REPORT ON ALARM CLOCK

NAME: RITHISH KUMAR R S **REG NO: 21BEC2521**

DATE OF SUBMMISION: 05-10-2023 **COURSE NAME: RISCV(DM+DI)**

RTL CODE:

COUNTER:

```
module counter (clk,
                reset,
                    one_minute,
                    load_new_c,
                    new_current_time_ms_hr,
                    new_current_time_ms_min,
                    new_current_time_ls_hr,
                    new_current_time_ls_min,
                    current_time_ms_hr,
                    current_time_ms_min,
                    current_time_ls_hr,
                    current_time_ls_min);
// Define input and output port directions
input clk,
    reset,
    one_minute,
    load_new_c;


input[3:0] new_current_time_ms_hr,
        new_current_time_ms_min,
        new_current_time_ls_hr,
        new_current_time_ls_min;


output [3:0] current_time_ms_hr,
         current_time_ms_min,
         current_time_ls_hr,
```

```
          current_time_ls_min;
// Define register to store current time
reg [3:0] current_time_ms_hr,
        current_time_ms_min,
        current_time_ls_hr,
        current_time_ls_min;

// Lodable Binary up synchronous Counter logic
/*****************************

// Write an always block with asynchronous reset
always@( posedge clk or posedge reset)
 begin
   // Check for reset signal and upon reset load the current_time register with default value (1'b0)
   if(reset)
     begin
           current_time_ms_hr  <= 4'd0;
       current_time_ms_min <= 4'd0;
       current_time_ls_hr  <= 4'd0;
       current_time_ls_min <= 4'd0;
      end
   // Else if there is no reset, then check for load_new_c signal and load new_current_time to
current_time register
    else if(load_new_c)
      begin
           current_time_ms_hr  <= new_current_time_ms_hr;              //   Corner cases: ms_hr
ls_hr :ms_min ls_min
           current_time_ms_min <= new_current_time_ms_min;             //          2     3     5     9
-> 00:00
           current_time_ls_hr  <= new_current_time_ls_hr;              //          0     9     5     9  ->
10:00
           current_time_ls_min <= new_current_time_ls_min;             //          0     0     5     9  ->
01:00
```

```
    end                                         //          0     0     0     9  -> 00:10
  // Else if there is no load_new_c signal, then check for one_minute signal and implement the counting algorithm
    else if (one_minute == 1)
     begin
  // Check for the corner case
  // If the current_time is 23:59, then the next current_time will be 00:00
      if(current_time_ms_hr == 4'd2 && current_time_ms_min == 4'd5 &&
        current_time_ls_hr == 4'd3 && current_time_ls_min == 4'd9)
        begin
          current_time_ms_hr  <= 4'd0;
              current_time_ms_min <= 4'd0;
              current_time_ls_hr  <= 4'd0;
              current_time_ls_min <= 4'd0;
        end
  // Else check if the current_time is 09:59, then the next current_time will be 10:00
       else if(current_time_ls_hr == 4'd9 && current_time_ms_min == 4'd5
            && current_time_ls_min == 4'd9)
         begin
                  current_time_ms_hr <= current_time_ms_hr + 1'd1;
           current_time_ls_hr <= 4'd0;
           current_time_ms_min <= 4'd0;
           current_time_ls_min <= 4'd0;
         end
  // Else check if the time represented by minutes is 59, Increment the LS_HR by 1 and set MS_MIN and LS_MIN to 1'b0
       else if (current_time_ms_min == 4'd5 && current_time_ls_min == 4'd9)
         begin
           current_time_ls_hr <= current_time_ls_hr + 1'd1;
                  current_time_ms_min <= 4'd0;
           current_time_ls_min <= 4'd0;
         end
```

```
    // Else check if the LS_MIN is equal to 9, Increment the MS_MIN by 1 and set MS_MIN to 1'b0
            else if(current_time_ls_min == 4'd9)
              begin
                    current_time_ms_min <= current_time_ms_min + 1'd1;
                    current_time_ls_min <= 4'd0;
          end
    // Else just increment the LS_MIN by 1
        else
          begin
            current_time_ls_min <= current_time_ls_min + 1'b1;
          end
      end
   end

endmodule
```

