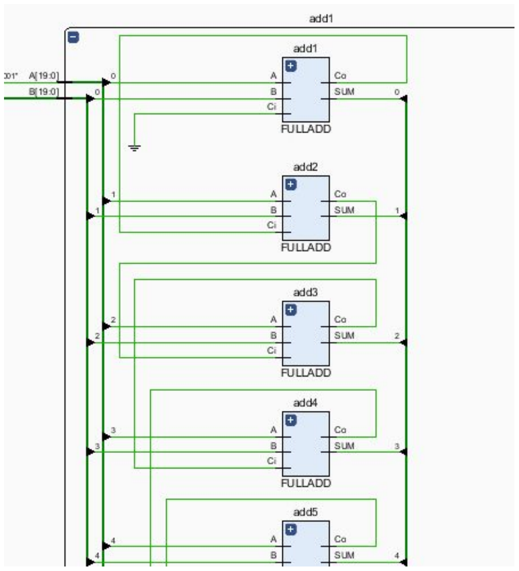
20 Bit Comparator [3]



Clock Divider [4]

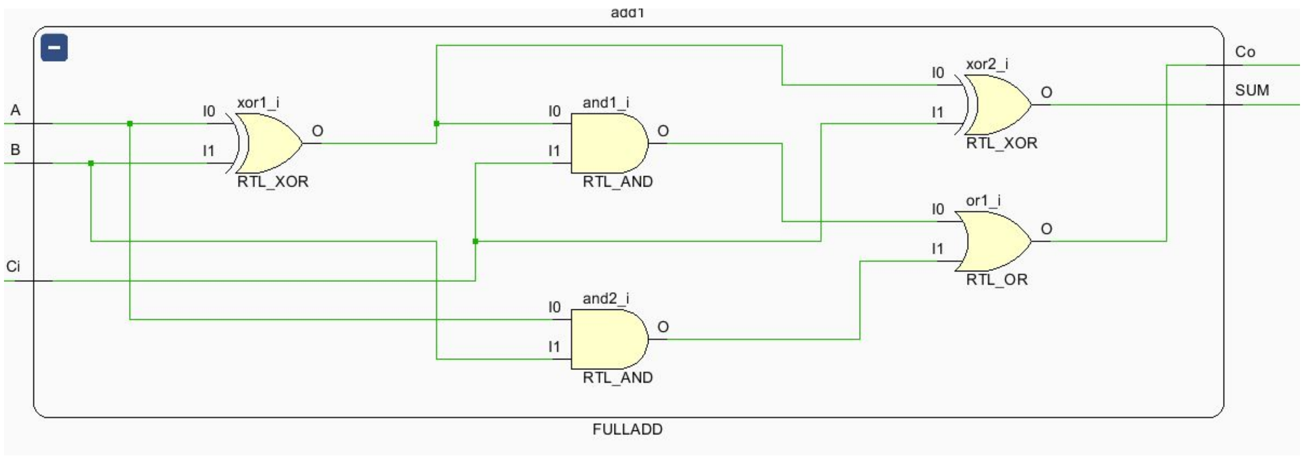


20 Bit Adder [5]

