Milestone 1

1. A way to know which frame of the sprite walking animation to show is needed. We can accomplish this by using a counter to select which sprite to display. To accomplish:
    a. **If** direction pressed **AND** count is less than some threshold **THEN** increment count
2. To determine the location in memory of the desired sprite to display a ternary statement which returns the required memory address based on game status (is the game over flag raised?) and directional button press duration.
3. The bomb explosion FSM functions as follows:
    a. No_bomb: This will function as an idle state, waiting for the button press that triggers the rest of the FSM. If the action button is press during the game bomberman's current position needs to be stored, so the bomb can be drawn in this location. An offset must be added, bomberman's dimensions are 16X25 vs 16X16. We want the bomb we want the bomb centered at bomberman's hitbox, to make things easier we can place the bomb exclusively in 16x16 tiles.
    b. Bomb: The bomb must disappear after some time. Using a counter, it is possible to wait in this state for the counter's duration then de-assert a flag which controls when the bomb is drawn on screen. To prepare for the explosion the block map RAM must be write enabled during this state.
    c. Exp_1 – Exp_4: These states are used to select the locations in memory that will be written to. One tile left, right, above, and below are selected. The location of the bomb is used to reference the location of the tiles. In memory, the locations of the blocks are stored as 1 bit 1 values, but rewriting 0s to all location effected by the blast the block will be removed from the map. Pillars are stored in a separate block of memory and are thus unaffected.
    d. Post_exp: The write enable of the block map RAM must be set low. Flags will also need to be set to alert other modules that the explosion is no longer active/visible and post explosion effects must be started, such as enemy damage.

Milestone 1 Code

```
//********************** ANIMATION FRAME TIMER*****************************//
//Frame timer for bomberman walking animation: duration increases as bomberman moves in same direction
// infer register for frame_timer
always @(posedge clk, posedge reset)
    if(reset)
      frame_timer_reg <= 0;
    else
      frame_timer_reg <= frame_timer_next;

// next state logic for frame_timer
assign frame_timer_next = ((L | R | U | D) & (frame_timer_reg < FRAME_REG_MAX))? frame_timer_reg + 1 : 0;
```

