Building a Snake Game Using FPGA

Morgan Harmon, Jiekang Xu
Department of Electrical and Computer Engineering, Miami University, Oxford, Ohio
harmonm2@miamioh.edu, xuj8@miamioh.edu

I. INTRODUCTION

"Snake", a game that originated from the 1976 arcade game Blockade shown in Figure 1, shows a decade when videogames were rudimentary, relying upon the hardware making use of clocks, registers, and wires [1]. The simple game allows a user to control a snake on a VGA screen. The objective of the game is to eat as many randomly generated apples as possible without colliding into walls or the snake's tail. This paper discusses our research into building the game, Snake, from the hardware level. We were motivated to build the snake game to advance our experience with Verilog and to understand the complexity of th seemly simple game.

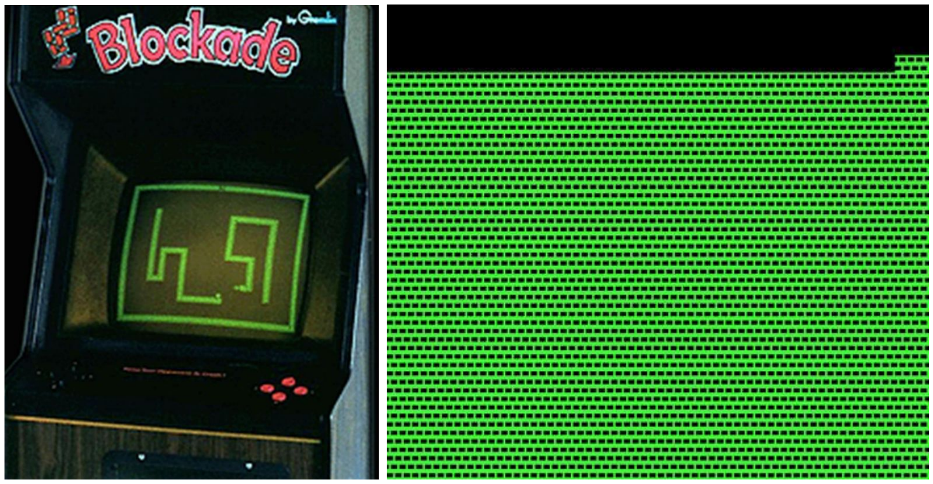


Figure 1: Shows the game, Blockade, that the snake game is based upon [6][7].

II. BACKGROUND

The following project was split into four parts to improve development and simplicity. These four parts are working with the keyboard input, creating VGA output, generating apples and collisions, and coding the snake and game specifics.

A. Keyboard Input

To get input from the keyboard there are several concepts to make the implementation simple. Here are the principles for implementing a keyboard input module: 1) having the keyboard send a 1 for data and clock when not sending input, 2) a single key press will have 33 bits of input from the keyboard, 3) when a key is pressed a ‘make’ code of 11 bits of data is released and then a 11 bit ‘break’ code is sent, 4) the 11-bit make code will be sent repeatedly while the key is held down, 5) the keyboard sends the data on the negative edge of the keyboard clock, and 6) the keyboard has its own clock that does not need to be defined with the FPGA clock [2]. Every 11 bits contains four elements. The first bit is a zero to signal start. Then eight bits of data are sent to determine the specific key pressed. After that a parity bit and a one to signify stop is sent. This works specifically with a PS/2 keyboard.

B. VGA Output

VGA is an analog video standard that does not require high clocking speeds or complex encoding [3]. The VGA has five main signal pins one for red, one for green, one for blue, and two for syncing. A typical VGA display is 640x480 at 60 Hz. This requires a clock speed of 25 MHz, where each tick of the clock is a pixel [3]. The VGA is made up of frames that are made up of a series of horizontal lines, each line is then made up of a series of pixels [4]. Transmission of these pixels are from top to bottom and left to right within the pixel lines [4]. There are a front and back porch within the timing of syncing information. This information is synced during the horizontal blanking period in the analog video. The front porch is the interval period between the end of information and start of a horizontal pulse. The back porch is the duration between the end of the horizontal pulse and start of next line of information. This process of sending and receiving information is shown in Figure 2. The transition between the front porch and sync pulse shows the resetting of information for the next frame. The back porch then sends the next round of information.