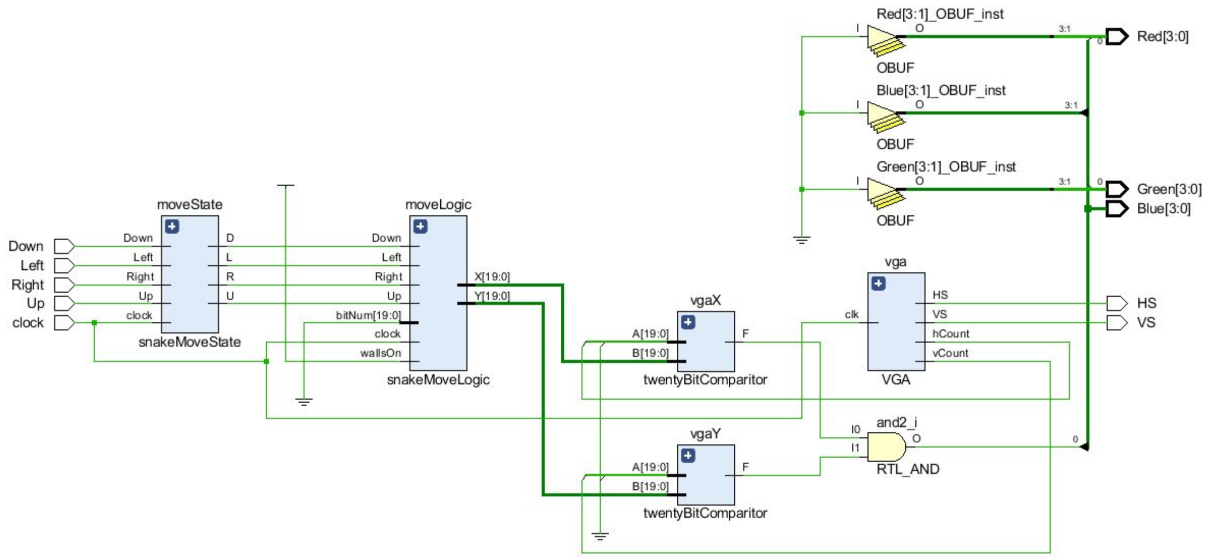


(Image Cut due to size, contains 20 full adders)

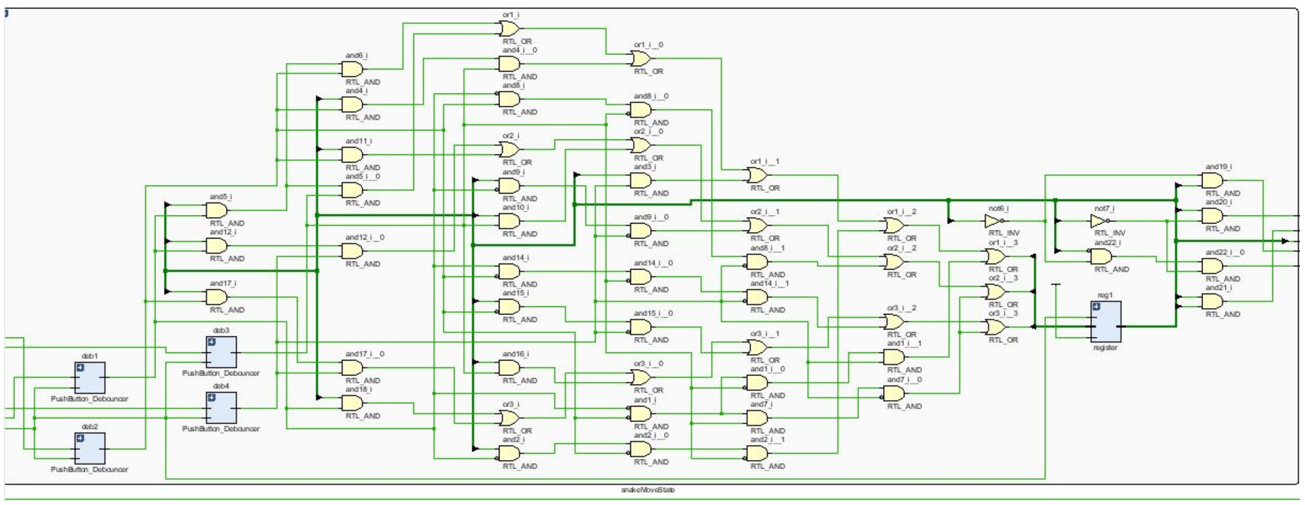
Full Adder [6]



