UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería de Telecomunicaciones

Universidad Politécnica de Valencia

Memoria Laboratorio Proyecto 3

Diseño de un controlador de motor paso a paso

Autores:

Domínguez Martínez, Carlos

Jiménez Bou, Óscar

Telecommunications Engineering,

May 20, 2023

Índice

1. Controlador de motor paso a paso

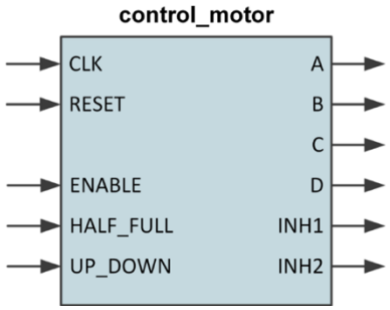

Figura 1.0. control por motor

Para este parte del trabajo, se va a diseñar un controlador para un motor paso a paso que debe emular el comportamiento del circuito integrado L297. El controlador consistirá en una máquina de ocho estados que determinará la situación del bobinado del motor. Cada cambio de estado de la máquina generará un paso y provocará el giro del motor. Además, tal y como se indica en *figura 1.0*, el controlador tendrá cinco líneas de entrada, y 6 líneas de salida.

En este código (*figura 1.1)*, se describe el código que controla el motor con las entradas: CLOCK, RESET, ENABLE, HALF_FULL y UP_DOWN. Y las salidas: A, B, C, D, INH1 e INH2.

```
18
19    module control_motor(CLK, RESET, ENABLE, HALF_FULL, UP_DOWN, A, B, C ,D, INH1, INH2);
20
21    input CLK, RESET, ENABLE, HALF_FULL, UP_DOWN;
22    output reg  A, B, C ,D;
23    output INH1, INH2;
24
25    reg [2:0] state, next_state;
26    localparam S1=3'b000, S2=3'b001, S3=3'b010, S4=3'b011, S5=3'b100, S6=3'b101, S7=3'b110, S8=3'b111;
27
28    always @(posedge CLK or negedge RESET)
29  begin
30    if(~RESET)
31       state <= S1;
32    else if (ENABLE)
33       state <= next_state;
34    end
35
36  always @(*) begin
37  case(state)
38       S1: next_state = (HALF_FULL == 1) ? ((UP_DOWN == 1) ? S2 : S8) : ((UP_DOWN == 1) ? S3 : S7);
39       S2: next_state = (HALF_FULL == 1) ? ((UP_DOWN == 1) ? S3 : S1) : ((UP_DOWN == 1) ? S4 : S8);
40       S3: next_state = (HALF_FULL == 1) ? ((UP_DOWN == 1) ? S4 : S2) : ((UP_DOWN == 1) ? S5 : S1);
41       S4: next_state = (HALF_FULL == 1) ? ((UP_DOWN == 1) ? S5 : S3) : ((UP_DOWN == 1) ? S6 : S2);
42       S5: next_state = (HALF_FULL == 1) ? ((UP_DOWN == 1) ? S6 : S4) : ((UP_DOWN == 1) ? S7 : S3);
43       S6: next_state = (HALF_FULL == 1) ? ((UP_DOWN == 1) ? S7 : S5) : ((UP_DOWN == 1) ? S8 : S4);
44       S7: next_state = (HALF_FULL == 1) ? ((UP_DOWN == 1) ? S8 : S6) : ((UP_DOWN == 1) ? S1 : S5);
45       S8: next_state = (HALF_FULL == 1) ? ((UP_DOWN == 1) ? S1 : S7) : ((UP_DOWN == 1) ? S2 : S6s);
46       default next_state = S1;
47       endcase
48    end
49
50    always @(*)
51  case(state)
52       S1: begin A = 0 ; B = 1; C = 0; D = 1; end
53       S2: begin A = 0 ; B = 0; C = 0; D = 1; end
54       S3: begin A = 1 ; B = 0; C = 0; D = 1; end
55       S4: begin A = 1 ; B = 0; C = 0; D = 0; end
56       S5: begin A = 1 ; B = 0; C = 1; D = 0; end
57       S6: begin A = 0 ; B = 0; C = 1; D = 0; end
58       S7: begin A = 0 ; B = 1; C = 1; D = 0; end
59       S8: begin A = 0 ; B = 1; C = 0; D = 0; end
60       default begin A = 0 ; B = 1; C = 0; D = 1; end
61    endcase
62
63    assign INH1 = A + B;
64    assign INH2 = C + D;
65
66    //
67
68    endmodule
69
```

Figura 1.1. Código control_motor

El comportamiento del controlador se implementa mediante una maquina de estados de manera que el estado actual se almacena en un registro de 3 bits llamado *state*, que se actualiza en el flanco de subida de la señal de CLOCK o en el flanco negativo de la señal de RESET. El siguiente estado se determina mediante una tabla de estados, que toma en cuenta los valores de las entradas HALF_FULL y UP_DOWN.

