Final Report: Bomberman Project

Cory Brynds

cory.brynds@ucf.edu

EEL5722C: FPGA Design
Section 0012

Due: 6 December 2023
Submitted: 2 December 2023

1.0 Objectives

The objective of this project is to implement a fully functional retro Bomberman video game on the BASYS 3 FPGA development board. The graphics will be displayed on a monitor via a VGA connector, and the input is taken from the user using pushbuttons on the BASYS 3. In the game, the user plays as the Bomberman and must evade and attack an enemy randomly moving around the arena, using bombs to injure the enemy and score points. The Bomberman has five lives and loses a life when hit by an enemy or one of its bombs. Additionally, a random assortment of blocks is distributed around the map, and the Bomberman can destroy these blocks using bombs.

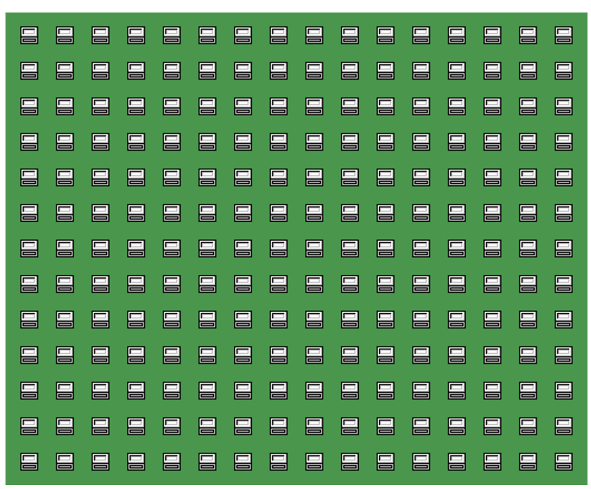
2.0 Equipment

- Xilinx Vivado Design Suite
- BASYS 3 FPGA Development Board
- USB Keyboard
- VGA Monitor

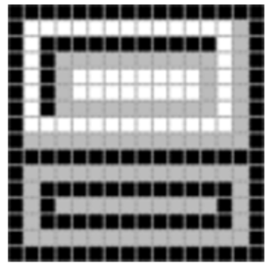
3.0 Module Breakdown

I. **Top module**: Receives user input from the controller (buttons), instantiates all modules and routes signals between them, and handles all display logic and game mechanics on the VGA monitor.

II. **Pillar display module**: instantiates the ROM for the pillar sprite and asserts pillar_on when the pixel coordinates are within the play area of the pillar.



A. **Pillar ROM**: Distributed ROM to store the .coe file for the pillar sprite



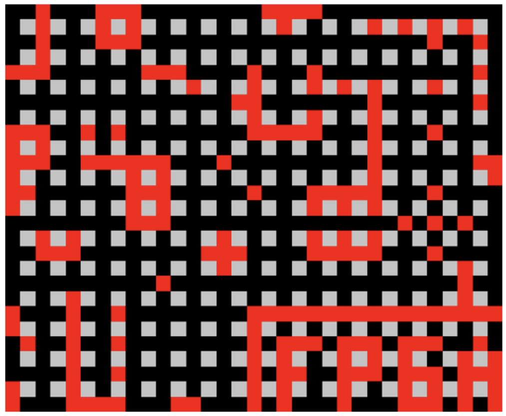
III. **Bomberman module**: Displays the Bomberman on screen, updates his location in response to user input and collisions, handles the logic for walking animations, outputs RGB color data for Bomberman based on sprite ROM, and outputs Bomberman's location to other modules for collision detection.

A. **Bomberman sprite ROM**: Single-port block ROM to store the 10 Bomberman sprites used to animate movement.

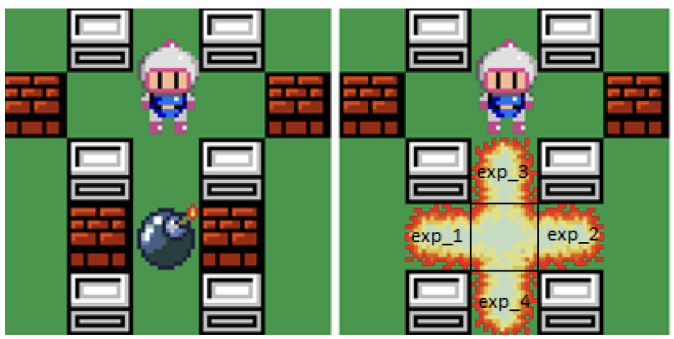
IV. **Block module**: Outputs RGB data to display block map to the monitor and handles Bomberman collision detection with blocks.

A. **Block ROM**: Distributed ROM used to store the .coe file for the block sprite.

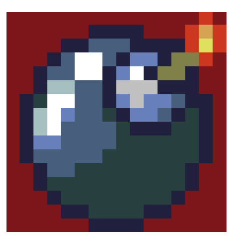
B. **Block map RAM**: Dual-port block RAM used to store 1-bit locations of blocks around the arena. Can be written to add, re-arrange, and remove blocks in response to game mechanics (i.e. bombs blowing up blocks).



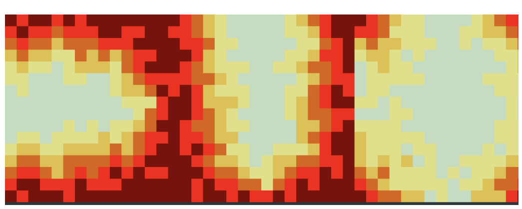
V. **Bomb module**: Implements logic for dropping a bomb to the tile closest to the center of the Bomberman's hitbox, which then remains on screen for a duration of time before exploding. The explosion can destroy surrounding blocks, but not pillars.



A. **Bomb ROM**: Distributed ROM to store the .coe file for the bomb sprite.



B. **Explosion ROM**: Block ROM to store the .coe file for the explosion sprites.



VI. **Enemy module**: Creates a single enemy that moves around the screen in response to a pseudorandom number generator. If it overlaps with an explosion from the bomb module, the enemy will be hit, increasing in speed and moving around more erratically.

