Space Monsters

Haomei Liu, EE
Denny Shen, CECS

EE 354
2020 Fall

Abstract

Space Monsters is a simple game developed using Verilog with Digilent Nexys4 FPGA board as the target device. The game is inspired by the classic arcade game "Space Invaders." It involves a tank which can only move at the bottom of the screen, and monsters that attack the tank with bullets. Users can interact with the game through buttons on the FPGA board and a monitor using VGA connection.

Introduction and Background

We referenced Lab 5 for SSDs display, including the conversion of hex to SSDs and the use of ssdscan_clk to power only one anode at one time. We also used contents from Lab 6a to process button press from users to clean signals that can be used directly in the state machine. Compared to the given lab file, we do not need multi-step pulses, so we simplified the state machine and removed two extra outputs, only using DPB and SCEN as module's output. The VGA demo file is also a great help for us. The core of our design, the moving tank, is very similar to the moving block demo provided in the VGA lab. We modified block_controller.v for custom graphic displays and other objects and display_controller.v for vga control, and we implemented a revised version of vga_top.v as the top file that links the board and the state machine.

The Design

a. Design Description

The objective of our design is to provide a functional game with the purpose of entertaining. To smoothly interact with users, buttons are debounced and some trigger a single step to achieve a good user experience. We also put considerable effort into designing the user interface to make the game more enjoyable.

Main functions of our design involve the following: a) A tank with its position controlled by user inputs (button press), b) Monsters that can attack the tank with bullets and can be destroyed to win points, c) Graphic display that is straightforward and entertaining. Correspondingly, main objects in our design include a tank, monsters (3 in level 1, 5 in level 2), and bullets. The tank has 3 bullets at one time, so it cannot fire when no bullet is available. The bullets are restored to the tank if it hits a monster (score+1) or reaches the edge of the screen. Monsters also have 3 bullets each, firing in a pseudorandom frequency. Once the user takes out all monsters, they enter the next level, and win if they again take out all monsters in level 2. If the tank is hit by a monster bullet at any time, the game is over.

We also make sure the design is easy to build upon. Different objects are separated to different blocks of codes, and modification can be achieved by simply adding more blocks of the same object. For instance, if we want to design the game with more levels in the future, we only need to add a new state in the state machine, expand the size of registers, and produce more monster blocks to represent a larger number of monsters.

b. State Machine

To make our design more readable, we seperate codes to four files: debounce.v to process button press, space_monsters_sm.v as the overall state machine, block_controller.v that manages tank, monster, and bullets' behavior and display, display_controller.v that uses 25MHz clock to control vga, and vga_top.v as the top file.