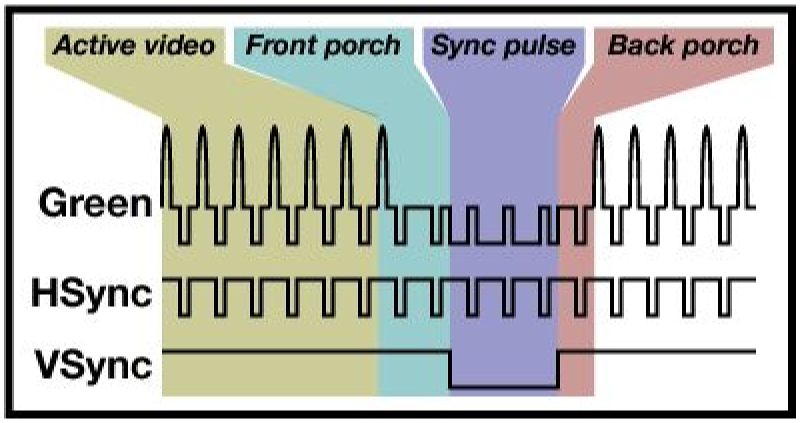
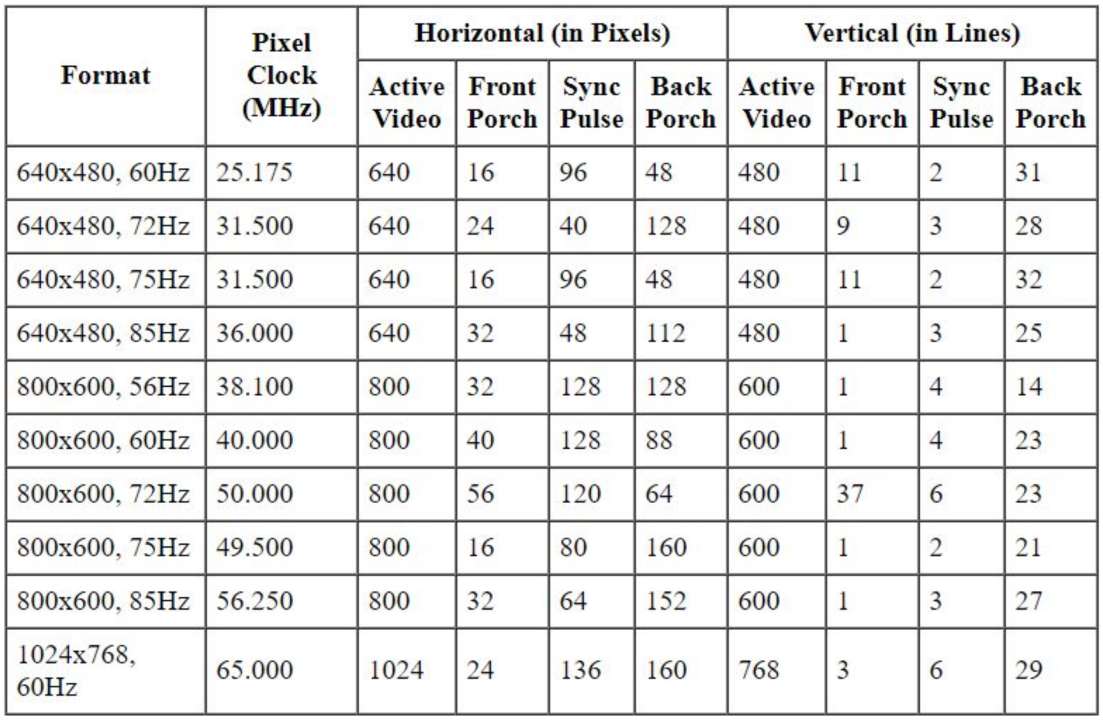


Figure 2: The clocking diagram for VGA sending/receiving information [4].

Because there are different video formats ranging from 640x480, 60Hz to 1024x768, 85Hz, timing values for popular resolutions are given for the four different modes for both the horizontal pixels and vertical lines of the VGA, as shown in Figure 2.