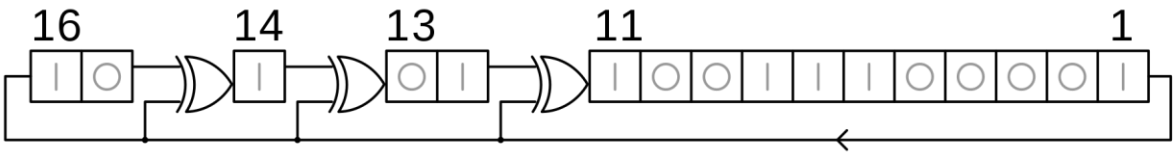
```
//******************REGISTER TO INDEX INTO SPRITE ROM **************************//
//Based on duration of frame timer selects animation sprite for walking direction
// infer register for rom_offset_reg
always @(posedge clk, posedge reset)
    if(reset)
      rom_offset_reg <= 0;
    else
      rom_offset_reg <= rom_offset_next;

// next state logic for frame_timer
assign rom_offset_next = (gameover)   ?  G_O :
                (cd == CD_U) ? (frame_timer_reg < FRAME_CNT_1 ? U_1 :
                        frame_timer_reg < FRAME_CNT_2 ? U_2 :
                        frame_timer_reg < FRAME_CNT_3 ? U_1 : U_3) :
                (cd == CD_R) ? (frame_timer_reg < FRAME_CNT_1 ? R_1 :
                        frame_timer_reg < FRAME_CNT_2 ? R_2 :
                        frame_timer_reg < FRAME_CNT_3 ? R_1 : R_3) :
                (cd == CD_D) ? (frame_timer_reg < FRAME_CNT_1 ? D_1 :
                        frame_timer_reg < FRAME_CNT_2 ? D_2 :
                        frame_timer_reg < FRAME_CNT_3 ? D_1 : D_3) :
                (cd == CD_L) ? (frame_timer_reg < FRAME_CNT_1 ? R_1 :
                        frame_timer_reg < FRAME_CNT_2 ? R_2 :
                        frame_timer_reg < FRAME_CNT_3 ? R_1 : R_3) : rom_offset_reg;
```

```
//******************Bomb Module Explosion FSM**************************//
// infer registers used in FSM
always @(posedge clk, posedge reset)
  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;
    end
  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;
    end


// bomb and explosion FSM: idle until bomb button pressed
// controls bomb and explosion placement and duration, address of block removal, and triggers post
explosion affects on enemy and player
always @*
  begin
  // defaults
  bomb_exp_state_next = bomb_exp_state_reg;
  bomb_active_next   = bomb_active_reg;
  exp_active_next    = exp_active_reg;
  bomb_x_next        = bomb_x_reg;
  bomb_y_next        = bomb_y_reg;
  exp_block_addr_next = exp_block_addr_reg;
  block_we_next      = block_we_reg;
  post_exp_active    = 0;

  case(bomb_exp_state_reg)
   no_bomb:   if (A && !gameover)
              begin
              bomb_active_next = 1;                 //allows bomb counter to run: displays till counter max
              bomb_x_next = x_b[9:4] - X_BOMB_OFFSET ;   //current tile bomberman occupies
              bomb_y_next = y_b[9:4] - Y_BOMB_OFFSET ;   //current tile bomberman occupies
              bomb_exp_state_next = bomb;
```

```
            end
  bomb:     if (bomb_counter_reg == BOMB_COUNTER_MAX)
            begin
            bomb_active_next = 0;              //bomb disappers after counter period
            exp_active_next = 1;               //allows explosion counter to run: displays till counter max
            block_we_next = 1;                 //allow write to block map RAM
            bomb_exp_state_next = exp_1;
            end
  exp_1:    if (bomb_x_reg != 0)
            begin
            exp_block_addr_next = ((bomb_x_reg - 1) + (bomb_y_reg)*33); //location of left explosion:
                                                                        used to remove blocks
            bomb_exp_state_next = exp_2;
            end
          else   bomb_exp_state_next = exp_2;
  exp_2:    if (bomb_x_reg != (X_WALL_R - X_WALL_L))
            begin
            exp_block_addr_next = ((bomb_x_reg + 1) + (bomb_y_reg)*33); //location of right explosion:
                                                                        used to remove blocks
            bomb_exp_state_next = exp_3;
            end
          else   bomb_exp_state_next = exp_3;
  exp_3:    if (bomb_y_reg != 0)
            begin
            exp_block_addr_next = ((bomb_x_reg) + (bomb_y_reg - 1)*33); //location of top explosion:
                                                                        used to remove blocks
            bomb_exp_state_next = exp_4;
            end
          else   bomb_exp_state_next = exp_4;
  exp_4:    if (bomb_y_reg != (Y_WALL_D - Y_WALL_U))
            begin
            exp_block_addr_next = ((bomb_x_reg) + (bomb_y_reg + 1)*33); //location of bottom
                                                                    explosion: used to remove blocks
            bomb_exp_state_next = post_exp;
            end
          else   bomb_exp_state_next = post_exp;
  post_exp: if (exp_counter_reg == EXP_COUNTER_MAX)
            begin
            post_exp_active = 1;          //flag to trigger status of enemies affected by blast
            exp_active_next = 0;          //end explosion
            block_we_next = 0;            //end write access to block map RAM
            bomb_exp_state_next = no_bomb;
            end
  endcase


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

Milestone 2

1. The Linear Shift Feedback Register was designed based on the Galois LFSR seen in the figure below, with taps (inputs to XOR gates) at bits 16, 15, 13, and 4. Combinational logic was used within the FSM to stored the XOR'd bits back into the shifted register.



2. For the enemy FSM performs the following:
    a. Idle state: This state needs to control how fast the enemy moves and set flags if it is hit. The first was done via a counter that delayed the enemy movement, a secondary counter controls how many tiles it moves. If the explosion from the bomb module and the enemy sprite are both "on" at the same location, we know it was hit and can move to the 'hit' state.
    b. Move_btwn_tiles: We need to make sure the enemy moves between tiles and stays within the map. We can do this by adding conditions to our directional checks such as "If moving up **AND** next location is greater than end of map **AND** we are not moving into an odd number row and column (pillars are placed in these locations) **THEN** we can move up 1 space.
    c. get_rand_dir: To produce random directions we can use the LFSR as a Pseudo-random number generator. Each individual bit of the 16 bit binary number as a ½ chance of being a zero, if we set a condition were we source a random number to choose our next direction if we have 3 consecutive zeros in 3 bits of the random number, we generate a 1/8 chance for the enemy to change direction each time it reaches this state. The directions are coded using a 2 bit value, so any 2 bits of the random number can be used to set the enemy's direction.
    d. Check_dir: We must ensure that the chosen path is legal. This can be done by using similar conditions as those used in the move_btwn_tiles state. "If moving up **AND** next location is greater than end of map **AND** we are not moving into an odd number row and column **THEN** proceed in this direction **ELSE** generate new direction and check again.
    e. Exp_enemy: Once hit by an explosion the enemy must stand in place until the explosion event is over. An explosion event flag from the bomb_module can be used for this. While the flag is high stay in this state, once over reset back to the idle state.

Milestone 2 Code:

//*****************LFSR*************************//**