La asignación condicional se encarga de actualizar las salidas A, B, C y D, en función del estado actual. Los valores de estas salidas corresponden a la secuencia de activación de las bobinas del motor paso a paso, que se utilizan para generar el movimiento.

La asignación *assign* se utiliza para generar las señales INH1 e INH2, que son la suma de algunas de las salidas A, B, C y D, y se utilizan para habilitar o deshabilitar la corriente en las bobinas del motor.

2. Verificación del controlador

2.1. Con el Simulador Visual en Hardware

En esta parte memoria de laboratorio se describirá el procedimiento utilizado para verificar físicamente el funcionamiento del controlador para motor paso a paso utilizando el emulador realizado por los profesores de la asignatura.

En primer lugar, se descargó el archivo motor_alumno.qar desde PoliformaT y una vez abierto el proyecto, se compiló el código siguiendo los pasos indicados en las instrucciones. Finalmente, se procedió a probar el funcionamiento en la placa mediante la interacción con los diferentes botones y computadoras.

Como se puede comprobar en las siguiente Figurae, el funcionamiento del controlador es correct

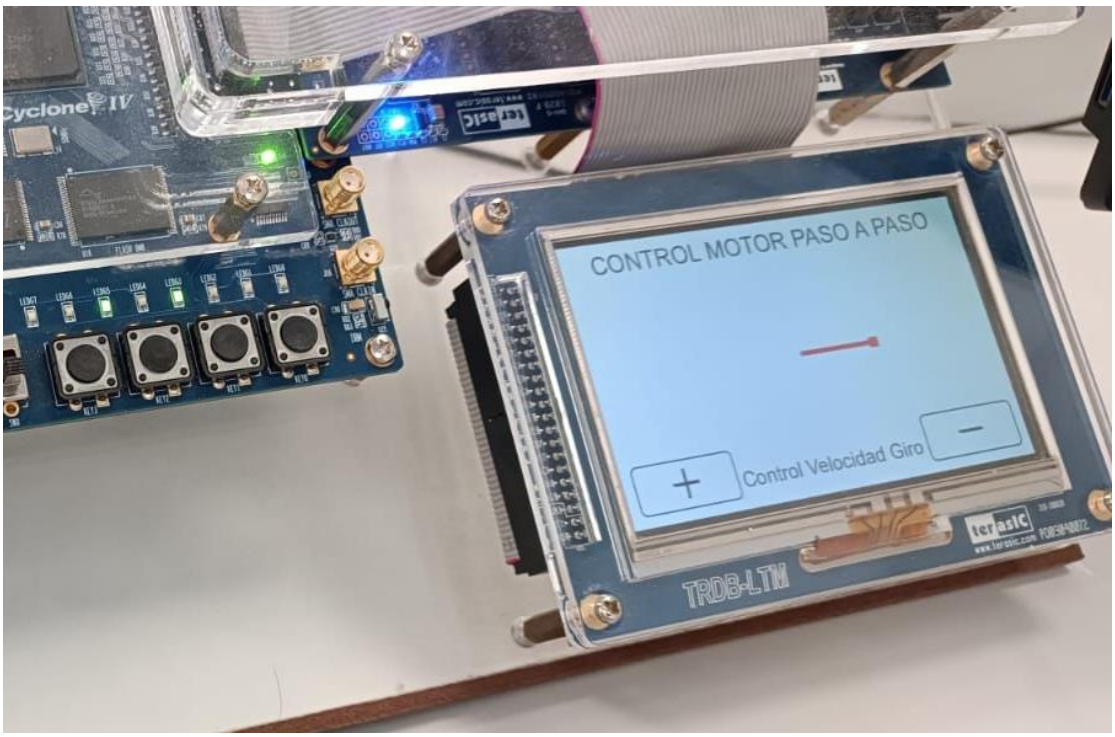

Figura 2.0. Verificación en placa del controlador

2.2. Testbench con Madelin

Además de la comprobación mediamye el simulador, hemos realizado un testbench con ModelSim para visualizar como cambian las variables y poder corroborar el funcionamiento de este.

Antes de comenzar con el módulo como tal definimos la escala de tiempo (***timescale 1ns/100ps)*** para la simulación. Una vez hecho esto tenemos que definir las entras y salidas con *wire* o *reg* respectivamente usadas en nuestro código a comprobar. Después de generar las señales para probar el módulo, lo instanciamos.

En todos nuestros programas necesitamos un clock (CLK), por lo que en todos los test bench lo tenemos que definir de la misma manera: mediante un bloque *always*, para simular el cambio según el periodo le hayamos definido.

Más tarde inicializamos las variables de entrada y las modificaremos tras cierto tiempo arbitrario para visualizar su funcionamiento en las salidas ante las combinaciones que sean de nuestro interés. En nuestro caso hemos de comprobar los 3 diferentes modos de funcionamiento de los motores en ambas direcciones.

