ECE 253 EMBEDDED SYSTEMS DESIGN PROJECT REPORT

-Submitted by Barath Kumar Ramaswami and Gokul Prasath Nallasami

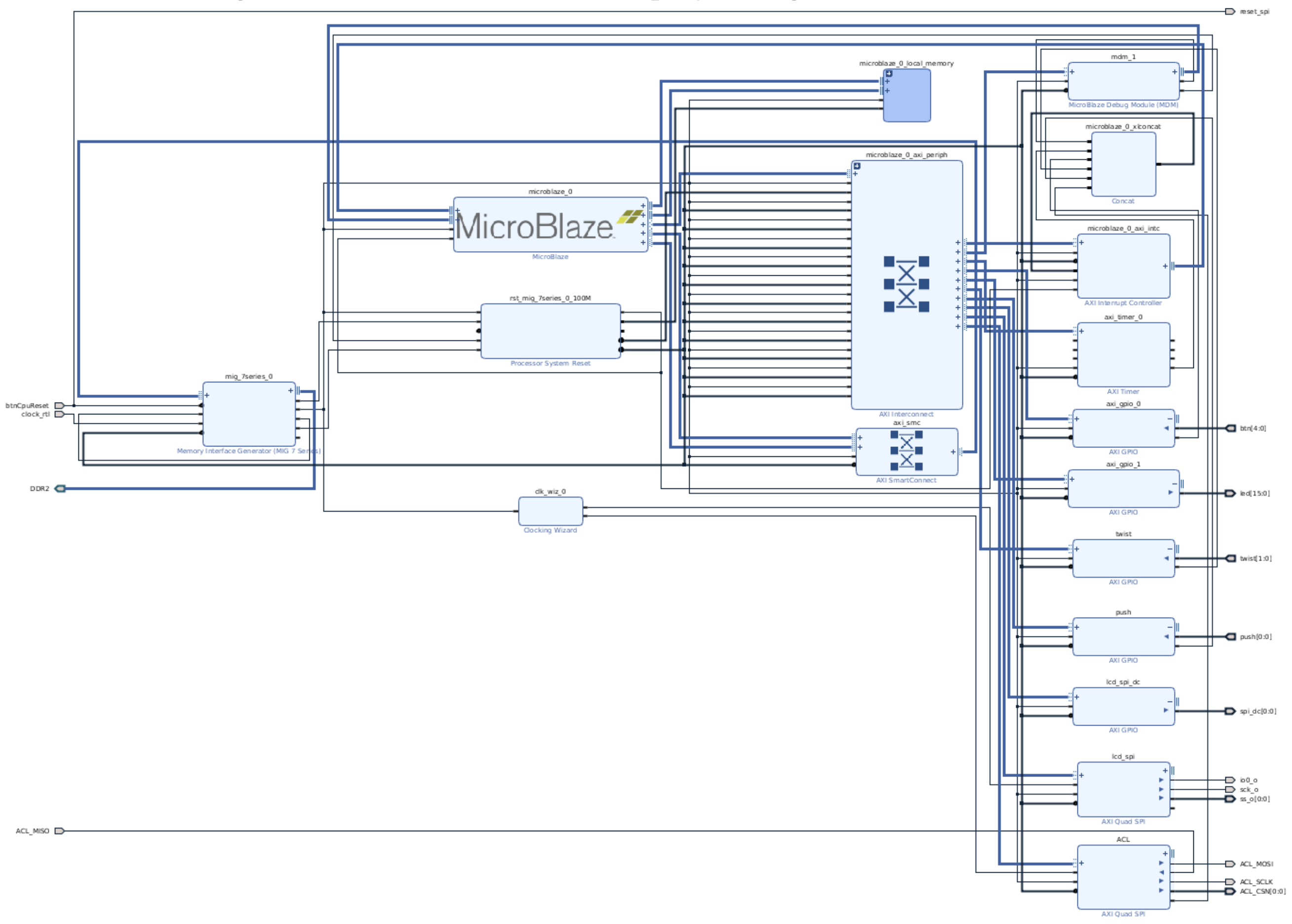
1. OVERVIEW

The basic overview of the whole project can be seen as a game that utilizes the use of an accelerometer to navigate the maze while collecting certain elements to increase the score and avoiding certain spots that can kill you. Game of Balance is an accelerometer based maze navigation game that is built on Nexys 4 DDR development board. The hardware project is built on Xilinx Vivado HLS suite using a variety of IP's. It uses Microblaze soft-processor and communicates using SPI interface with the 2.2" TFT LCD display and also with the on-board 3 axis MEMS accelerometer. The accelerometer is used to control the movement of the player designated ball and it employs a sophisticated collision detection system that enables it to detect walls and distinguish it from objects of other type so that the player cannot pass through walls. In the same way, the collision detection logic is used to differentiate the **"STARS"** and **"HOLES".** Stars and holes are objects in the game logic that determines the playthrough aspects of the game.

In this project, Nexys 4 DDR development board is used. This board boasts an arsenal of features that enables a larger degree of customization and tweaking to the projects in addition to the solid hardware capabilities of the board itself.

1.1 HARDWARE DESIGN