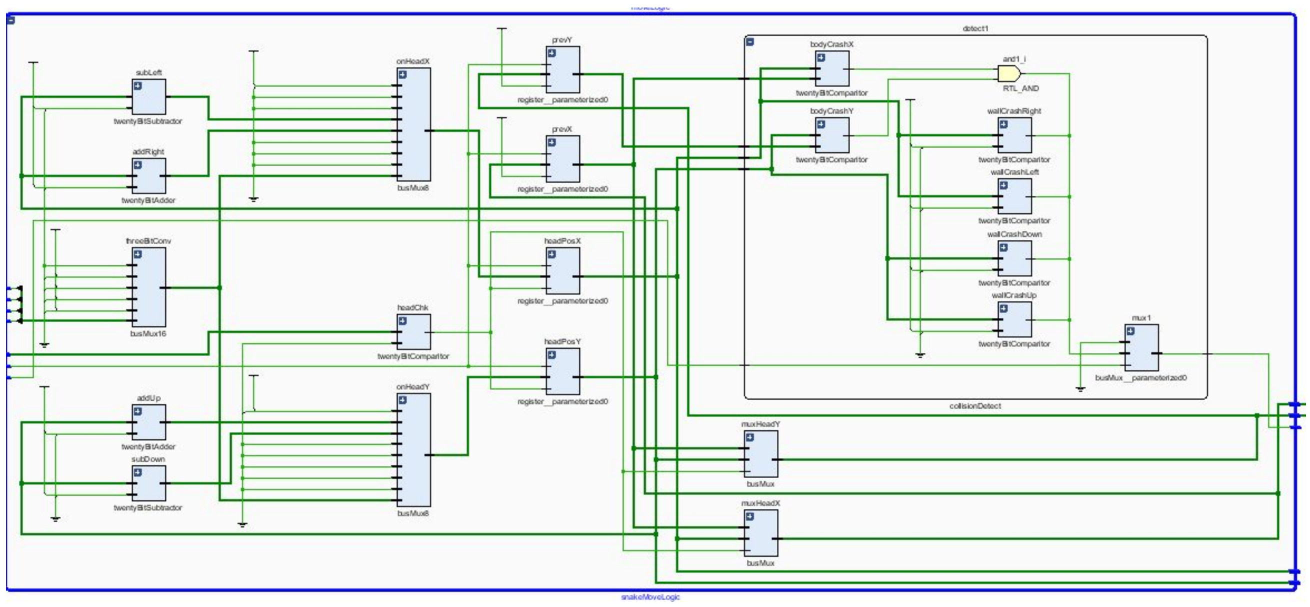
Main File [7]



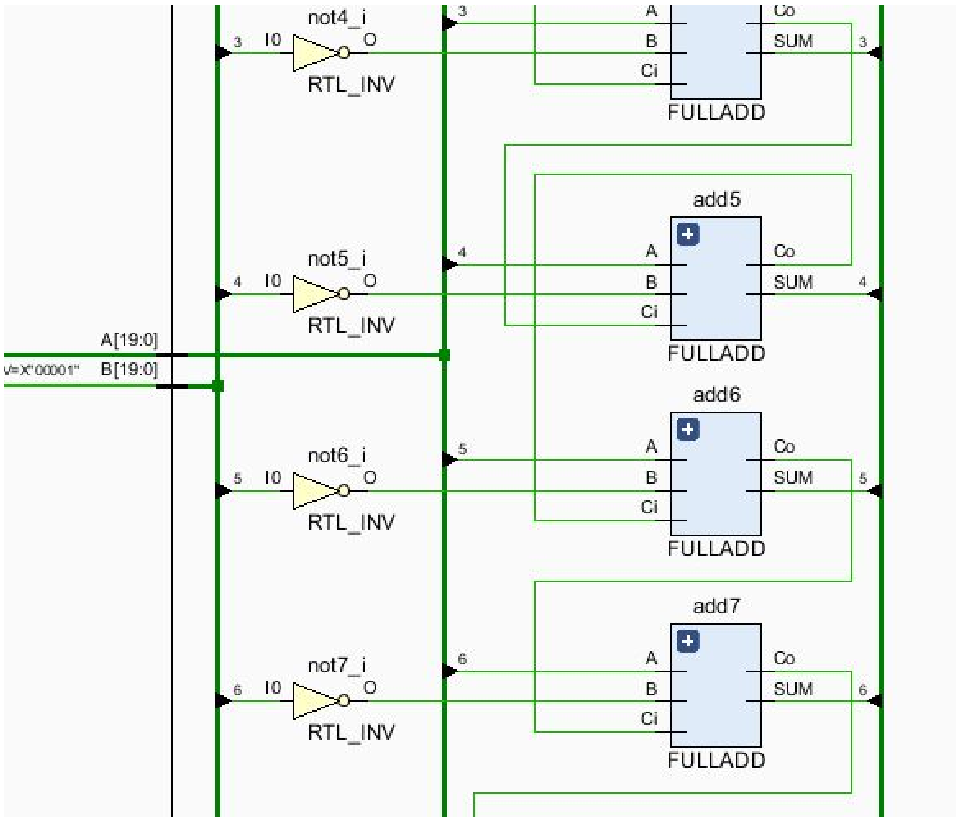
Snake Movement State [8]