Como podemos observar en la *figura 2.3*, esto lo provocamos con las distintas combinaciones de las variables *HALF_FULL* y *UP_DOWN*. Como particularidad de este test bench, para poder activar el *modo Normal* hemos de entrar en un tiempo impar con un retardo que es donde se ejecuta, por lo que hemos puesto un *reset* para separarlo visualmente del modo Wave. Finalmente añadimos un *$stop* para parar la simulación.

```
8   // Nombre del archivo: control_motor_tb.v
9   //
10  // Descripción: testbench del codigo que controla el motor
11  //
12  // -----------------------------------------------------------------------
13  // Versión: v1.0 | Fecha Modificación: 30/04/2023
14  //
15  // Autores: Carlos E Dominguez Martinez, y  Oscar Jimenez Bou
16  //
17  // -----------------------------------------------------------------------
18  `timescale 1ns/100ps
19  module control_motor_tb;
20  reg CLK, RESET, ENABLE, HALF_FULL, UP_DOWN;
21  wire A, B, C, D, INH1, INH2;
22  always #1 CLK = ~CLK; //Definimos el reloj
23
24  control_motor cm(CLK, RESET, ENABLE, HALF_FULL, UP_DOWN, A, B, C ,D, INH1, INH2);
25
26  initial begin //Inicializacion de las variables
27      CLK = 0;
28      RESET = 1;
29      ENABLE = 0;
30      HALF_FULL = 0;
31      UP_DOWN = 0;
32  end
33
34  initial begin
35      #10;
36     // Medio paso 1
37      ENABLE = 1;
38      HALF_FULL = 1;
39      UP_DOWN = 1;
40      #100;
41     // Medio paso 2
42      UP_DOWN = 0;
43      #100;
44     // Wave 1
45      HALF_FULL = 0;
46      UP_DOWN = 1;
47      #100;
48      // Wave 2
49      UP_DOWN = 0;
50      #100;
51     // Reset
52      RESET = 0;
53      #9 //Tiempo impar para entrar en modo normal
54     // Normal 1
55      RESET = 1;
56      HALF_FULL = 0;
57      UP_DOWN = 1;
58      #100;
59      // Normal 2
60      UP_DOWN = 0;
61      #100;
62      $finish;
63  end
64  endmodule
```

Figura 2.3. Código control_motor_tb

En la *figura 2.4* se presenta de forma total el resultado del testbench completo para una visión global de toda la simulación, mientras que en las *figuras 5-7* hemos ampliado la simulación para facilitar la comprobación teórica de cada modo de funcionamiento, y como se observa el resultado programa coincide con el esperado.