A block diagram of the Vivado Hardware project is given below.



The hardware used for the project is **Nexys 4 DDR development board** under the following configuration. The Microblaze soft-processor is configured for **Maximum performance** and has the high-performance preset enabled for offering the maximum performance needed to run the game logic and the QP Nano state machine in tandem.

The accelerometer used is the on-board **3 axis MEMS accelerometer** called **ADXL362**. The ADXL362 is a 3-axis MEMS accelerometer that consumes less than 2µA at a 100Hz output data rate and 270nA when in motion triggered wake-up mode. Unlike accelerometers that use power duty cycling to achieve low power consumption, the ADXL362 does not alias input signals by under-sampling; it samples the full bandwidth of the sensor at all data rates. The ADXL362 always provides 12-bit output resolution; 8-bit formatted data is also provided for more efficient single-byte transfers when a lower resolution is sufficient. Measurement ranges of ±2 g, ±4 g, and ±8 g is available with a resolution of 1 mg/LSB on the ±2 g range. The FPGA can talk with the ADXL362 via SPI interface. While the

ADXL362 is in Measurement Mode, it continuously measures and stores acceleration data in the X-data, Y-data, and Z-data registers. The accelerometer interface is mentioned in the figure below.

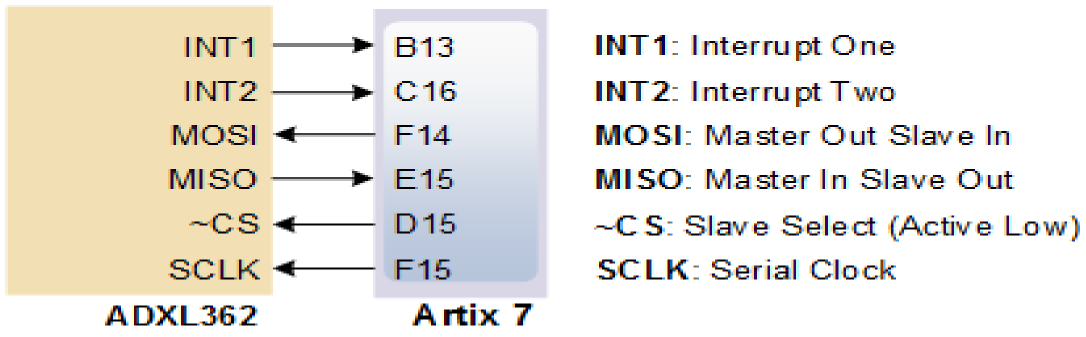


Figure 23. Accelerometer interface.

