

ECE 385 Final Project Lab Report

Final Project:

Bomberman on the FPGA

Spring 2022

Trusha Vernekar & Soham Manjrekar

Dohun Jeong (DJ)

tnv2 & sohammm2

Introduction/Overview

Our Final Project for ECE 385 was to make a multiplier version of the 1983 NES classic video game Bomberman. In our version of Bomberman the objective of the game is very simple, to blow up the other user. The movement inputs are WASD for user 1 and the four arrow keys for user 2. The action inputs are V for user 1 and P for user 2.

Both users start on a playable board with 3 lives each and the ability to drop bombs. The board users play on is constructed of 5 different types of tiles: Grey Walls, Pink Bricks, Paths, Safeports, and Hazards. Grey Walls are unbreakable walls that restrict movement and cannot be blown up. Pink Bricks, like Grey Walls, restrict movement however they can be blown up to create Paths. Paths are traversable tiles that do not restrict movement, users can stand or drop bombs on these tiles. Safeports are in hard to reach positions on the top and bottom of the board, these tiles teleport users to their starting position, however there is a pseudo random chance that they will teleport you to the opposing users location instead. Hazards are placed around the center of the board, if a user comes into contact with one of these they instantly die losing all of their lives and the opposition wins the game. The playing board is made from a .mif file leaving a certain level of customization within the game which we will discuss later as well.

Users can drop bombs by using the V and P keys respectively for user 1 and user 2, bombs can only be dropped one at a time, and once dropped there is a short delay before they explode. If a user is within the blast radius of the bomb while it explodes they will die and lose one life. The bombs explode in a cross, blowing up the tiles above, below, left and right of the bombs initial placement. If an explosion's radius reaches any Pink Bricks the explosion will break the bricks leaving Path tiles, however, for any other tiles (Grey Walls, Safeports, existing Path tiles, and Hazards) the explosions will not change the nature of that specific tile.

Moving on now to some qualities that impact the flow of the game. Once the FPGA is programmed and the software is running, the game begins on a start screen introducing the tester to Bomberman, upon pressing space, the game begins and follows the rules we have established. During a game, either user can pause by pressing the escape key which brings up a pause screen, during this time no inputs fed by either user will impact the game until the spacebar is pressed resuming the game. Finally, once a user either loses all of their lives, by getting exploded or by hitting a Hazard tile, a screen comes up indicating which user has won. Using the Key1 on the FPGA resets the game, returning to the beginning so that you can play again if you choose.

Official Feature List

- 2 playable characters
 - Movement: Able to traverse playing board
 - Action ability: Dropping Bombs
- Bombs
 - Have a grace period followed by an animated explosion
 - Affects the board as well as impacts the players
- Board
 - RAM module instantiated with a .mif file to create a board
 - 5 Different Tiles that all have different properties affecting different aspects of play.
 - Tiles are:
 - Grey Walls:
 - Pink Bricks
 - Paths
 - Safeports
 - Hazards
- Game Fundamentals
 - All game fundamentals are true fundamentals, the clocks and timings are paused, and no inputs are allowed or registered during these states
 - Start Menu
 - Pause Menu
 - Player 1 or Player 2 Win Screen