Format	Pixel Clock (MHz)	Horizontal (in Pixels)				Vertical (in Lines)			
		Active Video	Front Porch	Sync Pulse	Back Porch	Active Video	Front Porch	Sync Pulse	Back Porch
640x480, 60Hz	25.175	640	16	96	48	480	11	2	31
640x480, 72Hz	31.500	640	24	40	128	480	9	3	28
640x480, 75Hz	31.500	640	16	96	48	480	11	2	32
640x480, 85Hz	36.000	640	32	48	112	480	1	3	25
800x600, 56Hz	38.100	800	32	128	128	600	1	4	14
800x600, 60Hz	40.000	800	40	128	88	600	1	4	23
800x600, 72Hz	50.000	800	56	120	64	600	37	6	23
800x600, 75Hz	49.500	800	16	80	160	600	1	2	21
800x600, 85Hz	56.250	800	32	64	152	600	1	3	27
1024x768, 60Hz	65.000	1024	24	136	160	768	3	6	29

Figure 2: Timing values for popular resolutions [4].

C. Generating Apples and Collisions

Having completed the keyboard and VGA output, apple collisions could now be generating and tested on the screen. The rules of Snake are simple, eat as many apples as you can, and do not hit the walls or your tail. The apples typically appear randomly upon the screen. This meant creating a pseudo-random coordinate generator module in Verilog. Also, a collision detection module was needed to determine what would be a lethal collision for the snake. This meant checking if the snake hit itself or any of the walls. However, hitting an apple would be an okay hit. The meant the snake actually had to grow in size when it collided with an apple.

D. Coding the Snake and Game Specifics

The creation of the snake involves having the pixels regenerate at a certain speed on the VGA, determining the speed of the snake, and having the direction of the snake be based upon the last keystroke. To keep the coloring of the game simple, the snake is white on a black background with the apples being red, and a blue border. The classic design of the snake game is shown in Figure 3.

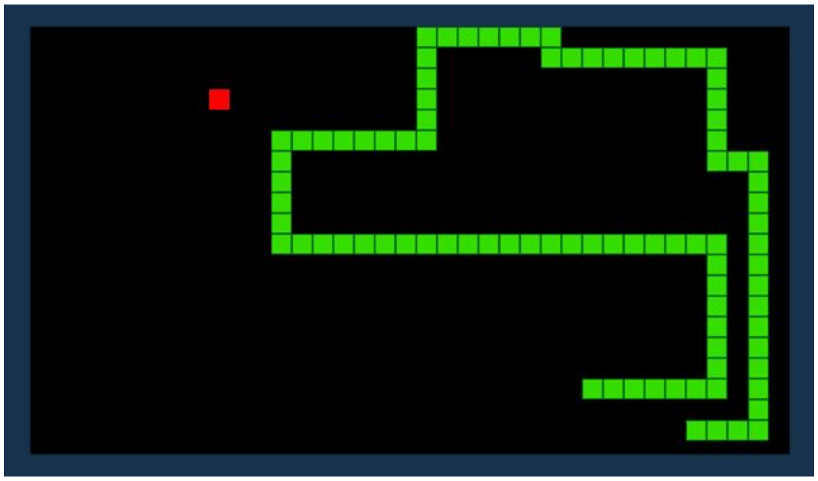


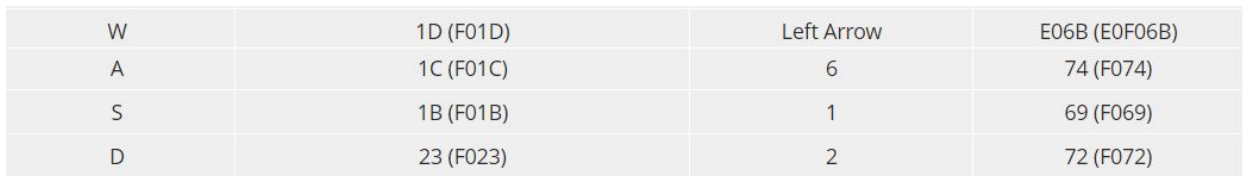
Figure 3: The basic design of the snake game [5].

III. PROJECT DESCRIPTION

The following section shows the procedures taken to complete the snake game. The keyboard input, VGA output, apple generation and collisions, and coding the snake are explained in implementation means to show the correlation between game functions and registers, clocks, and wires.