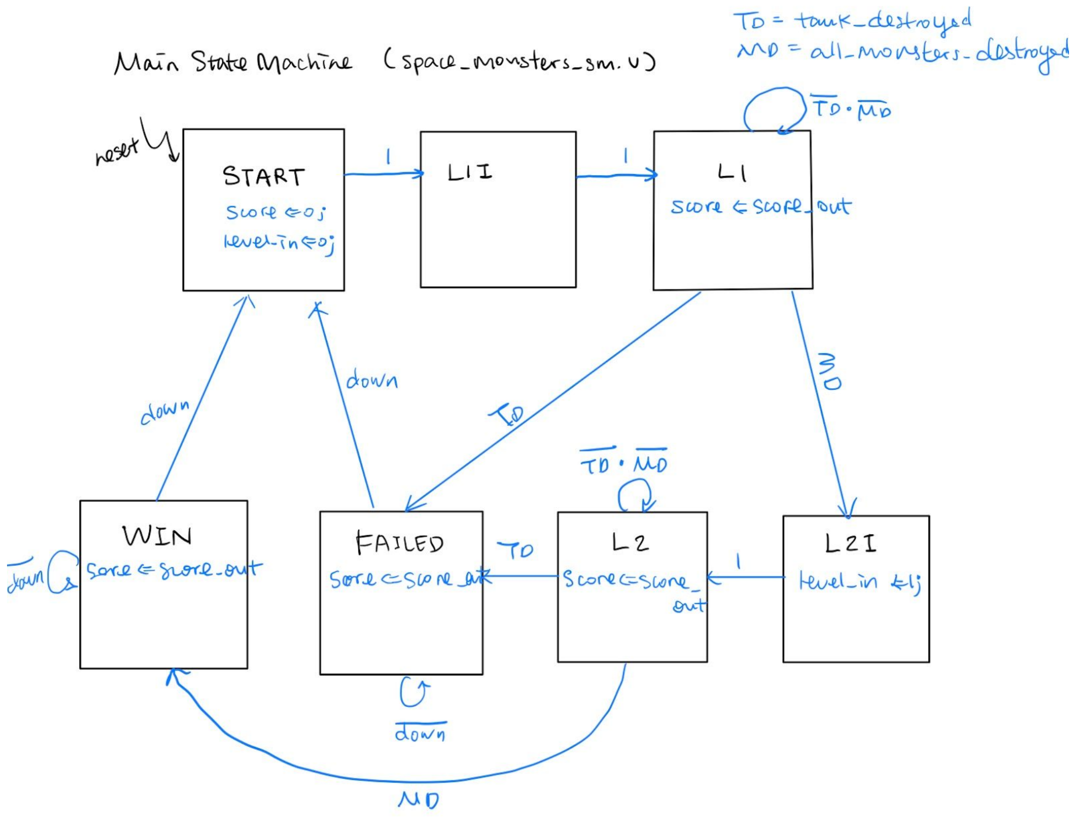


Figure 1: Main State Machine

The Main State Machine controls the general flow of the game. It has 7 states: START, L1I (Level 1 initiation), L1, L2I (Level 2 initiation), L2, FAILED, and WIN. State transitions, being relatively simple, are indicated with arrows. Two initiation states are added for clock considerations. Apart from user inputs, the control signals (tank_detroyed and monster_destroyed) come from the block_controller module. This separation makes interaction with block_controller and adding additional levels an easy task.

The detailed logic that controls movement and status of tank, monsters, and bullets are all in the block_controller module. Each object's logic is simple, but requires lots of repetition as objects in verilog cannot be dynamically allocated. Therefore, rather than using states, we choose the simpler if else statements. The module receives a trigger from the main state machine to start a round, and also level_in signal to decide the number of monsters to display on the screen. We use many registers as flags to represent the status of an object in a round. Score is incremented when any monster is killed and sent back to the main state machine, then to the top file for SSD output.