Snake Movement Logic [9]



20 Bit Subtractor [10]



Module Code

VGA Driver [1]

```
mmod
ul e
VGA
                (
                   output HS,
                   output VS,
                   output hCount,
                   output vCount,
                   output [3:0] R,
                   output [3:0] G,
                   output [3:0] B,
                   //input tempSwR, tempSwG, tempSwB,
                   input clk
                );


                // Temp Switches to drive RGB Pins
                /*wire color;
                and BUFR0(R[0], tempSwR, color);
                and BUFR1(R[1], tempSwR, color);
                and BUFR2(R[2], tempSwR, color);
                and BUFR3(R[3], tempSwR, color);

                and BUFG0(G[0], tempSwG, color);
                and BUFG1(G[1], tempSwG, color);
                and BUFG2(G[2], tempSwG, color);
                and BUFG3(G[3], tempSwG, color);
```

```
        and BUFB0(B[0], tempSwB, color);

        and BUFB1(B[1], tempSwB, color);

        and BUFB2(B[2], tempSwB, color);

        and BUFB3(B[3], tempSwB, color);*/


        wire [19:0] hCountWire, vCountWire;

        wire clk25, hSyncWire, vSyncWire, hSetWire, hResetWire, vSetWire,
vResetWire;


        // Clock Divider for Pixel Clock


        clockDivide divide

        (

           .clock100(clk),

           .clock25(clk25)

        );


        // Horizontal Counter + Sync


        twentyBitCounter hCounter

        (

           .max(20'b00000000001100100000), //800

           .en(1'b1),

           .clock(clk25),

           .count(hCountWire)

        );
```

```
twentyBitComparitor hSet

(

   .A(hCountWire),

   .B(20'b00000000000000000000), // Zero Detect

   .F(hResetWire)

);



twentyBitComparitor hReset

(

   .A(hCountWire),

   .B(20'b00000000000001100000), // 3.84us (96 clks) Detect

   .F(hSetWire)

);



sr_latch hLatch

(

   .S(hSetWire),

   .R(hResetWire),

   .Q(hSyncWire),

   .Q_Not()

);



// Vertical Counter + Sync



twentyBitCounter vCounter

(

   .max(20'b01100101110000100000), //416,800

   .en(hSyncWire),

   .clock(clk25),
    .count(vCountWire)
);


twentyBitComparitor vSet
(
    .A(vCountWire),
    .B(20'b00000000000000000000), // Zero Detect
    .F(vResetWire)
);


twentyBitComparitor vReset
(
    .A(vCountWire),
    .B(20'b00000000011001000000), // 64us (1600 clks) Detect
    .F(vSetWire)
);


sr_latch vLatch
(
    .S(vSetWire),
    .R(vResetWire),
    .Q(vSyncWire),
    .Q_Not()
);


// Prop wires to sync pins
buf bufHS(HS, hSyncWire);
buf bufVS(VS, vSyncWire);
```

```
        /*wire colorSetWireH, colorResetWireH, colorSetWireV,
colorResetWireV, colorSetWire, colorResetWire, colorWireV, colorWireH;
        twentyBitComparitor colorSetH
        (
           .A(hCountWire),
           .B(20'b00000000000010010001),
           .F(colorSetWireH)
        );

        twentyBitComparitor colorResetH
        (
           .A(hCountWire),
           .B(20'b00000000001100010001),
           .F(colorResetWireH)
        );

        twentyBitComparitor colorSetV
        (
           .A(vCountWire),
           .B(20'b00000110000011100001),
           .F(colorSetWireV)
        );

        twentyBitComparitor colorResetV
        (
           .A(vCountWire),
           .B(20'b01100011110011100001),
           .F(colorResetWireV)
        );
```

```
sr_latch colorLatchH
(
   .S(colorSetWireH),
   .R(colorResetWireH),
   .Q(colorWireH),
   .Q_Not()
);

sr_latch colorLatchV
(
   .S(colorSetWireV),
   .R(colorResetWireV),
   .Q(colorWireV),
   .Q_Not()
);

and and1(color, colorWireV, colorWireH);*/


// Output count addresses
buf bufHCount(hCount, hCountWire);
buf bufVCount(vCount, vCountWire);


endmodule
```