FSM:

```
module fsm (clock,
          reset,
          one_second,
          time_button,
          alarm_button,
          key,
          reset_count,
          load_new_a,
          show_a,
          show_new_time,
          load_new_c,
          shift);
```

```
// Define the input and output port direction
input clock,
    reset,
    one_second,
    time_button,
    alarm_button;

input [3:0] key;

output load_new_a,
     show_a,
     show_new_time,
     load_new_c,
     shift,
     reset_count;

// Define internal register for present state and next state
reg [2:0] pre_state,next_state;
// Define internal signal for timeout logic
wire time_out;
// Define registers for counting 10 secs in KEY_ENTRY and KEY_WAITED state
reg [3:0] count1,count2;

//states definition
parameter SHOW_TIME       = 3'b000;
parameter KEY_ENTRY       = 3'b001;
parameter KEY_STORED      = 3'b010;
parameter SHOW_ALARM      = 3'b011;
parameter SET_ALARM_TIME  = 3'b100;
parameter SET_CURRENT_TIME = 3'b101;
parameter KEY_WAITED      = 3'b110;
```

```
parameter NOKEY           = 10;

//Counts 10 seconds pulses for KEY_ENTRY state
always @ (posedge clock or posedge reset)
begin
 // Upon reset, set the count1 value to 1'b0
 if(reset)
   count1 <= 4'd0;
 // Else check if present state is a state other than KEY_ENTRY, then set the count1 value to 1'b0
 else if(!(pre_state == KEY_ENTRY))
   count1 <= 4'd0;
 // Else check if the count1 value reaches 'd9, then set the count1 to 1'b0
 else if (count1==9)
   count1 <= 4'd0;
 // Else increment the count for every one_second pulse
 else if(one_second)
   count1 <= count1 + 1'b1;
end

//Counts 10 seconds pulses for KEY_WAITED state
always @ (posedge clock or posedge reset)
begin
 // Upon reset, set the count2 value to 1'b0
 if(reset)
   count2 <= 4'd0;
 // Else check if present state is a state other than KEY_WAITED, then set the count2 value to 1'b0
 else if(!(pre_state == KEY_WAITED))
   count2 <= 4'd0;
 // Else check if the count2 value reaches 'd9, then set the count2 to 1'b0
 else if (count2==9)
   count2 <= 4'd0;
```

```
 // Else increment the count for every one_second pulse
 else if(one_second)
   count2 <= count2 + 1'b1;
end

//Time out logic  // Assert time_out signal whenever the count1 or count2 reaches 'd9
assign time_out=((count1==9) || (count2==9)) ? 0 : 1;

//Present state logic
always @ (posedge clock or posedge reset)
begin
 // Upon reset, assign the present_state as "SHOW_TIME"
 if(reset)
   pre_state <= SHOW_TIME;
 // Else if there is no reset then assign the present_state as next_state
 else
   pre_state <= next_state;
end

//Next state logic
// Whenever there is a change in input, check for present_state and assign next_state with
approriate state
always @ (pre_state or key or alarm_button or time_button or time_out)
begin
 case(pre_state)
    // State transition from SHOW_TIME to other state
    SHOW_TIME  : begin
              // Check if alarm_button is pressed, then the next state is SHOW_ALARM
               if (alarm_button)
                 next_state = SHOW_ALARM;
              // Else check if the key is pressed or not, If key pressed, then next_state is KEY_STORED
```

```
            else if (key != NOKEY)
              next_state = KEY_STORED;
            // Else if the key is not pressed, then next_state is SHOW_TIME state
                    else
              next_state = SHOW_TIME;
            end
    // In KEY_STORED state assign next_state as KEY_WAITED
    KEY_STORED : next_state = KEY_WAITED;
    // State transition from KEY_WAITED state
    KEY_WAITED : begin
            // Check if the pressed key is released, If the key is released then next_state is
KEY_ENTRY state
            if (key == NOKEY)
              next_state = KEY_ENTRY;
            // Else check if active low time_out signal is asserted, If asserted, then next_state is
SHOW_TIME state
            else if(time_out == 0)
              next_state = SHOW_TIME;
            // Else the next_state is KEY_WAITED state
            else
              next_state = KEY_WAITED;
               end
    // State transition from KEY_ENTRY state
    KEY_ENTRY  : begin
            // Check if the alarm_button is pressed, if pressed then set the next_state as
SET_ALARM_TIME state
             if (alarm_button)
              next_state = SET_ALARM_TIME;
            // Else if the time_button is pressed, then set the next_state as SET_CURRENT_TIME state
             else if (time_button)
              next_state = SET_CURRENT_TIME;
            // Else if 10sec timeout is asserted, then set the next_state as SHOW_TIME state
```

```
                else if (time_out == 0)
                 next_state = SHOW_TIME;
               // Else if the key is pressed, then set the next_state as KEY_STORED state
                else if (key != NOKEY)
                 next_state =KEY_STORED;
               // Else the next_state is KEY_ENTRY state
                        else
                 next_state = KEY_ENTRY;
               end
    // State transition from SHOW_ALARM state
    SHOW_ALARM  : begin
               // If alarm_button is pressed, then set next_state as SHOW_ALARM state else next_state
is SHOW_TIME state
                if (!alarm_button)
                 next_state = SHOW_TIME;
                        else
                 next_state = SHOW_ALARM;
               end
  // In SET_ALARM_TIME state assign next_state as SHOW_TIME
  SET_ALARM_TIME : next_state = SHOW_TIME;
  // In SET_ALARM_TIME state assign next_state as SHOW_TIME
  SET_CURRENT_TIME : next_state = SHOW_TIME;
  // Set default state as SHOW_TIME state
       default : next_state = SHOW_TIME;

 endcase
end

//Moore FSM outputs

// Assert show_new_time signal, when present state is either KEY_ENTRY or KEY_STORED or
KEY_WAITED state
```

```
assign show_new_time = (pre_state == KEY_ENTRY||
                        pre_state == KEY_STORED||
                        pre_state == KEY_WAITED) ? 1 : 0;
// Assert show_a signal when present state is SHOW_ALARM
assign show_a       = (pre_state == SHOW_ALARM) ? 1 : 0;
// Assert load_new_a signal when present state is SET_ALARM_TIME state
assign load_new_a    = (pre_state == SET_ALARM_TIME) ? 1 : 0;
// Assert load_new_c signal when present state is SET_CURRENT_TIME state
assign load_new_c    = (pre_state == SET_CURRENT_TIME) ? 1 : 0;
// Assert reset_count signal when present state is SET_CURRENT_TIME state
assign reset_count   = (pre_state == SET_CURRENT_TIME) ? 1 : 0;
// Assert shift signal when present state is KEY_STORED state
assign shift        = (pre_state == KEY_STORED) ? 1 : 0;

endmodule
```