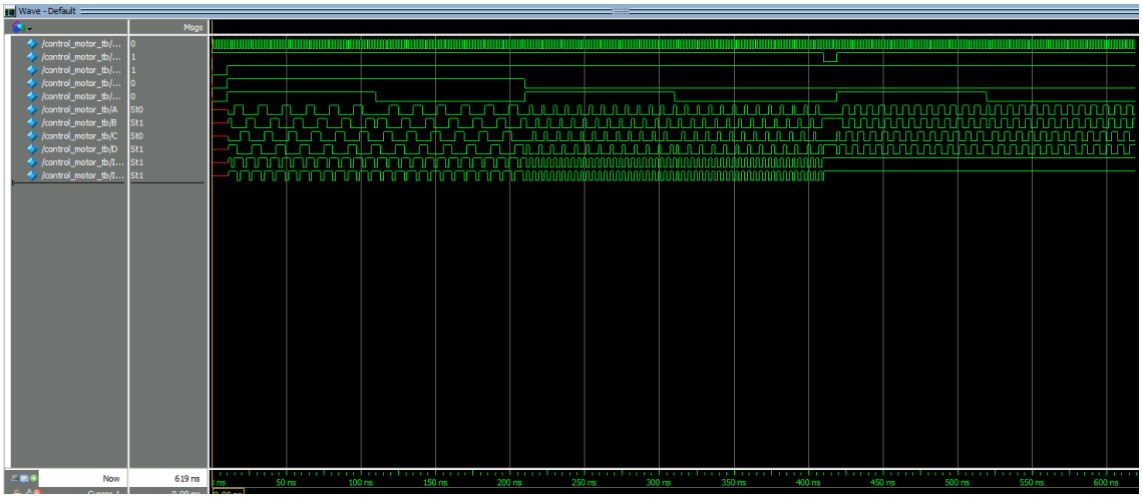

Figura 2.4. Wavefrom del control_motor_tb

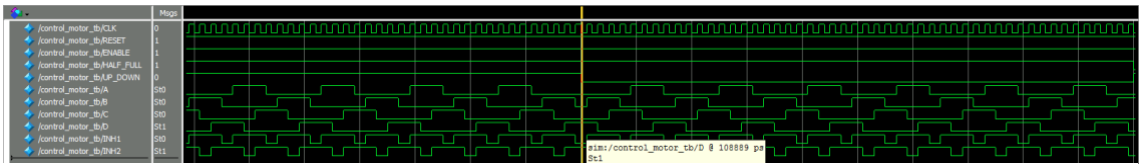

Figura 2.5. Wavefrom del modo Medio Paso

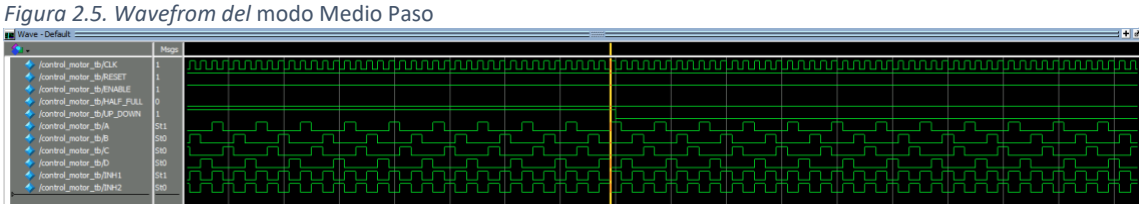

Figura 2.6. Wavefrom del modo Wave

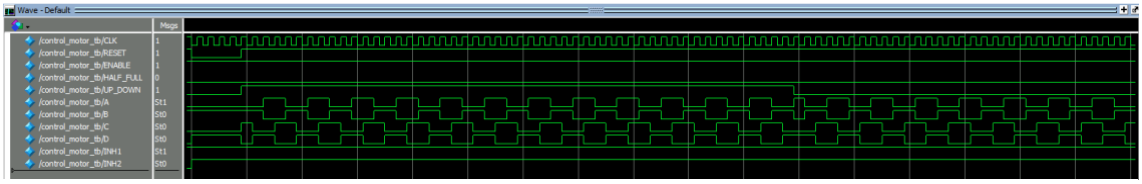

Figura 2.7. Wavefrom del modo Normal

3. Rediseño Juego de Luces

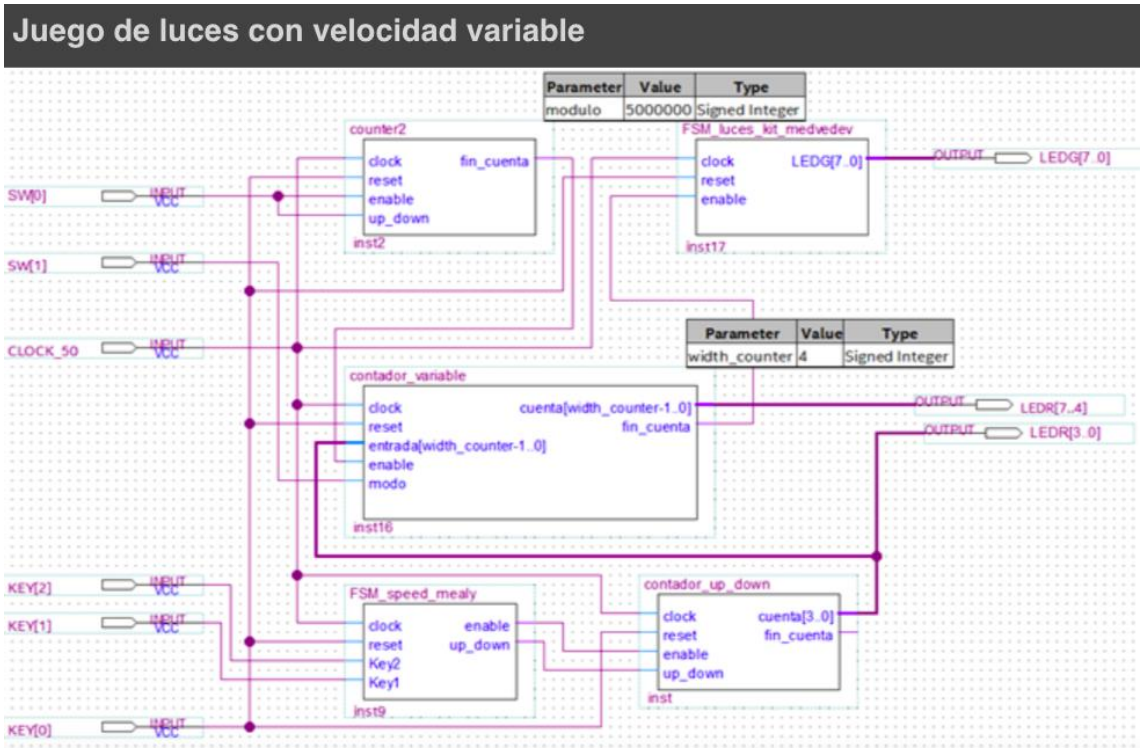

Figura 3.0. Esquemático del programa

En esta parte del proyecto se ha creado un programa general que implementa distintos módulos distingos para controlar el mismo juego de luces realizado anteriormente en el proyecto 1, con la novedad de que incluiremos también un variador de velocidad. De manera que hemos aprendido a diseñar dos máquinas de estados finitos, una de Medvedev que defina y controle los estados de los LED, y otra de Mealey para implementar el variador de velocidad en un circuito digital.

Los distintos módulos se pueden separar por contadores y máquinas de estado, comenzaremos comentando los contadores por encima ya que han sido explicado previamente en otros proyectos

```
module counter2(clock,reset,enable,up_down,fin_cuenta,cnt);

parameter MODULE = 5000000; //para que la frecuencia sea de 10Hz
parameter n =$clog2(MODULE);

input clock, reset, enable,up_down;
output fin_cuenta;
output reg [n-1:0] cnt;

always @(posedge clock, negedge reset) begin

   if(~reset)
      cnt <= 0;
   else if(enable)
      if(up_down)
         if(cnt == cnt - 1'b1)
            cnt <= 0;
         else cnt <= cnt + 1'b1;
      else if(cnt == 0)
         cnt <= MODULE - 1'b1;
         else cnt <= cnt - 1'b1;

   end

assign fin_cuenta = (up_down == 1'b1)? ((cnt == MODULE -1)? enable:1'b0):((cnt == 1'b0)? enable:1'b0);
endmodule
```

Figura 2.1. Contador de 10Hz

El primero de todos, *figura 3.1*, lo usamos simplemente para que la frecuencia a la que visualizamos el cambio de LEDs sea de 10Hz, ya que el microprocesador tiene una frecuencia de base de 50MHz, para lograr la frecuencia deseada ajustamos el módulo a 5.000.000. De manera que el fin de la cuenta de este sea la entrada enable del contador principal.

```
module contador_up_down(clock,reset,enable,up_down,fin_cuenta,cuenta);

parameter MODULE =16;
parameter n =$clog2(MODULE);

input clock, reset, enable,up_down;
output fin_cuenta;
output reg [n-1:0] cuenta;

always @(posedge clock, negedge reset) begin

   if(~reset)
      cuenta <= 0;
   else if(enable)
      if(up_down)
         if(cuenta == MODULE - 1'b1)
            cuenta <= 0;
         else cuenta <= cuenta + 1'b1;
      else if(cuenta == 0)
         cuenta <= MODULE - 1'b1;
         else cuenta <= cuenta - 1'b1;

   end

assign fin_cuenta = (up_down == 1'b1)? ((cuenta == MODULE - 1'b1)? enable:1'b0):((cuenta == 1'b0)? enable:1'b0);
endmodule

```

Figura 3.2. Contador up_down

Por otro lado tenemos el contador que se muestra en la *figura 3.2*, el cual está conectado con la máquina de estados de Mealey cuya función es proporcionar el cambio de velocidades (16 distintas), es por ello que usamos un contador up_down de modulo 16.

```
module contador_variable

#(parameter width_counter = 4)

(input clock, reset,enable, modo,
 input [width_counter-1:0] entrada,
 output fin_cuenta,
 output reg[width_counter-1:0] cuenta);

 localparam [width_counter-1:0] modulo_good = 2**width_counter-1'b1;

 wire [width_counter-1:0] cuenta_fin;

always@(posedge clock, negedge reset) begin
    if(~reset)
       cuenta <= 0;
    else if(enable)
       if(cuenta == cuenta_fin)
          cuenta <= 0;
       else cuenta <= (cuenta + 1'b1);
 end

 assign cuenta_fin = (modo == 1'b1)? entrada : modulo_good;
 assign fin_cuenta = ((cuenta == cuenta_fin) && (enable == 1'b1))? 1'b1:1'b0;

 endmodule

```

Figura 3.3. Contador Variable

Para poder agrupar el conjunto de programas y que funcionen de la manera esperada hemos implementado el código que se muestra en la *figura 3.3*, cuya función es poder variar la velocidad y el tamaño de la cuenta en función de los parámetros de entrada.

En cuanto a las máquinas de estado, comenzamos realizando una maquina de Moore de tipo Medvedev cuya función consiste en indicarle al circuito digital cuales son los estados iniciales y siguientes de la posición de los LED, de tal manera que sigan el mismo comportamiento en el juego de luces del proyecto 1.

Como se muestra en la *figura 3.4*, comenzamos el código definiendo los parámetros locales con los distintos estados que hay, para después usarlos en la codificación de estados siguientes y su respectiva codificación de salida. Para que los estados reaccionen de manera correcta hemos tenido que especificar como interaccionan respecto al reloj y al reset en el *state memory*.

```
module FSM_luces_kit_medvedev(clk, reset, LEDG);
  input clk, reset;
  reg [3:0] cuenta;
  output reg [7:0] LEDG;


  localparam ESTADO_0 =  4'b0000;
  localparam ESTADO_1 =  4'b0001;
  localparam ESTADO_2 =  4'b0010;
  localparam ESTADO_3 =  4'b0011;
  localparam ESTADO_4 =  4'b0100;
  localparam ESTADO_5 =  4'b0101;
  localparam ESTADO_6 =  4'b0110;
  localparam ESTADO_7 =  4'b0111;
  localparam ESTADO_8 =  4'b1000;
  localparam ESTADO_9 =  4'b1001;
  localparam ESTADO_10 = 4'b1010;
  localparam ESTADO_11 = 4'b1011;
  localparam ESTADO_12 = 4'b1100;
  localparam ESTADO_13 = 4'b1101;

  reg [3:0] estado_actual, estado_siguiente; // Declaración de los registros que guardan el estado actual y el estado siguiente

  //State memory
  always @(posedge clk, negedge reset) begin
    if (~reset) begin
      estado_actual <= ESTADO_0;
      cuenta <= ESTADO_0;
    end else begin
      estado_actual <= estado_siguiente;
      cuenta <= estado_actual;
    end
  end

  //Next state logic
  always @(estado_actual) begin
    case(estado_actual)
      ESTADO_0:  estado_siguiente = ESTADO_1;
      ESTADO_1:  estado_siguiente = ESTADO_2;
      ESTADO_2:  estado_siguiente = ESTADO_3;
      ESTADO_3:  estado_siguiente = ESTADO_4;
      ESTADO_4:  estado_siguiente = ESTADO_5;
      ESTADO_5:  estado_siguiente = ESTADO_6;
      ESTADO_6:  estado_siguiente = ESTADO_7;
      ESTADO_7:  estado_siguiente = ESTADO_8;
      ESTADO_8:  estado_siguiente = ESTADO_9;
      ESTADO_9:  estado_siguiente = ESTADO_10;
      ESTADO_10: estado_siguiente = ESTADO_11;
      ESTADO_11: estado_siguiente = ESTADO_12;
      ESTADO_12: estado_siguiente = ESTADO_13;
      ESTADO_13: estado_siguiente = ESTADO_0;
      default:   estado_siguiente = ESTADO_0;
    endcase
  end

    //Output logic
    always @(estado_actual) begin
      case(estado_actual)
        ESTADO_0:  LEDG = 8'b10000000;
        ESTADO_1:  LEDG = 8'b01000000;
        ESTADO_2:  LEDG = 8'b00100000;
        ESTADO_3:  LEDG = 8'b00010000;
        ESTADO_4:  LEDG = 8'b00001000;
        ESTADO_5:  LEDG = 8'b00000100;
        ESTADO_6:  LEDG = 8'b00000010;
        ESTADO_7:  LEDG = 8'b00000001;
        ESTADO_8:  LEDG = 8'b00000010;
        ESTADO_9:  LEDG = 8'b00000100;
        ESTADO_10: LEDG = 8'b00001000;
        ESTADO_11: LEDG = 8'b00010000;
        ESTADO_12: LEDG = 8'b00100000;
        ESTADO_13: LEDG = 8'b01000000;
        default:   LEDG = 8'b00000000;
      endcase
  end
  endmodule
```

Figura 3.4. Maquina de estados LEDs

Por último, se va a comentar en la figura 3.5 la máquina de Mealey, cuya función consiste en un selector de velocidades para que los LEDs se “desplacen” más o menos rápido por la placa.

```
module FSM_speed_mealey(clock, reset, key, enable, up_down);

input clock, reset;
input [1:0] key; //1 -> Mas velocidad, 0 -> menos velocidad.

output reg up_down, enable;
reg estado_actual, estado_siguiente;

localparam s0 = 1'b0;
localparam s1 = 1'b1;

always@(posedge clock, negedge reset)
    begin
        if(!reset)
            estado_actual <= s0;
        else
            estado_actual <= estado_siguiente;
    end

always @(estado_actual, key)
    begin

        if(estado_actual == s0)

        case (key)
            2'b10:   begin estado_siguiente = s1; enable = 1; up_down = 1; end
            2'b01:   begin estado_siguiente = s1; enable = 1; up_down = 0; end
            default: begin estado_siguiente = s0; enable = 0; up_down = 0; end
            endcase

        else

        case (key)
            2'b11:   begin estado_siguiente = s0; enable = 0; up_down = 0; end
            default: begin estado_siguiente = s1; enable = 0; up_down = 0; end
        endcase

    end |

endmodule
```

Figura 3.5. Maquina de Mealey

Esta máquina de estado tiene dos estados principales: s0 y s1, y las salidas de la máquina dependen tanto del estado actual como de la entrada 'key'. De forma que dependiendo de la entrada ‘key’, determinará si la maquina se moverá al estado s1 y se activará “up_down” e incrementará la cuenta. Si por el otro lado, up_down esta desactivado se decrementará la cuenta. Entre las salidas de este módulo está el valor de up_down, que marcara si sube o baja. Dicha salida estará conectada a la entrada del siguiente módulo, el contador_up_down.

4. Verificación del Juego

4.1. Con los LEDS en Hardware

Figura 4.0. Verificación en Placa

Para poder comprobar el correcto funcionamiento del programa, hemos asignado los pines de la placa correspondiente a las entradas y salidas de nuestro programa para poder controlar y visualizar el juego de luces. De esta forma los outputs serían los grupos de LEDs A, B y C.

En cuanto al funcionamiento de esta simulación, los switches D y E controlan el *modo* y el *Enable* respectivamente. Si apagamos el switch D, el juego de luces irá siempre a una velocidad constante independientemente de si queramos y si lo activamos dependerá de la velocidad que seleccionemos con los botones F. Por último tenemos el botón G que actúa como un *reset*.

A la hora de las salidas, el grupo A muestra la cuenta del contador en binario titilando de manera síncrona con nuestro reloj. El grupo B indica la velocidad en binario de forma que cuando aumentemos la velocidad se irán encendiendo más LEDs dependiendo del número de la velocidad, sucediendo lo contrario si disminuimos la velocidad. Finalmente, el juego de luces se ejecuta en el grupo de LEDs C pudiéndose observar el vaivén de un lado a otro con distintas velocidades.

4.2. Mediante un Testbench

Para asegurarnos que el programa funciona perfectamente hemos realizado un testbench para cada uno de los módulos de este.

Comenzando con los contadores, *figuras 4.1-6*, todos los códigos del testbench siguen la misma estructura: indexamos el módulo original, inicializamos las variables y después realizamos distintas combinaciones de estas para verificar que todas las posibilidades de funcionamiento no tienen ningún error.

Si analizamos todos los ModelSim con detenimiento podemos comprobar como los contadores funcionan según lo esperado, por lo que podemos pasar a comprobar el último bloque del proyecto, las máquinas de estado.

```
`timescale 1ns/100ps

module tb_counter2();

reg clock;
reg reset;
reg enable;
reg up_down;
wire fin_cuenta;

// Instanciar el contador
counter2 dut(clock,reset,enable,up_down,fin_cuenta);
defparam dut.MODULE = 10;

// Generador de reloj
always #5 clock = ~clock;

// Estímulo
initial begin
    // Inicializar señales
    clock = 0;
    reset = 0;
    enable = 0;
    up_down = 0;

    // Reset
    #10 reset = 1;

    // Prueba 1: Contar hacia arriba
    enable = 1;
    up_down = 1;

    // Prueba 2: Contar hacia abajo
    #50 up_down = 0;

    // Terminar simulación
    #50 $finish;
end
endmodule
```

Figura 4.1. tb_counter2

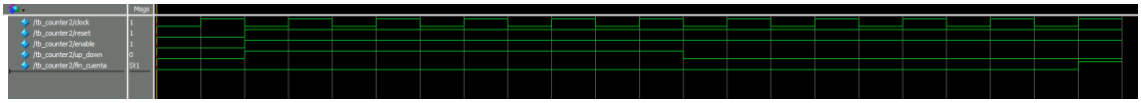

Figura 4.1. ModelSim tb_counter2

```
`timescale 1ns/100ps

module tb_contador_up_down();

reg clock;
reg reset;
reg enable;
reg up_down;
wire fin_cuenta;
wire [3:0] cuenta;

// Instanciar el contador
contador_up_down dut(clock,reset,enable,up_down,fin_cuenta,cuenta);

// Generador de reloj
always #5 clock = ~clock;

// Estímulo
initial begin
    // Inicializar señales
    clock = 0;
    reset = 0;
    enable = 0;
    up_down = 0;
    reset = 0;


    // Prueba 1: Contar hacia arriba
    #10 reset=1;
    enable = 1;
    up_down = 1;


    // Prueba 2: Contar hacia abajo
    #80 up_down = 0;

    // Terminar simulación
    #100 $finish;
end
endmodule
```

Figura 4.3. tb_contador_up_down

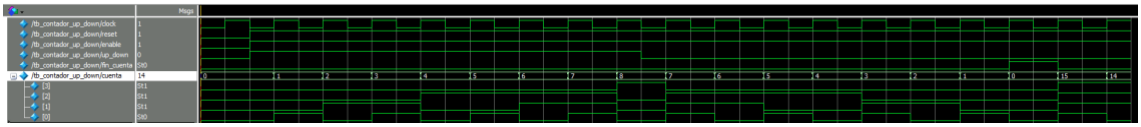

Figura 4.4. ModelSim tb_contador_up_down

```
`timescale 1ns / 100ps

module tb_contador_variable();
reg clock, reset,enable,modo;
reg [3:0] entrada;
wire fin_cuenta;
wire [3:0] cuenta;

// Instanciar el contador
contador_variable #(4) contador_inst (
    .clock(clock),
    .reset(reset),
    .enable(enable),
    .modo(modo),
    .entrada(entrada),
    .fin_cuenta(fin_cuenta),
    .cuenta(cuenta)
);
// Generador de reloj
always #5 clock = ~clock;

// Estímulo
initial begin
    // Inicializar señales
    clock = 0;
    reset = 0;
    enable = 0;
    modo = 0;
    entrada = 4'b0000;

    // Reset
   #5 reset = 1;

    // Prueba 1: Contar con modo = 0 (cuenta hasta el módulo máximo)
     enable = 1;
    modo = 0;

    // Prueba 2: Contar con modo = 1 (cuenta hasta el valor de entrada)
    #100 entrada = 4'b0110; // Cuenta hasta 6

    #10 modo = 1;

    // Terminar simulación
    #1000 $finish;
end
endmodule
```