Main File [2]

```
        mod
 ule main
                         (
                            input Up, Down, Left, Right,
                clock,
                            output VS, HS,
                            output [3:0] Red, Green, Blue
                         );


                         wire U, D, L, R, noMove;


                         snakeMoveState moveState
                         (
                            .Up(Up),
                            .Down(Down),
                            .Left(Left),
                            .Right(Right),
                            .clock(clock),
                            .U(U),
                            .D(D),
                            .L(L),
                            .R(R),
                            .noMove(noMove)
                         );


                         wire gameOver;
                         wire [19:0] X, Y;
                         snakeMoveLogic moveLogic
                         (
                            .Up(U),
                            .Down(D),
                            .Left(L),
                            .Right(R),
                            .clock(clock),
                            .wallsOn(1'b1), // Temp

                 .bitNum(20'b00000000000000000000), //
                 Temp
                            .gameOver(gameOver),
                            .X(X),
                            .Y(Y),
                            .headX(),
                            .headY()
                         );


                         wire hCount, vCount;


                         VGA vga
                         (
                            .clk(clock),
                            .HS(HS),
                            .VS(VS),
                            .hCount(hCount),
                            .vCount(vCount)
                         );
```

```
wire xOut, yOut;


twentyBitComparitor vgaX
(
   .A(hCount),
   .B(X),
   .F(xOut)
);


twentyBitComparitor vgaY
(
   .A(vCount),
   .B(Y),
   .F(yOut)
);


wire draw;


and and2(draw, xOut, yOut);
buf red1(Red[0], draw);
buf red2(Red[1], draw);
buf red3(Red[2], draw);
buf red4(Red[3], draw);


buf gr1(Green[0], draw);
buf gr2(Green[1], draw);
buf gr3(Green[2], draw);
buf gr4(Green[3], draw);
buf blu1(Blue[0], draw);

buf blu2(Blue[1], draw);

buf blu3(Blue[2], draw);

buf blu4(Blue[3], draw);


endmodule
```