A. **Enemy ROM**: block ROM to store the .coe file for the 10 enemy sprites.

B. **LFSR Module**: 16-bit linear feedback shift register that generates a 16-bit pseudorandom number.

VII. **Debounce button module**: Debounces the five buttons on the BASYS 3 used for controlling Bomberman to prevent glitches during gameplay.

VIII. **Game lives module**: Maintains a counter for the number of lives the Bomberman has remaining. It begins at 5 and decreases by one each time the enemy or an explosion collides with the Bomberman's hitbox. When the lives counter reaches 0, gameover is asserted.

IX. **Numbers ROM module**: ROM template used to display the 4 BCDs for the user's score to the screen, roughly at the top center.

X. **Score display module**: Implements a score counter, which increments by 10 each time the enemy is hit by an explosion. The counter starts at 0 and counts up to 9999, and it is displayed on the screen using 4 digits.

A. **Binary to BCD module**: Converts the binary value of the score to four binary-coded decimal values between 0 and 9 to be displayed on the screen.

XI. **VGA sync module**: Handles the logic necessary to drive the VGA display.

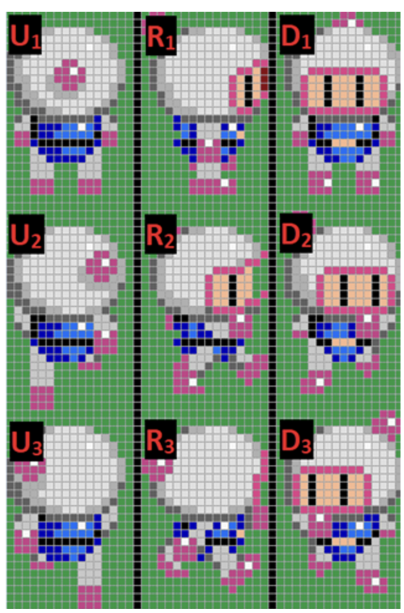
- Divides 100MHz clock down to a 25MHz pixel clock, the frequency at which pixels are updated on the VGA monitor.
- Generates the hsync and vsync timing signals.
    - Vsync defines the rate at which the display is refreshed, or redrawn, 60 Hz for the monitor used in this experiment.
    - Hsync signals the number of lines to be redrawn at a given refresh frequency.

4.0 Animate Bomberman

4.1 Description of Implementation

To create the Bomberman's walking animation, logic must be implemented to index into the block ROM at the correct offset to retrieve the proper walking sprite (up, down, right, etc). To accomplish this, a frame timer is implemented that starts counting when user input is received and stops counting once it reaches a maximum value, which in this case is 50 million clock cycles.

This maximum timing period is subdivided into four intervals. After each 12.5 million cycles (T/4), a new Bomberman sprite is displayed on the screen. For a given direction, the module cycles through the sprites below in the order 1, 2, 1, 3 (to obtain the left direction, the right direction is mirrored).



To index into the proper location in ROM, additional logic is added in the form of a switch case statement based on Bomberman's current direction. Based on the current count of the frame timer, the ROM offset is set to one of the 9 sprites pictured above.

4.2 Verilog Code

```
// ANIMATION FRAME TIMER
always @ (posedge clk, posedge reset) begin
   if (reset)
       frame_timer_reg <= 0;
   else begin
       frame_timer_reg <= frame_timer_next;
   end
end

// next state logic for motion timer: increment when Bomberman to move and timer less
than max, else reset.
assign frame_timer_next = ((L | R | U | D) & (frame_timer_reg < FRAME_REG_MAX)) ?
frame_timer_reg + 1 : 0;
```

In the animation frame timer section of the module, the frame timer is updated each clock cycle and set to 0 upon reset. A continuous assignment statement is used to increment the frame timer if a valid direction is detected or the timer has not reached its maximum value, or else reset the timer to 0 on the next clock cycle.

```
// REGISTER TO INDEX INTO SPRITE ROM
always @ (posedge clk, posedge reset) begin
   if (reset)
       rom_offset_reg <= 0;
   else
       rom_offset_reg <= rom_offset_next;
end

always @ (*) begin
   if (gameover)
       rom_offset_next = G_O;
   else
   case(current_dir)
       CD_U: begin
           case(frame_timer_reg)
```

