Dungeon Maze: A FPGA-based Maze Game

Tarik Wilkinson and Kyle Manke
Computer Engineering and Computer Science
EE 354
Fall 2020

1. Abstract

For this project we wanted to create a fun maze game with replay value. Our idea for this maze is based on a combination of the simple computer maze games you see on the internet. All good games have a compelling story behind it so we created a story for the game about the player needing to escape this dungeon after collecting all five magic golden coins.

2. Introduction

All in all, the implementation of our maze game drew on several prior experiences and original ideas. Firstly, a majority of the VGA code came from the Moving Block demo. Additionally, we referred to several of our past labs for help with the clock divider and seven-segment displays. Finally, the homeworks helped to give inspiration for the practice behind state machine design, but the overall state machine was novelly designed.

3. The Design

I. An Overall Description of the Implementation

Simply put, the objective of our design was to create a fun dungeon-like game. Due to our technical abilities, we were not looking to create the next Doom, rather, we had a few small goals that we wanted to accomplish: displaying a maze and character, taking in user input, and implementing an effective collectible and collision system. To begin, let's explore the processes behind the VGA output. The VGA output relied greatly on the display_controller module. In this module, all necessary formalities, such as controlling hSync and vSync, were handled. Additionally, this module contained the necessary logic to keep track of hCount and vCount: the current location of the pixel being printed. The values of hCount and vCount were wired to the maze_controller module so that the actual graphics could be displayed. Within the maze_controller module, hCount and vCount were used in conjecture with several other registers to decide what color to print. For instance, by using the maze register and the current values of hCount and vCount, the logic would recognize that a black pixel, to represent a wall, needed to be printed. The maze register, which stores the hard coded maze, consisted of fifteen sixty-bit wide rows. This allowed us to store a fifteen by fifteen maze with only four bits per cell. For each cell, the four bits represented where the walls were present. For example, for a cell with walls on the top and left, the four bit representation would be 1001.

Besides the state machine, as explained in the section below, the rest of our design consisted of several smaller sub-systems. For the collision detection system, a collision occurred when a pixel was slated to be user-colored and wall-colored. This detection occurred during VGA output, so it was under the 25 MHz clock domain, and had to be sent to the state machine, under a slower clock domain. To manage this, a two-way handshake was implemented where the VGA output would flip its collision flag, the state machine would see the collision flag flipped, flip their own collision flag to acknowledge the collision, which would then prompt both systems to then revert their flags back to zero. Additionally, another interesting implementation detail was needed to correctly keep track of the users collision, or death, count. For its implementation, a special counter was used to display the score in decimal. Both digits, the ones and the tens, were 4-bit registered that only ever contained a maximum value of nine.

This allowed us to increment the ones digit to nine and then on the next increment, the tens digit would increment while the ones digit flipped back to zero. Additionally, as part of the gameplay, golden coins were included to act as collectibles and check-points. This means that the user has to collect all five golden coins before they can finish. As a check-point, the location of the last collected coin was stored in two registers and then when the user collided with a wall, their position was set to the check-point registers.

II. An Explanation of the State Machine

Note: All assignments in the state machine for the game use non-blocking assignments.

Description of some of the variable names:

- xcheck and ycheck are the x-position and y-position respectively, of the last checkpoint
- xpos and ypos are the x-position and y-position respectively, of the player
- xcoin_pos and ycoin_pos are the x-position and y-position respectively, of the current golden coin the player is trying to find
- score is the current number of golden coins already found
- death_count_0 controls first digit of death counter
- death_count_1 controls second digit of death counter
- coin_flag is one when all coins are collected
- wall_flag_vga is the vga wall collision flag
- wall_flag_sm is the state machine collision flag

On RESET, set state to initial state (INI) on the next clock.