Twenty-bit Counter [3]

```
        mod
ule
twentyBit
Counter
                    (
                       input [19:0] max,
                       input en,
                       input clock,
                       output [19:0] count
                    );


                    //wire en_pulse;


                    // Pulser for Input
                    /*pulser pulse1
                    (
                       .D(en),
                       .clock(clock),
                       .Q(en_pulse)
                    );*/


                    // Logical Comparitor for Max


                    wire opcode_not, opcode;


                    wire [19:0] muxout;
                    wire [19:0] addout;

                    wire [19:0] dffwire;


                    twentyBitComparitor comp1

                    (

                       .A(max),

                       .B(dffwire),

                       .F(opcode_not)

                    );


                    not not1(opcode, opcode_not);


                    /*wire [9:0]xnorwire;


                    xnor xnor1(xnorwire[0], max[0], dffwire[0]);


                    xnor xnor2(xnorwire[1], max[1], dffwire[1]);


                    xnor xnor3(xnorwire[2], max[2], dffwire[2]);


                    xnor xnor4(xnorwire[3], max[3], dffwire[3]);


                    xnor xnor5(xnorwire[4], max[4], dffwire[4]);


                    xnor xnor6(xnorwire[5], max[5], dffwire[5]);


                    xnor xnor7(xnorwire[6], max[6], dffwire[6]);


                    xnor xnor8(xnorwire[7], max[7], dffwire[7]);
```

```
	xnor xnor9(xnorwire[8], max[8], dffwire[8]);

	xnor xnor10(xnorwire[9], max[9], dffwire[9]);

	nand and1(opcode, xnorwire[0], xnorwire[1], xnorwire[2],
xnorwire[3], xnorwire[4], xnorwire[5], xnorwire[6], xnorwire[7],
xnorwire[8], xnorwire[9]);
	*/


	// Mux for ALU
	mux #(2,1) mux1
	(
	   .data_out(muxout[0]),
	   .select_in({opcode}),
	   .data_in({dffwire[0],1'b0})
	);


	mux #(2,1) mux2
	(
	   .data_out(muxout[1]),
	   .select_in({opcode}),
	   .data_in({dffwire[1],1'b0})
	);


	mux #(2,1) mux3
	(
	   .data_out(muxout[2]),
	   .select_in({opcode}),
	   .data_in({dffwire[2],1'b0})
);


mux #(2,1) mux4
(
   .data_out(muxout[3]),
   .select_in({opcode}),
   .data_in({dffwire[3],1'b0})
);


mux #(2,1) mux5
(
   .data_out(muxout[4]),
   .select_in({opcode}),
   .data_in({dffwire[4],1'b0})
);


mux #(2,1) mux6
(
   .data_out(muxout[5]),
   .select_in({opcode}),
   .data_in({dffwire[5],1'b0})
);


mux #(2,1) mux7
(
   .data_out(muxout[6]),
   .select_in({opcode}),
   .data_in({dffwire[6],1'b0})
);
```

```
mux #(2,1) mux8
(
   .data_out(muxout[7]),
   .select_in({opcode}),
   .data_in({dffwire[7],1'b0})
);


mux #(2,1) mux9
(
   .data_out(muxout[8]),
   .select_in({opcode}),
   .data_in({dffwire[8],1'b0})
);


mux #(2,1) mux10
(
   .data_out(muxout[9]),
   .select_in({opcode}),
   .data_in({dffwire[9],1'b0})
);


mux #(2,1) mux11
(
   .data_out(muxout[10]),
   .select_in({opcode}),
   .data_in({dffwire[10],1'b0})
);
```

```
mux #(2,1) mux12
(
   .data_out(muxout[11]),
   .select_in({opcode}),
   .data_in({dffwire[11],1'b0})
);


mux #(2,1) mux13
(
   .data_out(muxout[12]),
   .select_in({opcode}),
   .data_in({dffwire[12],1'b0})
);


mux #(2,1) mux14
(
   .data_out(muxout[13]),
   .select_in({opcode}),
   .data_in({dffwire[13],1'b0})
);


mux #(2,1) mux15
(
   .data_out(muxout[14]),
   .select_in({opcode}),
   .data_in({dffwire[14],1'b0})
);
```

```
mux #(2,1) mux16

(

   .data_out(muxout[15]),

   .select_in({opcode}),

   .data_in({dffwire[15],1'b0})

);



mux #(2,1) mux17

(

   .data_out(muxout[16]),

   .select_in({opcode}),

   .data_in({dffwire[16],1'b0})

);



mux #(2,1) mux18

(

   .data_out(muxout[17]),

   .select_in({opcode}),

   .data_in({dffwire[17],1'b0})

);



mux #(2,1) mux19

(

   .data_out(muxout[18]),

   .select_in({opcode}),

   .data_in({dffwire[18],1'b0})

);



mux #(2,1) mux20
```

```
(
   .data_out(muxout[19]),
   .select_in({opcode}),
   .data_in({dffwire[19],1'b0})
);


//Flip-Flops for Enable


register #(20) reg1
(
   .clk(clock),
   .load(en),
   .d(addout),
   .q(dffwire)
);


/*dff_en dff1
(
   .data_out(dffwire[0]),
   .in_D(addout[0]),
   .in_CLK(clock),
   .in_EN(en_pulse)
);


dff_en dff2
(
   .data_out(dffwire[1]),
   .in_D(addout[1]),
   .in_CLK(clock),
   .in_EN(en_pulse)
);

dff_en dff3
(
   .data_out(dffwire[2]),
   .in_D(addout[2]),
   .in_CLK(clock),
   .in_EN(en_pulse)
);

dff_en dff4
(
   .data_out(dffwire[3]),
   .in_D(addout[3]),
   .in_CLK(clock),
   .in_EN(en_pulse)
);

dff_en dff5
(
   .data_out(dffwire[4]),
   .in_D(addout[4]),
   .in_CLK(clock),
   .in_EN(en_pulse)
);

dff_en dff6
(
   .data_out(dffwire[5]),
   .in_D(addout[5]),

   .in_CLK(clock),

   .in_EN(en_pulse)

);


dff_en dff7

(

   .data_out(dffwire[6]),

   .in_D(addout[6]),

   .in_CLK(clock),

   .in_EN(en_pulse)

);


dff_en dff8

(

   .data_out(dffwire[7]),

   .in_D(addout[7]),

   .in_CLK(clock),

   .in_EN(en_pulse)

);


dff_en dff9

(

   .data_out(dffwire[8]),

   .in_D(addout[8]),

   .in_CLK(clock),

   .in_EN(en_pulse)

);


dff_en dff10
```

```
(
   .data_out(dffwire[9]),
   .in_D(addout[9]),
   .in_CLK(clock),
   .in_EN(en_pulse)
); */


// Logical 10-Bit Adder


twentyBitAdder add1
(
   .A(1'b1),
   .B(muxout),
   .S(addout)
);


assign count = muxout;



endmodule
```