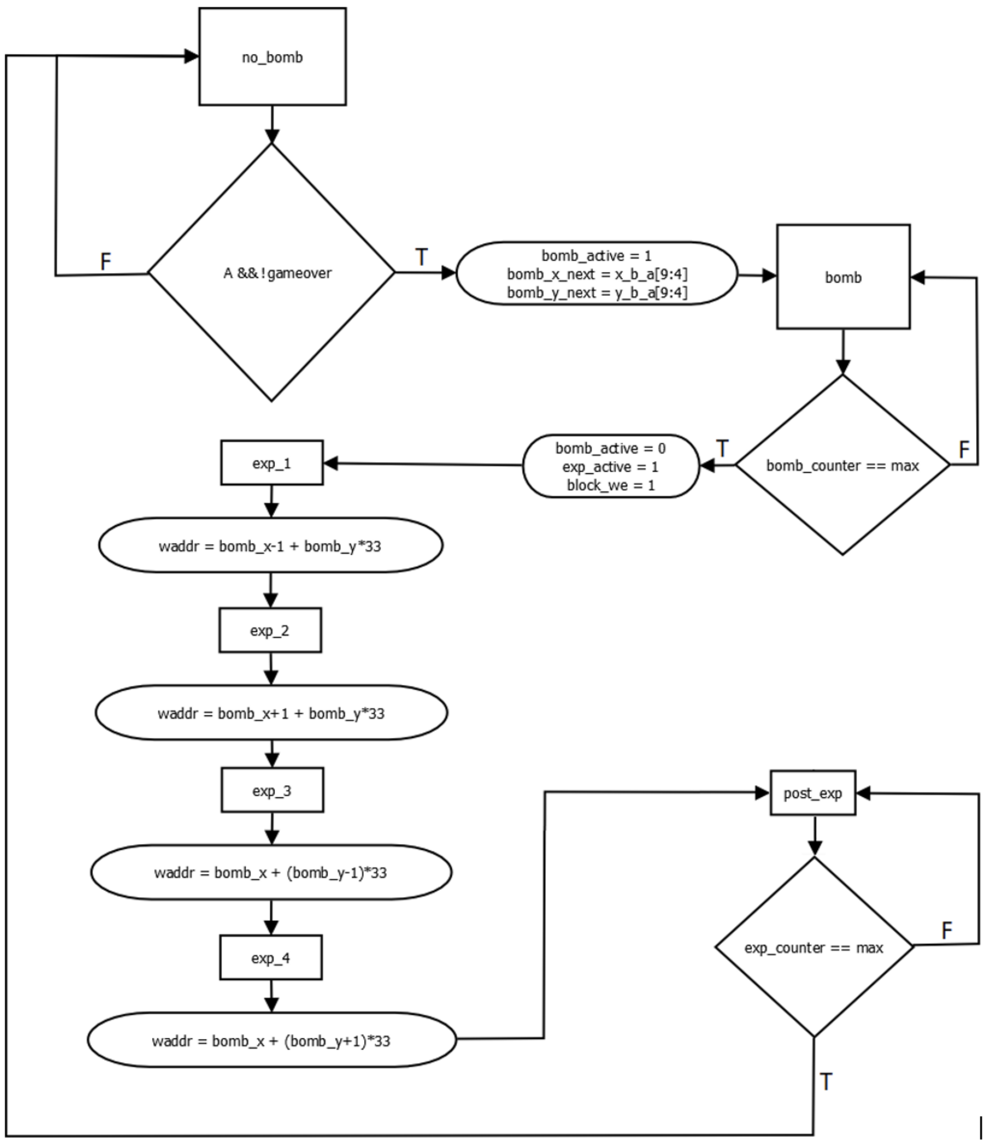
```
                0: rom_offset_next = U_1;
                FRAME_CNT_1: rom_offset_next = U_2;
                FRAME_CNT_2: rom_offset_next = U_1;
                FRAME_CNT_3: rom_offset_next = U_3;
            endcase
        end
        CD_D: begin
            case(frame_timer_reg)
                0: rom_offset_next = D_1;
                FRAME_CNT_1: rom_offset_next = D_2;
                FRAME_CNT_2: rom_offset_next = D_1;
                FRAME_CNT_3: rom_offset_next = D_3;
            endcase
        end
        CD_L: begin
            case(frame_timer_reg)
                0: rom_offset_next = R_1;
                FRAME_CNT_1: rom_offset_next = R_2;
                FRAME_CNT_2: rom_offset_next = R_1;
                FRAME_CNT_3: rom_offset_next = R_3;
            endcase
        end
        CD_R: begin
            case(frame_timer_reg)
                0: rom_offset_next = R_1;
                FRAME_CNT_1: rom_offset_next = R_2;
                FRAME_CNT_2: rom_offset_next = R_1;
                FRAME_CNT_3: rom_offset_next = R_3;
            endcase
        end
    endcase
end
```

In this section of the module, the ROM offset register is updated each clock cycle with the next value and set to 0 upon reset. A procedural block with a switch case statement is used to select the proper ROM offset to animate the Bomberman's movement. Following the process explained earlier in this section to animate the Bomberman, the case for each direction has a nested case statement to select the proper ROM sprite offset based on the value of the frame timer register.

5.0 Bomb and Explosion Finite State Machine

5.1 Description of Implementation

To complete the bomb module, a state machine was needed to handle the next-state logic for dropping the bomb and causing the explosion to damage blocks, the enemy, and Bomberman. The state machine was constructed using the following ASMD chart:



In the first state, no_bomb, the state machine waits until the user presses the button A, which is used to drop a bomb. Upon A being pressed, bomb_active_reg is asserted and a bomb is dropped at the middle of the Bomberman's hitbox before transitioning to the bomb state.

In the bomb state, the bomb is displayed on screen for a duration defined by BOMB_COUNTER_MAX, which is 220 million clock cycles. After this duration is reached bomb_active_reg is deasserted and exp_active_reg is asserted, signaling the start of the explosion animation. The state machine then transitions to exp1.

In exp1, the arena block map coordinate to the left of the bomb is sent to the write address of the block map RAM, clearing the block from RAM, if it exists. The state machine then transitions to exp2, which sets the write address to the block map RAM for the arena block map coordinate to the right of the bomb. A similar process is followed in exp3 and exp4, which set the write address to the block map RAM for the top and bottom of the bomb, respectively. Exp4 transitions to the post_exp state.

In the post_exp state, the explosion counter counts up to a maximum value of 120 million cycles to keep the explosion on the screen for a duration, after which exp_active_reg and block_we_reg are set to zero. The state machine then transitions to no_bomb.

5.2 Verilog Code

```
case(bomb_exp_state_reg)
      no_bomb: begin
           if (A && !gameover) begin
               bomb_active_next = 1;
               bomb_x_next = x_bomb_a[9:4];
               bomb_y_next = y_bomb_a[9:4];
               bomb_exp_state_next = bomb_exp_state_reg + 1;
           end
       end
       bomb: begin
           if (bomb_counter_reg == BOMB_COUNTER_MAX) begin
               bomb_active_next = 0;
               exp_active_next = 1;
               block_we_next = 1;
```

```
                bomb_exp_state_next = bomb_exp_state_reg + 1;
            end
        end
        exp_1: begin
            if (bomb_x_reg > 0)
                exp_block_addr_next = (bomb_x_reg - 1) + bomb_y_reg * 33;
            bomb_exp_state_next = bomb_exp_state_reg + 1;
        end
        exp_2: begin
           if (bomb_x_reg < ARENA_WIDTH)
                exp_block_addr_next = (bomb_x_reg + 1) + bomb_y_reg * 33;
            bomb_exp_state_next = bomb_exp_state_reg + 1;
        end
        exp_3: begin
            if (bomb_y_reg > 0)
                exp_block_addr_next = bomb_x_reg + (bomb_y_reg - 1) * 33;
            bomb_exp_state_next = bomb_exp_state_reg + 1;
        end
        exp_4: begin
           if (bomb_y_reg < ARENA_HEIGHT)
                exp_block_addr_next = bomb_x_reg + (bomb_y_reg + 1) * 33;
            bomb_exp_state_next = bomb_exp_state_reg + 1;
        end
        post_exp: begin
            post_exp_active = 1;
            if (exp_counter_reg == EXP_COUNTER_MAX) begin
                exp_active_next = 0;
                block_we_next = 0;
                bomb_exp_state_next = 0;
            end
        end
        default:
            bomb_exp_state_next = bomb_exp_state_reg + 1;
    endcase

end         // END FSM next-state logic
```

6.0 Linear Feedback Shift Register

6.1 Description of Implementation

A pseudorandom number generator module is needed to create the random movement for the enemy around the arena. A linear feedback shift register is a shift register that XORs an arbitrary combination of its input bits every cycle, generating a pseudorandom output. In this case, a 16-bit LFSR is created, which outputs a 16-bit random number, used in the enemy finite state machine to move the enemy around the arena.

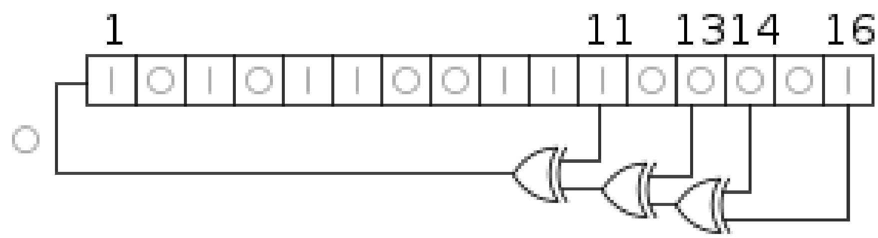


Figure: 16-bit LFSR (Source: Wikipedia)

To generate pseudorandom numbers, specific bits of the 16-bit shift register are "tapped" with an XOR gate. The result of this 4-input XOR gate is then placed into the LSB of the shift register. The following formula is used to generate this feedback bit:

$$Feedback = LFSR[10] \oplus LFSR[12] \oplus LFSR[13] \oplus LFSR[15]$$

6.2 Verilog Code

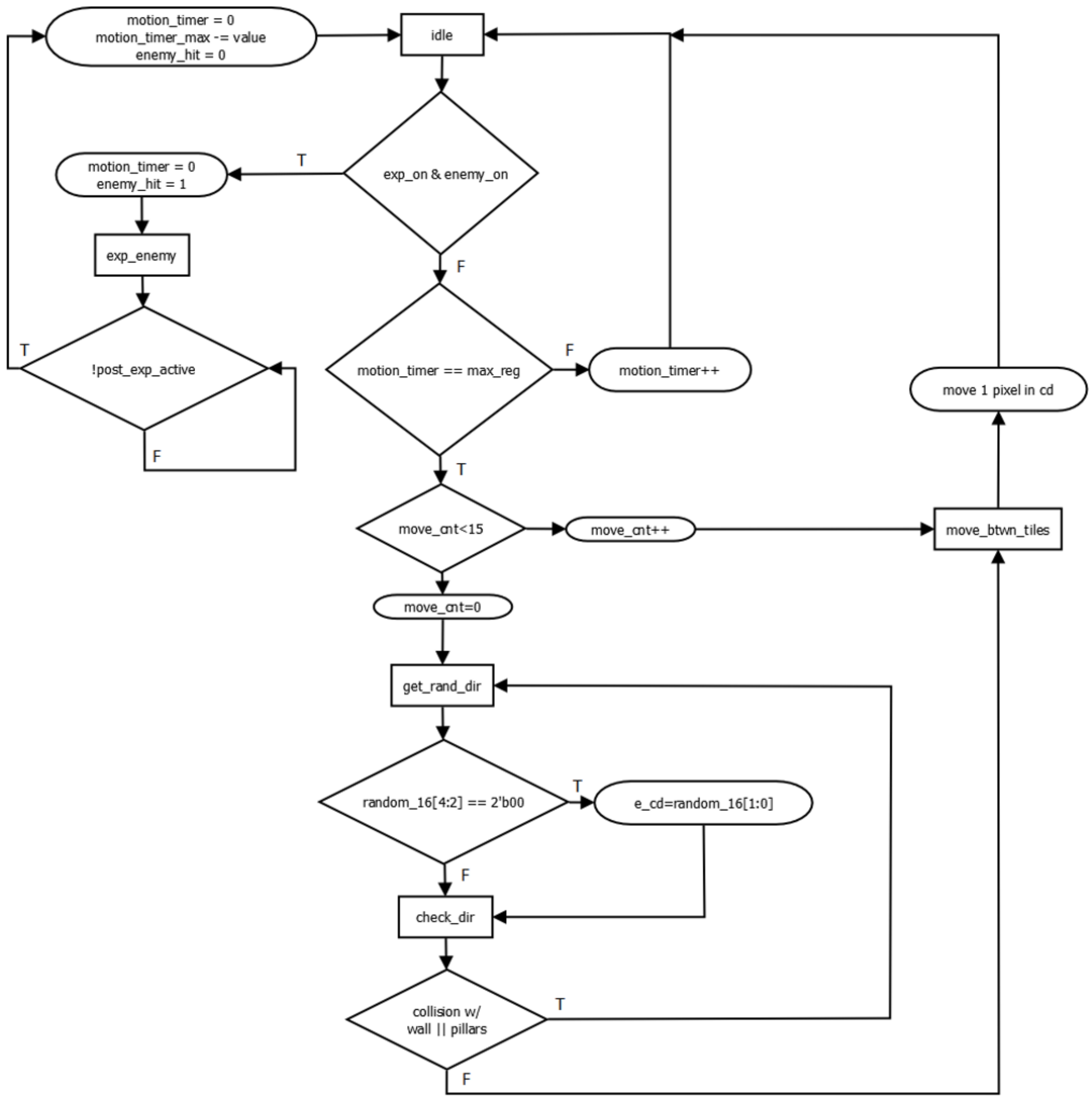
```
// Linear Feedback Shift Register
module LFSR_16(
    input clk, rst, w_en,
    input [15:0] w_in,
    output [15:0] out
);

reg [15:0] LFSR_reg;
wire feedback_next;
```

```
reg feedback_reg;
wire [15:0] LFSR_next;

assign feedback_next = LFSR_reg[15] ^ LFSR_reg[13] ^ LFSR_reg[12] ^
LFSR_reg[10];
assign LFSR_next = {LFSR_reg[14:0], feedback_reg};

always @ (posedge clk, posedge rst) begin
    if (rst) begin
        LFSR_reg <= 1; // Reset reg to a value of 1; if 0, no random
      sequence would be generated
        feedback_reg <= 0;
        end
    else if (w_en)
        LFSR_reg <= w_in;
    else begin
        feedback_reg <= feedback_next;
        LFSR_reg <= LFSR_next;
    end
end

assign out = LFSR_reg;

endmodule
```

7.0 Enemy Finite State Machine

7.1 Description of Implementation

To finish the enemy module, a finite state machine must be created that handles its idle states, retrieves a random direction, and moves the enemy without colliding with pillars or blocks. An ASMD for the enemy finite state machine is pictured below:



To start, the enemy is centered in a tile on the arena block map in the idle state. It then moves 16 pixels in the current direction, stored in e_cd_reg. For every move, the enemy has a 1/8 chance, determined by 3 bits in random[4:2], of receiving a random direction from random[1:0]. This random direction is checked to be a valid move in the check_dir state. A valid move is defined as not allowing the enemy to pass through the arena walls or a pillar. In the case that it is invalid, a new random move is chosen until a valid move is found.

If the enemy is hit by an explosion, denoted by exp_on and enemy_on both being asserted, the state machine will transition to the exp_enemy state, where the enemy does not move until the duration of the explosion is over. After the enemy is hit, the FSM transitions back to the idle state, and the enemy starts moving faster and more erratically (higher probability of random movement).

7.2 Verilog Code

```
// FSM next-state logic
always @ (*)
   begin
   // defaults
   e_state_next           = e_state_reg;
   x_e_next               = x_e_reg;
   y_e_next               = y_e_reg;
   e_cd_next              = e_cd_reg;
   motion_timer_next      = motion_timer_reg;
    motion_timer_max_next = motion_timer_max_reg;
   move_cnt_next          = move_cnt_reg;
   enemy_hit              = 0;

   case(e_state_reg)
      idle: begin
         if (exp_on && enemy_on) begin
            motion_timer_next = 0;
            enemy_hit = 1;
            e_state_next = exp_enemy;
         end
         else begin
            if (motion_timer_reg != motion_timer_max_reg)
                 motion_timer_next = motion_timer_reg + 1;
             else begin
                if (move_cnt_reg == 15) begin
                   move_cnt_next = 0;
                    e_state_next = get_rand_dir;
                end
                else begin
                   move_cnt_next = move_cnt_reg + 1;
                   e_state_next = move_btwn_tiles;
                end
                motion_timer_next = 0;
             end
        end
```

```
      end

      exp_enemy: begin
         if (!post_exp_active) begin
            motion_timer_next = 0;
            motion_timer_max_next = motion_timer_max_reg / 2; //
           Arbitrarily speed up the enemy's movement
            enemy_hit = 0;
            e_state_next = idle;
         end
     end

      move_btwn_tiles: begin
        // Move one pixel in current direction
        x_e_next = x_e_reg - (e_cd_reg == CD_L) + (e_cd_reg == CD_R);
        y_e_next = y_e_reg - (e_cd_reg == CD_U) + (e_cd_reg == CD_D);
        e_state_next = idle;
      end

      get_rand_dir: begin
         e_cd_next = (random_16[4:2] == 2'b00) ? random_16[1:0] :
                                                     e_cd_reg;
         e_state_next = check_dir;
      end
      // Check if colliding with walls or pillar
      check_dir: begin
        case (e_cd_reg)
            CD_U: e_state_next = (y_e_abm == 0 || x_e_abm[0] == 1) ?
                 get_rand_dir : move_btwn_tiles;
            CD_L: e_state_next = (x_e_abm == 0 || y_e_abm[0] == 1) ?
                 get_rand_dir : move_btwn_tiles;
            CD_R: e_state_next = (x_e_abm == 32 || y_e_abm[0] == 1) ?
                 get_rand_dir : move_btwn_tiles;
            CD_D: e_state_next = (y_e_abm == 26 || x_e_abm[0] == 1) ?
                 get_rand_dir : move_btwn_tiles;
            default: e_state_next = move_btwn_tiles;
        endcase
    end
   endcase
end
```

8.0 Binary to BCD Converter

8.1 Description of Implementation

This module is used by the score display module to convert a user's score (a value between 0 and 9999) to four binary-coded decimals, which can then be displayed on the screen. The input to the module is a 14-bit binary number, and the output is four BCDs between 0 and 9. To efficiently implement the binary to BCD converter, the double dabble algorithm is used. Pictured below is a diagram of the double dabble algorithm for an 8-bit binary input.

BCD2	BCD1	BCD0	Binary Input	
0000	0000	0000	10010111	
0000	0000	0001	0010111.	← 1
0000	0000	0010	010111..	← 2
0000	0000	0100	10111...	← 3
0000	0000	1001	0111....	← 4
0000	0000	1100	0111....	**ADD**
0000	0001	1000	111.....	← 5
0000	0001	1011	111.....	**ADD**
0000	0011	0111	11......	← 6
0000	0011	1010	11......	**ADD**
0000	0111	0101	1.......	← 7
0000	1010	1000	1.......	**ADD ADD**
0001	0101	0001	....	← 8

Figure: Double dabble for 8-bit input (Source: Real Digital)

In double dabble, the binary input is shifted left each iteration, with the MSB being shifted into the LSB of the BCD sequence. After each shift, a check is performed to ensure that no single BCD has exceeded five. If any BCD has exceeded five, the value three is added to it. While this module could be implemented with purely combinational logic, it is implemented using state machine logic, where the input start signals the beginning of the conversion state machine. No output signal is needed in this implementation.

8.2 Verilog Code

```
module binary2bcd (
```

```verilog
    input clk, reset, start,
    input [13:0] in,
    output [3:0] bcd3, bcd2, bcd1, bcd0, count,
    output [1:0] state
);
    localparam  IDLE = 2'b00,
                SHIFT = 2'b01,
                ADD = 2'b10,
                COUNT_MAX = 14;

    reg [1:0] state_reg, state_next;
    reg[13:0] binary_reg, binary_next;
    reg [3:0] shift_count_reg, shift_count_next;
    reg [15:0] bcd_out_reg, bcd_out_next;

    always @ (posedge clk, posedge reset) begin
        if (reset) begin
            state_reg <= IDLE;
            bcd_out_reg <= 0;
            binary_reg <= 0;
            shift_count_reg <= 0;
        end
        else if (start) begin
            state_reg <= SHIFT;
            binary_reg <= in;
            shift_count_reg <= 0;
            bcd_out_reg <= 0;
        end
        else begin
            bcd_out_reg <= bcd_out_next;
            shift_count_reg <= shift_count_next;
            state_reg <= state_next;
            binary_reg <= binary_next;
        end
    end

    always @ (*) begin
        bcd_out_next = bcd_out_reg;
        shift_count_next = shift_count_reg;
        state_next = state_reg;
```

```
binary_next = binary_reg;

case(state_reg)
    IDLE: begin
        if (start) begin
            binary_next = in;
            bcd_out_next = 0;
            shift_count_next = 0;
            state_next = SHIFT;
        end
    end
    SHIFT: begin
        if (shift_count_reg == COUNT_MAX) begin
            state_next = IDLE;
            shift_count_next = 0;
            bcd_out_next = bcd_out_reg;
        end
        else begin
            bcd_out_next = bcd_out_reg << 1;
            bcd_out_next[0] = binary_reg[13]; // MSB from binary
               input
            binary_next = binary_reg << 1;

            state_next = ADD;
        end
    end
    ADD: begin
        if (shift_count_reg < COUNT_MAX - 1) begin
            if (bcd_out_next[3:0] > 4)
                bcd_out_next[3:0] = bcd_out_next[3:0] + 3;
            if (bcd_out_next[7:4] > 4)
                bcd_out_next[7:4] = bcd_out_next[7:4] + 3;
            if (bcd_out_next[11:8] > 4)
                bcd_out_next[11:8] = bcd_out_next[11:8] + 3;
            if (bcd_out_next[15:12] > 4)
                bcd_out_next[15:12] = bcd_out_next[15:12] + 3;
        end
        shift_count_next = shift_count_reg + 1;
        state_next = SHIFT;
    end
```

```
        endcase
    end

    assign state = state_reg;
    assign count = shift_count_reg;
    assign {bcd3, bcd2, bcd1, bcd0} = bcd_out_reg;

endmodule
```

8.3 Verilog Test Bench

To verify the functionality of this converter, a test bench was written to test four separate 14-bit input vectors. The Verilog code for the test bench module is shown below.

```
module bcd_converter_tb();

reg clk, reset, start;
reg [13:0] in;
wire[15:0] bcd_out;
wire[3:0] count;
wire[1:0] state;

binary2bcd DUT(
    .clk(clk),
    .reset(reset),
    .start(start),
    .in(in),
    .bcd3(bcd_out[15:12]),
    .bcd2(bcd_out[11:8]),
    .bcd1(bcd_out[7:4]),
    .bcd0(bcd_out[3:0]),
    .count(count),
    .state(state)
);

always #10 clk = ~clk;

initial begin
    clk = 0;
    reset = 0;
    start = 0;
```

```
in = 0;

// Test Case 1: in = 0000
#100
reset = 1;
#20 reset = 0;
in = 0000;
start = 1;
#20 start = 0;

while (state != 2'b00)
    #20

// Check result
if (bcd_out[15:12] == 0 && bcd_out[11:8] == 0 && bcd_out[7:4] == 0 &&
  bcd_out[3:0] == 0 && count == 0)
    $display("Test passed: 0000 -> %d %d %d %d", bcd_out[15:12],
        bcd_out[11:8], bcd_out[7:4], bcd_out[3:0]);

    // Test case 2: in = 1234
    #100
    reset = 1;
    #20 reset = 0;
    in = 1234;
    start = 1;
    #20 start = 0;

    while (state != 2'b00)
        #20

    // Check result
    if (bcd_out[15:12] == 1 && bcd_out[11:8] == 2 && bcd_out[7:4] == 3
        && bcd_out[3:0] == 4 && count == 0)
        $display("Test passed: 1234 -> %d %d %d %d", bcd_out[15:12],
        bcd_out[11:8], bcd_out[7:4], bcd_out[3:0]);

    // Test case 3: in = 6781
    #100
    reset = 1;
    #20 reset = 0;
```

```
        in = 6781;
        start = 1;
        #20 start = 0;

        while (state != 2'b00)
            #20

        // Check result
        if (bcd_out[15:12] == 6 && bcd_out[11:8] == 7 && bcd_out[7:4] == 8
      && bcd_out[3:0] == 1 && count == 0)
            $display("Test passed: 6781 -> %d %d %d %d", bcd_out[15:12],
           bcd_out[11:8], bcd_out[7:4], bcd_out[3:0]);

        // Test case 4: in = 9999
        #100
        reset = 1;
        #20 reset = 0;
        in = 9999;
        start = 1;
        #20 start = 0;

        while (state != 2'b00)
            #20

        // Check result
        if (bcd_out[15:12] == 9 && bcd_out[11:8] == 9 && bcd_out[7:4] == 9
           && bcd_out[3:0] == 9 && count == 0)
            $display("Test passed: 9999 -> %d %d %d %d", bcd_out[15:12],
           bcd_out[11:8], bcd_out[7:4], bcd_out[3:0]);

    $finish;
end
endmodule
```

8.4 Test Bench Results

The following table summarizes the input vectors used, their binary representations, and the corresponding BCD output. The behavioral simulation waveforms are also shown below.

Test input	Binary value	BCD0	BCD1	BCD2	BCD3

0	0	0000	0000	0000	0000
1234	10011010010	0001	0010	0011	0100
6781	1101001111101	0110	0111	1000	0001
9999	10011100001111	1001	1001	1001	1001

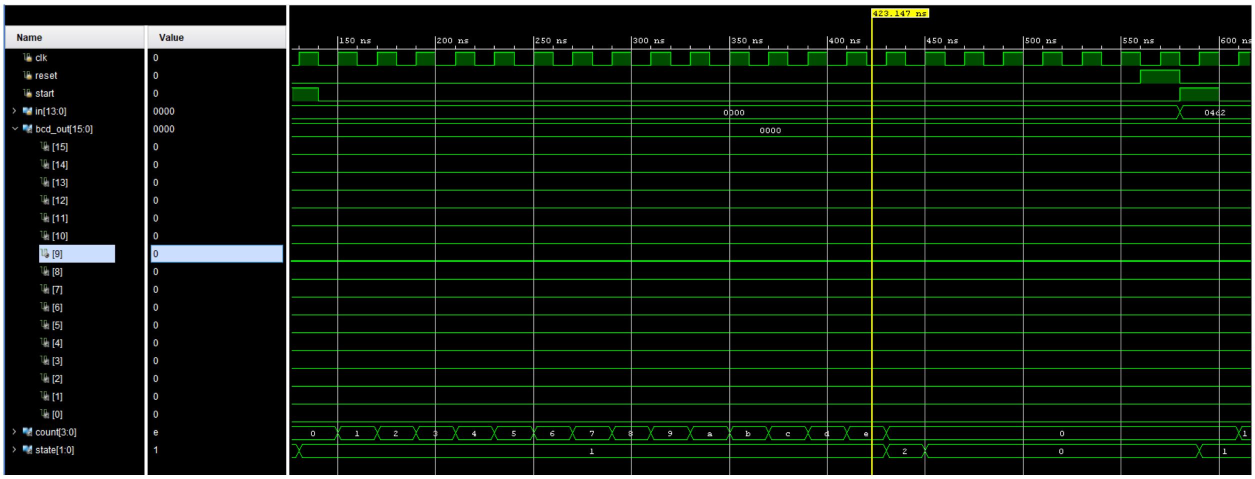


Figure: Simulation waveform for input = 0

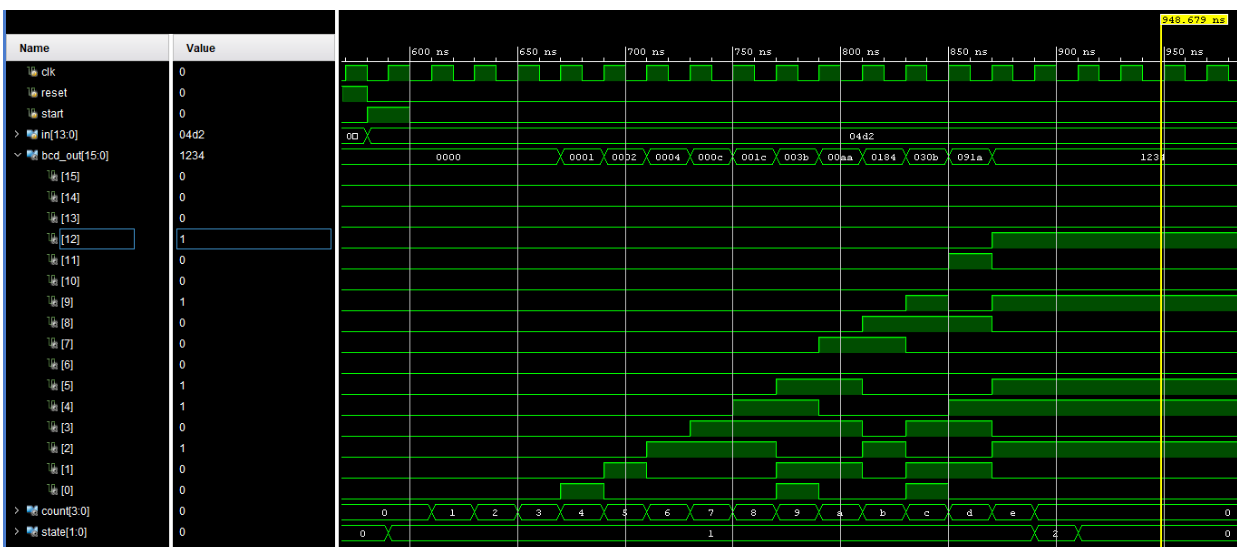


Figure: Simulation waveform for input = 1234

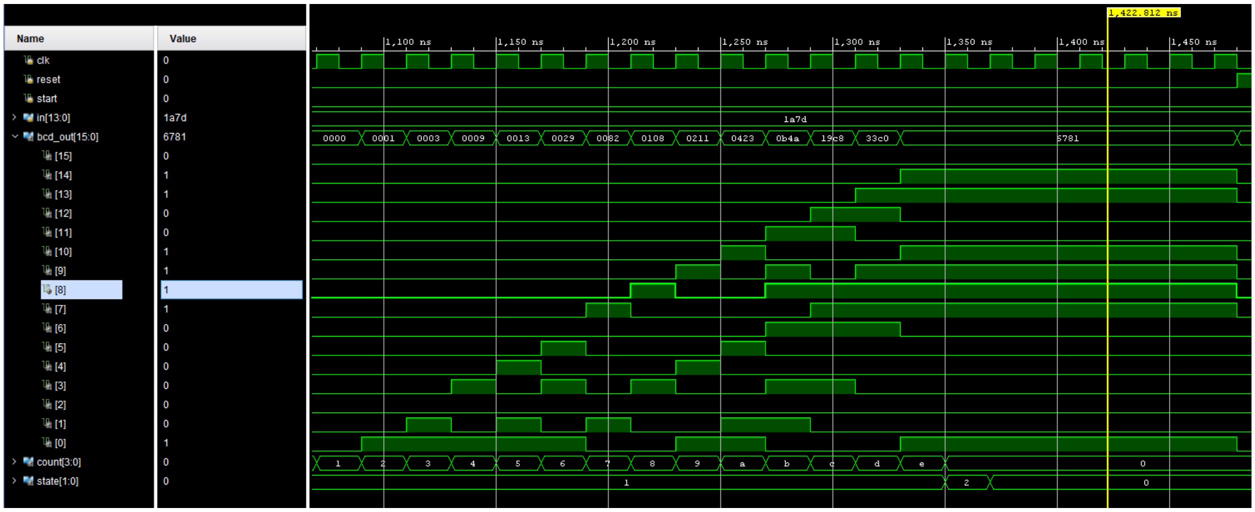


Figure: Simulation waveform for input = 6781

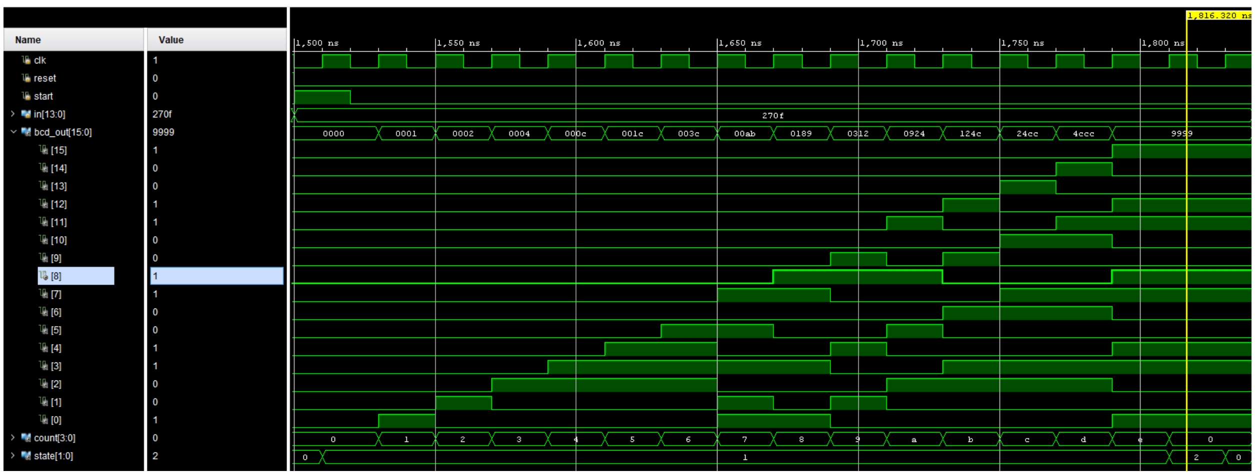


Figure: Simulation waveform for input = 9999

9.0 Animate Life Bar

9.1 Description of Implementation

This module displays a 20-pixel by 8-pixel red life bar for the Bomberman's five lives. It is displayed in the top right-hand corner and is filled with black color each time the Bomberman is hit by an explosion or enemy. Once all five lives have been consumed, the life bar is entirely black.

9.2 Verilog Code

```
// output the amount of lives the Bomberman has left
assign lives = lives_reg;
// Assert healthbar_on when coordinates of pixel are within the health bar rectangle
assign healthbar_on = ((x > X_WALL_R) & (x < X_WALL_R + lives_reg*4) & (y > 5) & (y <
13));
assign healthbar_rgb = 12'b111100000000;
```

The code above is implemented as part of the game lives module. To display the life bar, a "lifebar_on" signal was created, which is asserted when the (x,y) coordinates of the pixel are within the rectangular region of the life bar in the top right-hand corner. Additionally, a 12-bit lifebar_rgb register is created to hold the RGB values for the life bar pixels.

```
// RGB register next-state logic
assign rgb_next = p_tick ?
                   (bomberman_on & bomberman_rgb != 2049) ? bomberman_rgb :
                   (enemy_on     & enemy_rgb     != 2049) ? enemy_rgb :
                   (pillar_on)                            ? pillar_rgb :
                   (block_on)                             ? block_rgb :
                   (bomb_on      & bomb_rgb      != 2049) ? bomb_rgb :
                   (exp_on       & exp_rgb       != 2048) ? exp_rgb :
                   (score_on)                             ? 12'b111111111111:
                   (healthbar_on)                         ? healthbar_rgb :
                   (wall_on)                              ? background_rgb :
                   12'b001000100000 : rgb_reg;