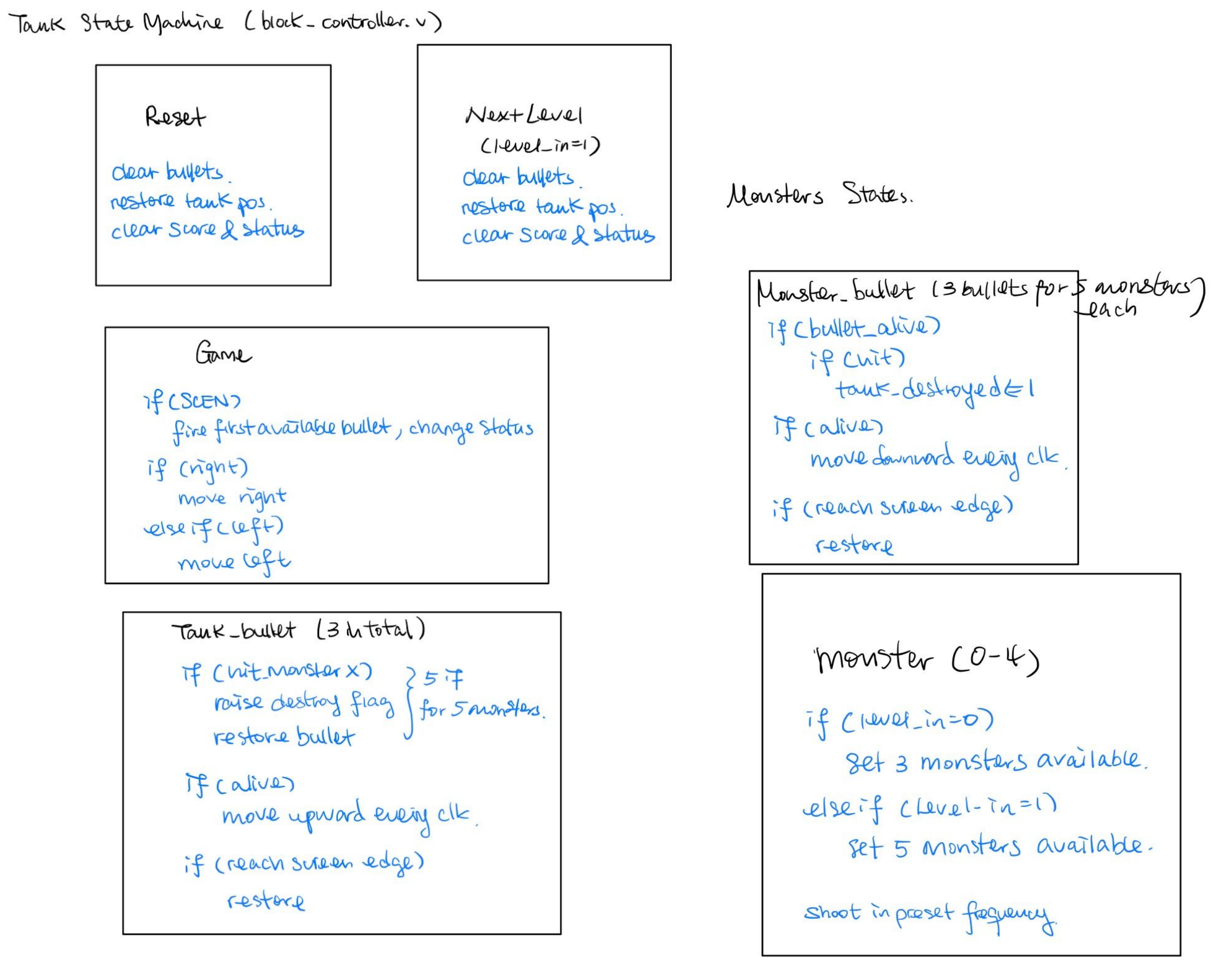


Figure 2 and 3: Tank States and Monster States

c. User Interface

The significant inputs and outputs of our design include buttons, SSDs, and VGA.

All five buttons are used in the design: left and right to move the tank horizontally, up to fire bullets, the middle one for reset, and down to start a new round after entering WIN state or FAILED state. Buttons are debounced with a separate module debounce.v, and up button is additionally processed so that each press only generates one single step pulse. By referencing lab 6a's files, we wrote these codes in a separate file so that it can be called multiple times, in both main state machine and display.

```
module debounce(rst, clk, btn_press, clean, single);
```

For SSDs display, we record the score users get during one round of the game, one monster killed corresponding to one point. This part of codes references seven segment display verilog files from lab 5.

We use hard-coded positions to display images on VGA. We have a tank with iron armor and monsters with different colors of eyes. A simple letter of W or L is displayed for winning or losing the game.

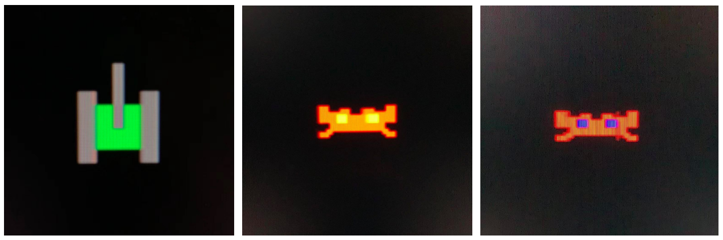
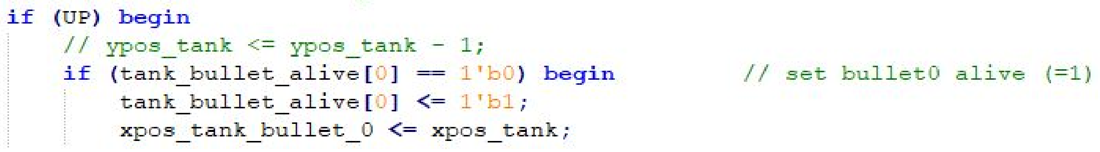


Figure 4: Design of our tank and two example monsters.

Test Methodology

Testing was challenging in this case since the error messages were usually ambiguous and recompiling the verilog was time consuming. The error message could give us a big picture of where the code went wrong, and actually acted as a learning tool. By googling and researching online, it was not too hard to find a suitable solution. Compiling the verilog could take a long time as it has to go through simulation, synthesis, implementation, and then generation of bitstream.

However, the above two aspects were not the most challenging part of the project. Usually, in a C-like program, the programmer is able to run debugger or simply utilize print statements to keep track of the program and check for potential errors along the way, which this project has been having difficulty on. Groping for a better method for testing and debugging, one method being we managed to take good use of vga display. The project is based on LAB9, the vga_moving_block, which has a completed vga interaction. Built upon that, we were able to test if our project had entered a specific block of code by moving a block by one pixel on the vga display. For example, in our case, we wanted to check if the user tank was shooting bullets continuously, we added a line right after the bullet shoot that would move the tank upward by exactly one pixel (see Figure 5). We found that the tank was only shooting one bullet, but moved continuously upward as long as we held onto the button UP. We realized that it was the problem of the button debouncing that the user tank exhausted all of its bullets in one shot due to the fast clock and slow human button press which triggered multiple shooting processes at the same time.



```
if (UP) begin
    // ypos_tank <= ypos_tank - 1;
    if (tank_bullet_alive[0] == 1'b0) begin          // set bullet0 alive (=1)
        tank_bullet_alive[0] <= 1'b1;
        xpos_tank_bullet_0 <= xpos_tank;
```

Figure 5: Testing Script

Another method we used for testing is by testing one object of its class at a time. There will always be multiple objects with the same class in a game project. It is time efficient to only implement one object at a time and test with it until it is completely functional, so that the rest of the objects that take the same class would only take a minor fix after the copy-paste. Although copy-pasting code is never a good practice, writing every single object by hand without it is likely going to take longer, verilog hdl coding does require a lot of repetitive blocks of code for each object, and thus, we decided to take the risk and were exceptionally careful when doing copy-pasting. For example, when we are implementing bullets for monsters, there were 15 bullets in total with 3 bullets for each of the 5 monsters. We implemented 3 of the bullets for monster_0 and tested with it until working, and then copy-pasted the results for the result 12 bullets with a minor change in their naming indices.

Last but not least, math matters when trying to draw and display objects via vga in detail. We would first take time to calculate the position of each pixel for the objects (see Figure 6), before testing with them as compiling does take a long time, and careful math would help us to get things in the right positions with the least amount of trails.

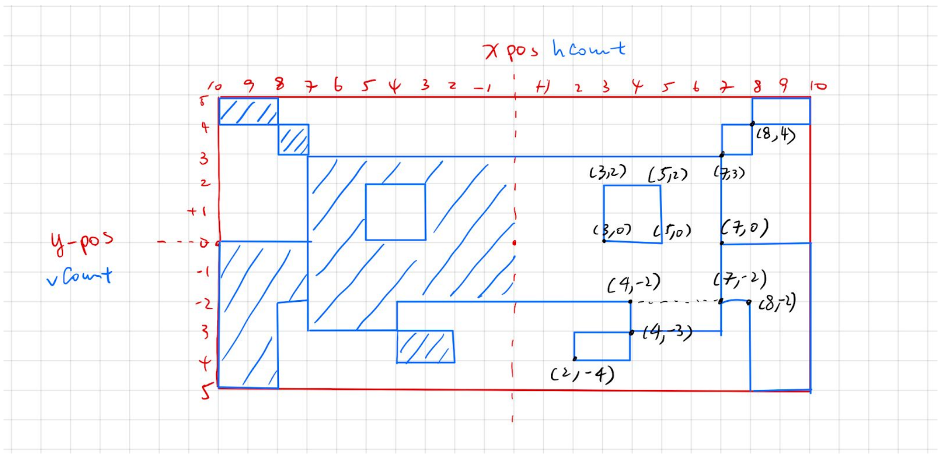


Figure 6: Monster Design

The most challenging part of the project remains to be the error debugging without a debugger or even a print statement when the error is not reported in Xilinx. However, there might be a functionality in Xilinx of mimicking the print statement which would help debugging that we did not discover, but the overall programming experience and the testing method we utilized were acceptable and sufficient for this project.

Conclusion and Future Work

The main game structure and most functionalities in the initial proposal were successfully implemented, as the game turned out to be a two-round score-recorded Space Invader type of game with monsters of different shooting frequencies, press-to-fire feature, and decent graphics.

Some adjustments made on the bullet-count limited to 3 per object since we later discovered that every object needs to be declared at the start of the project, and cannot be allocated during the game dynamically. We also changed the plan and made the monsters stay at a fixed position throughout the game instead of appearing in an emerging style.

Due to the time limitation, we could only go as far as we could. Additional features we would like to add to the game with time permission include switching between bullet-mode of press-to-fire and hold-to-fire, more bullet-count and more rounds, pause, and moving monsters. Some of these features would improve the user experience while the other would allow the game to have a progressive difficulty. Below are some of the strategies we originally planned to implement these additional features. Bullet-mode could be implemented with a switch button that will activate or deactivate button debouncing and time-elapsed firing. More bullet-count and more rounds could simply be achieved with more copy-pasting with little fixes. Moving monsters could be realized by giving each monster a walking pattern attached to their xy-coordinates. Pause might be possible to implement with a pause-attribute attached to every object that would stop their movement when triggered.

The amount of functionalities we had implemented for this project were satisfactory considering the time limitation, and it was a great learning experience and practice in all.