FSMs

snakeMoveState [1]

d2	d1	d0	U	D	L	R		q2	q1	q0	Up	Down	Left	Right	No Move
0	0	0	0	0	0	0		0	0	0	0	0	0	0	1
0	0	0	0	0	0	1		1	0	0	0	0	0	0	1
0	0	0	0	0	1	0		0	1	1	0	0	0	0	1
0	0	0	0	0	1	1		0	0	0	0	0	0	0	1
0	0	0	0	1	0	0		0	1	0	0	0	0	0	1
0	0	0	0	1	0	1		0	0	0	0	0	0	0	1
0	0	0	0	1	1	0		0	0	0	0	0	0	0	1
0	0	0	0	1	1	1		0	0	0	0	0	0	0	1
0	0	0	1	0	0	0		0	0	1	0	0	0	0	1
0	0	0	1	0	0	1		0	0	0	0	0	0	0	1
0	0	0	1	0	1	0		0	0	0	0	0	0	0	1
0	0	0	1	0	1	1		0	0	0	0	0	0	0	1
0	0	0	1	1	0	0		0	0	0	0	0	0	0	1
0	0	0	1	1	0	1		0	0	0	0	0	0	0	1
0	0	0	1	1	1	0		0	0	0	0	0	0	0	1
0	0	0	1	1	1	1		0	0	0	0	0	0	0	1
0	0	1	0	0	0	0		0	0	1	1	0	0	0	0
0	0	1	0	0	0	1		1	0	0	1	0	0	0	0
0	0	1	0	0	1	0		0	1	1	1	0	0	0	0
0	0	1	0	0	1	1		0	0	1	1	0	0	0	0
0	0	1	0	1	0	0		0	1	0	1	0	0	0	0
0	0	1	0	1	0	1		0	0	1	1	0	0	0	0
0	0	1	0	1	1	0		0	0	1	1	0	0	0	0
0	0	1	0	1	1	1		0	0	1	1	0	0	0	0
0	0	1	1	0	0	0		0	0	1	1	0	0	0	0
0	0	1	1	0	0	1		0	0	1	1	0	0	0	0
0	0	1	1	0	1	0		0	0	1	1	0	0	0	0
0	0	1	1	0	1	1		0	0	1	1	0	0	0	0
0	0	1	1	1	0	0		0	0	1	1	0	0	0	0
0	0	1	1	1	0	1		0	0	1	1	0	0	0	0
0	0	1	1	1	1	0		0	0	1	1	0	0	0	0
0	0	1	1	1	1	1		0	0	1	1	0	0	0	0