```
module LFSR_16
(
   input wire clk, rst, w_en,                    //active high write enable
   input wire [15:0] w_in,
   output wire [15:0] out
);

reg [15:0] data_reg, data_next;                    //register to hold LSFR data

//infer registers for FSM
always @ (posedge clk, posedge rst) begin
   if(rst)
      data_reg <= 0;
   else
      data_reg <= data_next;
end

//LFSR FSM next state logic: Galois LFSR with taps at bits 16, 15, 13, and 4
always @*   begin
   if(w_en)                                  //load data when write enable active
      data_next = w_in;
   else
      begin
      data_next = data_reg[15:1];               //shift right one bit
      data_next[15] = data_reg[0];               //feedback bit 1
      data_next[14] = data_reg[15] ^ data_reg[0];   //XOR input of each tap with feedback
      data_next[12] = data_reg[13] ^ data_reg[0];
      data_next[3] = data_reg[4] ^ data_reg[0];
      end
end

assign out = data_reg;


endmodule
```

//*****************Enemy FSM*************************//**

```
// 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:    if(exp_on && enemy_on)                    //default state: determines if hit, how often it
changes direction, how fast it moves
             begin
             motion_timer_next = 0;
             enemy_hit = 1;
             e_state_next = exp_enemy;
             end
           else
             begin
             if  (motion_timer_next == motion_timer_max_next)
                begin
                if(move_cnt_next < 15)
                   begin
                   motion_timer_next = 0;
                   move_cnt_next = move_cnt_next + 1;
                   e_state_next = move_btwn_tiles;
                   end
                else
                   begin
                   motion_timer_next = 0;
                   move_cnt_next = 0;
                   LFSR_w_en = 0;
                   e_state_next = get_rand_dir;
                   end
                end
             else
                motion_timer_next = motion_timer_next + 1;
             end
  move_btwn_tiles: begin                          //move one space in current direction
```

