NOTES:

An addition has been made to the project. While the 'Clock', 'Stopwatch', and 'Count Down Timer' work exactly as expected (nothing has been changed regarding these modes; the buttons still affect the timer in the expected manner), a new feature has been implemented in the form of a completely new and fun mode.

Have you ever looked at your clock and just wondered to yourself "I really want to gamble. I have an insatiable thirst to get the dopamine rush of gambling." Well now you can with the new 'Gambling' mode. Upon initialisation, you are in 'Clock' mode. When KEY3 is pressed, you will be put into 'Stopwatch' mode, another press and 'CountDownTimer' mode. A final third press of KEY3 will place you into 'Gambling' mode. LEDs 6 to 9 also correspond to the different modes.

As to how the 'Gambling' mode actually works, upon entering this mode. You will see the display showing 00:00:00. Now this is representative of *Your bet:00:Your tokens*. Pressing the reset button (KEY2) will instantly change the display to 00:00:25. This can be pressed at any time throughout the playing process in order to reset and gamble anew (like if you lost all your tokens). The 25 is representative of the number of tokens you currently have (you start with 25). Now KEY1 is connected to a FastAdvance module. As it is pressed or held, the number of tokens you have will decrease (and thus the number on the right two seven segment displays will gradually decrease). You will notice the number on the left two seven segment displays begin to increase at an equal rate. This is representative of how many tokens you are betting. After deciding how many tokens you are willing to bet, press KEY0 to start the gambling process. Now LED3 indicates that you gambled and will turn on when KEY0 is pressed and off when you release the button. LED2 indicates the outcome of your gamble. If it turns on, this means your gamble was successful and if it is off, this means your gamble was not successful. There is roughly a 50/50 chance of winning or losing any gamble in this game. If you win the gamble, you will notice that the number of tokens (the number on the 2 far right seven segment displays) will be the number of tokens you had previously, now added with the number of tokens you bet while the number displayed in the far left seven segment displays will be returned back to 00 and the game continues, this time you have more tokens then you had previously. This way you gain more if you bet more. Now if you lose the gamble, you will notice that the number of tokens (the number on the 2 far right seven segment displays) will not change at all, ie. you lost the number of tokens you bet. The number displayed in the far left seven segment displays will be returned back to 00 and the game continues, this time you have less tokens then you had previously. This way you'll be worried about betting a large number of tokens. It is not possible to bet more tokens than you currently have, it won't let you, and you can only increase your bet. Once you have placed down a token to bet, you can't take it back. I'd make it so you could but there are not enough buttons on the board to allow for this functionality. The middle two seven segment displays serve no purpose in this game.

Here is an example (You start with 25 tokens so the initial display will be 00:00:25):

Previous Display	**Bet**	**Current Display**	**Outcome**	**New Display**
00:00:25	5	05:00:20	Win	00:00:30
00:00:30	15	15:00:15	Win	00:00:45
00:00:45	20	20:00:25	Lose	00:00:25
00:00:25	3	03:00:22	Win	00:00:28
00:00:28	16	16:00:12	Lose	00:00:12
00:00:12	12	12:00:00	Lose	00:00:00

Your goal is to acquire as many tokens as you possibly can (maximum of 99). Your other goal is to have fun giving in to your crippling gambling addiction.

```
module Watch(input CLOCK_50, input [2:0] KEY,
             output [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);

    // Time module
    wire [6:0] secs;
    wire [6:0] mins;
    wire [4:0] hrs;
    Time timer(.clk(CLOCK_50), .edit(edit),
               .secs(secs), .mins(mins), .hrs(hrs));

    // For button control module
    wire [3:0] edit;
    ButtonControl buttoncontrol(.clk(CLOCK_50), .key(KEY), .display(edit));

    // For the flash control
    wire [3:0] flash_digit;
    FlashControl flashcontrol(.clk(CLOCK_50), .key(KEY[2]), .led(flash_digit));

    // For the display
    Display display_secs(.clk(CLOCK_50), .num(secs), .flash(flash_digit[1]),
                         .sseg_tens(HEX1), .sseg_ones(HEX0));
    Display display_mins(.clk(CLOCK_50), .num(mins), .flash(flash_digit[2]),
                         .sseg_tens(HEX3), .sseg_ones(HEX2));
    Display display_hrs(.clk(CLOCK_50), .num(hrs), .flash(flash_digit[3]),
                         .sseg_tens(HEX5), .sseg_ones(HEX4));
endmodule
```