0	1	0	0	0	0	0		0	1	0	0	1	0	0	0
0	1	0	0	0	0	1		1	0	0	0	1	0	0	0
0	1	0	0	0	1	0		0	1	1	0	1	0	0	0
0	1	0	0	0	1	1		0	1	0	0	1	0	0	0
0	1	0	0	1	0	0		0	1	0	0	1	0	0	0
0	1	0	0	1	0	1		0	1	0	0	1	0	0	0
0	1	0	0	1	1	0		0	1	0	0	1	0	0	0
0	1	0	0	1	1	1		0	1	0	0	1	0	0	0
0	1	0	1	0	0	0		0	0	1	0	1	0	0	0
0	1	0	1	0	0	1		0	1	0	0	1	0	0	0
0	1	0	1	0	1	0		0	1	0	0	1	0	0	0
0	1	0	1	0	1	1		0	1	0	0	1	0	0	0
0	1	0	1	1	0	0		0	1	0	0	1	0	0	0
0	1	0	1	1	0	1		0	1	0	0	1	0	0	0
0	1	0	1	1	1	0		0	1	0	0	1	0	0	0
0	1	0	1	1	1	1		0	1	0	0	1	0	0	0
0	1	1	0	0	0	0		0	1	1	0	0	1	0	0
0	1	1	0	0	0	1		1	0	0	0	0	1	0	0
0	1	1	0	0	1	0		0	1	1	0	0	1	0	0
0	1	1	0	0	1	1		0	1	1	0	0	1	0	0
0	1	1	0	1	0	0		0	1	0	0	0	1	0	0
0	1	1	0	1	0	1		0	1	1	0	0	1	0	0
0	1	1	0	1	1	0		0	1	1	0	0	1	0	0
0	1	1	0	1	1	1		0	1	1	0	0	1	0	0
0	1	1	1	0	0	0		0	0	1	0	0	1	0	0
0	1	1	1	0	0	1		0	1	1	0	0	1	0	0
0	1	1	1	0	1	0		0	1	1	0	0	1	0	0
0	1	1	1	0	1	1		0	1	1	0	0	1	0	0
0	1	1	1	1	0	0		0	1	1	0	0	1	0	0
0	1	1	1	1	0	1		0	1	1	0	0	1	0	0
0	1	1	1	1	1	0		0	1	1	0	0	1	0	0
0	1	1	1	1	1	1		0	1	1	0	0	1	0	0
1	0	0	0	0	0	0		1	0	0	0	0	0	1	0
1	0	0	0	0	0	1		1	0	0	0	0	0	1	0
1	0	0	0	0	1	0		0	1	1	0	0	0	1	0
1	0	0	0	0	1	1		1	0	0	0	0	0	1	0
1	0	0	0	1	0	0		0	1	0	0	0	0	1	0
1	0	0	0	1	0	1		1	0	0	0	0	0	1	0
1	0	0	0	1	1	0		1	0	0	0	0	0	1	0

1	0	0	0	1	1	1		1	0	0	0	0	0	1	0
1	0	0	1	0	0	0		0	0	1	0	0	0	1	0
1	0	0	1	0	0	1		1	0	0	0	0	0	1	0
1	0	0	1	0	1	0		1	0	0	0	0	0	1	0
1	0	0	1	0	1	1		1	0	0	0	0	0	1	0
1	0	0	1	1	0	0		1	0	0	0	0	0	1	0
1	0	0	1	1	0	1		1	0	0	0	0	0	1	0
1	0	0	1	1	1	0		1	0	0	0	0	0	1	0
1	0	0	1	1	1	1		1	0	0	0	0	0	1	0
1	0	1	0	0	0	0									
1	0	1	0	0	0	1									
1	0	1	0	0	1	0									
1	0	1	0	0	1	1									
1	0	1	0	1	0	0									
1	0	1	0	1	0	1									
1	0	1	0	1	1	0									
1	0	1	0	1	1	1									
1	0	1	1	0	0	0									
1	0	1	1	0	0	1									
1	0	1	1	0	1	0									
1	0	1	1	0	1	1									
1	0	1	1	1	0	0									
1	0	1	1	1	0	1									
1	0	1	1	1	1	0									
1	0	1	1	1	1	1									
1	1	0	0	0	0	0									
1	1	0	0	0	0	1									
1	1	0	0	0	1	0									
1	1	0	0	0	1	1									
1	1	0	0	1	0	0									
1	1	0	0	1	0	1									
1	1	0	0	1	1	0									
1	1	0	0	1	1	1									
1	1	0	1	0	0	0									
1	1	0	1	0	0	1									
1	1	0	1	0	1	0									
1	1	0	1	0	1	1									
1	1	0	1	1	0	0									
1	1	0	1	1	0	1									

1	1	0	1	1	1	0									
1	1	0	1	1	1	1									
1	1	1	0	0	0	0									
1	1	1	0	0	0	1									
1	1	1	0	0	1	0									
1	1	1	0	0	1	1									
1	1	1	0	1	0	0									
1	1	1	0	1	0	1									
1	1	1	0	1	1	0									
1	1	1	0	1	1	1									
1	1	1	1	0	0	0									
1	1	1	1	0	0	1									
1	1	1	1	0	1	0									
1	1	1	1	0	1	1									
1	1	1	1	1	0	0									
1	1	1	1	1	0	1									
1	1	1	1	1	1	0									
1	1	1	1	1	1	1									

GitHub Repository

For further insight into the development process and access to unused files, please refer to the relevant GitHub repository: https://github.com/Audrax/eece-snake