Top Level RTL Block Diagram

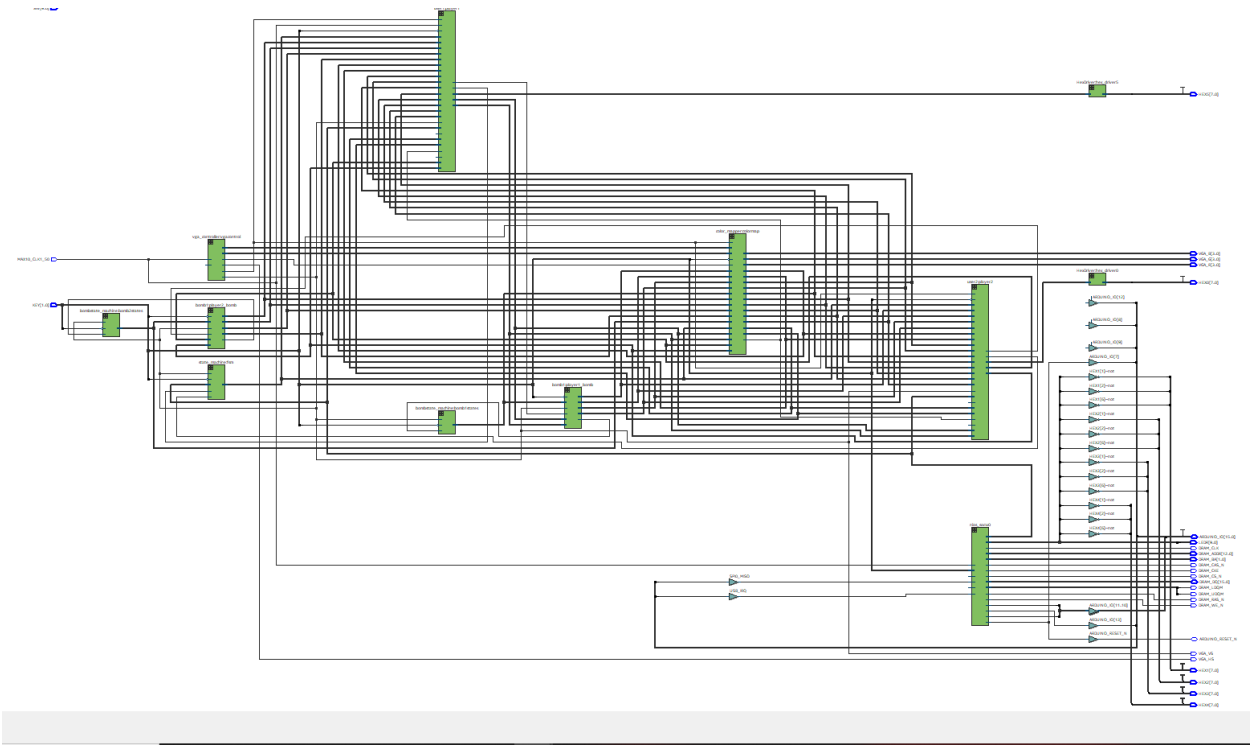


Figure 1: Top Level Block Diagram

State Machines

1. Game State Machine

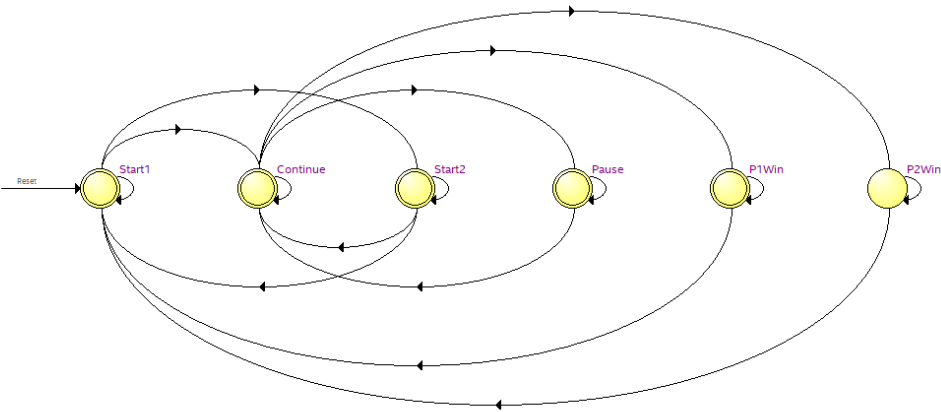


Figure 2:Game State Machine

2. Bomb State Machine

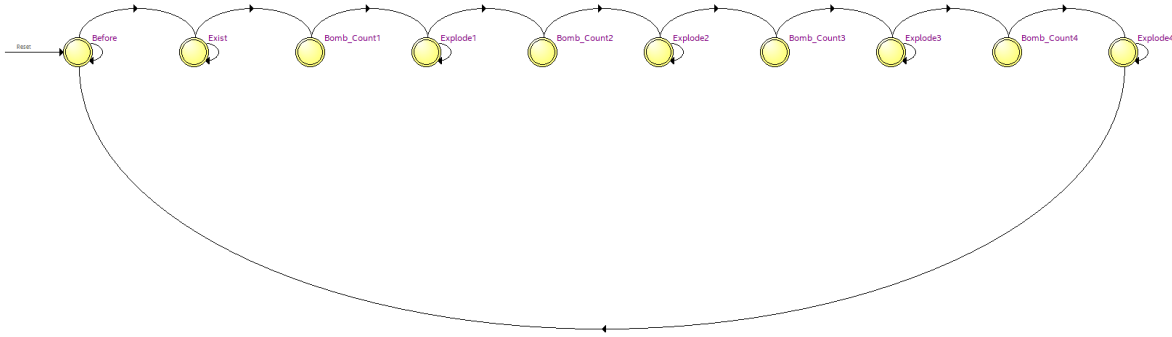


Figure 3: Bomb State Machine

Written descriptions of Hardware

1. User Movement and Action Items

The movement of the users was handled in the same way as the movement of the ball in Lab 6.2. The main difference between the two was that we had 2 players and each was controlled with a separate set of keys on the keyboard. Player 1's movement was controlled with wasd keys and the player 2's movement was controlled with the arrow keys. Along with this, since each of our player could drop a bomb in any location on the map which was a path, the action was controlled by key V for player 1 and key P for player 2.

The way the keys were linked to the movement of the players was the same as Lab 6.2. Depending on which key is being pressed at any specific time, the keycode is read and the movement of the user is assigned accordingly. For example, if the up arrow is being pressed, the Y motion for user 2 is set to be -1 and X motion is set to be 0 (user 2 moves up). The user keeps going in the same direction until another keycode for the user is pressed or it collides with a wall. Instead of bouncing off a wall (like in Lab 6.2), our users would collide with the wall and stay in the same position with X and Y motion of 0. The user can be moved again when another keycode is pressed.

Similarly, to drop a bomb, the keycode is read and if the Key V or P is pressed, a flag is set and a bomb is moved from onscreen to the users position who dropped a bomb. To draw the bomb on the map positions are passed into colormapper which are abstracted out into the 4 directions where an explosion is imminent this handles both the drawing as well as some of the explosion logic. The set flag in user1 and user2 is also checked in the bomb_statemachine which begins the countdown to explosion as well as the explosion states.

2. Creating the Game Board

To construct our board we instantiated a RAM Module (map.v) with a .mif file. To do this we established that our tiles would be 32x32 pixels, this meant that we could divide our screen dimensions by 32, leaving us with a 20x15 screen, which is 300 addresses. So to make our .mif we defined 300 addresses with 4 bit values that correspond to a specific tile (for example, 0000 is a Path, 0001 is a Grey Wall, etc).

The addressing to translate our .mif from 300 4 bit values to on screen locations was fairly straightforward as well, using the top 5 bits of DrawX and DrawY we can do the calculation:

$$\text{DrawY}[9:5] * 20 + \text{DrawX}[9:5];$$

to address into our RAM module to pull the 1st value for the first 32 bit block and the 300th value for the last 32 bit block. With everything in between coming in row major order. This lets us both assign RGB values in our color mapper and lets us slightly off the hook for some collision logic for both of our users as well (we will discuss this a bit later as well).

In our ColorMapper module and our user1/user2 modules we have instantiations of Map.v whose write values are related. This is because once an explosion occurs and a Pink Brick is affected we need to rewrite the value at that address in memory for Color Mapper to be a Path tile so that on screen the Brick tile disappears, we must do this in user1 and user2 as well so that RAM in colormapper is consistent with the RAM in our user modules, this is important since we use the data coming off of memory for collision logic, if we hadn't done this our collision logic would still be operating off of the initial instantiation of the RAM module while on screen the tile would explode.

3. Use of Rishi's tools for png → text

Rishi's tools were used very liberally and exhaustively in this project, making use of the palletizer and resizer to cut down on the number of bits required in memory from 24 for hex values to 3 or 4 depending on the size of our palette and to resize images from 16x16 to 32x32 in the case of our tiles and explosion animation frames, without the loss of defining characteristics.

The main purpose of this tool is to turn a png into a row major order txt file that defines the RGB value of every pixel in the png. Then using a ram module that instantiates itself with the txt file, you can address through the RAM using DrawX and DrawY in colormapper and display out the png, we will discuss this calculation for addressing later. This made drawing bitmap sprites very easy since there was a standard process to display them.

4. Color Mapper

Our Color Mapper is split up into a few sections. Firstly, we make all the logic variables for users positions, and bomb positions, including their offset from DrawX and DrawY ($\text{distX} = \text{DrawX} - \text{Xpos}$, $\text{distY} = \text{DrawY} - \text{Ypos}$, for user1, user2, bomb1, and bomb2). We also create all of the addresses we need to index into our ram modules for our sprites. The address calculation is a straightforward manipulation of the following calculation:

$$(\text{DrawX} + (\text{Xsizeofsprite} * \text{DrawY})).$$

The next section is combinational logic. The first block pertains specifically to the explosion of bombs, it determines if the tile that is being exploded is a Pink Brick and sets the write enable to change it to a Path, this is the same write enable we are passing to our users to ensure that our RAM modules are consistent. The next block of combinational logic handles resets, this block sets a write enable on the second port of the ram module to rewrite every address in the colormapper ram to the original data, since our ram modules in user1 and user2 are related, on a reset, they are also written with the original data making the game replayable.

The next section is the instantiation of every RAM module that holds a sprite, using the addressing method from before.

Following that is a block of sequential logic that sets temporary RGB values which will later get set to the output RGB values. One of the larger sections in this block of code are the conditionals checking wall_data. Wall_data is the 4 bit value of our .mif file at the current DrawX and DrawY values addressed using the calculation from before. Depending on what our

wall_data is we use the values coming off of our sprite ram modules addressed by a manipulation of the calculation above.

(specifically: wall_addr = DrawX[4:0] + (32 * DrawY[4:0]);)

to set temporary rgb values for each pixel of the 32x32 wall. In this section the other checks that occur are for our explosion animations, since those are tiles as well, we can treat them as such by modifying the conditional slightly, instead of checking the data from our mif, we check if the position of an explosion is valid by comparing the position of the bomb with the DrawX and DrawY position as well as what state the explosion is to ensure that the temp RGB values are set only when necessary. Finally, the last section of this block is the other case statements to set temp RGB values for sprites that are not tiles which includes, user1 and user2, bomb1 and bomb2, as well as the start, pause and win screens.

Next we have a small block of combinational logic which sets flags for if the non tile and non screen sprites should be drawn, and handles transparency as well. A flag is set if the sprite offset from DrawX and DrawY is (introduced before as dist) is within the size of the sprite to indicate the sprite should be drawn. To make the sprites transparent an additional condition checks to see if the data (palette index) coming off the sprite ram modules does not correspond to the background color for our sprites before setting the flag.

Finally we have our last sequential block which sets RGB values and handles the information being passed into user1 and user2. Based on game state RGB values are set, during the start pause and win states the RGB values are the values of their respective temporary RGB values. Here the data values being passed to user1 and user2 are set to be offscreen ensuring that an accidental death does not occur. During the continue state however, all of the tiles are drawn depending on what the data coming off of Map.v is, which sets the output RGB values to the temp values that we set earlier. For bomb explosions the explosions are referenced to the 5 positions where it will overlap and rewrite tiles, and saves those positions to logic variables that will be passed to user1 and user2. Following an explosion these logic variables will be set back to offscreen like they were during the other game states.

5. User Collision Logic with 5 types of tiles

In user1 and user2 there are two methods of collision. The first is a hardcoded mathematical way which involves constricting the users location and then doing the modulus of

64 against the user position to check if the user is overlapping with a wall since the walls only exist from bits 32-64 in the X and Y direction. This method works for our Grey Walls however it does NOT work for the Pink Bricks or any other variation of collision logic. This method is still running in our user1 and user2 however it is redundant and superseded by the following method.

The second method of collision uses the instantiated RAM module of Map.v that we have referenced earlier. For collision, we instantiate 4 modules of Map.v, since it is dual port we can write the data colormap is writing while also reading from 4 addresses. The 4 addresses we are reading from are the 4 corners of our sprite, this calculation looks like this:

$$w_typebl = User_Y_Pos[9:5] * 20 + User_X_Pos[9:5];$$

Manipulating the X and Y coordinates to instead be the position plus the size of the sprite to get the X and Y offset we get the following calculations as well:

$$w_typebr = User_Y_Off[9:5] * 20 + User_X_Off[9:5];$$

$$w_typetr = User_Y_Pos[9:5] * 20 + User_X_Off[9:5];$$

$$w_typebl = User_Y_Off[9:5] * 20 + User_X_Pos[9:5];$$

This gives us the address of all 4 corners, which we can pass into our instantiated ram modules. To do collision detection, we check if a corner is intersecting with a wall, meaning the data coming off of the RAM module is a Grey Wall or a Pink Brick (4'b0001 or 4'b0010), then we can check motion of the sprite and set one of the four collision flags. The four collision flags that can be set are if the sprite hits the top, left, right, or bottom of a wall. Using this method of reading what data we have from memory at a position we can abstract the flags out to even more tiles including our Teleport and instant loss tiles. Currently this check is not used for explosions since explosions are done only in display and not with reads and writes from memory, however it could potentially be used for that as well, this is discussed further ahead as well.

6. Teleport and Instant loss tiles logic

These are the two special tiles in our game that add a little difficulty for each player to win the game.

The first kind was a tile that could be used for safe teleports. As mentioned earlier, the user can be teleported to their starting position or to the location of the opposing user (at

random). The pseudo random logic for this instiates a counter that counts to a 5 bit value iterating for every clock cycle of the MAX1050 clock made in our soc. Depending on if the value is odd or even, as well as 4 random bits from the opposing users position, the decision is made to either teleport the current user back to their starting position or to the position of the other user.

The second kind of special tile in our game was the Hazard tile which, as mentioned earlier, would cause instant death of the user if they reached this tile. There were 6 of these placed around the board. This feature was added to the game due to a bug that caused the users to die as soon as these tiles were reached. The feature that was originally planned to be implemented for these positions was to gain an additional life. The way we had planned to add additional lives for a user was by increasing the max_lives, which was initially 6. The variable max_lives which was suppose to track the max number of lives of the player was calculated as follows: If the user comes into the special tile, the max_lives was updated as follows:

$$\text{Max_lives} = \text{max_lives} + 2$$

This variable was checked against the death_count variable to set a die_flag (lives logic along with the die_flag logic is discussed more in detail in the later sections).

However, the implementation, where we added an additional life to the max_lives variable, failed and caused the user to die instantly when they reached this tile. For this reason, instead of using this tile as a power up, this tile was used as a hazard in the game, adding more complexity for individuals to win in this game.

7. Life Count on HEX display

Initially, both users start out with 3 lives each, which is displayed on the seven segment displays on the FPGA. The variable live_count tracks this and is calculated with the following equation:

$$\text{live_count} = (\text{max_lives} - \text{death_count}) / 2;$$

Since the FPGA rounds down we thought to start out the users with an initial value of max_lives as 6 and death count as 0, and each time the user died due to a bomb explosion the death_count iterates, such that eventually we have (6-3)/2 which comes to 0, at this point under the flag which resets the user after a death there is a condition that sets a death_flag where, if the death_count is \geq max_lives then the game is over. For some reason, this logic results in the

game ending once either user hits 0 lives from bomb deaths. We discuss the uncertain nature of this section of our project later as well.

In the top level this comes out very easily, we just have variables of the same size going into the 0th and 5th hex displays to show the lives of user2 and user1 respectively.

8. Bomb Explosions and Board Response

The bomb explosions was one of the most essential parts of our game. Once a bomb is dropped, the bomb state machine takes care of the animations caused due an explosion in the cross which are the tiles above, below, left and right of the bombs initial placement. The way these animations show up on the screen will be discussed in the section about the state machines. The way Bomb explosion logic works is as follows: Once a bomb is placed on the board, the bomb's X and Y positions are set. In Color_mapper, the positions of the explosions are calculated. We needed the position according to our 20x15 map. The center of the explosion is calculated the same way as the users.

Bomb Center: $\text{bombY}[9:5] * 20 + \text{bombX}[9:5]$

Bomb Up: $(\text{bombY}[9:5] * 20 + \text{bombX}[9:5]) - 20$

Bomb Down: $(\text{bombY}[9:5] * 20 + \text{bombX}[9:5]) + 20$

Bomb Left : $(\text{bombY}[9:5] * 20 + \text{bombX}[9:5]) - 1$

Bomb Right: $(\text{bombY}[9:5] * 20 + \text{bombX}[9:5]) + 1$

These 5 positions are used to set the explosion sprites. As mentioned later, the explosions are controlled by the bomb state machine. In Color mapper, depending on the state of the bomb state machine, the corresponding sprite for the explosion is drawn in the 5 positions mentioned above. The sprites for Up and Down are the same and so are the Left and Right. Center has its own sprite. Each has a small and a big sprite. As we cycle between the exploding states in the bomb_state machine, the sprites are changed from Small to Big to Small to create the explosion animation (Will be explained in more detail later).

The 5 positions above are also used for the pink wall explosion logic. As mentioned earlier, an explosion can cause Pink Bricks to break and turn to paths. The way this is handled is by writing to memory. Since the explosion only has an effect on a pink wall apart from a player, and since the only thing the block can change to is a path, the value written to memory is always 0, corresponding to a path in our mif. The way we decided if the block addressed by DrawX and

$\text{DrawY}(\text{w_type} = \text{DrawY}[9:5] * 20 + \text{DrawX}[9:5])$ needs to be written to by setting a write enable. The write enable for any block on our board is set only when the block is being affected by an explosion and if the block was a pink wall. This was done by checking the data read from memory called `wall_data`. Depending on what the `DrawX` and `DrawY` are, the `wall_data` will be a value between 0-4 in decimal (0000- 1000 in binary) (0- path, 1- Grey Wall, 2- Pink Brick, 3 / 4 special tiles). The way we decided if the block was a pink wall, to set write enable was to check if `wall_data` was 2. Additionally, we had to check if the position was actually exploding or not. This was done by checking if the `w_type` was the same as the Up, down, left, right or center of the bomb. Once these 2 conditions are true, the write enable is set to true. This way, the only time `data = 0000` will be written to memory for any block would be when an explosion has occurred.

9. User Collisions with Explosions

The bomb collision logic was originally meant to exist as read and writes from the `Map.v` RAM module, however, there were many problems with the original implementation, this is discussed later in the report. So, instead of reads and writes from memory, the bomb logic depends on the values being passed out from `colormapper` to `user1` and `user2` discussed earlier in the analysis of our color mapper. These values are 10 bits wide and correspond to a position on screen. To check if a collision with a bomb explosion occurs, the position of the corners of a user, whose calculation is in the section prior regarding wall/tile collisions, is referenced against the bomb explosion positions, if this condition is true along with a condition checking the current state of the bomb confirming that it is in one of its explosion states means that the user has collided with an explosion and a flag indicating so is set.

Once this flag (`bomb_flag`) is set, the user loses a life, its position is sent back to the original starting position and the user's motion is halted. As discussed earlier if this occurs three times then the opposing user wins and the game is over.

10. State Machines

For our game, there were 2 simple state machines that controlled different aspects of the game. The game state machine was used to have different game fundamentals such as a start

screen pause screen, etc and the bomb state machine was used to control the explosions when a bomb is dropped.

In our game state machine, the game would start from two start states. The reason why we had 2 start states was because we wanted to create a blinking animation on our start screen for the text “Press space to start”. The way this was done was as follows: First we had a counter going which would increment at every Clk cycle. In the state Start1, this counter was checked and if the value of the counter was either 0x40 or 0xC9, then the next state was Start2.

Something similar is done in state Start2 where if the value of the counter was either 0xFF or 0x80, then the next state was Start1. The way this handled the blinking logic was by setting the pixels of the screen accordingly. In Color_mapper, the game state is checked and if the game was in the state Start1, the start screen without the text “Press space to start” would appear and if the game was in the state Start2, the start screen with the text “Press space to start” would appear. Since the counter is incremented every CLK cycle, the state changes only after a specific time, and the text appears to be blinking. In both the start states, the keycode is checked and if space is pressed, the state is changed to the Continue state. Any other keycode has no effect on the game (This means that the keys to control the 2 players have no effect).

In the Continue state, the 2 users can play the game by using the keyboard. The game state remains unchanged unless the keycode for the escape key is registered. If this key is pressed, the state is changed to Pause. The game stays in this state unless the space key is pressed. Once the space key is pressed, the game goes back to the continue state. In the pause state, any other keycode has no effect on the game (This means that the keys to control the 2 players have no effect). The pause state was to add the pause screen in our game. Again, if the game state was Pause, in Color_mapper, the pixels were set to pixels in our pause screen. Two other ways that the state changes from continue was when either of the players died due to 0 lives or because of the hazard tile. In the continue state, the p1die and p2die flag (set in User1 and User2 according to the die_flag in the modules) were checked. If p1die was true, then the state changes to P2Win and if p2die was true, then the state changes to P1Win. In both of these states, the game state does not change unless the Reset button is pressed on the FPGA. Along with this, any of the other keycodes have no effect on the players. Again, if the game state was any of the 2 Win states, in Color_mapper, the pixels were set to pixels in our Player X Win screen. Upon pressing Key0 on the FPGA, a Reset signal is generated, which is checked in this

state machine. Once the Reset signal is high, regardless of which state you are in, the game returns to the Start1 state of the game (the game is restarted).

As mentioned earlier, the bomb state machine was used to handle animations for explosions due to bombs. This state machine also had a counter which was used to create the animations of an explosion at specific time intervals. At the start of the game, or when Reset is pressed, the bomb state is set to Before. This state machine stays in this state unless either of the players drop a bomb. Once a player drops a bomb, the bomb_exist flag is set to true (in User1 and User2). This flag is checked, and if the flag is true, the bomb state changes to Exist. The bomb state stays in this state until the counter changed to 0xA0 from 0x00. In Color_mapper, the bomb is drawn to the screen only when the bomb state is in the Exist state. The wait time in this state gives the players time to escape the explosion radius. Once the counter hits 0xA0, the bomb state changes to Explode1.

For the explosion animation, we had 4 explosion and 4 bomb_count states. In each of the bomb_count states, the curr_state is changed to the corresponding explode state by default. In each of the explode states, the state changes to the next bomb_count state when the counter hits 0x20, 0x40 and 0x60 respectively. The way these explode state handle the explosion animation is as follows: In Color_mapper, the bomb state is checked. If the bomb state is in the explode 1 or explode 3 state, the sprites for the small explosions are outputted in the cross which are the tiles above, below, left and right of the bombs initial placement. If the bomb state is in the explode 2 state, the sprites for the big explosions are outputted in the cross which are the tiles above, below, left and right of the bombs initial placement. In explode4, the explosion is over and a path is drawn. Since the states move from Explode1 to Explode2 to Explode3 to Explode4 with a fixed time difference, the explosion looks like an animation. Due to the wait in the exploding states, each animation can be distinctly seen. In the explode4 state, the bomb state is changed to the Before state on default. The state machine will start changing states once a bomb is dropped once again.

11. Winning Bomberman on the FPGA

As mentioned in multiple places above, the way for any player to win is to not lose all 3 of their lives or to not hit the Hazard tile. As mentioned in the game state machine, if any player loses all of their 3 lives or hits the Hazard tile, the die_flag is set to high (which is the p1die or p2die flag in the state machine). If Player1 dies, Player2 wins and visa versa. In the P1Win or

P2Win states, the P1 or P2 win screen is outputted on the screen (in Color_mapper). The game stays in these states unless the Key0 is pressed on the FPGA. This was our Reset key. If the Reset key is pressed, the game state automatically changes to Start1 state no matter which state the game is in. Along with this, all the flags used in the game such as the die_flag or bomb_exist flags are set to 0. Also, the map is changed to the map with all the Pink Bricks (initial map). The user positions are also set to their corresponding starting positions with both X and Y motion set to 0. The game essentially restarts and this can be done any number of times.

Software Description

The software that we ran on the FPGA for our project is exactly the same as the software ran for lab 6.2. Everything for our project was implemented in verilog and did not make use of any C code aside from what we had written to complete the functions in MAX3421E.

Issues, Solutions, and Missed Features

1. Life Count

As mentioned earlier, the way we implemented the Hazard tiles was due to a bug in our max_lives. Initially, these tiles were meant to be used to increase the number of lives for the player. This was implemented by increasing the max_lives variable which is checked against the death_count variable. However, when we did this, the player would instantly die. We were unable to fix this to create this power up and hence, the logic remained unchanged but we decided to have them as a hazard tile, which increased the complexity for the individuals to win in the game.

2. Writing to Memory (logic with Pink Bricks to turn them into Path tiles)

This was one of the major issues we faced in our game. As mentioned earlier, explosions due to the bombs can cause Pink Bricks to explode and change to Paths. The way we had set up our map.v ram module was by using a mif file in which the value of 0000 indicates a path, a value of 0001 indicates an unbreakable Grey Wall and value of 0010 indicates a breakable Pink Brick. The way we thought of implementing the wall logic was to simply write to memory values that would indicate explosion states, which would end their write with a value of 0000 after the animation finished. However, we were unable to figure out a way to perform this many writes to memory correctly, you can actually see the remnants of this in our code, the explosion

temp RGB values are contained in the same section for logic that sets the other tile colors. Furthermore, we could not implement this method without running into compilation issues where our instantiated RAM modules of map.v were being synthesized away by Quartus, meaning that the output from this RAM did not determine outputs from the module it was instantiated in, even though it did influence logic elsewhere, which resulted in the RAM modules not being created in the first place. To avoid this issue we shifted strategies, we instead opted to only write 0000 (Paths) to memory after using some logic to determine whether or not the exploded tile should explode or not. This way we could use ram modules in locations that specifically would drive outputs but it meant we would have to instantiate multiple modules of map.v that were not related. The way we fixed this was by reading and writing from the same port in the RAM in Color_mapper while the Write enable signal was controlled by the explosion. If the explosion was set to occur in a block, the write enable was set to high only if the block was a Pink Brick. The data written to memory was always the value of 0 because we needed the block to always change to a path, but since the write enable was set only if the block was a pink wall, the RAM would only write in the positions of the Pink Brick that were affected due to an explosion. Passing the explosion controlled write enable out to our user1 and user2 modules, we were able to relate the instantiated RAM modules and effectively shifted our idea from having an explosion and reaction represented in memory that would be displayed on screen to having an explosion displayed on screen while its effects were reflected in memory.

3. Random Board

We wanted our board to be randomly designed. However, we weren't able to add this feature to our game. We set the positions of paths, Grey Walls and Pink Bricks using a mif file. However, if the mif file was changed, to have different positions of paths, Grey Walls and Pink Bricks, all the logic from collisions to bomb dropping to explosions to movement of the players would remain the same and the game would function in the same way as it does now. The problem with implementing this however, is the many rules that come with designing the board including, but not limited to, having the correct ratio of Grey Walls to Pink Bricks to Path tiles, making sure that the users do not get surrounded by Grey Walls, and distributing the special tiles in a way that adds to the game and does not detract.

4. Power Ups

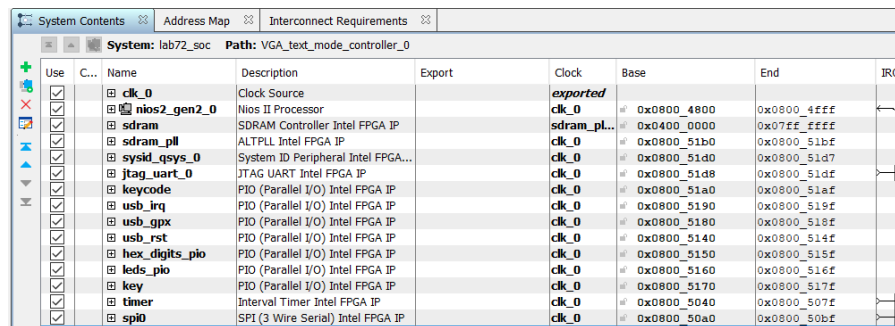
An additional feature that we wanted to include in our game were 2 power ups. One of them was for a player to gain lives and the other was for a player to increase their movement speed. As mentioned earlier in this section, we weren't able to implement the power up to gain lives and hence, this was changed to a hazard tile. As for the speed power up, we tried to increase the player's speed by 1 every time a player collected this power up. However, the increase of 1 would cause the speed to increase by a lot which made it super different for us to control the user's motions to actually play the game. Hence, we decided to not implement this power up and decided to implement the safe teleporting power up as mentioned earlier in the written description.

Design Resources and Statistics Table

LUT	6331
DSP	0
Memory (BRAM)	1016172
Flip-Flop	2871
Frequency	119.36 MHz
Static Power	96.41 mW
Dynamic Power	0.64 mW
Total Power	156.06 mW

Table 1 : Design resources and statistics table for Bomberman

Platform Designer



Use	C...	Name	Description	Export	Clock	Base	End	IRQ
<input checked="" type="checkbox"/>		clk_0	Clock Source		exported			
<input checked="" type="checkbox"/>		nios2_gen2_0	Nios II Processor		clk_0	0x0800_4800	0x0800_4fff	
<input checked="" type="checkbox"/>		sdram	SDRAM Controller Intel FPGA IP		sdram_pll	0x0400_0000	0x07ff_ffff	
<input checked="" type="checkbox"/>		sdram_pll	ALTPLL Intel FPGA IP		clk_0	0x0800_51b0	0x0800_51bf	
<input checked="" type="checkbox"/>		sysid_qsys_0	System ID Peripheral Intel FPGA...		clk_0	0x0800_51d0	0x0800_51d7	
<input checked="" type="checkbox"/>		jtag_uart_0	JTAG UART Intel FPGA IP		clk_0	0x0800_51d8	0x0800_51df	
<input checked="" type="checkbox"/>		keycode	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0800_51a0	0x0800_51af	
<input checked="" type="checkbox"/>		usb_irq	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0800_5190	0x0800_519f	
<input checked="" type="checkbox"/>		usb_gpx	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0800_5180	0x0800_518f	
<input checked="" type="checkbox"/>		usb_rst	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0800_5140	0x0800_514f	
<input checked="" type="checkbox"/>		hex_digits_pio	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0800_5150	0x0800_515f	
<input checked="" type="checkbox"/>		leds_pio	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0800_5160	0x0800_516f	
<input checked="" type="checkbox"/>		key	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0800_5170	0x0800_517f	
<input checked="" type="checkbox"/>		timer	Interval Timer Intel FPGA IP		clk_0	0x0800_5040	0x0800_507f	
<input checked="" type="checkbox"/>		spi0	SPI (3 Wire Serial) Intel FPGA IP		clk_0	0x0800_50a0	0x0800_50bf	

Figure 4: Platform Designer of our Final Project

The Platform Designer for Week 1 and Week 2 of Lab 7 is essentially the exact same, the only difference is that the address space of the memory mapped slave increases from 10 - 12 bits. Otherwise it is the same.

Our Platform Designer was adapted from Lab 6 so many modules are unnecessary yet present.

i. Modules

clk_0: This module is a 50 MHz clock created by the FPGA that provides clock and reset inputs to other modules

nios2_gen2_0: This module is the 32-bit processor, NIOS II/e, e for the economy, this processor is an optimized version of the full-fledged NIOS II/f, the processor gives out data and instruction as an output to other modules

onchip_memory2_0: This module is the on-chip memory, it can be used for higher speeds and quicker access to data.

SDRAM: This module is the off-chip memory, it is 512 MBits and takes instructions and data from the processor. This module unlike the others gets its clock from the sdram_pll module which accounts for the difference in phase between the master and slave clocks.

sdram_pll: This module is a clock created for the SDRAM, that has a 1ns delay

sysid_qsys_0: This module is used to verify that the loaded software on the FPGA is allowed to run on the hardware provided, ensuring that the FPGA is not incompatible with the code.

jtag_uart_0: This module is used to communicate a serial bit stream between the FPGA and connected computer

keycode: This module is a Parallel I/O (PIO) that is used to identify which key on the keyboard has been pressed, it is 8 bits wide

key: This module is a 2-bit PIO that is used to represent the two buttons on the FPGA

usb_irq, usb_gpx, usb_rst: This module is a 1-bit PIO that is used to connect the keyboard to the FPGA

hex_digits_pio: This module is a 16-bit PIO that is used to output the keycode on the Hex Display

leds_pio: This module is an 8-bit PIO that is used to output to the leds above the switches on the FPGA

timer: This module is used to track time for NIOS II

spi0: This module is used to transfer data between the attached peripherals and the FPGA, and controls which peripheral is being read from or written to or if the FPGA is being read from or written to

Conclusion

The final project was a great deal of fun, we appreciate the free reign we were given regarding what we could choose to do. As a team we are very pleased with the level of functionality we were able to achieve in 4 weeks. For this project some of the labs that were particularly useful were Lab 6 and Lab 7. Lab 6.2 was particularly helpful in understanding character movement, collisions and how keycodes came through as signals on the FPGA from MAX3421E. Lab 7 helped us to understand how we could modify and expand the capabilities of the Color_mapper module to set the pixels on the screen by manipulating DrawX and DrawY in very clever ways. Even though we are proud of what we have accomplished in the 3-4 weeks that we have been working on this project, we do wish that we were able to deliver all of the initially promised features. If we were to have an additional week to work on the project, one of the first things we would change is moving the life counter to be displayed on screen instead of being displayed on the FPGA Hex Display, alongside that we would work to implement Powerups, Save/Load Game functionality, and Sound into our game. Overall, we successfully created a version of the 1983 NES classic Bomberman that runs on an FPGA.

Appendix A

Module: bomberman.sv (similar to other labs)

Inputs: MAX10_CLK1_50, [1:0] KEY, [9:0] SW,

Outputs: [9:0] LEDR, [7:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, DRAM_CLK,
DRAM_CKE, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N,
DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS [3:0] VGA_R, VGA_G, VGA_B,
[12:0] DRAM_ADDR, [1:0] DRAM_BA

Inout: [15:0] DRAM_DQ, ARDUINO_IO, ARDUINO_RESET_N

Description: This is the top-level module used.

Purpose: The purpose of bomberman.sv is to create common logic and instantiate the modules required for our game

Module: VGA_controller.sv

Inputs: Clk,

Outputs: hs, vs, pixel_clk, blank, sync

[9:0] DrawX, DrawY

Description: Handle vertical and horizontal sync while setting a DrawX and DrawY

Purpose: The purpose of VGA_controller is to shoot the “electron gun” across the screen and determine the hs and vs to draw pixels accordingly

Module: hexdriver.sv

Inputs: [3:0] In0,

Outputs: [6:0] Out0

Description: Takes a nibble or 4 bits and converts it to a 7 bit equivalent that is displayable on the HEX# output LEDs of the FPGA.

Purpose: To allow the seven segment displays to display what the input and output of our code are.

Module: Color_Mapper.sv

Inputs: Clk, blank,

[9:0] user1X, user1Y, bomb1X, bomb1Y, bomb1XS, bomb1YS, DrawX, DrawY,

[9:0] user2X, user2Y, bomb2X, bomb2Y, bomb2XS, bomb2YS, [4:0] game_state,

[3:0] bomb1_state, bomb2_state

Outputs: [7:0] Red, Green, Blue

Description: Uses the positions of the two users, two bombs, and the state of the game to decide what needs to be outputted to the screen

Purpose: Displays all elements of the game out to a monitor

Module: bomb.sv

Inputs: Reset, frame_clk, make, [9:0] userX, userY, [3:0] bomb_state

Outputs: bomb_check, [9:0] bombX, bombY, bombXS, bombYS,

[9:0] explore_addr [5], [0:0] explore_flag [5], [3:0] explore_data [5]

Description: Instantiated twice for the two users. Determines the position of the bomb based on the specific key being pressed. Depending on the position of the bomb, decides which positions on the board need to be affected due to the bomb.

Purpose: Used to place bombs for both users which is later indexed to perform an explosion animation as well as a write to memory to update the board

Module: User1.sv

Inputs: Reset, frame_clk, [7:0] keycode, [4:0] allow,

[9:0] bomb2X, bomb2Y, bomb2XS, bomb2YS

Outputs: bomb_drop, collide, [9:0] userX, userY, [3:0] wall_out

Description: Determines the position and motion of the first user depending on which key is pressed on the keyboard. This module updates the motion and position of the ball when a key is pressed, or if the user collides with a wall, or if the user is caught in a bomb explosion.

Purpose: This module controls the motion and actions of user 1.

Module: User2.sv

Inputs: Reset, frame_clk, [7:0] keycode, [4:0] allow,

[9:0] bomb1X, bomb1Y, bomb1XS, bomb1YS

Outputs: bomb_drop, collide, [9:0] userX, userY, [3:0] wall_out

Description: Determines the position and motion of the second user depending on which key is pressed on the keyboard. This module updates the motion and position of the user when a key is pressed, if the user collides with a wall, or if the user is caught in a bomb explosion.

Purpose: This module controls the motion and actions of user2.

Module: state_machine.sv

Inputs: Clk, Reset, p1die, p2die, [7:0] keycode

Outputs: [4:0] state, [7:0] count_out

Description: State machine for our entire game, goes through the start states, continue state, pause state, and the player1 or player2 die state depending on which player has died

Purpose: Used to delegate game sections that corresponded to different screens showing the start of the game, the actual game, a pause game feature, and the player1 or player2 win screen feature and to only resume control of the game when in the “continue” state (and not in the start/pause/Win state).

Module: bomb_statemachine.sv

Inputs: bomb_exist

Outputs: [8:0] state, [7:0] count_out

Description: State machine for our bomb explosion animations. Goes through states like before, exist and 4 explode states, each with a wait state

Purpose: Used for timing and displaying bomb explosions with animation states.

Module: background.mif

Description: Consists of 300 values, each a 4 bit value. Values in the mif: 0 - blue paths, 1 - Grey Walls, 2 - Pink Bricks, 3 - special tiles: death zone, 4 - special tiles: teleport

Purpose: This was used to create our entire playing field with different types of tiles including unbreakable walls (grey tiles), breakable walls(Pink tiles), teleports (heart tiles) hazard tiles (fire tiles), and paths (blue tiles). Was used to instantiate a RAM module that was read by Color Mapper.

Module: map.qip

Inputs: rden_a, rden_b, wren_a, wren_b, Clk, [9:0] address_a, address_b, [3:0] data_a, data_b,

Outputs: [3:0] q_a, q_b

Description: Dual port RAM created using the mif file. This was created using the IP catalog.

Purpose: This RAM is used to read from and write to memory the data for creating the map in the game. This is also used for the collision logic in the game.

Module: every XYZ_ram.sv

Inputs: [18:0] read_address, we and Clk

Outputs: data_out \Rightarrow length of this output depends on the number of colors for the particular sprite

Description: Uses readmemh to read data about the colors from a txt file for a particular sprite

Purpose: Used to create sprites for many different components in the game such as the 2 users, bombs, and different kind of walls etc.

Special Thanks:

Very special thanks to Ian, Anushya, Zayd, Kevin P, and Ryan for helping us despite our ineptitude at times. And very special thanks to Dohun for being a wonderful TA. This semester was fun and challenging, and I learned a lot. I hope future students can have the same experience and I hope they'll enjoy it as much as I have. :)

- SMMM