KEYREG:

```
module keyreg(reset,
              clock,
              shift,
              key,
              key_buffer_ls_min,
              key_buffer_ms_min,
              key_buffer_ls_hr,
              key_buffer_ms_hr);
// Define input and output port direction
  input reset,
        clock,
        shift;
```

```
 input [3:0] key;
 output reg [3:0]  key_buffer_ls_min,
                   key_buffer_ms_min,
                   key_buffer_ls_hr,
                   key_buffer_ms_hr;



//////////////////////////////////////////////////////////////////
// This procedure stores the last 4 keys pressed. The FSM block
// detects the new key value and triggers the shift pulse to shift
// in the new key value.
//////////////////////////////////////////////////////////////////
always @(posedge clock or posedge reset)
begin
 // For asynchronous reset, reset the key_buffer output register to 1'b0
 if (reset)
 begin
  key_buffer_ls_min <= 0;
  key_buffer_ms_min <= 0;
  key_buffer_ls_hr <= 0;
  key_buffer_ms_hr <= 0;
 end
 // Else if there is a shift, perform left shift from LS_MIN to MS_HR
 else  if (shift == 1)
 begin
  key_buffer_ms_hr  <= key_buffer_ls_hr;
  key_buffer_ls_hr  <= key_buffer_ms_min;
  key_buffer_ms_min <= key_buffer_ls_min;
  key_buffer_ls_min <= key;
 end
```

```
end

endmodule
```

LCDDRIVE:

```
module lcd_driver (alarm_time,
                   current_time,
                   show_alarm,
                   show_new_time,
                   key,display_time,
                   sound_alarm);
//Define input and output ports direction
input [3:0] key;
input [3:0]alarm_time;
input [3:0]current_time;
input show_alarm;
input show_new_time;


output reg [7:0]display_time;
output reg sound_alarm;

//Define the internal signals
reg [3:0]display_value ;

//Define the Parameter constants to represent LCD numbers
parameter ZERO   = 8'h30;
parameter ONE    = 8'h31;
```

```
parameter TWO    = 8'h32;
parameter THREE  = 8'h33;
parameter FOUR   = 8'h34;
parameter FIVE   = 8'h35;
parameter SIX    = 8'h36;
parameter SEVEN  = 8'h37;
parameter EIGHT  = 8'h38;
parameter NINE   = 8'h39;
parameter ERROR  = 8'h3A;




always @ (alarm_time or current_time or show_alarm or show_new_time or key)
 begin
  //Displays the key_time,alarm_time or current_time as per the control signals
  if (show_new_time)
     display_value = key;
  else if (show_alarm)
     display_value = alarm_time;
  else
     display_value = current_time;
  //Generates sound_alarm logic i,e when current_time is equal to alarm_time
  if (current_time == alarm_time)
      sound_alarm = 1'b1;
  else
          sound_alarm = 1'b0;
 end

//Decoder logic
always @ (display_value)
```

```
 begin
  case (display_value)
       4'd0 : display_time = ZERO;
       4'd1 : display_time = ONE;
       4'd2 : display_time = TWO;
       4'd3 : display_time = THREE;
       4'd4 : display_time = FOUR;
       4'd5 : display_time = FIVE;
       4'd6 : display_time = SIX;
       4'd7 : display_time = SEVEN;
       4'd8 : display_time = EIGHT;
       4'd9 : display_time = NINE;
     default : display_time = ERROR;
   endcase
  end


endmodule
```

LCDDRIVER_4:

```
module lcd_driver_4 ( alarm_time_ms_hr,
                      alarm_time_ls_hr,
                      alarm_time_ms_min,
                      alarm_time_ls_min,
                      current_time_ms_hr,
                      current_time_ls_hr,
                      current_time_ms_min,
                      current_time_ls_min,
                      key_ms_hr,
                      key_ls_hr,
                      key_ms_min,
```

```
                key_ls_min,
                show_a,
                show_current_time,
                display_ms_hr,
                display_ls_hr,
                display_ms_min,
                display_ls_min,
                sound_a);
// Define input and output port directions
input [3:0] alarm_time_ms_hr,
        alarm_time_ls_hr,
        alarm_time_ms_min,
        alarm_time_ls_min,
        current_time_ms_hr,
        current_time_ls_hr,
        current_time_ms_min,
        current_time_ls_min,
        key_ms_hr,
        key_ls_hr,
        key_ms_min,
        key_ls_min;

input show_a,
    show_current_time;

output [7:0] display_ms_hr,
         display_ls_hr,
         display_ms_min,
         display_ls_min;

output sound_a;
```

```
wire sound_alarm1,sound_alarm2,sound_alarm3,sound_alarm4;
// Assert sound_a when all 4 digits matches
assign sound_a = sound_alarm1 & sound_alarm2 & sound_alarm3 & sound_alarm4 ;

//Instantiate lcd_driver as MS_HR_display
lcd_driver MS_HR (.alarm_time(alarm_time_ms_hr),
                  .current_time(current_time_ms_hr),
                  .key(key_ms_hr),
                  .show_alarm(show_a),
                  .show_new_time(show_current_time),
                  .display_time(display_ms_hr),
                  .sound_alarm(sound_alarm1));
//Instantiate lcd_driver as LS_HR_display
lcd_driver LS_HR (.alarm_time(alarm_time_ls_hr),
                  .current_time(current_time_ls_hr),
                  .key(key_ls_hr),
                  .show_alarm(show_a),
                  .show_new_time(show_current_time),
                  .display_time(display_ls_hr),
                  .sound_alarm(sound_alarm2));

//Instantiate lcd_driver as MS_MIN_display
lcd_driver MS_MIN (.alarm_time(alarm_time_ms_min),
                   .current_time(current_time_ms_min),
                   .key(key_ms_min),
                .show_alarm(show_a),
                .show_new_time(show_current_time),
                .display_time(display_ms_min),
                .sound_alarm(sound_alarm3));
```

```
//Instantiate lcd_driver as LS_MIN_display
lcd_driver LS_MIN(.alarm_time(alarm_time_ls_min),
                  .current_time(current_time_ls_min),
                  .key(key_ls_min),
                  .show_alarm(show_a),
                  .show_new_time(show_current_time),
                  .display_time(display_ls_min),
                  .sound_alarm(sound_alarm4));



endmodule
```

TIMEGEN:

```
module timegen(clock,
               reset,
               reset_count,
               fastwatch,
               one_second,
               one_minute
               );
 // Define input and output port directions
  input clock,
        reset,
        reset_count, //Resets the timegen whenever a new current time is set
        fastwatch;


  output one_second,
         one_minute;
  // Define internal registers required
  reg [13:0] count;
  reg one_second;
```

```
  reg one_minute_reg;
  reg one_minute;



//One minute pulse generation
always@(posedge clock or posedge reset)
begin
   // Upon reset, set the one_minute_reg value to zero
   if (reset)
   begin
     count<=14'b0;
     one_minute_reg<=0;
   end
   // Else check if there is a reset from alarm controller and reset the one_minute_reg and count
value
   else if (reset_count)
   begin
     count<=14'b0;
     one_minute_reg<=1'b0;
   end
   // Else check if the count value reaches 'd15359 to generate 1 minute pulse
   else if (count[13:0]== 14'd15359)
   begin
     count<=14'b0;
     one_minute_reg<=1'b1;
   end
   // Else for every posedge of clock just increment the count.
   else
   begin
     count<=count+1'b1;
     one_minute_reg<=1'b0;
```

```
  end
end

//One second pulse generation
always@(posedge clock or posedge reset)
begin
  // If reset is asserted, set one_second and counter_sec value to zero
  if (reset)
  begin
    one_second<=1'b0;
  end
  // Else check if there is reset from alarm_controller, and reset the one_second and counter_sec
value
  else if (reset_count)
  begin
    one_second<=1'b0;
  end
  // Else check if the count value reaches the 'd255 to generate and count 1 sec pulse
  else if (count[7:0]==8'd255)
  begin
    one_second<=1'b1;
  end
  // Else set the one_second and counter_sec value to zero
  else
  begin
    one_second<=1'b0;
  end
end

//Fastwatch Mode Logic that makes the counting faster
```

```
always@(*)
  begin
   // If fastwatch is asserted, make one_second equivalent to one_minute
   if(fastwatch)
     one_minute =one_second;
   // Else assert one_minute signal when one_minute_reg is asserted
   else
     one_minute =one_minute_reg;
  end

endmodule
```

ALARM_REG:

```
module alarm_reg (new_alarm_ms_hr,
                  new_alarm_ls_hr,
                  new_alarm_ms_min,
                  new_alarm_ls_min,
                  load_new_alarm,
                  clock,
                  reset,
                  alarm_time_ms_hr,
                  alarm_time_ls_hr,
                  alarm_time_ms_min,
                  alarm_time_ls_min );

// Define input and output port directions
input [3:0]new_alarm_ms_hr,
        new_alarm_ls_hr,
        new_alarm_ms_min,
        new_alarm_ls_min;
```

```
input load_new_alarm;
input clock;
input reset;

output reg [3:0] alarm_time_ms_hr,
                 alarm_time_ls_hr,
                 alarm_time_ms_min,
                 alarm_time_ls_min;

//Lodable Register which stores alarm time
always @ (posedge clock or posedge reset)
begin
  // Upon reset, store reset value(1'b0) to the alarm_time registers
  if(reset)
  begin
    alarm_time_ms_hr  <= 4'b0;
    alarm_time_ls_hr  <= 4'b0;
    alarm_time_ms_min <= 4'b0;
    alarm_time_ls_min <= 4'b0;
  end
// If no reset, check for load_new_alarm signal and load new_alarm time to alarm_time registers
  else if(load_new_alarm)
  begin
    alarm_time_ms_hr  <= new_alarm_ms_hr;
    alarm_time_ls_hr  <= new_alarm_ls_hr;
    alarm_time_ms_min <= new_alarm_ms_min;
    alarm_time_ls_min <= new_alarm_ls_min;
  end
end

endmodule
```