A. Keyboard Input

Implementation of the keyboard input involves the keyboard having its own clock and data variables. A register denotes the 5 directions, up, down, left, right, and stop. The key press code is an 8-bit register. The make and break code are 11-bit registers. Then there is a count to determine when 11-bits have gone through. The keyboard clock is on the negative edge. An always block is implemented to determine the state of the keyboard. The count increments by one until it reaches 11-bits. Then the previous code is examined for a button release. If the button has been release the key stroke is recorded. The count is then set back to zero, waiting for the next 11-bit instructions. The key press code is then used to determine the direction of the snake. If the key is W, the snake will move upwards. If the key is A, the snake will move left. If the key is D, the snake moves right. If the key is S, the snake will move down. This module provides the necessary correlation between the PS2 keyboard and the FPGA. Shown below in Figure 4 are the W, A, S, D keys on the PS2 keyboard to allow the user to interact with the VGA by changing the snake direction.



W	1D (F01D)	Left Arrow	E06B (E0F06B)
A	1C (F01C)	6	74 (F074)
S	1B (F01B)	1	69 (F069)
D	23 (F023)	2	72 (F072)

Figure 4: The assigned key values for the PS2 keyboard.

B. *VGA Output*

The VGA generator has its own clock. To generate visual output on the VGA, the parameters of horizontal front, horizontal sync, horizontal back, max horizontal, vertical front, vertical sync, vertical back, and max vertical must be set. The resolution used was 640x480. The VGA clock was used to increment the x and y coordinates on the screen. Loops are then implemented to display the area and continuously go through the values of the horizontal sync and the vertical sync.

C. *Apple Generation and Collisions*

The apples were randomly generated on the screen by using the modulus function to determine a random x coordinate and y coordinate within the bounds of the VGA window. This was specifically created for up to fifteen apple spaces. For wall collisions made by the snake, border boundaries were specified. This was defined by a ten pixel buffer between the 640x480 screen. If the x-value for the snake went above 630 or y-value for the snake went above 470, the game ended.

D. *Coding the Snake and Game Specifics*

The generation of the snake involved respawning upon eating food, generating the RGB value for the snake, and linking together the head to an incremental number of body squares. The coordinates of the snake head and the apple would be compared to determining if the snake would be an additional body part added. The snake head and snake body coordinates were also compared for collisions. If the coordinates were the same between the snake head and a body part, the game over screen would appear. The game also ends if the snake head and border coordinates match. The snake head was given x,y coordinates based off of a x and y counter. To move across the screen. This meant updating the clock to lower the speed of the snake to an acceptable level. The clocks between the FPGA and VGA also had to match. This meant reducing the clock from 50 MHz to 25 MHz.

IV. RESULTS

The results of this project can be seen in Figure 5 and Figure 6. The main objectives of development of the snake game were completed. The user can reset the game, play using the W, A, S, D keys, and obtain randomly generated apples that appear on the screen.

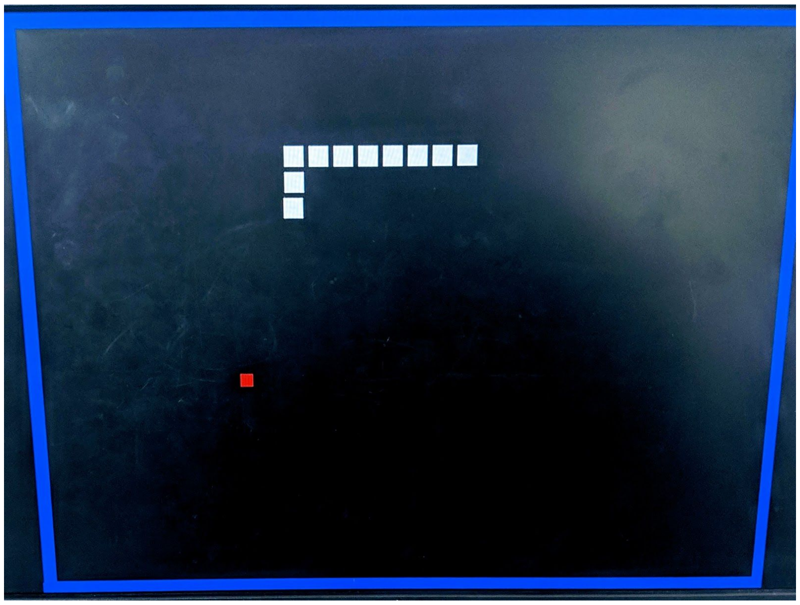


Figure 5: Gameplay of our project on the Altera DE2 board.