ADXL362 acts as a slave device using an SPI protocol. The recommended SPI clock frequency ranges from 1 MHz to 5 MHz, we have configured it to the maximum recommended SPI clock frequency of 5MHz. The SPI operates in SPI mode 0 with CPOL = 0 and CPHA = 0. All communications with the device must specify a register address and a flag that indicate whether the communication is a read or a write. Actual data transfer always follows the register address and communication flag. Device configuration can be performed by writing to the control registers within the accelerometer. Access accelerometer data by reading the device registers.

The display used is a **QVGA LCD display** that has a resolution of 320 x 240 pixels and communicates to the Nexys4 DDR board through a PMOD connection port. The LCD unit is interfaced to the Microblaze soft-processor through a Quad-SPI interface IP. The LCD unit used in this project is from the manufacturer called Adafruit. Further information can be found in the references section at the end of this report. **[5]**

2. THE GAME

2.1 OVERVIEW

The objective of the player in the game is to navigate successfully from point A to point B without dying. The game features two modes in which the player can choose to play. The modes can be described as difficulty settings where the complexity of the maze is changed to meet the difficulty standard to be met for that level. The levels are generated dynamically by a random maze generation algorithm that is a part of the game logic itself. The algorithm used is called depth first search algorithm or a Rosetta maze generator **[1][2][3]**. The random maze generation algorithm is a simple c code that uses depth-first search algorithm to generate the walls. The player starts at a random position in the maze with a destination area that is also randomized. The player completes the level when the destination area is reached. Along the way to the destination area there are multiple obstacles and objects called **"Stars" and "Holes".** Stars refer to the objects that grant the player 1 point to the total score accumulated for the game and holes are objects that when touched decrease the life of the player by 1.

By default, the player has 3 lives at the start of each maze map in any difficulty mode. The player must navigate from start to stop and if he does so by gathering all the stars without losing even a single point in life, then a perfect completion is awarded to the player at the end screen. If by any chance the player loses all the 3 lives before reaching the destination area, then the game ends prematurely and the user is given the option to either retry or quit. The above logic holds good for both the difficulty modes. However, it should be noted that in the second difficulty mode, the holes will be present in the way of navigation of the maze.

The movement logic in the game has a special function to help navigate the player at these crunch points. By game terminology, this is called jumping and by this the user can navigate the corner of the maze by jumping over a hole or across it without touching it. The movement function in the game logic intelligently senses the position of the ball and if the accelerometer is given a diagonal input (moved diagonally instead of normal tilting), the jump functionality is implemented instead of a normal move when the ball is at the corners.

2.2 DESIGN METHODOLOGY

The design of the game can be functionally split into two different parts. The first part of the game is the core logic involved and the second is a wrapper module that allows the game logic to be implemented and interfaced with all the other components in the system. In this case the wrapper is the QEP Nano hierarchical state automata code. The reason why QEP- Nano was chosen for the was because of the robust nature of the state machine itself and the sheer flexibility offered by it for new feature addition. It offers a structured way of handling events that can occur at different time frames while effectively handling the state machine to interface with individual sub-systems.
The game logic can be split into the following parts.

- **Maze generation**
- **Collision detection and navigation**

The maze generation is done dynamically by a simple C code that uses depth first search algorithm to generate walls and ways. The output of the C code is an array of elements that consists of elements called “Walls” and “Ways”. This array is stored as a global resource that is used at the start of every game mode by the LCD display function. The Walls and ways are color coded and hence using the array the map for the maze is generated. The objects such as stars and holes are also dynamically generated in the generated maze map array by a random cell pick and place algorithm. However, it should be noted that the holes are placed only in corners, to effectively use the jump feature to avoid the same, whereas the stars are randomly placed on the maze map.