ALARM_CLOCK_TOP:

```
module alarm_clock_top(clock,
                       key,
                           reset,
                           time_button,
                           alarm_button,
                           fastwatch,
                           ms_hour,
                           ls_hour,
                           ms_minute,
                           ls_minute,
                           alarm_sound);
// Define port directions for the signals
input clock,
    reset,
    time_button,
    alarm_button,
    fastwatch;

input [3:0] key;

output [7:0] ms_hour,
             ls_hour,
             ms_minute,
             ls_minute;

output alarm_sound;

//Define the Interconnecting internal wires
wire one_second,
   one_minute,
```

```
    load_new_c,
    load_new_a,
    show_current_time,
    show_a,
    shift,
    reset_count;

wire [3:0] key_buffer_ms_hr,
        key_buffer_ls_hr,
        key_buffer_ms_min,
        key_buffer_ls_min,
        current_time_ms_hr,
        current_time_ls_hr,
        current_time_ms_min,
        current_time_ls_min,
        alarm_time_ms_hr,
        alarm_time_ls_hr,
        alarm_time_ms_min,
        alarm_time_ls_min;

//Instantiate lower sub-modules. Use interconnect(Internal) signals for connecting the sub modules

// Instantiate the timing generator module
timegen tgen1   (.clock(clock),
            .reset(reset),
            .fastwatch(fastwatch),
            .one_second(one_second),
            .one_minute(one_minute),
            .reset_count(reset_count));

// Instantiate the counter module
```

```
counter count1 (.one_minute(one_minute),
            .new_current_time_ms_min(key_buffer_ms_min),
            .new_current_time_ls_min(key_buffer_ls_min),
            .new_current_time_ms_hr(key_buffer_ms_hr),
            .new_current_time_ls_hr(key_buffer_ls_hr),
            .load_new_c(load_new_c),
            .clk(clock),
            .reset(reset),
            .current_time_ms_min(current_time_ms_min),
            .current_time_ls_min(current_time_ls_min),
            .current_time_ms_hr(current_time_ms_hr),
            .current_time_ls_hr(current_time_ls_hr)
            );

// Instantiate the alarm register module
alarm_reg alreg1  (.new_alarm_ms_hr(key_buffer_ms_hr),
          .new_alarm_ls_hr(key_buffer_ls_hr),
          .new_alarm_ms_min(key_buffer_ms_min),
          .new_alarm_ls_min(key_buffer_ls_min),
          .load_new_alarm(load_new_a),
          .clock(clock),
          .reset(reset),
          .alarm_time_ms_hr(alarm_time_ms_hr),
          .alarm_time_ls_hr(alarm_time_ls_hr),
          .alarm_time_ms_min(alarm_time_ms_min),
          .alarm_time_ls_min(alarm_time_ls_min));

// Instantiate the key register module
keyreg keyreg1(.reset(reset),
           .clock(clock),
           .shift(shift),
```

```
            .key(key),
            .key_buffer_ls_min(key_buffer_ls_min),
            .key_buffer_ms_min(key_buffer_ms_min),
            .key_buffer_ls_hr(key_buffer_ls_hr),
            .key_buffer_ms_hr(key_buffer_ms_hr)
            );

// Instantiate the FSM controller
fsm   fsm1  (.clock(clock),
          .reset(reset),
          .one_second(one_second),
          .time_button(time_button),
          .alarm_button(alarm_button),
          .key(key),
          .load_new_a(load_new_a),
          .show_a(show_a),
          .reset_count(reset_count),
          .show_new_time(show_current_time),
          .load_new_c(load_new_c),
          .shift(shift)
           );
// Instantiate the lcd_driver_4 module
lcd_driver_4 lcd_disp  (.alarm_time_ms_hr(alarm_time_ms_hr),
            .alarm_time_ls_hr(alarm_time_ls_hr),
            .alarm_time_ms_min(alarm_time_ms_min),
            .alarm_time_ls_min(alarm_time_ls_min),
            .current_time_ms_hr(current_time_ms_hr),
            .current_time_ls_hr(current_time_ls_hr),
            .current_time_ms_min(current_time_ms_min),
            .current_time_ls_min(current_time_ls_min),
            .key_ms_hr(key_buffer_ms_hr),
```

```
        .key_ls_hr(key_buffer_ls_hr),
        .key_ms_min(key_buffer_ms_min),
        .key_ls_min(key_buffer_ls_min),
        .show_a(show_a),
        .show_current_time(show_current_time),
        .display_ms_hr(ms_hour),
        .display_ls_hr(ls_hour),
        .display_ms_min(ms_minute),
        .display_ls_min(ls_minute),
        .sound_a(alarm_sound));

endmodule
```