Figure 6 shows the speed of the generated snake and the pixel being redrawn per frame cycle. Once the snake eats an apple, the body length increases by one. The user can move the snake using the W, A , S, D keys for direction of the snake..

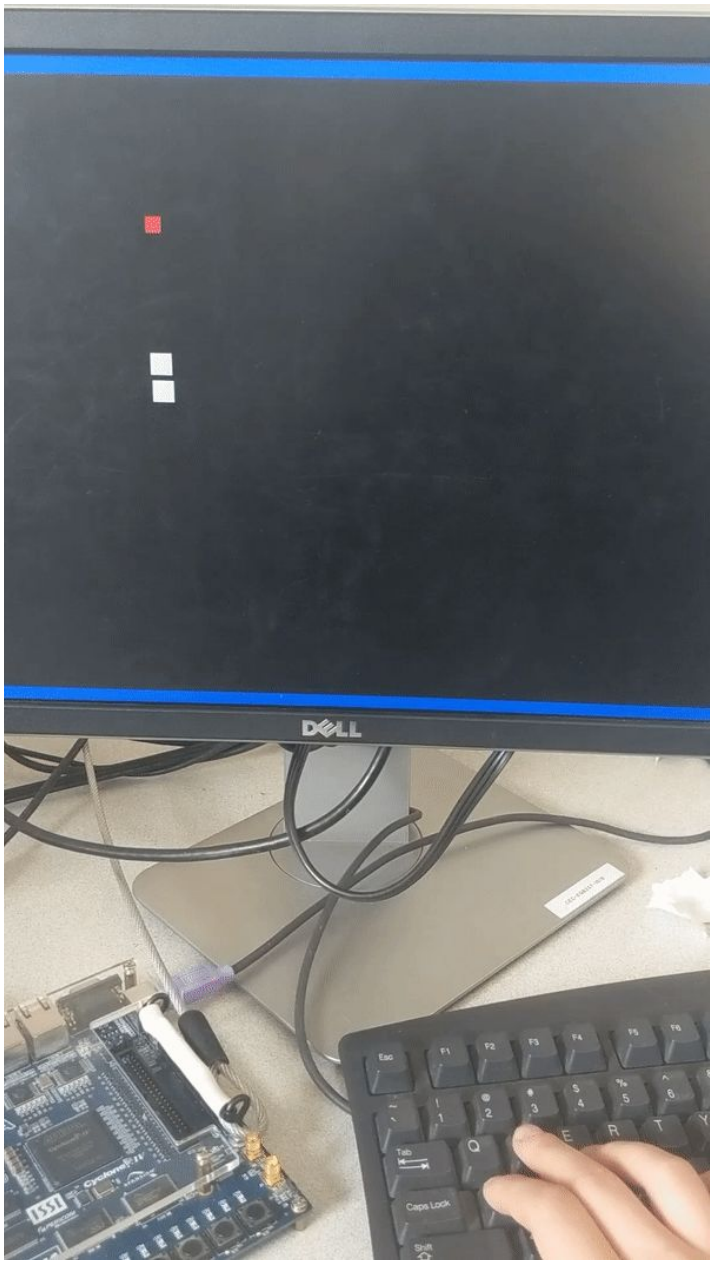


Figure 6: Additional gameplay of the developed snake game showing use of the PS2 keyboard

Although the main foundation of the game is developed and the functionality is adequate. Our team would have liked more time to continue fixing minor issues. Minor issues found in the game were the reset button would on rare occasion lead to a green screen or a won game, the snake had a maximum length of fifteen, which leads to a game won, and the snake could back tracking on itself resulting in a gameover.

V. CONCLUSION

This paper has described the development of the game Snake on the FPGA. It has shown the four main aspects of building the game: the keyboard input, VGA output, apple generation and collisions, and coding the snake and game specifics. The keyboard provides the user with a means of interacting with the VGA. The mains features of the keyboard are that data is sent on the negative edge of the clock, eleven bits of make and break code are send when a key is held, and the keyboard has its own clock. The VGA provides a visual screen to view the generated pixel design. The mains features of the VGA output are the change in horizontal and vertical values when resolution changes, the clock between the VGA and the FPGA must match for exchange of information, and setting the RGB values for the pixels being sent. The apple generation and collision detection is vital for providing an engaging user experience. The main features of this is pseudo-random generation of x, y coordinates within the VGA window, and creating a buffer between the edge of the VGA window that works as the boundary of the game. The generation of the snake and the game specifics provide the foundation of the game. The main features are correcting the speed of the snake, generating additional square body parts of the snake, and assigning the color value. This paper provides a foundational understanding of the development of the game snake on the Altera DE2 board.

VI. ACKNOWLEDGEMENTS

Special thanks goes to Ian Sweetland, Kristjan Jacobson, and Daniel Lovegrove who provided the modules for the keyboard input and VGA output, as well as, inspiration for building the apple collisions and snake generation. These modules can be found on https://www.instructables.com/id/Snake-on-an-FPGA-Verilog/, where the group put a guide to building the game on the Altera DE2 board. The keyboard input module and VGA output code created by the group can also be found at the bottom of the report.

VII. REFERENCES

[1] https://en.wikipedia.org/wiki/Snake_(video_game)

[2] https://www.instructables.com/id/Snake-on-an-FPGA-Verilog/

[3] https://timetoexplore.net/blog/arty-fpga-vga-verilog-01

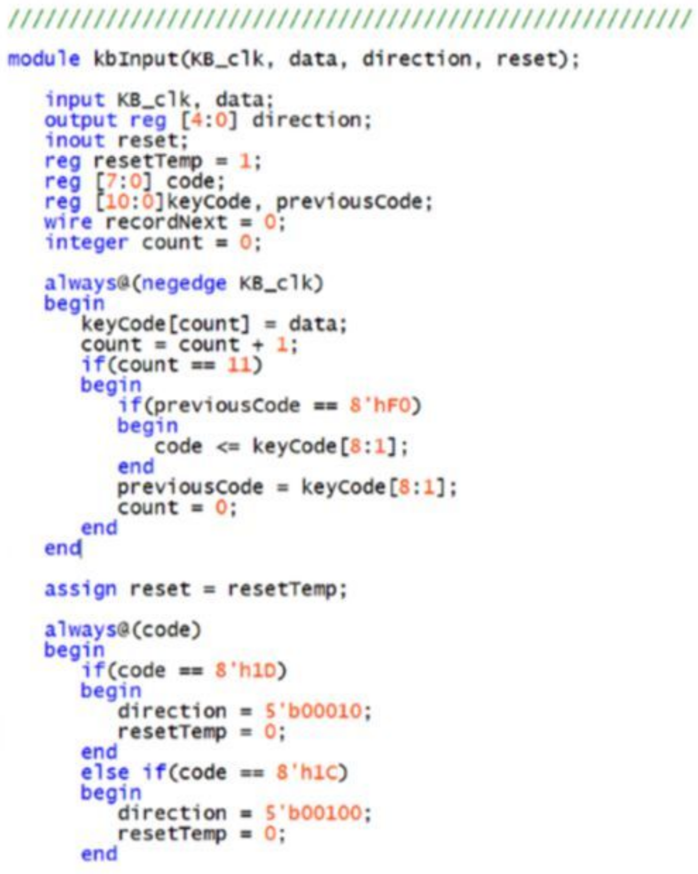
[4] http://web.mit.edu/6.111/www/s2004/NEWKIT/vga.shtml

[5] https://www.coolmathgames.com/0-snake

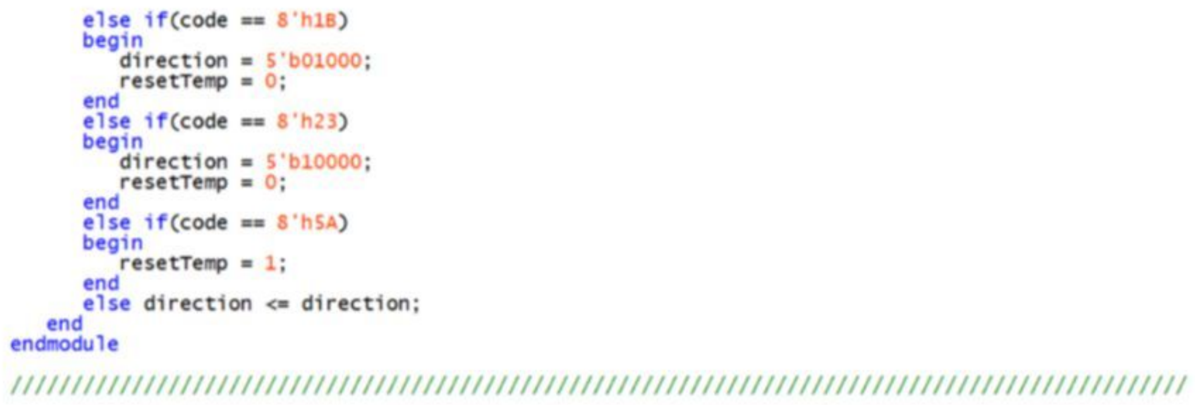
[6]https://lh5.googleusercontent.com/X396gZA1hKhaEPChpFiB12D4G2hvycjcG4aS6fOkZOGOBtSB4qKXQC7PZGZwzeesNL6ZvOHE9Z5FzURP1cMcwbcRKpgXIRGqeFmvsc9NaDcsnlPz9ysinlMw9cfzzachatSJhFar

[7] https://thumbs.gfycat.com/CoarseHarmoniousGroundbeetle-size_restricted.gif

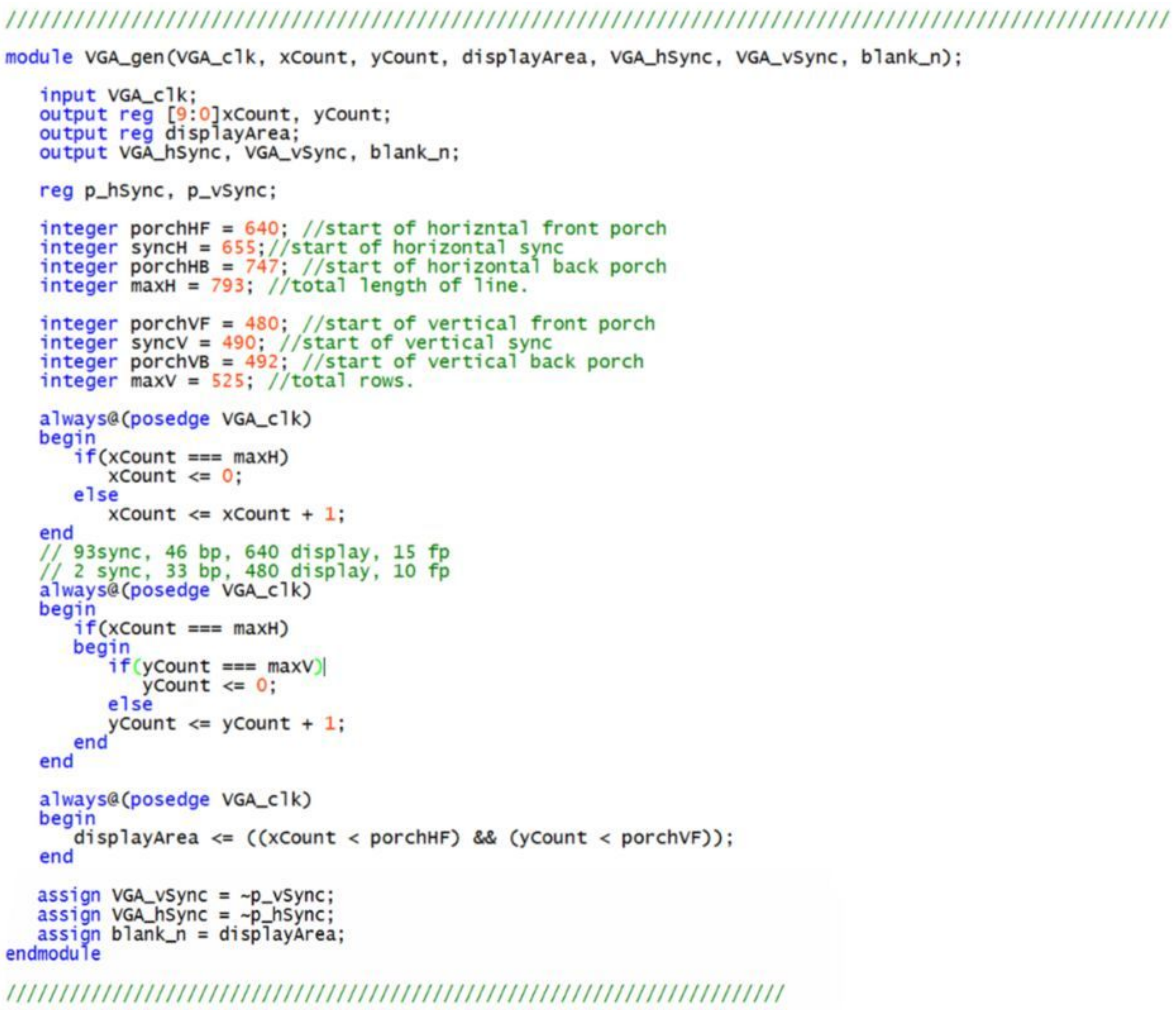
The following keyboard and VGA output modules were produced by Ian Sweetland, Kristjan Jacobson, and Daniel Lovegrove.



```
/////////////////////////////////////////////////////////
module kbInput(KB_clk, data, direction, reset);

   input KB_clk, data;
   output reg [4:0] direction;
   inout reset;
   reg resetTemp = 1;
   reg [7:0] code;
   reg [10:0]keyCode, previousCode;
   wire recordNext = 0;
   integer count = 0;

   always@(negedge KB_clk)
   begin
      keyCode[count] = data;
      count = count + 1;
      if(count == 11)
      begin
         if(previousCode == 8'hF0)
         begin
            code <= keyCode[8:1];
         end
         previousCode = keyCode[8:1];
         count = 0;
      end
   end

   assign reset = resetTemp;

   always@(code)
   begin
      if(code == 8'h1D)
      begin
         direction = 5'b00010;
         resetTemp = 0;
      end
      else if(code == 8'h1C)
      begin
         direction = 5'b00100;
         resetTemp = 0;
      end
```



```
      else if(code == 8'h1B)
      begin
         direction = 5'b01000;
         resetTemp = 0;
      end
      else if(code == 8'h23)
      begin
         direction = 5'b10000;
         resetTemp = 0;
      end
      else if(code == 8'h5A)
      begin
         resetTemp = 1;
      end
      else direction <= direction;
   end
endmodule
//////////////////////////////////////////////////////////////////////////////////////
```



```
////////////////////////////////////////////////////////////////////////////////////////////////////
module VGA_gen(VGA_clk, xCount, yCount, displayArea, VGA_hSync, VGA_vSync, blank_n);

   input VGA_clk;
   output reg [9:0]xCount, yCount;
   output reg displayArea;
   output VGA_hSync, VGA_vSync, blank_n;

   reg p_hSync, p_vSync;

   integer porchHF = 640; //start of horizntal front porch
   integer syncH = 655;//start of horizontal sync
   integer porchHB = 747; //start of horizontal back porch
   integer maxH = 793; //total length of line.

   integer porchVF = 480; //start of vertical front porch
   integer syncV = 490; //start of vertical sync
   integer porchVB = 492; //start of vertical back porch
   integer maxV = 525; //total rows.

   always@(posedge VGA_clk)
   begin
      if(xCount === maxH)
         xCount <= 0;
      else
         xCount <= xCount + 1;
   end
   // 93sync, 46 bp, 640 display, 15 fp
   // 2 sync, 33 bp, 480 display, 10 fp
   always@(posedge VGA_clk)
   begin
      if(xCount === maxH)
      begin
         if(yCount === maxV)
            yCount <= 0;
         else
         yCount <= yCount + 1;
      end
   end

   always@(posedge VGA_clk)
   begin
      displayArea <= ((xCount < porchHF) && (yCount < porchVF));
   end

   assign VGA_vSync = ~p_vSync;
   assign VGA_hSync = ~p_hSync;
   assign blank_n = displayArea;
endmodule

///////////////////////////////////////////////////////////////////////
```