```
            if (e_cd_next == CD_U & y_e_next > UP_LEFT_Y)
              y_e_next = y_e_next - 1;
            else if (e_cd_next == CD_D & y_e_next < LOW_RIGHT_Y)
              y_e_next = y_e_next + 1;
            else if (e_cd_next == CD_R & x_e_next < LOW_RIGHT_X)
              x_e_next = x_e_next + 1;
            else if (e_cd_next == CD_L & x_e_next > UP_LEFT_X)
              x_e_next = x_e_next - 1;
            else
              begin
              y_e_next = y_e_next;
              x_e_next = x_e_next;
              //go to rand dir if we want to stop hitting walls
              end
            e_state_next = idle;
            end
  get_rand_dir:   if (random_16[4:2] == 3'b000)              // # from LFSR determines if we change
direction, 1/8 chance of changing direction
              begin
              e_cd_next = random_16[1:0];
              e_state_next = check_dir;
              end
            else
              e_state_next = check_dir;
  check_dir:     if ((e_cd_next == CD_U) & (y_e_top[4] == 1) & (x_e_hit_l[4] == 1 | x_e_hit_r[4] == 1) &
(y_e_next > UP_LEFT_Y))
              begin
              LFSR_w_en = 0;
              e_state_next = get_rand_dir;
              end
            else if ((e_cd_next == CD_D) & (y_e_bottom[4] == 1) & (x_e_hit_l[4] == 1 | x_e_hit_r[4] == 1)
& (y_e_next < LOW_RIGHT_Y))
              begin
              LFSR_w_en = 0;
              e_state_next = get_rand_dir;
              end
            else if ((e_cd_next == CD_L) & (x_e_left[4] == 1) & (y_e_hit_t[4] == 1 | y_e_hit_b[4] == 1) &
(x_e_next > UP_LEFT_X))
              begin
              LFSR_w_en = 0;
              e_state_next = get_rand_dir;
              end
            else if ((e_cd_next == CD_R) & (x_e_right[4] == 1) & (y_e_hit_t[4] == 1 | y_e_hit_b[4] == 1) &
(x_e_next < LOW_RIGHT_X))
              begin
              LFSR_w_en = 0;
              e_state_next = get_rand_dir;
              end
```

```
            else
               e_state_next = move_btwn_tiles;
  exp_enemy:      if (~post_exp_active)
               begin
                  motion_timer_next = 0;
                  enemy_hit = 0;
                  motion_timer_max_next = motion_timer_max_next - 500;
                  e_state_next = idle;
               end
            else
               begin
               enemy_hit = 0;
               e_state_next = exp_enemy;
               end
  endcase

end
```

Final Milestone

For the Binary to BCD converter the Double Dabble algorithm was employed within an FSM. The picture below illustrates the methodology.

100's	10's	1's	Binary	Operation
			1010 0010	162
		1	010 0010	<< #1
		10	10 0010	<< #2
		101	0 0010	<< #3
		1000		add 3
	1	0000	0010	<< #4
	10	0000	010	<< #5
	100	0000	10	<< #6
	1000	0001	0	<< #7
	1011			add 3
1	0110	0010		<< #8
1	6	2		

1. An FSM with next state logic was created.
    a. Idle: During the idle state we wait until the start signal is received, indicating that the enemy has been hit. The current score is taken in and registers which store the BCD are cleared, as these registers are used each time the start signal is received.
    b. Shift: The shifting operations shown above must be done. This was accomplished by shifting the bcd register to the left by 1 bit, this state will execute *bit width of input* times, so this initial shifting will be necessary in later iterations. The MSB of the input value will be stored in the LSB of the bcd register, this was the purpose of the earlier shift. Register storing the input will then be shifted by one before leaving the state.
    c. Check_shift: the shifting process executes bit width of input times, so a counter is utilized to keep track of the number of shifts.
    d. Add: Each 4 bit nibble of the bcd register is checked. **If** the binary value of the 4 bits is greater than 4 **then** 3 is added **else** it is left alone. This allows each 4 bit nibble of the register to contain no binary value greater than 9 once the shifting process is over, each 4 bit group is to represent 1 digit.
2. Once shifting is complete, the FSM returns to the idle state and the BCD is routed out of the module into the score_display module.
3. As stated in the Project Manual, the module was simulated, and waveforms generated to verify functionality.

Final Milestone Code

```
module binary2bcd
  (
    input wire clk, reset,
    input wire start,
    input wire [13:0] in,
    output wire [3:0] bcd3, bcd2,
                bcd1, bcd0,
    output wire [3:0] count,
    output wire [1:0] state
  );

localparam [2:0]    idle            = 3'b000,  // wait for motion timer reg to hit max val
                shift           = 3'b001,  // move enemy in current dir 15 pixels
                check_shift     = 3'b010,  // get random_dir from LFSR and set r_addr to block module
block_map
                add             = 3'b011,  // check if new dir is blocked by wall or pillar
                done            = 3'b100;

localparam  input_width             = 14;

reg [2:0] bcd_state, bcd_state_next;
reg [15:0] bcd, bcd_next;
reg [13:0] bin_in, bin_in_next;
reg [3:0] shift_index, shift_index_next;

// infer registers for FSM
always @(posedge clk, posedge reset)
  if (reset)
    begin
    bcd_state       <= idle;
    bcd             <= 0;
    bin_in          <= 0;
    shift_index     <= 0;
    end
  else
    begin
    bcd_state       <= bcd_state_next;
    bcd             <= bcd_next;
    bin_in          <= bin_in_next;
    shift_index     <= shift_index_next;
    end

 always @ *
```

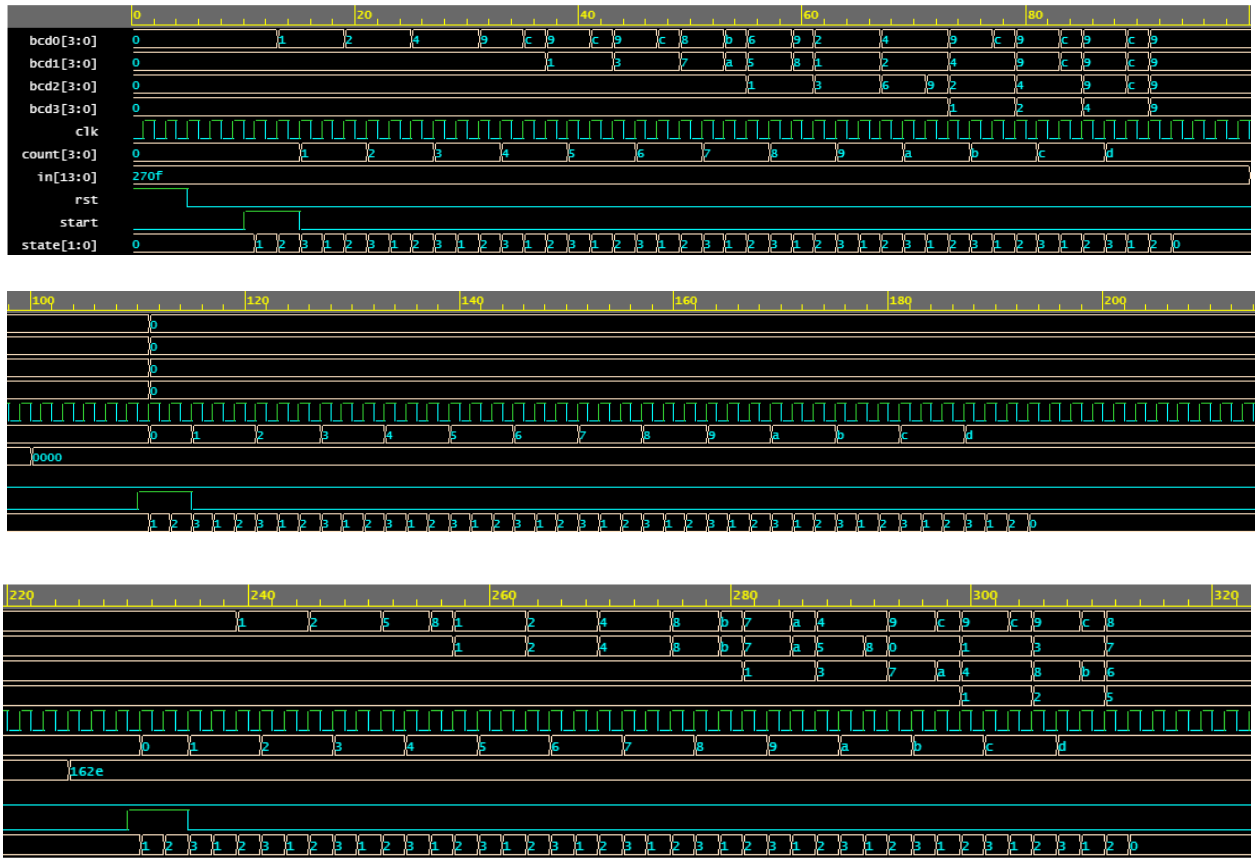
```
begin
//defaults
 bcd_state_next         = bcd_state;
 bcd_next               = bcd;
 bin_in_next            = bin_in;
 shift_index_next       = shift_index;

 case (bcd_state)
 idle:       begin
             if(start)
                begin
                bin_in_next = in;
                bcd_next = 0;
                shift_index_next = 0;
                bcd_state_next = shift;
                end
             else
                bcd_state_next = idle;
             end
 shift:      begin
                bcd_next = bcd_next << 1;
                bcd_next[0] = bin_in_next[(input_width - 1)];
                bin_in_next = bin_in_next << 1;
                bcd_state_next = check_shift;
             end
 check_shift:  begin
             if (shift_index_next == (input_width - 1))
                begin
                bcd_state_next = done;
                end
             else
                begin
                shift_index_next = shift_index_next + 1;
                bcd_state_next = add;
                end
             end
 add:        begin
             if (bcd_next[15:12] > 4)
                begin
                bcd_next[15:12] = bcd_next[15:12] + 3;
                end
             if (bcd_next[11:8] > 4)
                begin
                bcd_next[11:8] = bcd_next[11:8] + 3;
```

```
            end
          if (bcd_next[7:4] > 4)
            begin
            bcd_next[7:4] = bcd_next[7:4] + 3;
            end
          if( bcd_next[3:0] > 4)
            begin
            bcd_next[3:0] = bcd_next[3:0] + 3;
            end
          bcd_state_next = shift;
          end
  done:     begin
            bcd_state_next = idle;
          end
  endcase
 end

assign bcd0 = bcd[3:0];
assign bcd1 = bcd[7:4];
assign bcd2 = bcd[11:8];
assign bcd3 = bcd[15:12];

assign count = shift_index;
assign state = bcd_state;

endmodule
```

BCD TestBench and Wave Forms

```
module test;

 reg clk = 0;
 reg rst = 0;
 reg start = 0;
 reg [13:0] in = 14'd9999;

 wire [3:0] bcd3,bcd2,bcd1,bcd0,count;
 wire [1:0] state;


 //Instantiate design under test
 binary2bcd DUT(.clk(clk),
              .reset(rst),
              .start(start),
              .in(in),
              .bcd3(bcd3),
              .bcd2(bcd2),
              .bcd1(bcd1),
              .bcd0(bcd0),
              .count(count),
              .state(state)
             );

 initial begin
  fork
    clk = 0;
    rst = 1;
    #5 rst = 0;
    #10 start = 1;
    #15 start = 0;
    #100 in = 14'd0000;
    #110 start = 1;
    #115 start = 0;
    #225 in = 14'd5678;
    #230 start = 1;
    #235 start = 0;
  join
 end

 always begin
  #1 clk =! clk;
 end
```

```
initial begin
  $dumpfile("dump.vcd");
        $dumpvars(1);
end

initial
  #350 $finish;

endmodule
```



Conclusion

The design function mostly has required. The portion that I found most difficult, likely due to time constraints, was attempting to get the score counter to function correctly. As can be seen from the simulations above, the Binary to BCD convert work properly but when ran on hardware the score counter rapidly increases to 9980. It could be due to the explosion state triggering multiple 'hits' on the enemy. I plan to continue work on the project and would like to pursue additional features such as more interesting path planning for the enemies and the addition of sound effects. An attempt was made to add multiplayer functionality, but as the deadline approached I decided to revert back to the last working build.