The Collision detection and navigation is the other main part of the game logic that determines how the player can navigate the maze. The game logic is designed in such a way that the player’s ball moves according to the accelerometer data register changes. The direction of movement is sensed by reading the accelerometer data registers for the X-axis and the Y-axis, more information about accessing the register and register bit descriptions can be obtained from 'serial communications' section of the ADXL362 Datasheet **[6].** The third axis is not used since this is a 2D maze. Whenever a new game is started, calibration process occurs, where the current x and y axes are set as the initial value until the end of the game. Whenever the accelerometer readings are crossing this initial value plus/minus the preconfigured sensitivity value, the respective direction is taken as the expected ball movement direction.

The collision detection technique is intelligently implemented to avoid cheating in the game and jump across walls and complete it in an unfair manner. The navigation system senses the player’s current position and the surrounding neighbor cell values from the fixed maze map array. If the next cell to be moved is a “Wall” then the navigation system will not allow the user to move into the wall or across it thus preventing the user from going “into the wall”. To navigate the maze effectively while avoiding holes, a special move function is implemented at the corner cases alone. The move function intelligently senses the corners and whether the player is in such a corner and helps him jump across the hole without coming in contact with the hole.

The wrapper module that is implemented to serve as a general skeleton for the entire project is the QEP-Nano hierarchical state automata. The basic function of the QEP-Nano code is to serve as framework for attaching and creating various sub-systems for the game and to serve as a platform that enables easy changes in the future or for further addition of features such as using a different display etc... The Hierarchical state automata used closely models the functional and operational behavior of the game logic implemented. It enables easy event handling which would otherwise lead to unattended interrupts or events missed due to atomicity of the code itself. The state machine consists primarily of three states. The state diagram is given in the figure below.

The state machine utilizes both interrupts and polling techniques. There will be this 'USER_TIMEOUT' event that will be generated for every 10ms by an auto reload Timer0, which will perform different actions based on the present state. When in Idle state, it will perform pseudo random number generation. When in Play state, it will perform the key operation of polling/reading the accelerometer x and y axis data, identifying movement and collision; it may update life and score values, move the ball or end the game.

In the Idle state, the state machine listens for the user input to start the game. If the user does not give any input (game start input) and times out, as mentioned above a pseudo random number will keep generated, using linear feedback shift registers [7], which will be used in randomizing the maze generation logic when the game is started. The Press of the left button (BTNL) triggers the start of the game with the easy difficulty setting. The press of the center button (BTNC) triggers the start of the game in the hard difficulty setting with both the events making the state machine to transition from idle to play state.

From figure 1 the following elements are explained for better understanding of the reader.

Events: BTN_RIGHT_CLICK, BTN_LEFT_CLICK, BTN_DOWN_CLICK

Functions:

- Reset_data_create_maze() - Takes the pseudorandom seed and generates the map for the maze based on the difficulty chosen. (Easy or hard depending on the button pressed)
- Acl_calibrate() - Calibration function for the accelerometer. Takes the present value of x and y as standard reference(default).
- Acl_access() - Accesses the data registers of the accelerometer(reading values) and updates the location variables based on the direction moved.
- Move_ball() - Moves the user ball on the maze map depending on the accelerometer data output. Input and output destination values are fed as arguments to the function.
- Start_screen() - Displays the start screen of the game with the legend for user information. If the game was force stopped, then a separate message is displayed by passing a different argument to the function (0/1).
- End_screen() - Displays the end screen with information on how to play the game further if the user has failed a particular level.

- Victory_screen() - displays the victory screen when the user has successfully completed a maze level.