ALARM_CLOCK_TB:

```
module tb_alarm_clock();

reg clk,
   reset,
   fast_watch,
   alarm_button,
   time_button;

reg [3:0] key;

wire [7:0] display_ms_hr,
        display_ms_min,
        display_ls_hr,
        display_ls_min;
```

```
wire sound_alarm;

parameter cycle = 2;

alarm_clock_top DUV(.clock(clk),
                    .reset(reset),
                    .fastwatch(fast_watch),
                    .alarm_button(alarm_button),
                    .time_button(time_button),
                    .key(key),
                    .alarm_sound(sound_alarm),
                    .ms_hour(display_ms_hr),
                    .ls_hour(display_ls_hr),
                    .ms_minute(display_ms_min),
                    .ls_minute(display_ls_min));

 //Clock generation logic
 initial
  begin
     clk = 1'b0;
     forever
     #(cycle/2) clk = ~clk;
   end

 //

 //Stimulus logic
 initial
  begin
   //Hard reset the design
```

```
 reset = 1;
 #10;
 reset = 0;
//Set fastwatch to 1 to make counting faster
fast_watch = 1;
//Set  key time to current time :11:23
key = 1;
repeat(3)
@(negedge clk);
key = 10;
@(negedge clk);
key = 1;
repeat(3)
@(negedge clk);
key = 10;
@(negedge clk);
key = 2;
repeat(3)
@(negedge clk);
key = 10;
@(negedge clk);
key = 3;
repeat(3)
@(negedge clk);
key = 10;
@(negedge clk);
time_button = 1;
@(negedge clk);
time_button = 0;

//Set  key time to alarm time :11:30
```

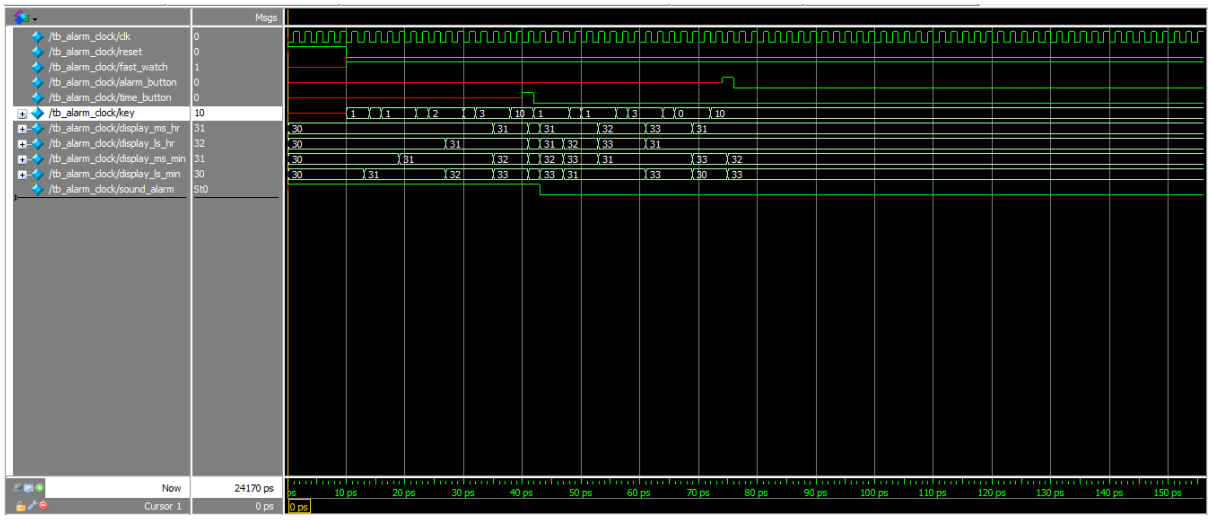
```
key = 1;
repeat(3)
@(negedge clk);
key = 10;
@(negedge clk);
key = 1;
repeat(3)
@(negedge clk);
key = 10;
@(negedge clk);
key = 3;
repeat(3)
@(negedge clk);
key = 10;
@(negedge clk);
key = 0;
repeat(3)
@(negedge clk);
key = 10;
@(negedge clk);
alarm_button = 1;
@(negedge clk);
alarm_button = 0;
#(7*256*2);//7 -> 7minutes ->7seconds->7 *256 clock cycles ->7*256*2(Time period of clock)
//Time out for Alarm clock
 //key = 7;
 repeat(4*2564) //Wait for minimum 10second pulses i.e (10*256) clock cycles
 @(negedge clk);
$finish;
end
```

```
 initial

  $monitor($time,"\-ns\t MAVEN SILICON : \tDISPLAY_MS_HR =%H >>> DISPLAY_LS_HR =%H>>>
DISPLAY_MS_MIN =%H>>>
DISPLAY_LS_MIN=%H",display_ms_hr[3:0],display_ls_hr[3:0],display_ms_min[3:0],display_ls_min[3:
0]);


endmodule
```

WAVE FORM:



SYNTHESIS: