Snake Game on Nexys A7

By: James Hamm, Louis Turriaga, Jordan Tejada, Triet Vo

## Table of Contents:

### 3 - Abstract

### 4. Introduction

### 5 - Display\_Controller.v

### 9 - Snake\_Game.v

### 12 - Snake\_Controller.v

### 15 - TOP.v

### 17 - Conclusion

### 18 - Addendum

## Abstract:

Our group project delves into the realm of hardware-based gaming by implementing the classic Snake game on a Field-Programmable Gate Array (FPGA) platform. Leveraging the versatility of FPGA, we have integrated a myriad of interactive elements, including VGA for display, the onboard XADC for difficulty adjustment, 7-segment displays for score tracking, LEDs for dynamic visual feedback, and an array of switches and buttons for user input.

The FPGA serves as the backbone of our gaming system, executing complex algorithms to simulate the Snake game dynamics efficiently. The VGA interface facilitates a visually immersive gaming experience, projecting the game state onto a monitor in real-time. This is done through a hardware description language (HDL), specifically Verilog. With the understanding of how VGAs work and their essential components like horizontal and vertical syncs, the program maps out the screen however we desire. The VGA provides the creative outlet as the customization of colors and designs are entirely open to us as long as we know how to express it mathematically.

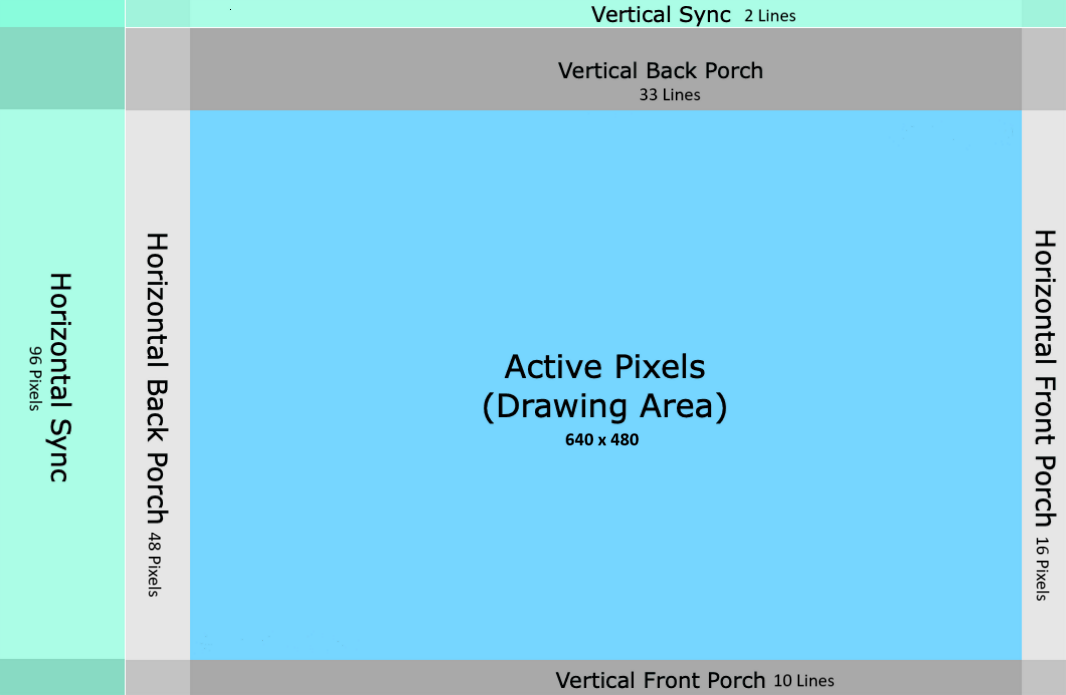
To enhance user engagement, the 7-segment displays showcase the player's score, creating a seamless connection between the virtual and physical worlds. Utilizing the Nexys A7s onboard XADC, the user’s engagement is further supplemented by implementing a difficulty change input. Through this, the user is offered a pseudo version of a difficulty screen similar to selecting ‘easy’, ‘normal’, or ‘hard’ difficulty on screen like many other games nowadays. With the strict usage of Verilog and an FPGA, our project aims to remake this classic game and we believe we have achieved this.

## Introduction

To make our snake game, we needed to break down the core components of the game. This includes a controller that would manage the VGA display, a controller for the logic of the game, and finally a controller for how the user perceives the game. These core components would become separate modules that would later be combined into our TOP file. It is also important to note that this alone wouldn’t be enough for the assignment. What we needed to include was the XADC feature. So, we decided to implement this by adding a variable game speed that could be altered through an external potentiometer. This would be done by utilizing Vivado’s build in IP catalog (Specifically XADC wizard), and adding it into the TOP module.

## Display\_Controller.v

This module generates timing signals for horizontal and vertical synchronization, tracks pixel positions, and controls the active area of pixels based on the VGA display timing specifications. It's a fundamental component in creating a VGA display interface and can generally be envisioned as drawing a rectangle on a canvas and stating that you can only paint on that defined area.

While this module does the timing signals which are responsible for synchronizing the various components involved in displaying an image on a screen it also sets up a needed foundation for the video signals in other modules to actually draw on the screen. Video Signals have two phases: drawing pixels and a blanking interval with the sync signal occurring within the blanking intervals. The sync occurs within the blanking interval which is separated from the drawing area by the front and back porch; horizontal sync demarcating a line and vertical sync a frame. 

Rewinding a bit, let's talk about the input and outputs of this module.

Input:

* clk: system clock of 100 Mhz

Outputs:

* hSync: indicator for a new line
* vSync: indicator for a new frame
* bright: designated area controller
* hCount: coordinate of current horizontal position
* vCount: coordinate of current vertical position

To start off this module’s code rundown, we’ll discuss the clock input. A clock of 25.175Mhz is standard for a 640x480 VGA display as it handles the timing and synchronization very well. Instead of using the built-in clock wizard in Vivado to get this value, we performed this:

always @(posedge clk) pulse = ~pulse;

always @(posedge pulse) clk25 = ~clk25;

This takes the 100 Mhz system clock and makes it 50 Mhz by making a new clock “pulse” based on the system clock’s positive edge transition and toggling a flip-flop. Then, the same action is done on the new clock’s positive edge transition, giving us a 25 Mhz clock called “clk25” which we use throughout this module. Although it is not quite the value we want, after testing we found that this simple manual solution works great for this project.

Next, this module goes through an always block that runs through each individual pixel on the display. For us with our 640x480 VGA, that means looping through 800 pixels (the horizontal) and 525 lines (the vertical) per line. 800 corresponds to the 640 active pixels plus the 16 pixels from its front porch, 48 from its back porch, and 96 from the horizontal sync width. While the 525 comes from the 480 active lines plus the 10 from the front porch, 33 from the back porch, and 2 from the vertical sync width. All of these values can be found on VGA 640x480 60 Hz documentations easily accessible online as well as their placement on the screen which is shown in the display visual above.

The module then checks if the current coordinate is in or outside of their respective sync area. This helps everything as a whole stay synced up and guarantee good timing as it gets outputted using the XDC file, allowing the FPGA to do the rest.

assign hSync = (hCount < 96) ? 0:1;

assign vSync = (vCount < 2) ? 0:1;

To finish off this module, unlike the earlier always block that kept updating the values of the hCount and vCount every clock posedge, this last always block checks the coordinates and designates it as either bright or not bright. It doesn’t concern actual brightness as it is just a single wire that goes to the Snake\_Controller module but based on if the coordinate is not bright then it will get an RGB value of 0. This essentially just makes it black and not the play area for the game. Afterwards, the Snake\_Controller module now has free reign to place whatever it wants in the pixels designated as “bright”.

For specifics, here’s the block:

always @(posedge clk25)

begin

if(hCount > 10'd143 && hCount < 10'd783 && vCount > 10'd34 && vCount < 10'd514)

bright <= 1;

else

bright <= 0;

end

The values that are being compared to the hCount and vCount are the starting and end points of the active area after accounting for the blanking interval components. For example, (144, 35) are the first coordinates the module goes through that is seen as bright. First we must remember that the module goes through the pixels from top left to bottom right. This means that for the horizontal, it must go through 96 pixels for the sync width then 48 for the back porch. While the for the vertical, it must go through 2 lines in the sync width and 33 for the back porch. Putting us at hCount = 144 and vCount = 35; them being the actual top left of our screen. The same logic applies to the bottom right.

## Snake\_Game.v

This module is in charge of performing the logic that would determine the snake size and location, the apple location, as well as the current state of the game. There are a total of seven inputs to this module and eleven outputs:

Input:

* Left: From the left button on Nexys A7
* Right: Left: From the right button on Nexys A7
* Up: From the up button on Nexys A7
* Down: Up: From the down button on Nexys A7
* Ack: From the center button on Nexys A7
* Reset: From Sw0 on the Nexys A7
* Clk: From the clock divider in the top module for the game clock

Outputs:

* Q\_init, Q\_move, Q\_check, Q\_hold, Q\_eat, Q\_win, Q\_lose, Q\_unkn: The various states of the game (one-hot encoded, so only one will be ‘1’ at any time)
* Apple: The apple location (8-bit for 256 possible locations)
* Size: The current size of the snake in the game (4-bit for up to size 16)
* Locations\_Flat: The current locations of the 16 snake parts (128-bit for up to 16 snake parts each with 8-bit locations, or 256 possible locations)

There are also 5 main registers which will be used, and one array of 8-bit registers:

Registers:

* Random\_loc: Register of next random location of the apple after it is eaten (8-bit for 256 possible locations)
* Apple: Register of current location of apple (8-bit for 256 possible locations)
* Size: Register of current size of snake (4-bit for up to size 16)
* state: Register of current state of game (8-bit for 8 different one-hot encoded states)
* Next\_dir: Register of next snake direction (2-bit for 4 different directions for the snake)
* locations: Array of registers for each of the 16 snake parts (16 registers of 8-bit for the 16 parts of the snake and 256 possible locations for each)

The Snake\_Game module is best explained from the beginning of a game to the end, so I will explain it in this order. There is a case statement in the module which contains the logic performed at any of the 8 states of the game (INIT, MOVE, CHECK, HOLD, EAT, WIN, LOSE, and UNKN). When BtnC is pressed, the Ack input goes high, and the game is first sent to the INIT state. The logic performed at this state sets the starting location of the first two parts of the snake, sets the snake size to ‘1’ (which is one less than it should be and will be fixed shortly), and changes the state to EAT. The EAT state will then increment the snake size (to the correct starting value of ‘2’ when the game starts), sets the apple to a random valid location in the game, checks if the snake hit a size of 15 (which it would not be near in the beginning of the game) which would change the state to WIN, but otherwise changes the state to MOVE. This MOVE state will then make the snake move in whatever direction was last in the Next\_dir register, which holds one of the Left, Right, Up, and Down latest inputs. After the snake movement is performed, the state changes to CHECK. This CHECK state checks if an apple was eaten (which will send the state to EAT), if the snake went out of bounds (which would send the state to LOSE), or if the snake ate itself (which would also send the state to LOSE). If none of those occurred, then the state will be sent to HOLD. The HOLD state simply sends the state back to MOVE, for this MOVE->CHECK logic to loop. When the snake head location (locations[0] 8-bit register) overlaps with the apple location (Apple 8-bit register), then the state changes to EAT, which as previously mentioned, would increment the snake size, set the apple to a random valid location in the game, check if the snake size is 15 (resulting in a win), but otherwise changing the state to MOVE for the MOVE-CHECK loop. If the size of the snake was 15 when in the EAT state, then the state of the game would change to WIN, resetting the game. It is important to note that at every occurrence of a state or state change, the one-hot encoded value of the states, the apple location, the snake size, and the locations of the snake parts are being sent out the module via their corresponding outputs listed above, to the top module. The data which is being output will then be used by other modules to create the visuals that would correspond to the logic that occurs in this module.

## Snake\_Controller.v

It’s important to point out that the Snake Controller’s purpose in the grand scheme of things is to control the color output for the snake game. By using the various states the game cycles through, the location data of the snake and apple, and the “hcount” and “vcount”, the Snake Controller outputs rgb data to provide the visual users will see on the screen. So despite having various inputs, only one 12 bit rgb data is outputted.

First and foremost there are the inputs for the module. These include:

* Clk: synchronizes and runs the entire module
* Bright: A flag to indicate addressable video time
* Q\_init: The flag to indicate the “Initializing” stage of the game
* Q\_win: Flag to indicate the “Win” state of the game
* Q\_lose: Flag to indicate the “Lose” state of the game
* Q\_check: Flag to indicate when the game is checking if the users won, lost, or scored a point
* hcount/vcount: The horizontal/vertical counters that indicate which pixel is currently active
* Apple: The 8bit data containing the location of the apple in the 16x16 grid
* Size: The size of the snake
* Locations\_Flat: The compressed data for the location of each individual part of the snake

For variables that we use throughout the project these include:

* Snakefill: A flag used to indicate if the current rendered pixel is part of the snake
* Applefill: A flag used to indicate is the current rendered pixel is part of the apple
* Borderedfill: A flag used to indicate if the current rendered pixel is part of the border
* Locations: A 16 variable array for the decompressed location data for each part of the snake
* s\_xpos/s\_ypos: data for the center pixel of snake parts after being scaled and offset from a 16x16 logical assumption to a display
* a\_xpos/a\_ypos: data for the center of the apple after being scaled and offset

With every piece of data and variable established, we can move forward onto the contents of the snake controller. First, we decompress the data from the 128bit Location\_Flat input by allocating the bits evenly into the 16 8bit location variables. We then have to establish the center positions of the snake. Under the assumption that the top left corner of the screen is at hcount 144 (horizontal back porch + horizontal sync pulse) and vcount 35 (veritcal back porch + vertical sync pulse), we have to scale and offset location data from the location data of the snake. For the x position of the snake, we convert this through the equation:

s\_xpos[i] <= (locations[i] % 16)\*40 + 144 + 15

With the %16 to find the exact column, then scaling it by multiplying the result by 40, and finally offsetting it by adding 144 & 15 to the product. For the y position of the snake, the center is calculated very similarly through the equation:

s\_ypos[i] <= (locations[i] / 16)\*30 + 35 + 15

With the /16 to determine the row, followed by \*30 to scale it, and finally adding 35+15 to offset the center. The x and y positions of the apple is calculated the exact same way, but using the variables a\_xpos and a\_ypos and the “Apple” input.

With the centers of the various pieces of the game known, the space of each piece can be established. For the snake parts, this is established through the line:

assign snake\_fill0 = (Size >= 1) ? (vCount>=(s\_ypos[0]-10) && vCount<=(s\_ypos[0]+10) && hCount>=(s\_xpos[0]-10) && hCount<=(s\_xpos[0]+10)) : 0  
First the current size of the snake is checked to see if that part of the snake even exists. If it does, then we can logically establish the space of that part of the snake. Because the pieces of the snake are going to be 20x20 squares, we add and subtract the x positions and y positions to make this square. And if the hcount and vcount is within this created space, the snake\_fill flag is triggered.

The apple’s space is created the exact same way, but there is only one flag because there is only ever one apple at a time. However the outer edge of the 16x16 grid, or the game’s border, is manually set because the border never changes so it is hard coded in the line:

assign border\_fill = (((hCount >= 10'd143) && (hCount < 10'd164) || (hCount >= 10'd764) && (hCount < 10'd784)) || ((vCount >= 10'd35) && (vCount < 10'd55) || (vCount >= 10'd495) && (vCount < 10'd516)));

After all the spaces of the various pieces of the game are established, the module can finally determine when to output which rgb value and this done by monitoring the hcount and vcount values. First, whether or not the current counter values are in the addressable video time. If not, the color black is outputted. Otherwise, If the counter values are in the snake's space, an rgb value coinciding with the color green is outputted. Else if the counter values are in the apple’s space, the color red is outputted. Else if the counter values are in the border space, the color black is outputted. And if none of these, the rgb output coincides with the current set background color.

Lastly, we check the inputted states of the games to determine the color of the background. If the game is reset or initialized, the background is going to be white. If we win or lose, the background will be blue and yellow respectively. If none of these, we want to keep the background white.

To summarize, the Snake\_Controller is responsible for controlling which rgb value is being outputted. This is done by comparing the current hcount and vcount values with the established spaces of each piece of the game and outputting a different color depending on which part of the screen is currently being rendered.

## TOP.v

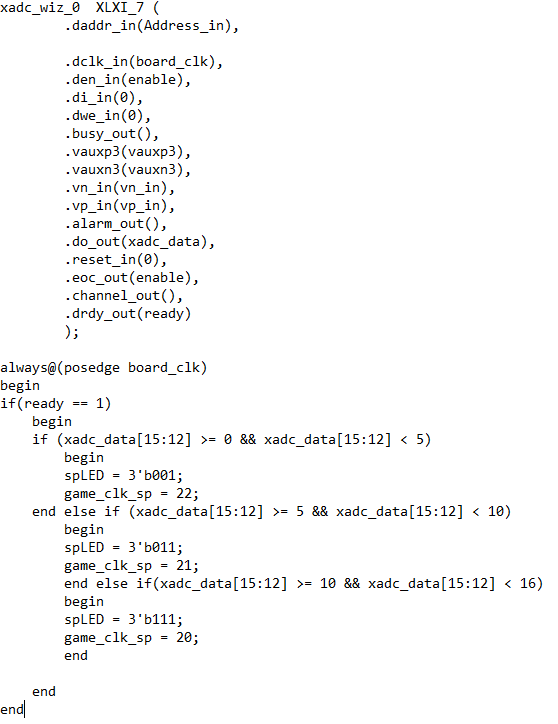
Our TOP.v file is the top module of our project, or the module that glues all of our game components together. The module also showcases various aspects of digital design, including clock management, input handling, VGA output, and game logic implementation. Without ‘TOP.v’, the other modules will not be able to interact with one another and have their features synthesize.

The top module includes:

* The clock port (ClkPort)
* Button inputs (BtnL, BtnR, BtnU, BtnD, BtnC)
* A switch input (Sw0)
* Analog inputs (vauxp3, vauxn3, vp\_in, vn\_in)
* Seven segment display LED outputs as well as their controls
* A 3-bit output (SpLED)
* And the VGA signals (hSync, vSync, vgaR, vgaG, vgaB).

The TOP module includes functionality for displaying the game on a VGA monitor, reading user inputs, and adjusting game speed based on analog inputs. It includes a variety of outputs for visual feedback, including LEDs and a seven-segment display to ensure this. The design uses clock division for timing control and interfaces with a snake game controller and a display controller to manage the game logic and display outputs.

The TOP module includes an XADC configuration, which is presented below. The XADC configuration allows for analog-to-digital conversion, which interfaces with the various analog inputs and outputs that are being used. For example, during our testing, we were able to change the speed of the snake using a potentiometer that we connected to the Nexys-A7 on a breadboard. This is possible with the XADC portion of this top module.



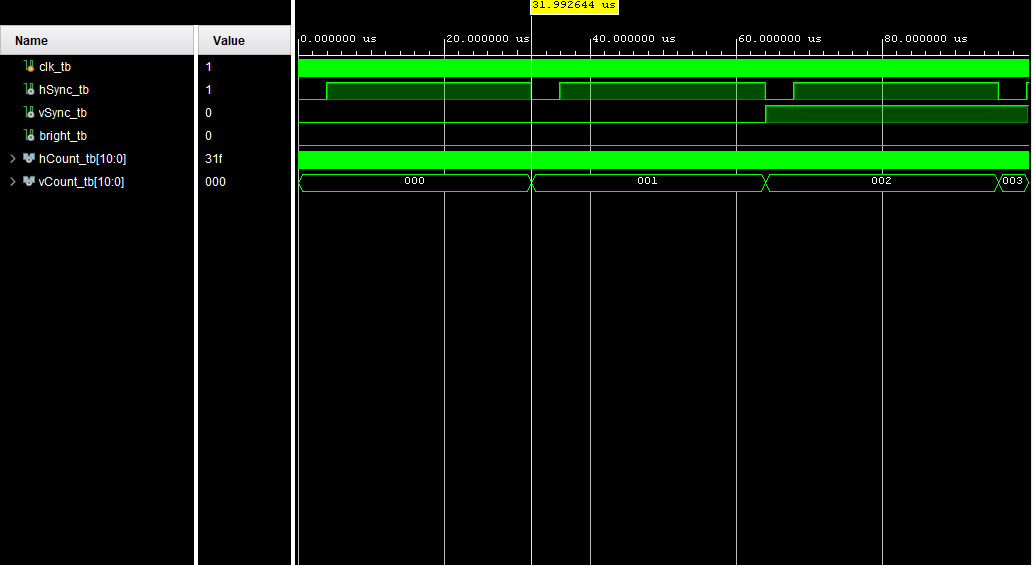
This module also includes the Snake\_Game core, which handles the logic of the game. The game takes inputs from the buttons and producing outputs that indicate the game's various states (Q\_init, Q\_move, Q\_check, Q\_hold, Q\_eat, Q\_win, Q\_lose, Q\_unkn), the location of the apple, the size of the snake, and the flat representation of the snake's locations.

## Conclusion:

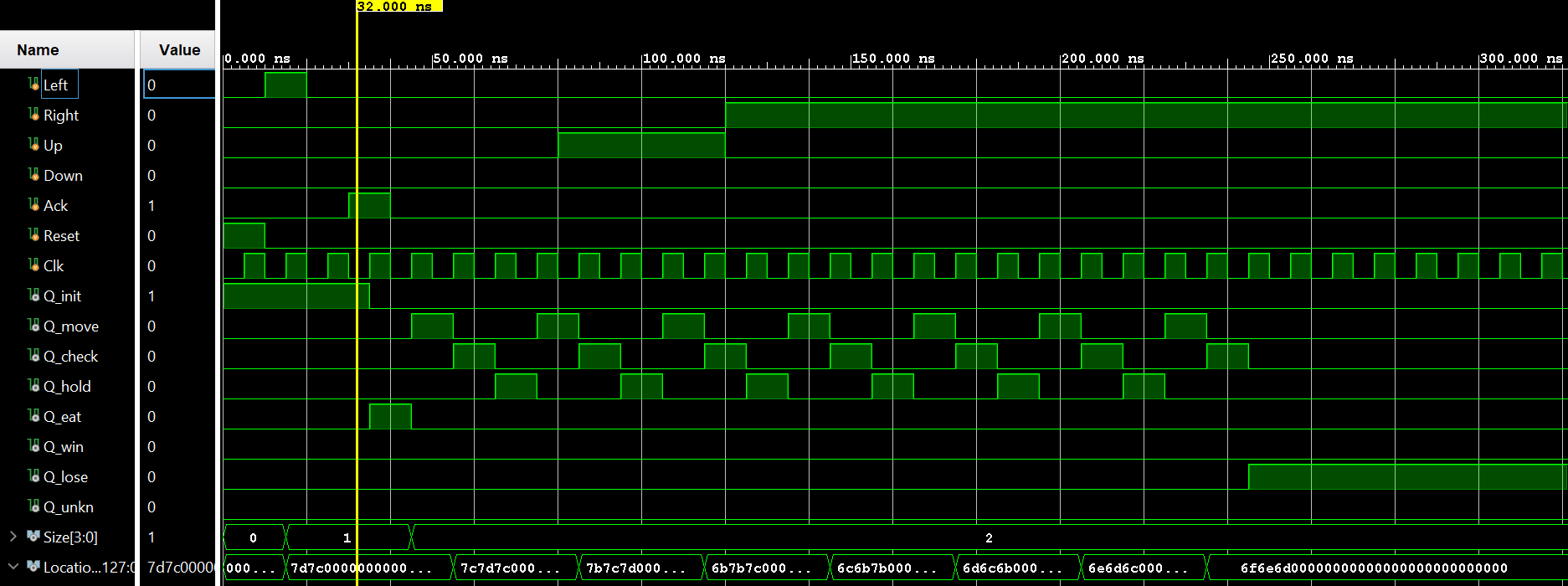
In conclusion, our group project has successfully explored the integration of hardware-based gaming using a Field-Programmable Gate Array (FPGA) platform. We incorporated interactive elements like VGA for display, XADC for difficulty adjustment, 7-segment displays for score tracking, LEDs for visual feedback, and user input through switches and buttons. The Display\_Controller module efficiently handled VGA timing signals, ensuring a real-time immersive gaming experience. Snake\_Game managed the game logic, determining snake size and location, apple location, and overall game state. Snake\_Controller played a crucial role in controlling color output, creating a visually engaging experience on the screen. The TOP module served as the glue, integrating clock management, user inputs, VGA output, and game logic as well as incorporating the XADC wizard to add a game difficulty aspect. Our project successfully showcases digital design principles, highlighting the capabilities of FPGA in hardware-based gaming.

## Addendum:

Display\_Controller Testbench;



Snake\_Game Testbench:



Snake\_Controller Testbench:

