CS 152A Lab 4: Creative Lab

Dodgebomb

Gee Won (Jennifer) Jo, Andrew Lin, Daxuan (Albert) Shu
704171725, 204455444, 204853061
CS 152A Lab 4
June 10th, 2017

Introduction

For our Lab 4 project, we designed a real-time game displayed on the VGA monitor. As shown in Figures 1 and 2 below, the game involves a player (the white square) dodging many bombs (the red squares) that are falling from the top of a blue screen. The player moves left and right to dodge the bombs using the buttons on the FPGA board. If the bombs touch the player, the game ends, the screen goes black, and everything resets after about five seconds.

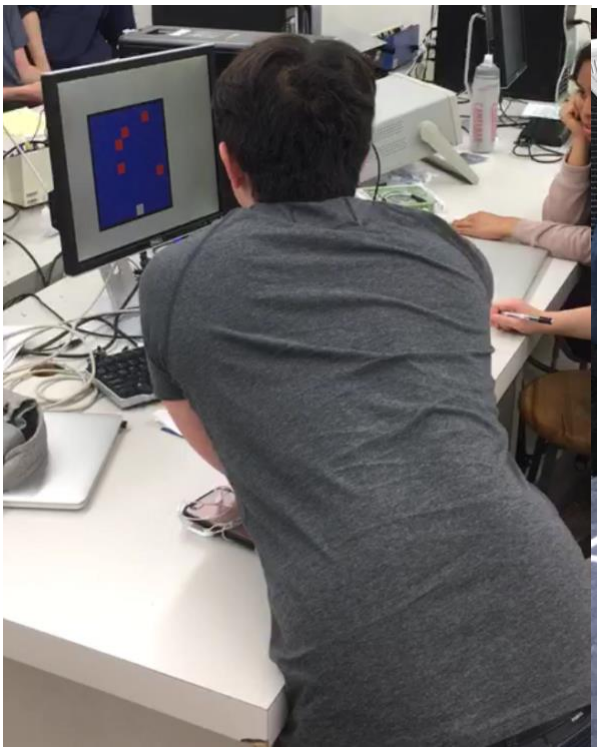

Figure 1

Figure 2

Dodgebomb Gameplay

Dodgebomb is a real-time game where the player, represented by the white square, has to dodge bombs, the red squares, falling from the top of the game field.

Design Description

Our game field is the blue square in Figures 1 and 2, and it is divided into 10 columns with a width of 30 pixels and a height of 300 pixels. Each bomb and the player take up an entire column's width, and they are all thirty pixels tall. The player is generated at the bottom of a column and the bombs are generated at the top. On each tick of the game's clock, the bombs move down by five pixels and the player can move one column over if they press a button.

The two main modules that were used to implement our game were a pseudo-random number generator and a game-object manager. Our VGA display module is based off of the sample code provided for this lab, with changes to correctly display our game.

Random generator module:

Verilog does not have a native, synthesizable random number generator. As such, we implemented a pseudo-random generator module to randomly generate bombs. We use the random generated number as the index of the column to tell the system in which column should

the new bombs be generated. Shown as figure 3 below, the random generator is made of a few D flip-flops and Xor gates. The three D flip-flop can provide at most $2^3 - 1$= 7 different states. In order to make sure there is no repeated state, the value of the seed must meet certain conditions.

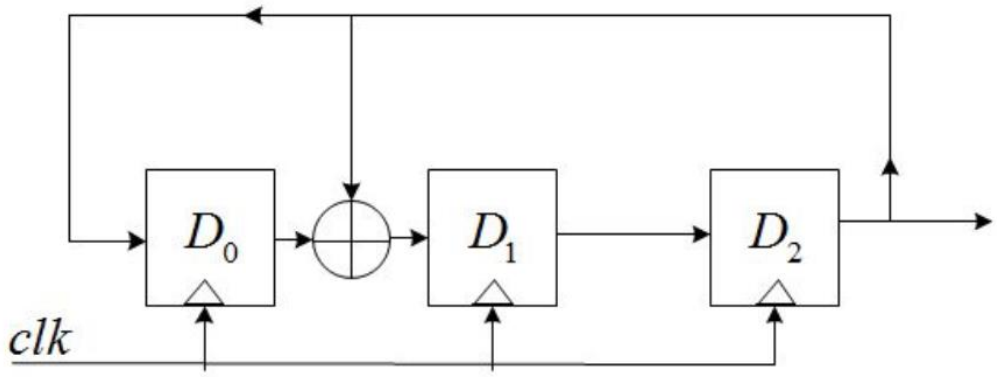

Figure 3. Schematic for random generator

By our testing, we use $D_2D_1D_0$= 3'b111 as our seed. In this case, we have 7 different states.

At the beginning, when the clk goes in,

$D_2 = D_{1out}$= 1;

$D_1 = D_{0out}\ xor\ D_{2out} = 0$;

D_0= D_{2out} = 1. Thus $D_2D_1D_0$= 3'b101 after the first clk goes in.

Similarly, $D_2D_1D_0$ = 3'b001 after the next clk goes in.

… .etc. The state-shifting is shown as figure 4 below.

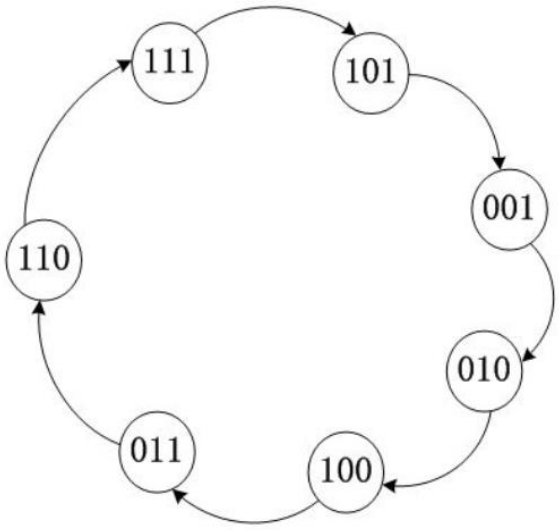

Figure 4. State-shifting graph

This graph explains why there are no bombs generated at the left most 3 columns. Since we only have 7 states and 10 columns. The value can never be 8, 9 and 10. Thus, the column

8, 9 and 10 can never have a chance to generate bombs. We did try to fix this problem by adding one more D flip-flop and fix the xor gates. However, in this case, there are total $2^4 - 1$= 15 states and we have to convert all values greater than 10 to some value less than 10. We tried two methods to do so: dividing all value greater than 10 by 2 and also using the modulus operator to get the remainder of all values greater than 10 when divided by 10. While these methods both cast values greater than 10 to values less than 10, and thus usable in our game, they all heavily skewed the results of the random number generator.

In the case of dividing greater values by 2, 10 was cast to 5, 11 and 12 were cast to 6,13 and 14 were cast to 7, and 15 was casted to 8. This caused the bombs to be generated much more often on the left side of the game field.

In the case of using the modulus operator, 10 through 16 were casted to 0 through 6, which meant that the bombs were generated much more often on the right side of the game field.

Additionally, we did not calculate an optimal suitable seed in time for the 4 flip-flop case. As such, we returned to our original method of using three flip-flops and only generating 8 columns of bombs.

If we had more time, some fixes we could have attempted would be to use even more registers, such that after applying the modulus, we could have a more even spread of values. For example, if we used 10 registers, then we would get numbers from 0 to 1023. After applying the modulus, the values 0 through 3 would still have a higher probability of occurring, but the relative frequencies would still be roughly the same (0 through 3 would have a probability of 103/1023, but 4 through 9 would have a probability of 102/1023).

Game-Object Manager:

Our Game-Object manager had to keep track of each object, which includes the bombs and the player, and keeps track of both their locations and their borders for movement, collision and display purposes. The Game-Object manager works very close to the VGA display module, so we decided to implement the logic for it in the same module.

Since our player is always at the very bottom of the field, it is a lot easier to manage the player. We keep two set parameters for the top of the player and the bottom of the player, and then have two variables for its left and right. When buttons are pressed, the player's left and right variables are moved thirty pixels (one square of the game field) in the appropriate direction.

The bombs' implementation was much more complex. The bombs needed to be randomly spawned, could potentially be in a not-spawned state, and each time they respawned they could change columns. Additionally, the manager needed to keep track of each bomb's location, and also move the bomb's down the screen on each tick.

In order to represent the bomb's, we gave each bomb a coordinate. Since Verilog does not support 2D-arrays, we instead just used two arrays, one to store each bomb's horizontal coordinate, and one for each bomb's vertical coordinate. We defined these coordinates to be the bomb's bottom left corner. Since the bombs' dimensions are preset (30 pixels by 30 pixels), we can calculate bomb collisions and display the bomb based just off of that coordinate. This way, when we move the bomb down the screen, all we have to do is move this one coordinate.

Additionally, since Verilog does not support dynamic variables, our arrays have a length of 12, which means that the highest number of bombs on the screen at one time is 12, which we deemed a suitable number. The bombs are generated using the output of the random number generator, as discussed above. A bomb that has yet to be spawned has its coordinates set to (0,0), which we set to be a not-spawned state. The bombs take the output from the random number generator, and spawns at the very top of that column.

One each tick of the game, the bombs are moved down the screen, and collisions are checked for each bomb. This is done by using the coordinate. We know that the bomb's right edge is thirty more than their horizontal coordinate (Verilog's VGA values place 0,0 at the top left corner) and since the player and bombs can only move and exist in columns, we can check for collisions by checking if the bombs' left and right edges match the player's, and the bomb's bottom is at a greater value than the player's top. Additionally, we check if each bomb has reached the bottom of the screen, in which case it is despawned by setting its coordinates back to (0,0).

Simulation Documentation

For the most part, this project was tested and simulated using the VGA output. We started out by simply displaying static (non-moving) players, bombs, and borders. Then, we incrementally expanded our game's functionality. For the player, we started out by having it automatically move across the screen in loops, and then hooked up the movement to button presses. For the bombs, we checked to see if we could control where the bombs were spawned, that we could pseudo-randomly spawn the bombs, that we could move the bombs down the screen, that we could despawn the bombs that reach the bottom of the screen, and finally that we could respawn bombs at the top of the screen.

In this part, some of our decisions included the sizing of our game field. We decided that using the entire monitor for our game would be distracting, and that it would make it harder for players to focus on where each bomb is dropping. We also found that smaller fields would be too small to easily see where the bombs were, and were harder to find a good, balanced number of bombs. If too many bombs were placed on a smaller field, the game got extremely difficult, whereas if there were not enough bombs, the game would become too easy, and thus also boring.

In the end, we decided that a 300x300 pixel squared field with ten columns was the best balance between being aesthetically pleasing and being just moderately difficult.

One problem that went unsolved for us throughout our testing was handling game-object movements. For some reason, the objects would only display movement correctly when they moved on a frequency of 1 Hz. When we increased the clock cycle to 100 Hz to 1 MHz and to even the board's native clock cycling, the movement became unresponsive, or too responsive. At higher frequencies, the movement would not obey our boundary checks, and the player and bombs could potentially jump right off of the screen. Our original solution was to have the bombs move at the working 1 Hz, but have the player move based on the posedge of the signals from the buttons. Despite having a debouncer and various other changes that we implemented, this still did not work. For some reason, when this happened, our player would

instead turn into a white bar extending the width of the entire game-field. In the end, we opted to keep the 1Hz movement frequency. The same occurred when we replaced

Conclusion

While we did run into problems with the random number generator and the game object movements as described above, we were still able to implement a playable game and a fairly complex game manager that was able to handle interactions between many game objects, as well as the player, all in real time and on a fairly limited board. All in all, this was a very rewarding experience, as we quickly learned a lot about controlling a VGA monitor using our board and also saw our game evolve in front of our eyes into our final product. Additionally, it also helped us understand just how complex commercial game engines such as Unity or Unreal really are.

Our group members worked together on all parts of the project, helping each other work on different parts of the module and brainstorming together to solve the bugs and difficulties we encountered.