INI state:

- set the background to RED
- set xcheck, ycheck, xpos, ypos, xcoin_pos, ycoin_pos all to their initial values
- set coin_flag, death_count_0, death_count_1, and score to 0
- always goes to PLAY state on next clock

PLAY state:

```
- if (Right button)
    - increment xpos by 1
- else if (Left button)
    - decrement xpos by 1
- else if (Up button)
    - decrement ypos by 1
- else if (Down button)
    - increment ypos by 1
- If (wall_flag_vga)
    - set xpos to xcheck and set ypos to ycheck
    - set wall_flag_sm to 1
    - increment death_count_0 by 1
    - if (death_count_0 is 9)
        - set death_count_0 to 0
        - increment death_count_1 by 1
        - if (death_count_0 is 9)
                - set death_count_1 to 0
    - if (not wall_flag_vga)
        - set wall_flg_sm to 0
    - if (found coin)
        - set xcheck to xcoin_pos and set ycheck to ycoin_pos
        - increment score by 1
        - if (score <  4)
                - set xcoin_pos to xcoin[score] and set ycoin_pos to ycoin[score]
        - else
                - set coin_flag to 1
    - if (coin_flag and found finish)
        - set state to DONE
DONE state:
    -   set background to GREEN
```

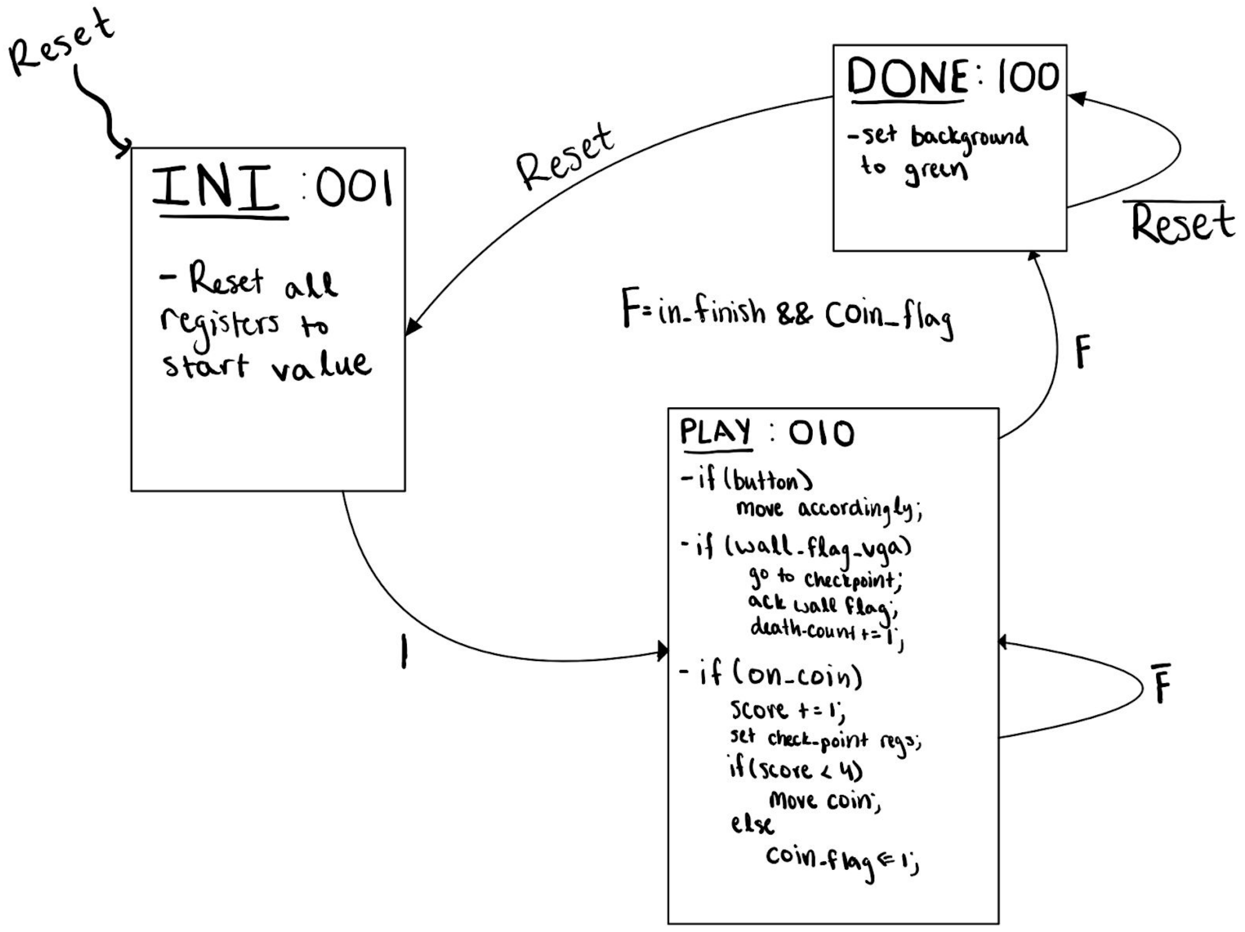
Reset
INI : 001
- Reset all registers to start value
Reset
DONE : 100
-set background to green
$\overline{\text{Reset}}$
F=in_finish && coin_flag
F
1
PLAY : 010
- if (button) move accordingly;
- if (wall_flag_vga) go to checkpoint; ack wall flag; death_count += 1;
- if (on_coin) score += 1; set check_point regs; if (score < 4) move coin; else coin_flag <= 1;
$\overline{F}$