Figura 4.5. tb_contador_variable

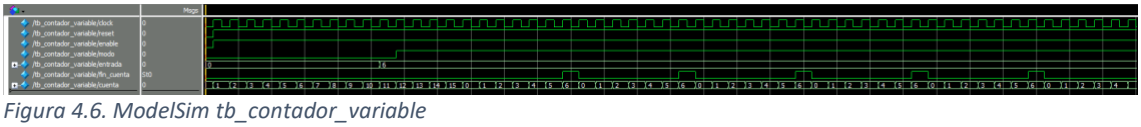

Figura 4.6. ModelSim tb_contador_variable

En este caso *(figura 4.7)*, el testbech es algo más simple puesto que hemos de comprobar que los estados cambian según lo esperado para el correcto funcionamiento del juego de luces. Por lo que hemos creado una simulación finita mediante un bucle lógico que cuenta hasta 100, donde se acaba la simulación. Y como se observa en la simulación *(figura 4.8)*, nuestro programa realiza el comportamiento deseado en el juego de luces.

```
`timescale 1ns/100ps
module tb_fsm_luces_kit_medvedev;

reg clk, reset; // Inputs
wire [7:0] LEDG; // Outputs

FSM_luces_kit_medvedev dut(clk, reset, LEDG);  // Instanciamos el modulo

always #5 clk = ~clk; // Generamos el reloj

  // Inicializamos las variables
  initial begin
    clk = 0;
    reset = 0;
    #10 reset = 1;
  end

  // Creamos un limite de 100 tiempos de reloj
  integer i;
  initial begin
    for (i = 0; i < 100; i = i + 1) begin
      #5;
    end
    $finish;
  end

endmodule
```

Figura 4.7. tb_fsm_luces_kit_medvedev

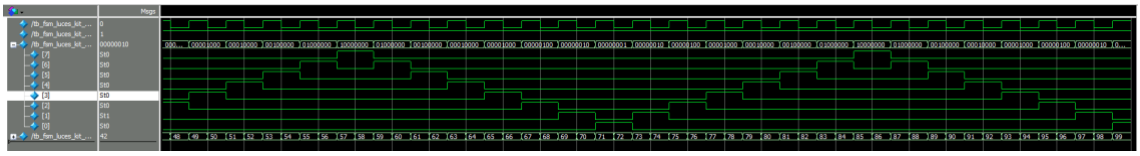

Figura 4.8. ModelSim tb_fsm_luces_kit_medvedev

Por último, en esta máquina de estado *(figura 4.9)* hemos de comprobar el funcionamiento de los “key”, que son nuestros selectores de velocidad, no generen *glitches* si los pulsamos a la vez. Cosa que no sucede en ningún momento *(figura 4.10)*

```
`timescale 1ns / 100ps
module tb_FSM_speed_mealey();

reg clock;
reg reset;
wire enable;
wire up_down;
reg [1:0] key;

// Instanciar la máquina de estados
FSM_speed_mealey dut(clock, reset, key, enable, up_down);

// Generador de reloj
always #5 clock = ~clock;

// Estímulo
initial begin
    // Inicializar señales
    clock = 0;
    reset = 0;
    key = 2'b00;

    // Reset
    #5 reset = 1;

    // Prueba 1: key = 2'b10 (enable = 1, up_down = 1)
    #20 key = 2'b01;
    #30 key = 2'b00;

    // Prueba 2: key = 2'b01 (enable = 1, up_down = 0)
    #10 key = 2'b01;
    #40 key = 2'b00;

    // Prueba 3: key = 2'b11 (enable = 0, up_down = 0)
    #20 key = 2'b11;
    #30 key = 2'b00;

    // Terminar simulación
    #50 $finish;
end

endmodule
```

Figura 4.9. ModelSim tb_fsm_luces_kit_medvedev

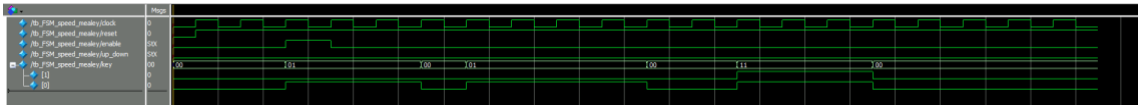

Figura 4.10. ModelSim tb_fsm_luces_kit_medvedev