Figure 22: Top-level module, Watch

```
module Time(input clk, input [3:0] edit,
            output [5:0] secs, mins, output [4:0] hrs);
    localparam N = 50000000;
    localparam BW = $clog2(N);
    wire [BW-1:0] tick;

    // Make it so only the flashing digit can be edited
    wire editablesecs;
    wire editablemins;
    wire editablehrs;
    // state 00 is seconds, state 10 is minutes and state 11 is hours
    assign editablesecs = (edit[1:0] == 2'b01) ? 1'b1 : 1'b0;
    assign editablemins = (edit[1:0] == 2'b10) ? 1'b1 : 1'b0;
    assign editablehrs = (edit[1:0] == 2'b11) ? 1'b1 : 1'b0;

    Counter #(.MAX(N-1), .WIDTH(BW)) divider(.clk(clk), .enable(edit[1:0] == 2'b00),
          .plus(edit[3]), .minus(edit[2]), .editable(1'b1), .cnt(tick));
    Counter #(.MAX(59), .WIDTH(6)) cs(.clk(clk), .enable(tick == 0),
          .plus(edit[3]), .minus(edit[2]), .editable(editablesecs), .cnt(secs));
    Counter #(.MAX(59), .WIDTH(6)) cm(.clk(clk), .enable(secs == 0 && tick == 1),
          .plus(edit[3]), .minus(edit[2]), .editable(editablemins), .cnt(mins));
    Counter #(.MAX(23), .WIDTH(5)) ch(.clk(clk), .enable(mins == 0 && secs == 0 && tick == 2),
          .plus(edit[3]), .minus(edit[2]), .editable(editablehrs), .cnt(hrs));

endmodule
```

Figure 23: Time module

```
module Counter
   #(parameter MAX = 1, WIDTH = 1, UP = 1) (
      input clk,
      input enable,
      input plus,
      input minus,
      input editable,
      output reg [WIDTH - 1:0] cnt
   );

   initial cnt = 0;

   reg [WIDTH-1:0] next_cnt;

   always @(posedge clk)
      cnt <= next_cnt;

   always @(*)
      // If both plus and minus are pressed, do nothing
      if (plus && minus);
      // If plus is pressed, increment count
      else if (plus && !minus && editable)
         next_cnt = (cnt == MAX) ? 0 : cnt + 1;
      // If minus is pressed, reduce count
      else if (minus && !plus && editable)
         next_cnt = (cnt == 0) ? MAX : cnt - 1;
      // If neither is pressed and enable is 1
      else if (enable && !plus && !minus)
         if (UP)
            next_cnt = (cnt == MAX) ? 0 : cnt + 1;
         else
            next_cnt = (cnt == 0) ? MAX : cnt - 1;
      else
         next_cnt = cnt;

endmodule
```

Figure 24: Counter module

```
module ButtonControl(input clk, input [2:0] key, output [3:0] display);

   // Mode module
   Mode mode(.clk(clk), .in(!key[2]), .out(display[1:0]));

   // The following FastAdvance module
   FastAdvance plus(.clk(clk), .in(!key[1]), .out(display[3]));
   FastAdvance minus(.clk(clk), .in(!key[0]), .out(display[2]));
endmodule
```

Figure 25: ButtonControl module

```
module Mode(input clk, in, output reg [1:0] out);
   localparam normal = 2'b00, secs = 2'b01, mins = 2'b10, hrs = 2'b11;

   reg [25:0] tracker = 0; // Tracks how much time has passed
   reg enabled = 0; // A flag to check whether a second has passed

   // Current state and next state
   reg [1:0] state, next_state;
   initial state = normal;

   // Check that the button has been held for 1 second
   always @(posedge clk) begin
      if (in) begin
         if (tracker < 50_000_000)
            tracker <= tracker + 1'b1;
         else
            enabled <= 1'b1;
   end else begin
        tracker <= 0;
        enabled <= 0;
      end
   end

   // The following is basically the RisingEdgeDetector module
   // to detect that the button has been pressed and not held
   reg prev = 0; // State
   wire next_prev;

   always @(posedge clk)
      prev <= next_prev;

   assign next_prev = in;

   wire press;
   assign press = (!prev && in);

   // Flip flop
   always @(posedge clk)
      state <= next_state;

   // Next state logic
   always @(*) begin
      case (state)
         normal: next_state = (enabled && in) ? secs : normal;
         secs: next_state = (press) ? mins : secs;
         mins: next_state = (press) ? hrs : mins;
         hrs: next_state = (press) ? normal : hrs;
         default: next_state = normal;
      endcase

      // Output logic
      out = state;
   end
endmodule
```

Figure 26: Mode module

```
module FastAdvance
    #(parameter LONG=25_000_000, PERIOD=5_000_000) (
    input clk,
    input in,
    output out);

    localparam BWL = $clog2(LONG+1);
    localparam BWP = $clog2(PERIOD);
    reg [BWL-1:0] cl = 0, next_cl;
    reg [BWP-1:0] cp = 0, next_cp;

    always @(posedge clk)
        {cl,cp} <= {next_cl,next_cp};

    // Next state logic
    always @(*) begin
    // If the button is being pressed and cl is less than the number of
    // clock cycles to wait before transitioning to the fast advance
    // mode, LONG, then increase cl by 1, otherwise if the button is
    // not pressed, set cl back to 0 and if it is, keep cl as what it
    // currently is.
       if (in == 1 && cl < LONG)
          next_cl = cl + 1;
       else
          if (in == 0)
             next_cl = 0;
          else
             next_cl = cl;
    // If the button is being pressed and cl is now equal to the number of
    // clock cycles to wait before transitioning to the fast advance
    // mode, LONG, and cp is equal to the period, set cp to 0, otherwise
    // increase cp to 0. If cl has yet to reach LONG, keep cp as 0
       if (in == 1 && cl == LONG)
          if (cp == PERIOD - 1)
             next_cp = 0;
          else
             next_cp = cp + 1;
       else
          next_cp = 0;
    end

    // While the button is pressed, output a 1 if cl is 0 (the button is
    // pressed or cl == LONG and cp == 0 (the button is held)
    assign out = (in == 1 && (cl == 0 || (cl == LONG && cp == 0))) ? 1 : 0;
endmodule
```

Figure 27: FastAdvance module

```
module FlashControl(input clk, input key, output [3:0] led);

    // Mode module
    wire [1:0] digit;
    Mode mode(.clk(clk), .in(!key), .out(digit));

    // DeMux4 module
    wire [3:0] led_output;
    DeMUX4 demux4(.in(!key), .sel(digit), .out(led_output));

    // The flashing module
    wire flash_out;
    Flash flash(.clk(clk), .out(flash_out));

    // Now make it flash
    assign led = (flash_out) ? led_output : 4'b0000;
endmodule
```

Figure 28: FlashControl module

```
module DeMUX4(input in, input [1:0] sel, output [3:0] out);
    reg [3:0] temp_out;

   always @(*)
      case(sel)
         2'b00: temp_out = 4'b0001;
         2'b01: temp_out = 4'b0010;
         2'b10: temp_out = 4'b0100;
         2'b11: temp_out = 4'b1000;
      endcase

   assign out = temp_out;
endmodule
```

Figure 29: DeMUX4 module

```
module Flash(input clk, output reg out);

    reg [24:0] counter = 0;

    always @(posedge clk)
       // 50000000Hz / 25000000 = 2 Hz
       if (counter < 25_000_000)
          counter <= counter + 1;
       else
          counter <= 0;

    // The duty cycle
    always @(posedge clk)
       // 25000000 * 0.8 = 20000000
       if (counter < 20_000_000)
          out <= 1;
       else
          out <= 0;

endmodule
```

Figure 30: Flash module

```
module Display(input clk, input [6:0] num, input flash,
               output [6:0] sseg_tens, sseg_ones);
   wire [3:0] digit_tens;
   wire [3:0] digit_ones;
   Binary2Decimal digit(.binary(num), .decimal_tens(digit_tens),
                        .decimal_ones(digit_ones));
   SSegDecoder digits_tens(.clk(clk), .digit(digit_tens), .flash(flash),
                       .sseg(sseg_tens));
   SSegDecoder digits_ones(.clk(clk), .digit(digit_ones), .flash(flash),
                       .sseg(sseg_ones));
endmodule
```

Figure 31: Display module

```
1  module Binary2Decimal(input [6:0] binary,
2      output [3:0] decimal_tens, decimal_ones);
3      assign decimal_tens = binary / 10;
4      assign decimal_ones = binary % 10;
5  endmodule
```

Figure 32: Binary2Decimal module

```
 1  module SSegDecoder(input clk, input [3:0] digit, input flash,
 2                     output reg [6:0] sseg);
 3
 4      always @(posedge clk)
 5          if (flash)
 6              sseg <= 7'b1111111;
 7          else
 8              case(digit)
 9                  0: sseg <= 7'b1000000;
10                  1: sseg <= 7'b1111001;
11                  2: sseg <= 7'b0100100;
12                  3: sseg <= 7'b0110000;
13                  4: sseg <= 7'b0011001;
14                  5: sseg <= 7'b0010010;
15                  6: sseg <= 7'b0000010;
16                  7: sseg <= 7'b1111000;
17                  8: sseg <= 7'b0000000;
18                  9: sseg <= 7'b0010000;
19                  default: sseg <= 7'b1111111;
20              endcase
21  endmodule
```

Figure 33: SSegDecoder module