```

When lifebar_on is asserted, logic in the top module will display the life bar to the monitor.

10.0 Results

10.1 Video of Experimental Implementation

Main gameplay and attacking enemy

Death animation

Reset switch

10.2 Explanation of Results

After finishing all of the modules, full functionality of the Bomberman video game was achieved. All modules worked as expected, from the Bomberman's walking animations to the enemy's pseudorandom movement to the score display when the enemy is hit. Some modules required extensive debugging, such as the FSM in the enemy module, but all issues were able to be resolved in the end. Simulations and test benches were used to verify the validity of the binary to BCD converter, and the simulation results indicated everything was functioning as expected.

11.0 Conclusion

All additional logic and modules were added to complete the Bomberman project. Through animating the Bomberman character, creating the finite state machine for the bomb feature, adding the finite state machine for the enemy character, implementing the BCD converter module using the double dabble algorithm, and updating the game lives module to create an on-screen life bar, a fully functional and playable Bomberman video game was created.

This project was an excellent introduction to working on large-scale Verilog projects with numerous signals passed between various submodules. Through designing finite-state machines, an understanding was gained of next-state logic and how sequential and combinational circuits interact. The process of instantiating distributed and block memory, both read-only and random-access, provided an opportunity to learn how to use the Vivado IP generator. To design the binary to BCD converter, the double dabble algorithm needed to be converted from pseudocode to Verilog, showing how high-level code maps to HDL.

Overall, this course project taught many fundamental HDL concepts, and the skills and tools used to complete this project will prove valuable in future applications.

12.0 References

Basys 3 Reference Manual. Digilent Reference. Digilent.
https://digilent.com/reference/programmable-logic/basys-3/reference-manual.

Binary to BCD. Real Digital.
https://www.realdigital.org/doc/6dae6583570fd816d1d675b93578203d

Final Report Manual. EEL5722. Rakin Muhammad Shadab.