Figure 1: Pseudo-Code State Machine

III. The User Interface

The user interface of our game consisted of three major systems: the seven-segment displays, the buttons, and the VGA. As stated above, the VGA was used as the major display for the game. Using output pins from the top file, the VGA displayed the maze, the user, and the coins. The seven segment displays were used to display two counters. SSDs 3 and 2 were used to display the tens and ones digit of the collision counter. SSD0 was used to display how many of the 5 coins had been collected. All of the other SSDs were disabled. Finally, the buttons were used to control the movement of the user and restart the game. BtnC was used to signify a reset. For the other four buttons, their placement signified which direction they would cause the user to move. For example, BtnL would move the user to the left while BtnU would move the user upwards.

4. Test Methodology

For the most part, testing was done through unit and play testing. Unit testing was conducted throughout the design and implementation process. After each new addition was added, we thoroughly debugged and tested it to ensure that no problems were encountered before continuing. Also, as our project was largely visual, most of the testing was done using the VGA instead of a standard test bench. As the project progressed past just representing the maze, we incorporated play testing into our testing regiment. The play testing consisted of us attempting to play the game while trying all possible cases. For example, once we had implemented the coin logic, we attempted to collect the coins, tried to finish without collecting them all, and collided with the walls multiple times to test the check-point hardware. Overall, our system did not have a ton of edge cases. A main one was what happens if a person collides with the walls more than 99 times. To handle this, we made the counter just loop back to 00, so if you want to excel, just die 100 times! Additionally, in order to handle the edge case of multiple button presses, we gave precedence to the button that was pressed first or, if both happen to be pressed at the exact same time, to whichever came first in the if-else statement.

5. Conclusion and Future Work

While the design for this game is already robust in its implementation and includes some interesting features there are always enhancements or improvements that could be added in the future. One such enhancement could be to add a timer that uses some of the unused SSDs that starts in the PLAY state and counts up until it stops in the DONE state. This way the player not only will want to replay in order to finish with the lowest possible death count, they will also have the incentive to finish with the fastest possible time. Another change could happen to the initial state in order to display a title screen for a few seconds to the VGA before going to the PLAY state. Even though this semester was shortened and taught remotely and the labs in this course were still enjoyable. The labs were always relevant to topics that were taught during the lecture and they provided the students with enough new skills to create some really fun and creative final projects. Overall, this whole course was really great across the board!