EE 254 – Introduction to Digital Circuits

spring 2015 | may 1, 2015

BOMBMERMAN Final Project

Christin carter – computer engineering and computer science anthony asuquo – computer engineering and computer science

Contents

[Abstract 2](#_Toc418119568)

[Introduction and Background 2](#_Toc418119569)

[The Design 2](#_Toc418119570)

[Test Methodology 4](#_Toc418119571)

[Conclusion and Future Work 5](#_Toc418119572)

# Abstract

# Introduction and Background

Bomberman is a computer multiplayer video game where players compete to be the last-man standing on different playing fields. In the original game, players have the ability to produce “bombs” that explode in a cross-like fashion, destroying objects and players in its path. The game is enhanced through players’ abilities to pick up perks and power-ups that allow players to move faster, release multiple “bombs”, and other enhancement.

# The Design

Our version of the game is not at the graphical level, only incorporates a two-player system, and does not consist the variety of power-ups that the original Bomberman game contains. The objective is to destroy your opponent utilizing the bomb-dropping capabilities every player is given. Players start in a main menu and transition to the game play state once both users are ready. During gameplay, players navigate the map and strategically place bombs in an attempt to beat each other. First player to win three rounds wins the game, and the players are asked if they want to replay the game. Below is the state diagram and a more technical description.

Reset

. . .

. . .

. . .

READY

. . .

READY

**MAIN\_MENU**

waitTimer1 <= 72 waiterTimer2 <= 48

countDigit <= 0 p1\_score <= 0

p2\_score <= 0 LCD <= “Pyromaster Press Start!”

//p1 variables

velocityX <= 0; velocityY <= 0;

direction <= 2'bXX; p1\_dead <= 0;

bombY <= 0; bombX <= 0;

bombRad <= 15; bombCount <= 0;

bombTimer <= 0; bombDelay <= 96;

explodeTimer <= 24; explode <= 0;

positionX <= 30; positionY <= 31;

//p2 has same variables

p2\_positionX <= 610 p2\_positionY <= 450

**WAIT**

LCD <= “Waiting for P1, Waiting for P2”

**WAIT\_P2**

LCD <= “P1 Ready! Waiting for P2”

**WAIT\_P1**

LCD <= “Waiting for P1, P2 Ready!”

**p2\_joystickStartButton &&**

**joystickStartButton**

**p2\_joystickStartButton**

**GameStart**

**joystickStartButton**

**joystickStartButton**

. . .

READY

**p2\_joystickStartButton**

. . .

. . .

. . .

. . .

**waitTimer1 == 0**

. . .

. . .

**READY**

if(waitTimer1 == 0)

waitTimer <= 72

else

if (waitTimer1 <= 24)

countDigit <= 1

else if (waitTimer1 <= 48)

countDigit <= 2

else if (waitTimer1 <= 72)

countDigit <= 3

waitTimer1 <= waitTimer1 – 1

LCD <= “Game Starts in” + countDigit

**­­­GAME**

positionX <= positionX + velocityX; //Update player location on VGA

positionY <= positionY + velocityY;

if (positionY < 26) //Collisions with border of playing field

positionY <= 27;

else if (positionY > 454)

positionY <= 453;

else if (positionX < 25)

positionX <= 26;

else if (positionX > 615)

positionX <= 614;

else if (direction == 0) // Collisions when Moving Up

if (maze\_block\_0 || maze\_block\_1 || maze\_block\_2 || maze\_block\_3)

positionY <= 131; //Resets players position if player overlaps with block

else if (maze\_block\_4 || maze\_block\_5 || maze\_block\_6 || maze\_block\_7)

positionY <= 235;

else if (maze\_block\_8 || maze\_block\_9 || maze\_block\_10 || maze\_block\_11)

positionY <= 339;

else if (maze\_block\_12 || maze\_block\_13 || maze\_block\_14 || maze\_block\_15)

positionY <= 443;

else if (direction == 1) // Collisions when Moving Down

. . .

. . .

else if (direction == 2) // Collisions when Moving Left

. . .

. . .

else if (direction == 3) // Collisions when Moving Right

. . .

. . .

**~p1\_dead && p2\_dead**

**p1\_dead && ~p2\_dead**

**p1\_dead && p2\_dead**

. . .

RND\_DRAW

. . .

RND\_WON\_P2

. . .

RND\_WON\_P1

**GAME (Continued)**

if (p1\_dead)

p2\_score <= p2\_score + 1;

else if (p2\_dead)

p1\_score <= p1\_score + 1;

if (p1\_killed)

p1\_dead <= 1;

LCD <= “P1 Score –“ + p1\_score + “P2 Score –“ + p2\_score //display scores on LCD

if (joystickY < 300 && ~(joystickX < 400) && ~(joystickX > 600)) //Pushed Down

velocityY <= -3; //Set speed of player in specific direction

direction <=1;

else if (joystickY > 700 && ~(joystickX < 400) && ~(joystickX > 600)) //Up

velocityY <= 3;

direction <=0;

else if (joystickX < 300 && ~(joystickY < 400) && ~(joystickY > 600)) //Left

velocityX <= -3;

direction <=2;

else if (joystickX > 700 && ~(joystickY < 400) && ~(joystickY > 600)) //RIght

velocityX <= 3;

direction <=3;

else

velocityX <= 0;

velocityY <= 0;

if (joystickBombButton && bombCount == 0 && bombTimer == 0 && explode == 0)

bombCount <= bombCount + 1; //Produce bomb where player pressed

bombTimer <= bombDelay; //the joystick button

bombY <= positionY;

bombX <= positionX;

if (!(bombTimer == 0)) //Countdown to detonation

bombTimer <= bombTimer - 1;

else if ((bombTimer == 0) && !(bombCount == 0)) //Displays the explosion of the

bombCount <= bombCount - 1; //bomb on screen

explode <= 1;

if (explode)

explodeTimer <= explodeTimer - 1;

if (explodeTimer == 0) //Resets the timers and state

explode <= 0; //of the players bomb

explodeTimer <= 24;

READY

. . .

. . .

. . .

. . .

READY

READY

. . .

. . .

**RND\_DRAW**

LCD <= “Round Draw”

. . .

WAIT

MAIN\_MENU

**p2\_score != 3**

**p1\_score != 3**

**RND\_WON\_P2**

LCD <= “P2 wins Round!”

**RND\_WON\_P1**

LCD <= “P1 wins Round!”

**PLAY\_AGAIN**

LCD <= “Play Again, Yes No”

. . .

Decline

Accept

~Accept && ~Decline

waitTimer2 == 0

waitTimer2 == 0

waitTimer2 != 0

waitTimer2 != 0

**GAME\_WON\_P2**

LCD <= “P2 wins Game!”

if (waitTimer2 == 0)

waitTimer2 <= 48

else

waitTimer2 <= waitTimer2 - 1

**GAME\_WON\_P1**

LCD <= “P1 wins Game!”

if (waitTimer2 == 0)

waitTimer2 <= 48

else

waitTimer2 <= waitTimer2 - 1

**p1\_score == 3**

**p2\_score == 3**

On Reset the state machine is initialized to the MAIN\_MENU state, in which most of the player variables, timers, and other signals are set. The LCD serves as our welcoming screen and displays the text “Pyromaster Press Start!” Players sit in this state until the “GameStart” signal goes high, which is accomplished by pressing btnU on the FPGA board, transitioning the machine to the WAIT state. The WAIT state acts as a lobby for the players as both users must signal they are ready by pressing the right button on the joystick. If only one of the players clicks their ready button the state machine goes into a dummy lobby (WAIT\_P1 or WAIT\_P2) where we continue to wait for the other player to be ready. Initially, the LCD will print to the screen that it is waiting for both players. Once one or both of the players click their respective start buttons, the LCD displays which players are ready or not. Once both have confirmed the state machine switches to the READY state that displays a three second countdown, allowing users to become situated and acquainted with the joysticks before transitioning to the GAME state.

The game state is the chunk of our program. Here we do a lot of multitasking and processing for the game. The state diagram is a bit deceiving as it shows to separate states for the GAME state. However, there is just so much code in this state that it could not all fit. In the first chunk of the GAME state, the X and Y position of the player are constantly updated based on the velocity, which is determined by the direction you push the joystick. There is also code for the collision detections in the game. To prevent the players from going off the screen, we established a perimeter that serves as the map edges and if a user’s position is greater or lesser than these boundaries, their position is set back to the pixel closest to the edge. For instance, our right wall boundary is set to be the 615th pixel, so the instant the player’s X value hits 616 or greater, their position value is set back to 614. This simulates the player no longer being able to move in the right direction. The same concept applies to maze blocks we have within the playing field. However, the collision between the maze blocks is a bit more complex because it is possible to “interact” with these objects from different sides and positions. Therefore, when righting the collision detection code for the blocks, we had to consider what the player’s updated position should be if they collided from the top, left, right, and bottom.

In the second chunk of the GAME we handle score updates when a player is killed and display the scores of each player on the LCD. In addition, we set the velocity and direction of the player after moving the joystick. The joystick returns two numeric values between 0 and 1023 based on the positioning of two-axis joystick. The values in our state diagram were the values we determined were optimal when detecting the direction a player requests to move in. To prevent from diagonal motion, the joystick values that represent going a certain direction were AND’d with a range of values that the other axis has to be in between so that a player can move in one specific direction. Lastly, we also handle the production of bombs. When a player presses the left button on the joystick the state diagram checks to make sure the players does not have any bombs currently on the map, hence the “bombCount == 0” expression, and that there is not a bomb in the process of detonating, hence the “bombTimer == 0 && explode == 0” expression. Once this is verified, the player’s bomb count is incremented by one, the bomb is placed at the X and Y coordinates of the player, and the detonation timer is initiated. Once the bomb timer expires, the explosion signal goes true which simulates that the explosion has happened. Once this process is over, the bomb count is decremented, the timers are reset and the process is ready to happen all over again.

Once one or both of the players is defeated, the state transitions to one of three states, RND\_DRAW if both players were killed, RND\_WON\_P1 if player one ended up being the last man standing, or RND\_WON\_P2 if player two survived the round. In the RND\_DRAW state, the LCD displays “Round Draw” to represent a tie match. For the other two states, depending on which player won the round, the LCD displays that the player won a round. RND\_DRAW will always return to the READY state to prepare the players for the next round. Until a player reaches three points the RND\_WON states will also transition back to the READY state. Once a player has a score of three the game end and the state machine transitions to the “GAME\_WON\_P1” or “GAME\_WON\_P2”state where the LCD displays which player won for two seconds and switches to the PLAY\_AGAIN state. In this final state, players decide if they want to rematch each other to another game, if so they will generate the “Accept” signal by pressing btnL or “Decline” by pressing btnR. Pressing Accept sends the players back to the WAIT state and players must signal that they are ready. Alternatively, pressing decline sends players back to the MAIN\_MENU state.

**Note: In the MAIN\_MENU and GAME states on variables and code for player one is shown, but the player two code is literally a replica of the player one variables, just with different. Because of this redundancy, player two’s declarations were not listed.**

# Test Methodology

The main testing method we utilized was trial and error. One of the toughest parts to implement was collision detections.

# Conclusion and Future Work