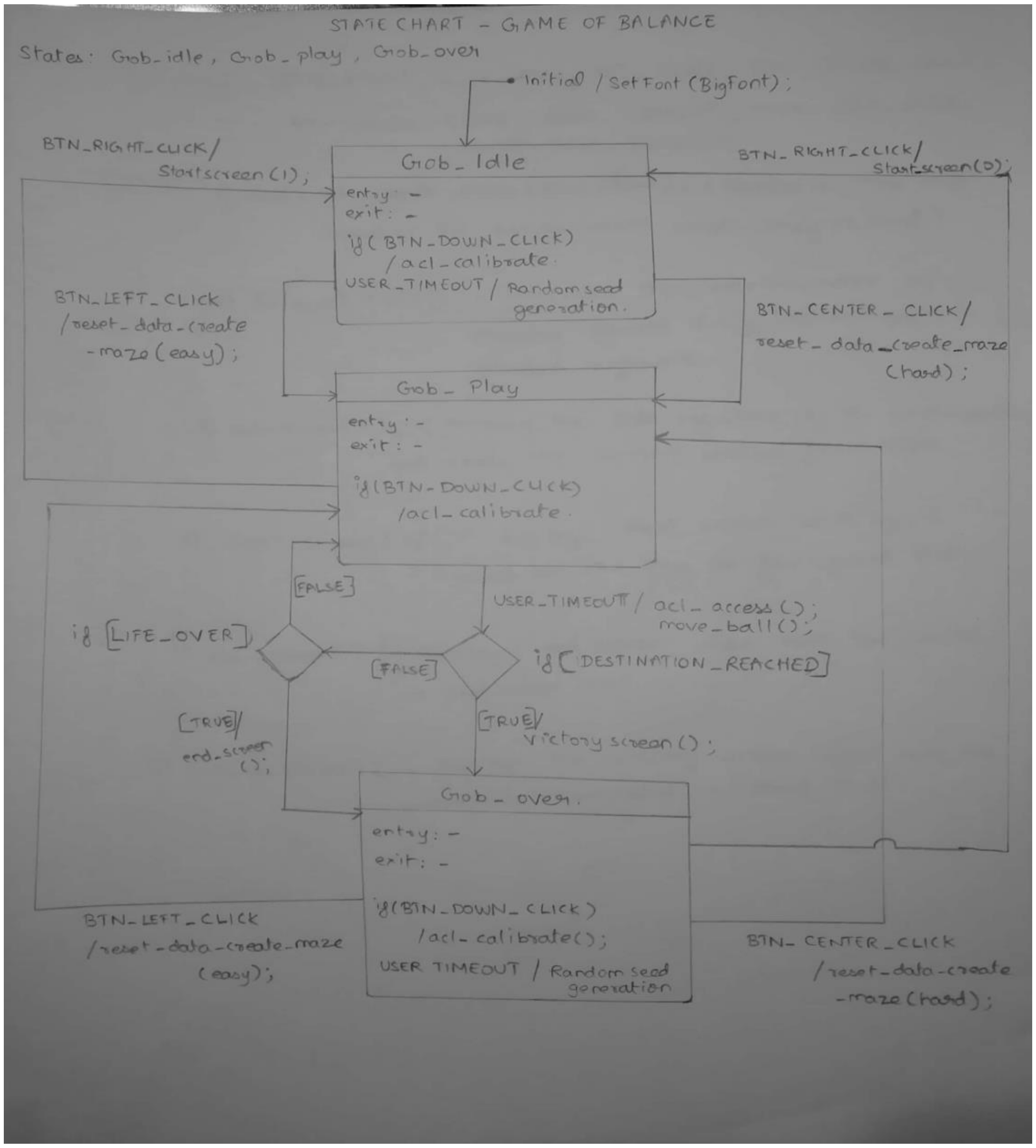

STATE CHART – GAME OF BALANCE
States: Gob_idle, Gob_play, Gob_over
Initial / Set Font (BigFont);
BTN_RIGHT_CLICK/ Startscreen (1);
Gob_Idle
entry: –
exit: –
if (BTN_DOWN_CLICK) /acl_calibrate.
USER_TIMEOUT / Random seed generation.
BTN_RIGHT_CLICK/ Start_screen(0);
BTN_LEFT_CLICK /reset_data_create_maze (easy);
BTN_CENTER_CLICK/ reset_data_create_maze (hard);
Gob_Play
entry: –
exit: –
if (BTN_DOWN_CLICK) /acl_calibrate.
[FALSE]
USER_TIMEOUT / acl_access(); move_ball();
if [LIFE_OVER]
[FALSE]
if [DESTINATION_REACHED]
[TRUE]/ end_screen ();
[TRUE]/ victory screen ();
Gob_over.
entry: –
exit: –
if (BTN_DOWN_CLICK) /acl_calibrate();
USER TIMEOUT / Random seed generation
BTN_LEFT_CLICK /reset_data_create_maze (easy);
BTN_CENTER_CLICK /reset_data_create_maze (hard);

Figure 1. State machine for the game logic using QEP-NANO

In the play state, the state machine listens for the stop button interrupt, which is the right button (BTNR), or the occurrence of the final conditions for the game to get over during the timeout event. If the stop button is pressed, the state machine transitions from play to idle state displaying the information that the game was stopped forcibly by the user and asks if the player wants to restart or not. Otherwise when the player reaches the destination or if the life count goes to zero, the state machine transitions from the play state to the over state by displaying the appropriate screen for victory or the game over screen. The User timeout is the main key event that activates the function to read the accelerometer input. Upon a timeout(100ms), the accelerometer is read and depending on the direction of the movement, the user ball is moved. This is like a polling mechanism that periodically checks for accelerometer input and depending on the direction of the accelerometer movement, the user's position is fed to the move ball function which correspondingly moves the ball based on the collision detection logic.

In the over state, state machine actively listens for all the possible inputs. If the user input is to restart the game, then the game is restarted with the appropriate difficulty (easy or hard). These cases trigger a transition back to the play state. If the stop button is pressed, the state machine transitions to the idle state displaying the default start screen.

3. EXPERIMENTAL RESULTS

The experimental results and observations from the project are as follows. As explained in the previous sections, the project encompasses all the features such as varying difficulty modes, dynamic maze generation for totally random map seeds and increases the quality of gameplay. The total time taken by the maze generation algorithm is in the order of 300-350 milliseconds, which will be executed only once for each game start and hence is barely noticeable.

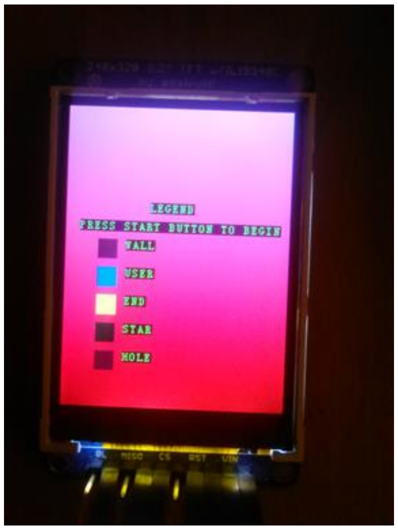


Figure 2. Start Screen

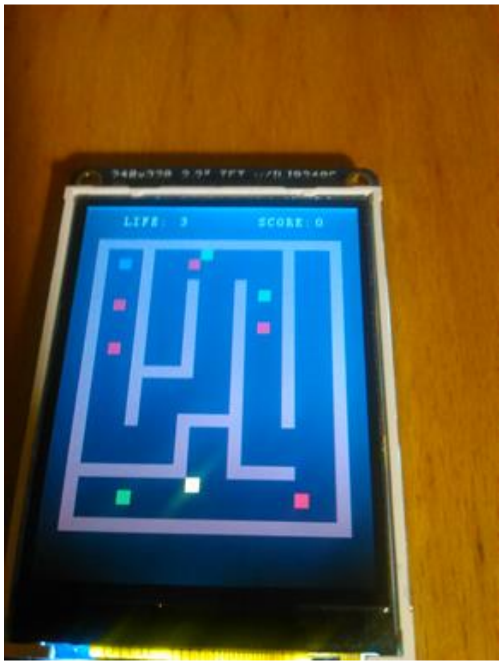


Figure 3. Maze (Easy difficulty)

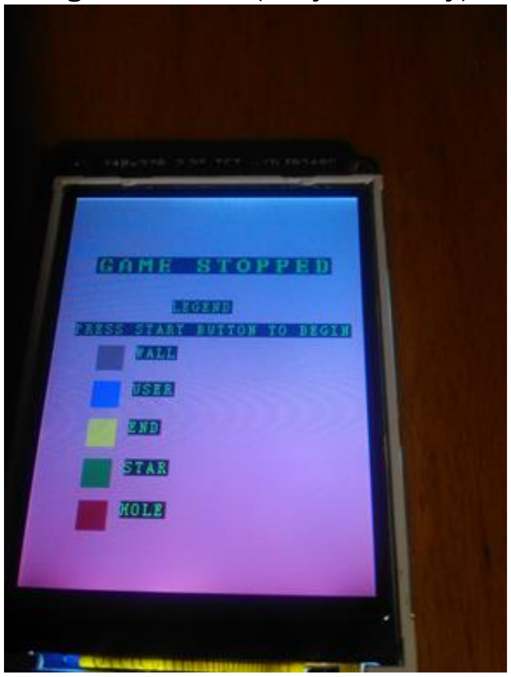


Figure 4. Stop Screen

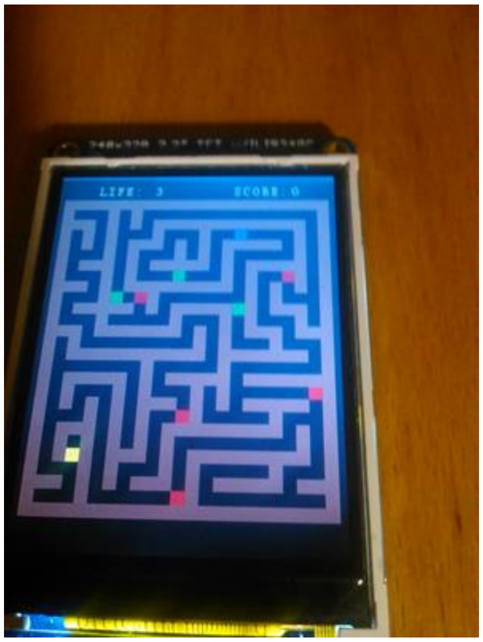


Figure 5. Maze (Hard Difficulty)

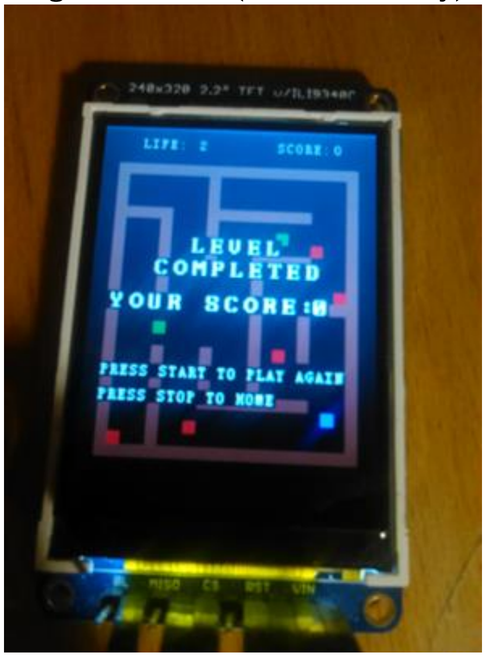


Figure 6. End screen

Figures 2 through 6 explain the different stages of the game logic. Figure 2 is the start screen of the state machine. This is the default screen that loads up when the game is started. A legend is displayed on the start screen to provide information for the user about the elements in the maze. When the left button is pressed(BTNL), the easy difficulty logic is loaded. This is shown in figure 3. It should be noted that the in the initial start screen unless user input is given, the state machine continuously

proceeds with a random seed generation for the map in both easy and hard difficulty modes using the logic mentioned in the above section. The maze generation logic was tested to ensure that the mazes generated were complete and that there was a path to completion. The maze generation algorithm also involved the placing of holes and stars. The method of placing the stars is random and can occur anywhere in the map other than the walls. However, the holes are always placed at the corners where the point at which it is placed will be having walls on two adjacent sides and will be free on the other two sides. This was done to ensure that the user can navigate through the higher difficulty maze without losing life using the corner jump functionality. On the lower difficulty level, the holes are placed randomly on the map only if all the adjacent spaces are empty (no walls present).Figure 4 shows the stop screen which is displayed when the game logic is prematurely terminated by the user by pressing the (BTNR) right button. This screen gives information to the user on how to begin a new game if the user desires to do so. Figure 5 is a sample maze on the hard difficulty level. Figure 6 is the completion screen when the user reaches the end destination on the maze map.
The following cases were tested for logic stability and game functionality.

- Collision detection and ball movement
- Game logic on both difficulty levels
- Maze generation

The Collision detection and movement logic for the player was tested exhaustively on both the difficulty levels over all possible cases, for example, the ball was placed at the end of the maze (near the border of the screen) and was tested to ensure that the ball would never go out of bounds. In a similar way tests were done to ensure that the user could never go into a wall or go across a wall. This logic testing was done on both the difficulty levels on an iterative basis encompassing a large number of random seeds(maps) and on all the possible locations in the maze. The ball movement logic was also tested to ensure that the jump functionality would be properly implemented in the corner cases where the user would need to traverse through the holes to reach the destination.

The game logic was tested by playing the game in both the difficulty levels on randomly generated maps exhaustively to ensure that the playability of the game was challenging and yet not impossible to finish. Upon loading of the game at each instance, the games were run to completion by manually playing the game to reach the destination or by losing life points by intentionally crossing over the holes. In both cases the game logic was tested exhaustively over random maps on an iterative basis as well as cold starting the game logic from the beginning as well. The games were tested to ensure that the score logic was also proper and that the scores updated in real time whilst displaying the proper screen after completion and the scoring mechanics were also proper.

The maze generation logic was tested exhaustively by iteratively loading the game screen after each stop and completion to ensure that the maps generated were truly random. The logic that involves the placement of holes and stars were checked to ensure that the map is always challenging but not impossible to finish. Collecting all the stars and not losing a single life point results in a perfect score that is displayed at the time of reaching the destination. This was also tested on both difficulty levels on a wide variety of map seeds.

4. LIMITATIONS AND FUTURE WORK

As mentioned in the above sections, the display used for the game is a QVGA LCD display unit that uses SPI interface for communication and data transfer. Due to the serial communication nature of the display, the updates and refreshing capabilities of the hardware limit the frame-rate of the images displayed on screen. Frame-rate is a crucial parameter that determines the playability of the game. Using the LCD QVGA display unit, the maximum frame-rate that we were able to achieve was 10 frame updates per second. In case of a VGA display, higher frame-rates can be achieved. Due to the non-availability of a direct IP core to interface the VGA display with the Micro-Blaze processor and the limited time availability, VGA was not implemented for this project. This may be viewed as one of the possible extensions for this project.

In addition to the adding the VGA display, a second state machine can be added to augment the functionality of the move logic enabling faster and responsive yet smooth movement instead of movement for every ball size that is being used currently. Also, another feature that can be added is a more complex and a bigger maze with a stop-clock mechanism that can count the number of seconds that the user took to complete the maze. A leaderboard feature can also be added to record the list of top 3 or 5 scores achieved.

5. REFERENCES

1. Sample maze patterns
2. Random maze generation techniques
3. Maze generation algorithm Overview
4. Accelerometer interfacing and functionality
5. QVGA LCD display reference
6. ADXL362 DataSheet
7. Pseudo Random Number Generation Using LFSR