Bomberman
EEL5722C
Report
Final Project

Esperandieu Elbon II

Contents

Figures

Milestone 1) Animate Bomberman & Speed Power-up: Frame Timer & ROM Offset Registers

For this section, I followed the advice and instruction of the lab manual. With it, I created the animation and frame timers necessary to animate Bomberman. I don't think anything I have to say of the implementation that would add much to this report, so I'll keep it short.

Under lab manual guidance, I added in the animation frame timer. It's simply a register that is ticked each time any button is pressed while its limit hasn't been reached. Further implementation can be found in my "bomberman_module.v" file in lines 177-189. I also added in the appropriate logic to index into the bomberman sprite based on the frame timer. It uses an always block and follows the 121312131213... pattern outlined within the manual. Finally, I instantiated the sprite ROM and assigned the outputs of the module. I had some struggles conceptually that bled into implementation issue. I couldn't understand how the left pattern would be shown. Because of this, I spent time looking for a sprite image that didn't exist of a left facing bomberman. After contemplation and looking a bit closer, I figured it out. Below are the main changes I made to the file.

sources_1 > imports > HDL_Incomplete > bomberman_module.v

```
177 //************************************************************ ANIMATION FRAME TIMER ************************************************************
178
179 always @(posedge clk or posedge reset) begin
180     // Reset logic.
181     if(reset) frame_timer_reg <= 0;
182
183     // Update animation frame timer.
184     else frame_timer_reg <= frame_timer_next;
185 end
186
187 assign frame_timer_next = ((L | R | U | D) &  (frame_timer_reg < FRAME_REG_MAX)) ? frame_timer_reg + 1 : 0;
188
189 //******************************************************** REGISTER TO INDEX INTO SPRITE ROM ********************************************************
190
191 always @ (frame_timer_reg or cd) begin
192     if(frame_timer_next < FRAME_CNT_1) begin
193         if(cd == CD_U) rom_offset_next <= U_1;
194         else if(cd == CD_D) rom_offset_next <= D_1;
195         else if(cd == CD_L) rom_offset_next <= R_1;
196         else if(cd == CD_R) rom_offset_next <= R_1;
197     end
198     else if(frame_timer_next < FRAME_CNT_2) begin
199         if(cd == CD_U) rom_offset_next <= U_2;
200         else if(cd == CD_D) rom_offset_next <= D_2;
201         else if(cd == CD_L) rom_offset_next <= R_2;
202         else if(cd == CD_R) rom_offset_next <= R_2;
203     end
204     else if(frame_timer_next < FRAME_CNT_3) begin
205         if(cd == CD_U) rom_offset_next <= U_1;
206         else if(cd == CD_D) rom_offset_next <= D_1;
207         else if(cd == CD_L) rom_offset_next <= R_1;
208         else if(cd == CD_R) rom_offset_next <= R_1;
209     end
210     else if(frame_timer_next < FRAME_CNT_4) begin
211         if(cd == CD_U) rom_offset_next <= U_3;
212         else if(cd == CD_D) rom_offset_next <= D_3;
213         else if(cd == CD_L) rom_offset_next <= R_3;
214         else if(cd == CD_R) rom_offset_next <= R_3;
215     end
216 end
217
218 always @(posedge clk or posedge reset) begin
219     // Reset logic.
220     if(reset) rom_offset_reg <= U_1;
221     // Update rom offset.
222     else rom_offset_reg <= rom_offset_next;
223 end
224
```

Figure 1: Changes to Bomberman.v File

Milestone 2) Bomb, Explosion FSM, and Radius Power-up

The bomb and explosion finite state machine was an interesting challenge, significantly more so than implementing animation on the bomberman. I ended up adding about 100 new lines of code to work with it. Unfortunately, the implementation I am submitting is not fully functional due to a bug that I couldn't find in time . Furthermore, what makes it worse is that through the entire week of struggling to work on this project between Final Reports and Exams, it worked properly until I added in the write section to my block module!

To get a functioning bomb, I needed to fill out the state machine given in the manual. I added the necessary bomb and explosion timers (very many clock cycles they both take) and then I began work on the FSM to drive it. There isn't much to say in the way of implementation that differs significantly from the state machine laid out in the manual. The only difference to see is the beginnings of the implementation of the Blast Radius Expansion power-up (something that I spent a considerable amount of time on).

I was unable to complete implementation of the power-up but the workings were all in my mind and they are as follows.

1. Successfully implement the power-ups and make sure they show on the screen (through logic implemented in the block module that I will discuss later)
2. Make it so that Bomberman detects whenever he passes over a power up and detect whenever he passes over this specific power send a signal from the block module to the bomb module to notify the bomb that its radius has expanded.

I was able to add this signal in time and I also added the beginnings of the workings of the signal in the block module. In the bomb module the signal would simply increment a register that is holding the value for the bomb's radius.

So, in sum every time the Bomberman would pass over the increased radius power up, the radius of the bomb would be incremented by 1 via the signal being sent from the block module to the bomb module. Within the bomb module, I decided to implement loop logic within the state machine using an internal index register that runs over the range of the bomb. This range would be defaulted to 1 for obvious reasons and it would be increased every time he walks over a power up.

Within the state machine, for the four explosion directions, if the range is more than 1 then the state machine remains in the explode state corresponding to that direction, incrementing not only the index but also the explosion address and it would with respect to the proper direction, moving 16 pixels in to make sure that it goes to the next block.

Implementing this state machine did not pose significant difficulty, it just took time as this is actually the first time I have been presented with directly implementing state machines in Verilog and implementing them in this way. So, before using the logic provided with the skeleton code I tried using different logic and the timing was all wrong. After some research I figured it out and using both the Combinational and Sequential always blocks, I had a state machine. Adding to disappointment that might be building as you skim this report, I was also unable to implement the exploding walls

because I ran out of time. Below is an example of one of the explosion states that demonstrates the sort of while loop logic I was talking about.

```
// Left hand side of the explosion occurs.
exp_left    :  begin
   // Remain in left explosion until range runs out.
   if(exp_index_reg != exp_range_reg) begin
      exp_block_addr_next = (bomb_x_next - 1 - exp_index_next*16) + bomb_y_next * 33;
      exp_index_next = exp_index_next + 1;
   end

   // Reset explosion index and leave left.
   else begin
      exp_index_next = 0;
      exp_block_addr_next = 0;
      bomb_exp_state_next = exp_right;
   end
end
```

Figure 2: Explosion State While Loop

Milestone 3) Animate Power Ups

Animating the powerups was the most difficult part of this project for me. I was unable to complete the animation. I spent most of my time on this section and the only things I was able to succeed was actually getting the power-up to show on the screen. Below is an explanation with pictures of my detailed plan that I was not able to implement because I ran out of time as well as thought process and struggles. As a preface, I used simulations and timing diagrams as a reference to try and figure things out and they were unhelpful do to the sheer amount of TIME that they took., that is until I realized that I was better off removing the push button input from the debouncing modules and decreasing the counters for the places of importance while keeping them the same for actual physical testing (VGA strict timing). Also, before I implemented all of the below. I should mention that I needed to port my VGA module to work properly with this project and as specified. All I changed were a few input and output names, but it worked fine and to that end, I will not discuss it further here. The below is a screenshot of a simulation run to verify the outputs and response of the Read State machine within the block module. It doesn't show much right now, zoomed out. It's placed here to prove that I was working the entire time, and that this late submission isn't for nothing.

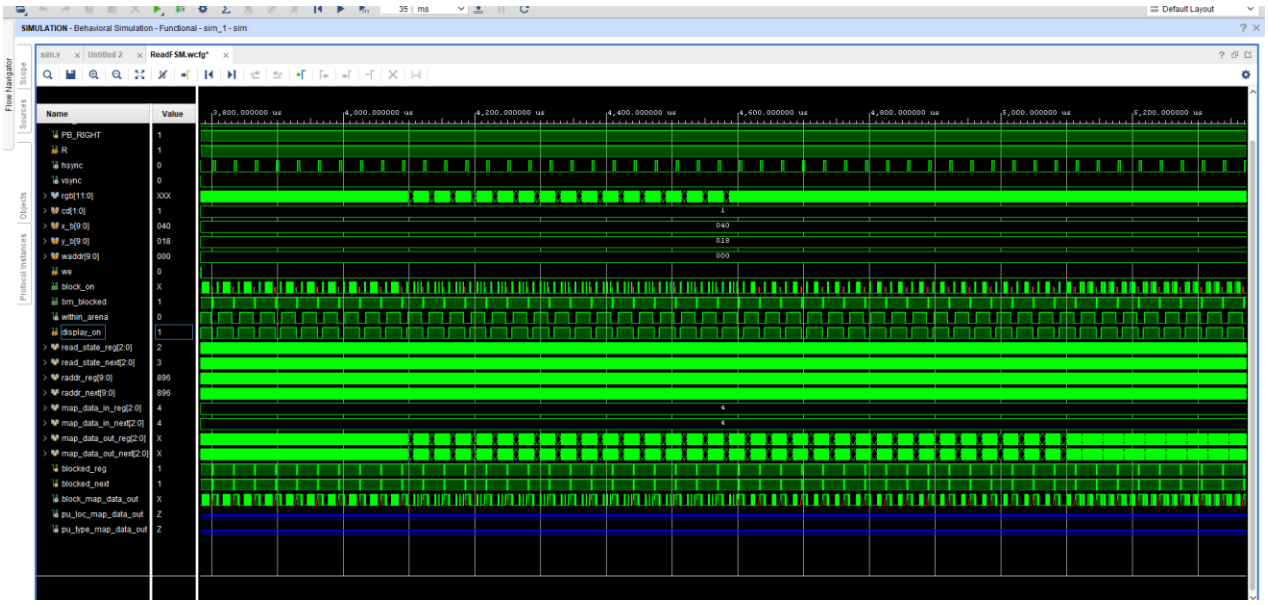


Figure 3: Block Module Read State Machine Simulation

In my pursuit of animating the powerups, I made significant annotations and edits to the block module file. I decided that in order for me to show power ups I needed to be able to track power ups and So what better way to track them and the bounds of the arena grid that we had then to use the arena grid that we had? There were two options before me: I could either expand the size of the grid and the .coe file that held it by adding two more dimensions, one representing if there was a power up at that location and the other representing the type of power up at that location, or I could save myself the trouble and simply implement 2 empty distributed RAMS that I could read and write to the same information. I did the latter and initialized them with 0s to indicate no power ups.

The image below shows what I conceived each map would show. You can see that the block map shows exactly what the block map was intended to show via the lab manual. The first map I added was the power up location map to show where power ups were (a 1 would indicate that there was a power up present at that address and a 0 would indicate that there was no power up). The 2nd map I added was to track power up type (a 1 indicates that it is a speed power up and a 2 indicates that it is a range increase power up). Each map requires its own input and its own output in order to show the proper information with respect to read and write addresses. To that end I used 2 global registers (one for data input and one for data output) so that I could simplify data transactions between these maps and my read and write addresses. I also came up with a couple of bit masks to identify the fields that I needed within this register. I will not explain here how bit masks work. An example of how I use them can be found in my code for the block module in the continuous assignment section starting at line 103.

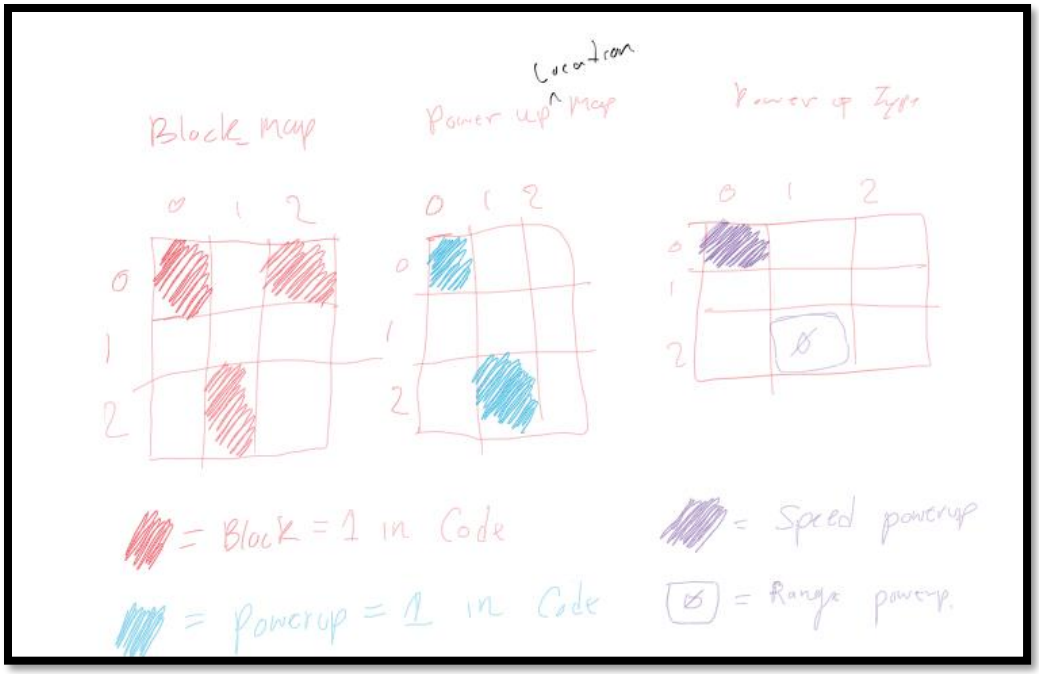


Figure 4: Description of Map Layout.

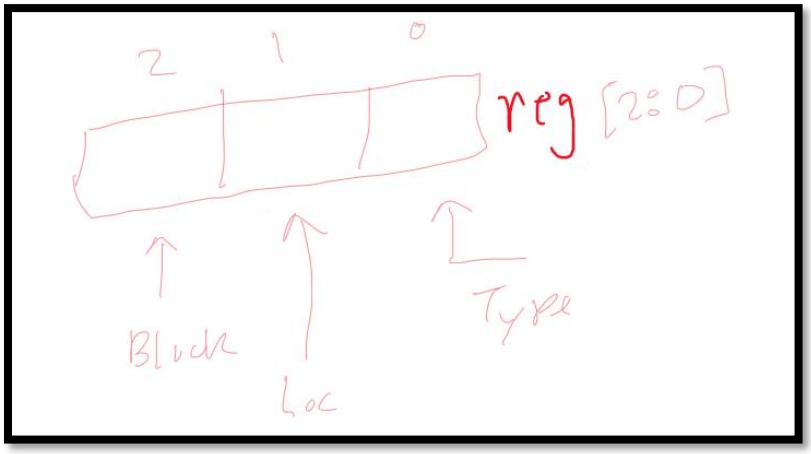


Figure 5. Description of Global Register Layout.

After deciding how I would get data with respect to power ups, I needed to decide how to show them. The code present in the skeleton was not that helpful for me personally and I ultimately ended up rewriting pretty much the entire module save the signal assignments into two distinct state machines that work together: the read state machine and the write state machine.

They both do what they sound like they do. The current read state machine works pretty much the same way that the module worked without it, to me the difference is that it is able to read from multiple addresses and multiple maps. It has four states: idle, set read address, read, and process

data output. Due to its similarity to the already present solution code I did not need to draw a state diagram in order to complete it. An interesting tidbit is that outside of assigning the blocked signal for bombermans movement in my process data out state for this state machine I also assign a power up signal that was intended to be active whenever the next block that the Bomberman was facing was a power up. This is a signal that I was speaking about earlier with respect to notifying the bomb module that it had its range increased via power up.

Of note is a previous iteration of the state machine for reading that I made. It is shown in the diagram below. Alongside the write state machine, it waited for the write enable signal from the bomb that signaled an explosion. Whenever this signal came, the read state machine would go into a wait state where it did nothing but wait for the write state machine to signal that it was finished writing. This presented problems in the video output as that extra couple clock cycles not to read the block rgb output presented all sorts of undefined behavior. There were points in time where blocks filled the screen as the Bomberman walked into a blocked state, and points where a bombs explosion would completely empty the screen, leaving nothing but the RGB output of the bombs explosion and the pillars in the background. Below is a picture of the first case because I thought it was interesting and it was the point in time at which I realized that I was not finishing this project on time so I thought it prudent to share in the report.

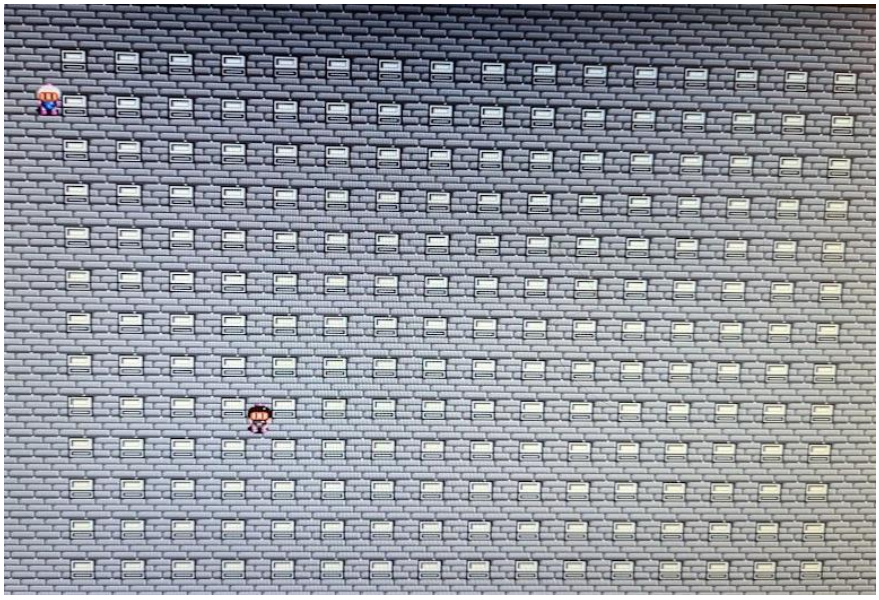


Figure 6: Undefined Behavior due to Faulty State Machine Logic.

The write state machine adds functionality that was not present in the skeleton code: it allows for us to write to the block map and it allows for us to write to the other two maps as well. I went through two designs before I had to stop working on this state machine. The first design had 4 states: idle, decide, write, and done. The idle state would be holding the state machine until it received the write enable signal from when the bomb module signaled that an explosion was happening. The decide state would decide whether or not what was written to the 3 maps was a power up or not.

This is where I made a mistake: in my write scheme, I deemed it intelligent to write not only to the power up location and type map but also to the block map. I did this so that I would be able to edit the top module and block module definitions as less as possible and to work within the constraints and confines of the skeleton code as closely as possible. My intention was to not have to write as much and in doing so I ended up writing a lot and wasting a lot of time, dealing with incorrect bit assignments and whatnot. Finally after deciding what would be written, the state machine would go into its write state where it would take a clock cycle to write the data to the maps with the RAMs (even though they are pretty much available on the fly after setting the address). The done state is unnecessary and it was only present to signal to the read state machine that the right machine was done.

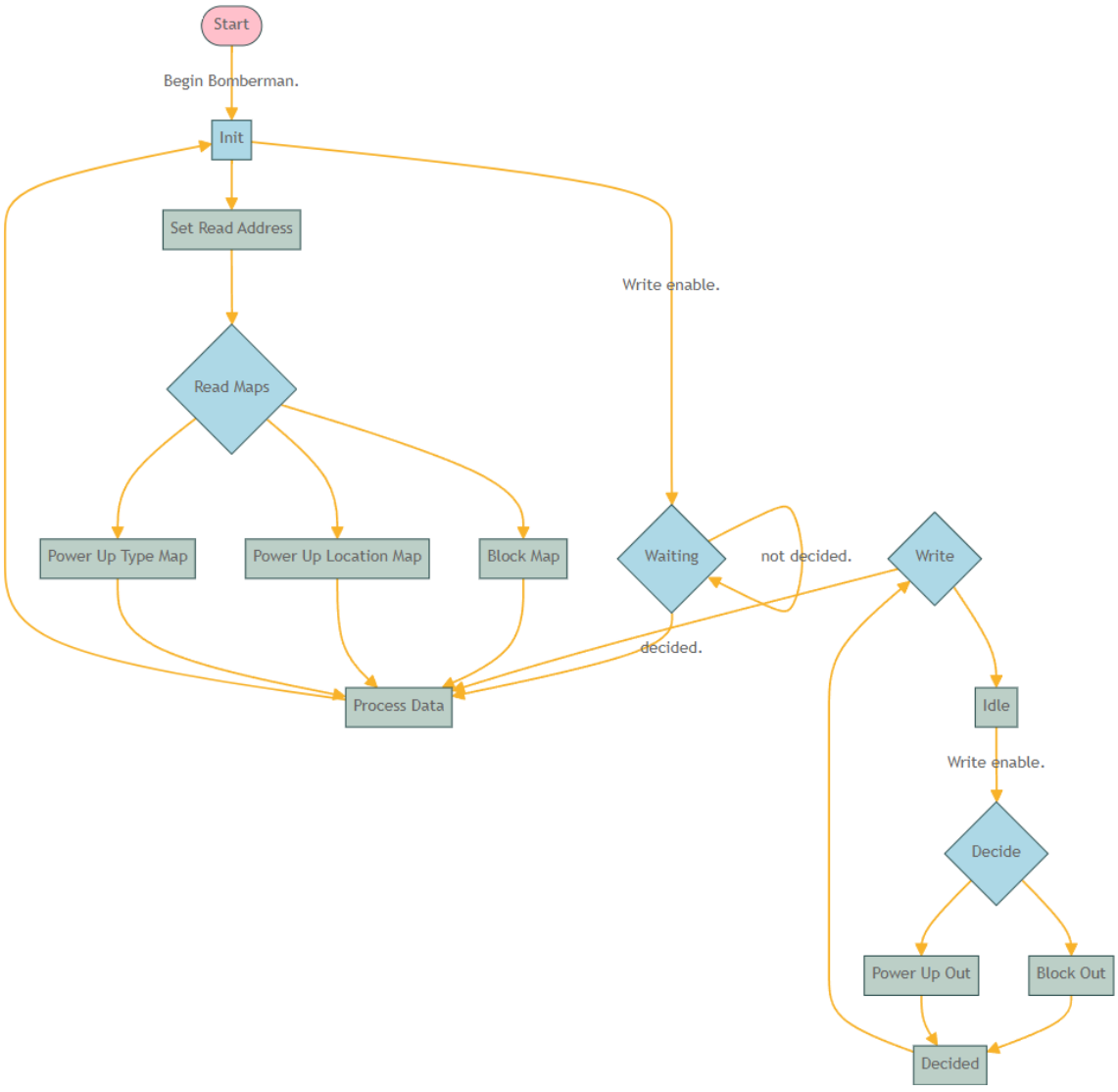


Figure 7: Initial Write State Machine

Speaking more about the write schemes with respect to the write state machine, below is a picture of a table that I drew in order to help me understand what fields I needed to assign my global registers to properly write and read block map data output, power up location data, and power up type data. It is from this table that the bit masks were derived that I use in my code. It has explanations nearby and highlights and annotations to show why specific choices were made. As a quick summary the table basically says that whenever I read the global register, the most significant bit dictates the presence of blocks. If that bit was not set, there were no blocks present. Bit 1 specifies power up location and due to how I read things, if the global register has bit 2 set (only in cases where the MSB is not 0), this indicated the presence of a power up. Finally, the least significant bit indicated what kind of power up was present: if the bit was 1, a speed power up was present and if not, it was a bomb increased range power up.

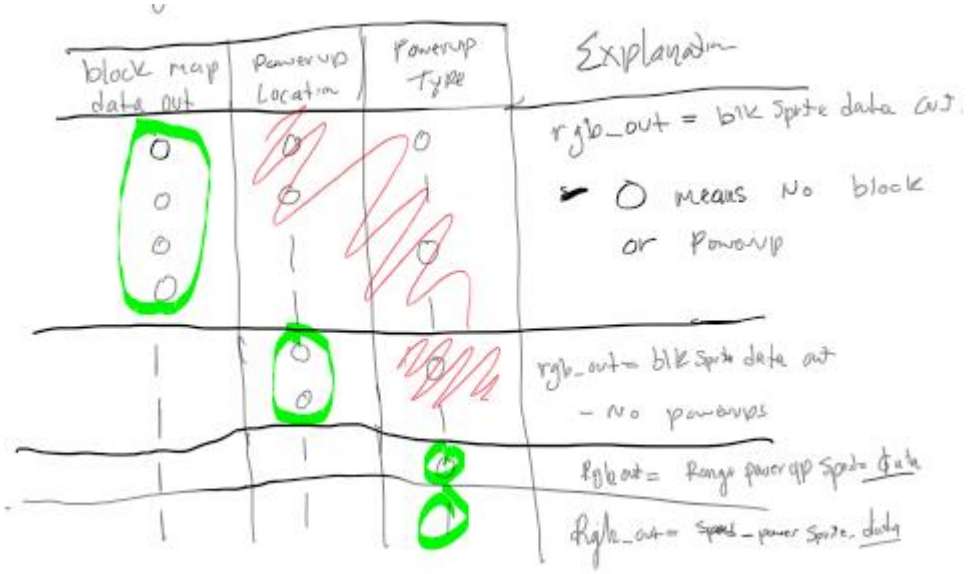


Figure 8: Bit Value Representation of Map States.

Below is the timing analysis that prompted me to redesign the write state machine. As I noticed when I went back and tried to implement the increase range power up for the bomb, the bomb sends its write addresses for writing the explosion right one after another, one cycle after sending the write enable signal. What this meant was that if the bomb had a radius of 1, the bomb state machine would send the write addresses within 4 clock cycles. If it was increased to five, it would take 20 clocks.

This gave me significant pause as I struggled against redesigning the skeleton code. Leaving the bomb state machine as undisturbed as possible, I shortened my write state machine into two states: idle and write. The idle state did the same thing that it did in the previous state machine and the write states combine the previous decide and write states, banking on the fact that the write address for the first explosion is sent in the clock cycle immediately after the write enable was sent, thus allowing it to be grabbed in the initialization state where the write state machine transitioned to

the write state. The decision was made within the write state and the writing was done within the write state. This cycles as necessary.

Of note, I saw it during my testing: I realized my frequency for generating power-ups was too high, so I lowered it artificially by setting the limit to 1 power up per explosion. This means if an explosion had a radius of 1, 2, 3, 4, …, it will always be only one power up that would result from that explosion. Speaking more on the decision to choose one power up type over the other, I simply use the linear feedback shift register that we were required to make to randomize the movement of the enemy in randomizing the power up choices as well.

The last work that I did on this module in order to animate power ups was to take in the bombs coordinates within the arena grid from the bomb module so that I could use that input within the block module and figure out whether or not the blocks that were hit by the explosion within the range were bricks, empty spaces, or pillars, a decision that would inform how far the explosion would go.

The powerups do show up on the screen – it shows up in my demo video. Intentionally I made it so that if a block is a power up, you are not blocked and you are allowed to pass through it, a result of the way that I read the power of location map. Also due to the fact that in the end I had intended to use the power up location map in conjunction with the power up type map to tell the bomberman that he had picked up a speed power up via the bomberman module or again, tell the bomb that it had picked up a range increase power up.

Figure 9: Bomb and Write Module Interconnected Timing Diagram.

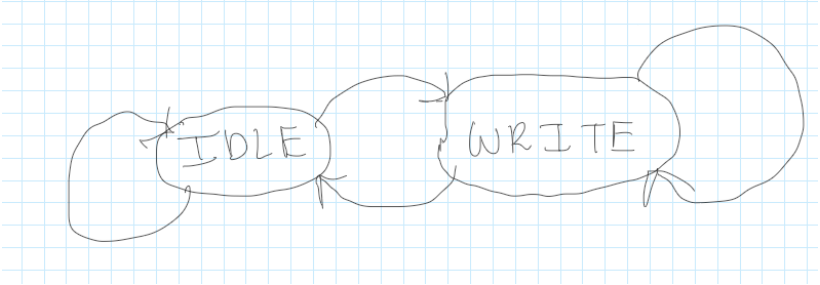


Figure 10: Updated Write State Machine

Milestone 4) Linear Feedback Shift Register

Creating the linear feedback shift register for enemy Sprite movement was pretty simple. After a quick Wikipedia search to figure out what it was, I got to coding using the example that Wikipedia gave as well as examples of non-Verilog C\C++ code to guide my hands. I use the polynomial specified in Wikipedia: x^16 + x^14 + x^3 +x^11 + 1, taps are at 16, 14, 13, 11. It is through the taps that the "randomness" is applied. As requested within the module prototype definition from the lab manual at right input was provided I assume to change or reassign the seed for the feedback shift register. I don't use it in my code.

Below is the code (just because) but, more importantly, the simulation files requested showing the functioning of the BCD from 0 to 9999.

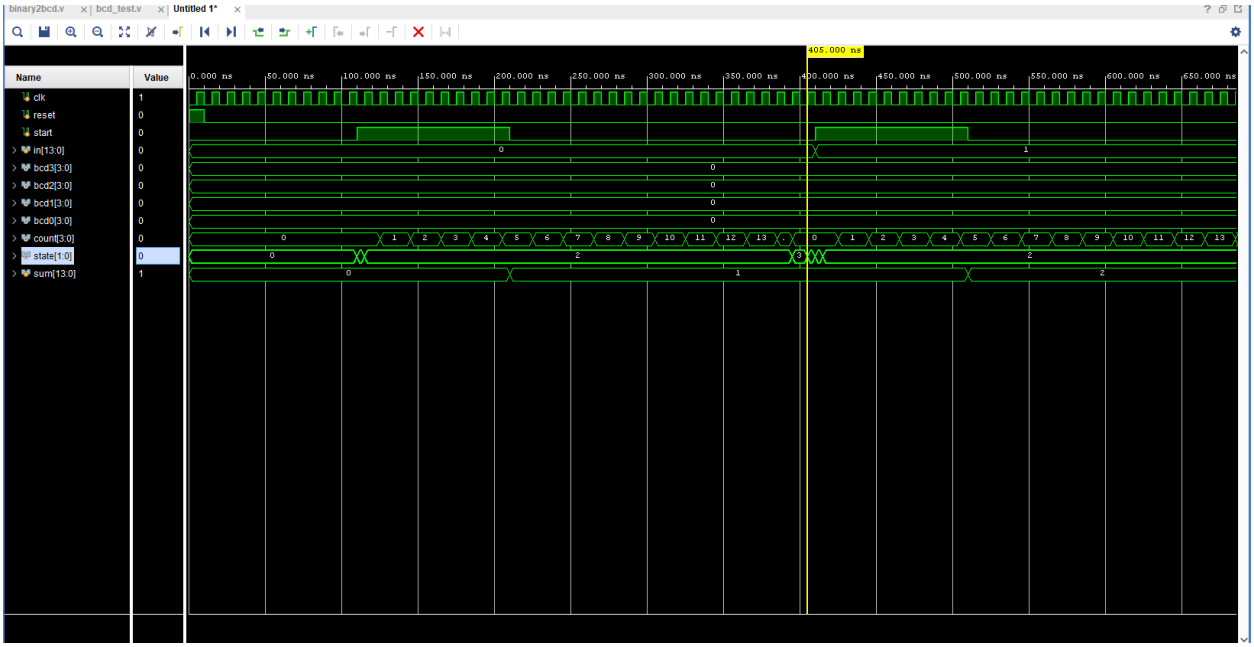


Figure 11:Proof of BCD (Displaying 0000).

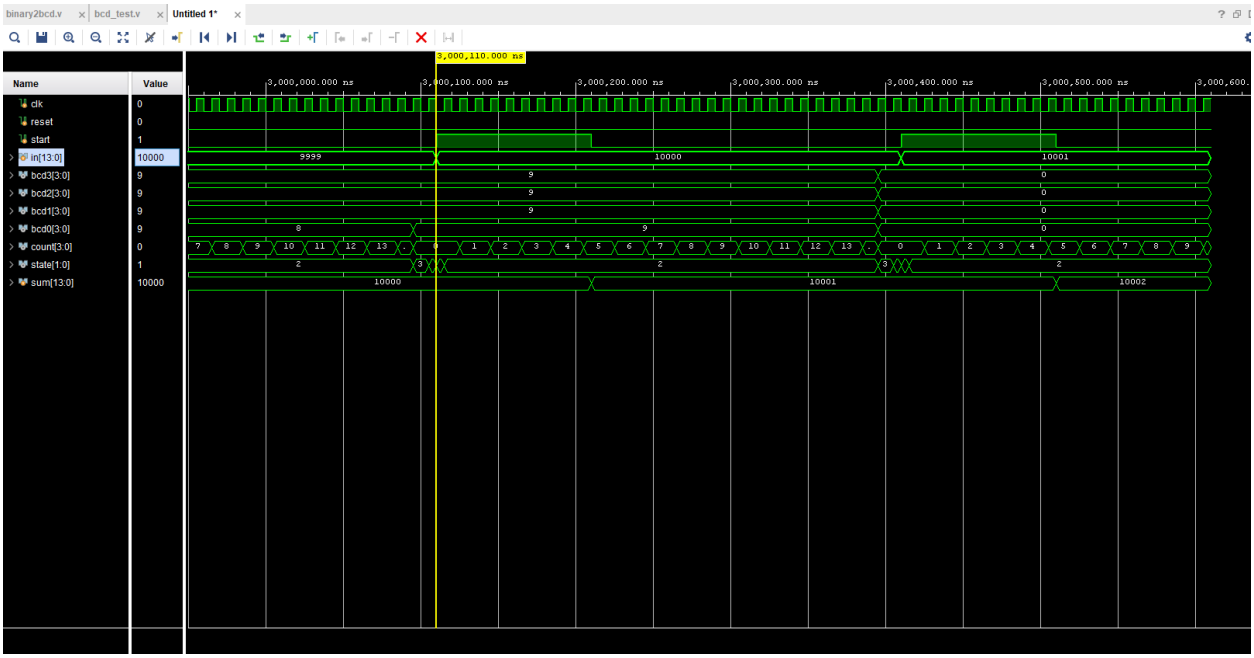


Figure 12: Proof of BCD (Displaying 9999).

```
/*
Linear Feedback Shift Register for Enemey Sprite Movement.
Using the polynomial x^16 + x^14 + x^3 +x^11 + 1,
taps are at 16, 14, 13, 11 .
*/
module LFSR_16 (
    input wire clk, rst, w_en,
    input wire [15:0] w_in,
    output wire [15:0] out
    );

    // Internal signals.
    reg [15:0] out_reg;

    // Assign the feedback and output.
    wire feedback = out[15] ^ out[13] ^ out[12] ^ out[10];
    assign out = out_reg;

    // Provide output or perform shift. Reset if needed.
    always @(posedge clk or posedge rst) begin

        // Initialize to defualt value.
        if (rst) out_reg <= 16'h4447;

        // Provide output if write enabled.
        else if (w_en) out_reg <= w_in;

        // Shift and feedback.
        else out_reg <= {out_reg[14:0], feedback};
    end

endmodule
```

Figure 13: Linear Feadback Shift Register.

Milestone 5) Enemy Finite State Machine

After creating the linear feedback shift register, it was time that animate the enemy and its movement. The state machine for this was similar to Bomberman state machine. Using the lab manual the regret of the same machine given within my own code with no significant differences between the two. It works as expected and is shown in the demo video with no issuses.

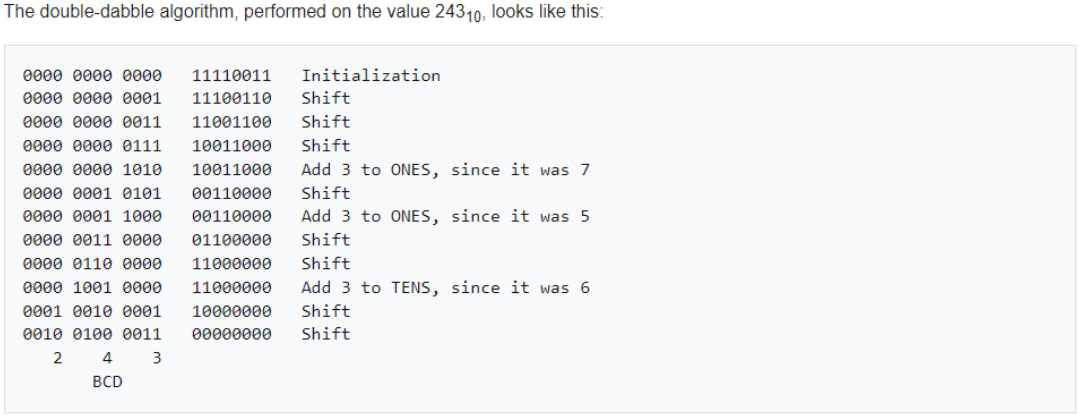
There's very little difficulties in implementation to speak of. It was primarily difficulties understanding servicing logic because like I said before I had not seen the same machines implemented in this way and so it was confusing.

This was a very uneventful section of programming the project until the implementation of the LSFR to randomly move the sprite. I cannot add much to the discussion that wasn't already in the report wrt to this module and that I haven't completed.

Milestone 6) Binary to BCD

Now this is the point where we remember we are making a game and there are scores. For the score display module we were tasked with implementing the binary to BCD converting module that took in the counter input by represented the number of points the user might have accumulated, and I'll put it as four separate binary coded decimals for display on the top of the screen. The implementation of this register I thought it was initially trivial. However, as I got to work on it I was presented with a challenge and so again I went to my good friend Wikipedia and followed an algorithm that they laid out in order to display the BCD. The algorithm I followed is known as a double dabble algorithm or the shif-and-add-3 algorithm and I'm genuinely surprised that I did not learn in any of the computer engineering courses that I've taken out of this university however I did learn how to display binary coded decimals in other ways so it's fine.

Very simply the algorithm, the algorithm applied to our 14-bit input says that you take your input, shift it over by 1 to the left, and then if any of your projected binary coded decimals add up to five or more you add 3 to that number. Shifting is continued until you have 4 fields of binary coded decimals. Below is wikipedia's example that I use to facilitate my construction of this module.


The double-dabble algorithm, performed on the value 243_{10}, looks like this:

```
0000 0000 0000   11110011   Initialization
0000 0000 0001   11100110   Shift
0000 0000 0011   11001100   Shift
0000 0000 0111   10011000   Shift
0000 0000 1010   10011000   Add 3 to ONES, since it was 7
0000 0001 0101   00110000   Shift
0000 0001 1000   00110000   Add 3 to ONES, since it was 5
0000 0011 0000   01100000   Shift
0000 0110 0000   11000000   Shift
0000 1001 0000   11000000   Add 3 to TENS, since it was 6
0001 0010 0001   10000000   Shift
0010 0100 0011   00000000   Shift
   2    4    3
      BCD
```

Figure 14: Double Dabble Algorithm

Milestone 7) Bomberman Life Bar

A sad story indeed the Barber man life module is. I got to this point earlier this week maybe on around Tuesday night where I thought I was good for the project and thought that I would finish everything on time so I took my time I spent time that I should not have spent creating a beautiful life bar for Bomberman. Here it is below all of its 20-by-48-pixel glory:

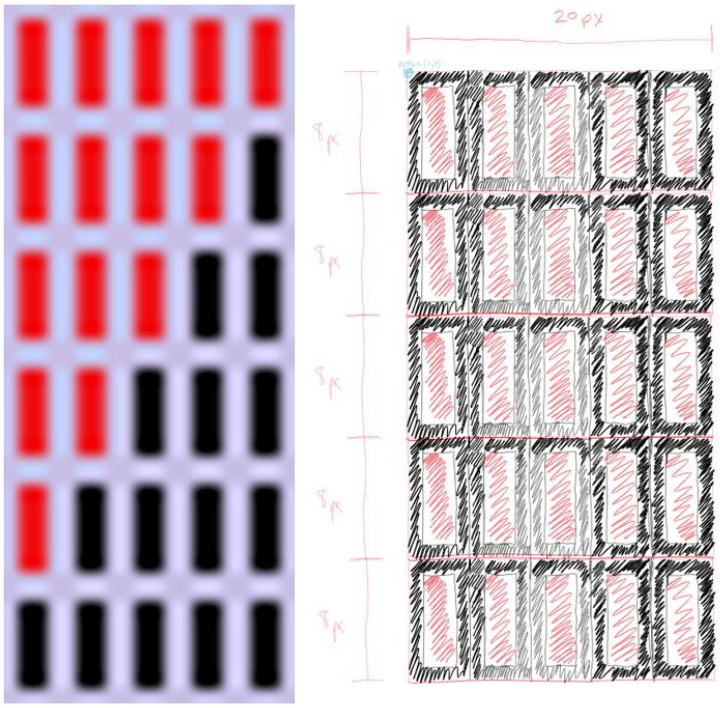


Figure 15: Unimplemented Life Bar

As you can see there are six life bars in the image: the top one has full red as specified in the lab manual and the bottom has full black. The idea was to load this image into a block ROM, much like we've done every other image in this project, and index through it changing state each time the Bomberman lost life. After realizing the gravity of my situation through the running of an unlikely test, I decided to put this on the back burner as it was only worth 5% of the grade and enough for it to complete it on time maybe I should have spent more time on this to actually have something nice to show in my project. The implementation plan for this was that I would attach it to the game lives module and use the signal within the games lives module that controlled the life bomberman had remaining to.

Milestone 8) Creative Expansion : Pause Signal

Finally, we have come to our creative expansion: this was the best part of the project, I enjoyed this a lot. All I had to do to implement this was go to the parts of the code that took live feedback. This meant I had to find a way to effectively shut out user input as if the user is unable to input anything then the game will not run as it is effectively paused, I had to stop the enemy from moving, and this meant that should the user pause the game after a bomb is placed or after an explosion begins, the bomb or explosion remains on the screen indefinitely until the game is unpaused.

After adding a switch to the constraints file and adding the pause signal from a switch to the top module, I added the pause signal to three modules that deal with what I discussed earlier: the debounce module used for debouncing buttons and provided by the skeleton code, the bomb module that deals with the bomb's states of course, and finally the enemy module. In the debounce module, I simply place the pause signal in the sensitivity list of the always block that runs the debouncer and added it in as to be "OR" - ed to the if statement within the block that set the debounce output to 0.

```
// infer debounce register and next-state logic
always @(posedge clk, posedge reset, posedge pause)
    if(reset || pause)
        db_reg <= 0;
    else
```

Figure 16: Pause for Debouncer.

To the bomb module I added the pause signal to the sensitivity lists for the always block that updated the signals within the state machine between the reset section and the actual update section. To stop it from leaving the explosion state or the bomb placed state , I also added the pause signal to the continuous assignment statements that served to increment the counter registers that drove the explosion and bomb display counters.

```
69  assign bomb_counter_next = (bomb_active_reg & bomb_counter_reg < BOMB_COUNTER_MAX) && (!pause) ? bomb_counter_reg + 1 : 0;
70  assign exp_counter_next = (exp_active_reg & exp_counter_reg < EXP_COUNTER_MAX) && (!pause) ? exp_counter_reg + 1 : 0;
```

```
// Update State Machine Current Registers on Clock or Reset.
always @(posedge clk, posedge reset, posedge pause) begin
    if(reset) begin
        bomb_exp_state_reg   <= no_bomb;
        bomb_active_reg      <= 0;
        exp_active_reg       <= 0;
        bomb_x_reg           <= 0;
        bomb_y_reg           <= 0;
        exp_block_addr_reg   <= 0;
        block_we_reg         <= 0;
        post_exp_active_reg  <= 0;
        exp_range_reg        <= 1;
        exp_index_reg        <= 0;
    end else if(pause); // Do nothing. just don's update values.
    else begin
        bomb_exp_state_reg   <= bomb_exp_state_next;
        bomb_active_reg      <= bomb_active_next;
        exp_active_reg       <= exp_active_next;
        bomb_x_reg           <= bomb_x_next;
        bomb_y_reg           <= bomb_y_next;
        exp_block_addr_reg   <= exp_block_addr_next;
        block_we_reg         <= block_we_next;
        post_exp_active_reg  <= post_exp_active_next;
        exp_range_reg        <= exp_range_next;
        exp_index_reg        <= exp_index_next;
    end
```

Figure 17: Pause for Bomb Module.

Finally, to stop the enemy from moving in light of the pause signal to the motion timers assignment, And at various different points within the state machine that dictated its movement to serve the potential various different points within its movement that a user might pause the program this can be found in the enemy module because there are significantly more segments where I put balls and I don't want to buff up this report with pictures.

Conclusion/Closing Remarks/Lament on Late Submission

This project was daunting to say the least. I believe it is the perfect project the perfect capstone for this class and I think Dr. Lin for allowing us to do it. However, the scope of this project within the time that we had to implement it is something that given the senior status of myself and other students within the class probably should have been taken into account before it was assigned. Other time conflicts with deadlines in senior design significantly impacted the submission of this project. For me personally within my undergraduate career, I haven't ever submitted a significant assignment late until today. I'm extremely disappointed in myself of course and many things could have gone differently many things could have changed but either way there's no changing things. Not only am I submitting the assignment late I'm also submitting it incomplete, not up to my own standards, and this is work I'm not proud of and for that I'm sorry I apologize.

My advice to doctor learn and to you if you're reading this, Rakin, you ***wonderful*** TA, please provide the project at the beginning of the semester. At the beginning of the semester or even at some point way earlier than this this semester, I had a lot more time deadlines had not come up and I most certainly would have completed this project on time and up to par.

For all his faults and for all my faults I did enjoy doing this project the problems that I posed the struggles that I had it was intellectually challenging, and I do like that that's half the reason I'm chose to be an engineering major. However, having not slept well since Friday rather having not slept at all since Friday night working on this project in conjunction with studying for final exams and working on another final project (collectively I believe I've gotten about 9 hours of sleep since last Friday) it really sucks that I wasn't able to complete it.

More to the mayor of this project though like I said before I really love it I think it is indicative of what this class should be. I look forward to continuing my FPGA education in the future I'm even considering thinking of purchasing if not the base is 3 then another FPGA board so that I can continue this project and work on other things. Before this class and before this project I viewed FPGA as as something that you could implement logic gates on as we did in Jericho systems. I was aware that you could use them for greater things but I've never actually seen it happen and let alone do them myself but through the lab not only did I do image processing not only did I take an input with this project I've used pretty much everything we learned throughout the entire semester and it showed me with this much control over what we can do how powerful FPGAs can be and for that I I actually don't regret taking this class and submitting the assignment late! I'm proud to say that I did indeed learn something from this class I do not at all feel like I've wasted time it's beautiful.

If you're reading this, Rakin, thank you and have a great summer!