Informe guía 4

Laboratorio de Electrónica Digital

Autor:
Simón Aulet

Noviembre 2025

Universidad Nacional de Río Negro
Ingeniería Electrónica y en Telecomunicaciónes

Índice

Parte I
Ejercicio 1

Se pide un secuenciador de dos luces usando máquina de Moore. Se trabaja generando módulos con propósitos específicos para mayor robustes y claridad. A continuación se detalla cada módulo

0.1 freq_divider

```
module freq_divider #(
  parameter mf_divider = 100_000,
  parameter lf_divider = 1_000
)(
input  wire clk_in,
output reg  tick_mf,
output reg  tick_lf
);
```

Listing 1: Módulo freq_divider

Para el divisor de frecuencia, se usa un sistema de pulsos síncronos llamados ticks en lugar de un reloj secundario, evitando problemas de timing. Las salidas tick_mf y tick_lf generan pulsos a 1 kHz y 1 Hz respectivamente. Los parámetros mf_divider y lf_divider definen las divisiones de frecuencia respecto al reloj principal y la frecuencia media, con lf_divider siendo un contador más pequeño por la división en cascada. En el testbench, se configuraron divisiones de 100 y 5 para acortar la simulación.

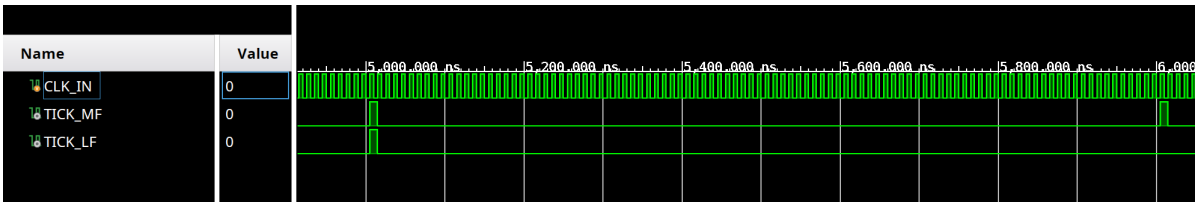

Figura 1: Waveform del divisor de frecuencia

Se puede ver cómo los pulsos están perfectamente sincronizados con el reloj principal y entre sí (en el momento que coinciden)

0.2 state_change

```
module state_change(
  input            btn,
  input            tick_mf,
  input            clk,
  output reg [1:0] state
);
```

Listing 2: Módulo state_change

Máquina de estados simple que varía de un estado a otro según cuando se pulsa el botón. Tiene incluido un sistema anti-bounce que aprovecha la media frecuencia de tick_mf para reducir la magnitud de los contadores.

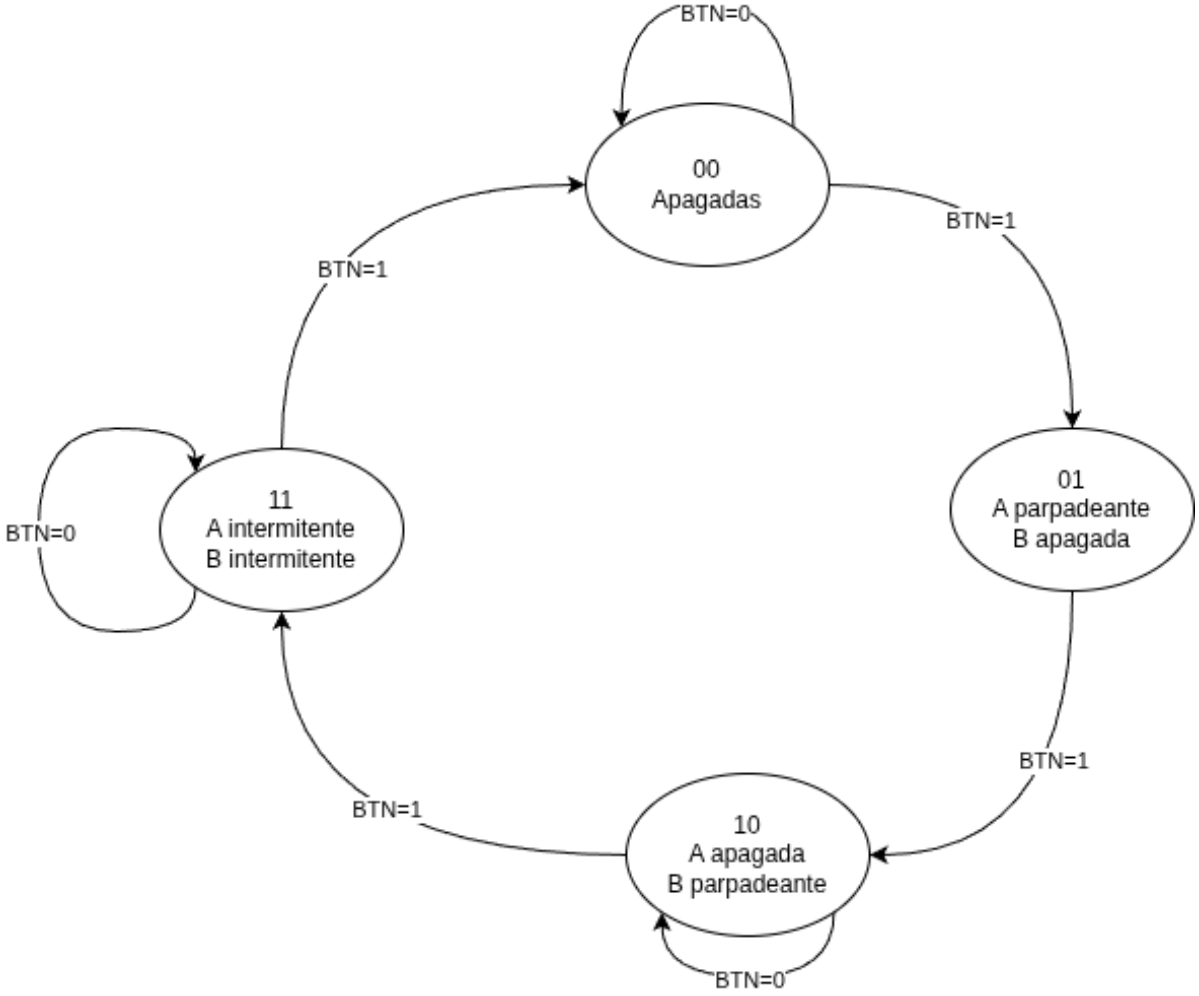

Figura 2: Diagrama de estados del ejercicio 1

0.3 led_change

```
module led_change(
  input  wire[1:0] state,
  input  wire      clk,
  input  wire      tick_lf,
  output reg       led_a,
  output reg       led_b
);
```

Listing 3: Módulo led_change

Controla los leds sincronizando su parpadeo cuando corresponde mediante tick_lf. Cuando ambos leds parpadean en simultáneo, el status del led_a se copia a led_b impidiendo que queden desfasados.

Parte II
Ejercicio 2

Para el circuito de alarma de 3 estados se trabaja usando el módulo freq_divider del módulo anterior. Se propone la siguiente lógica de funcionamiento artísticamente colorida:

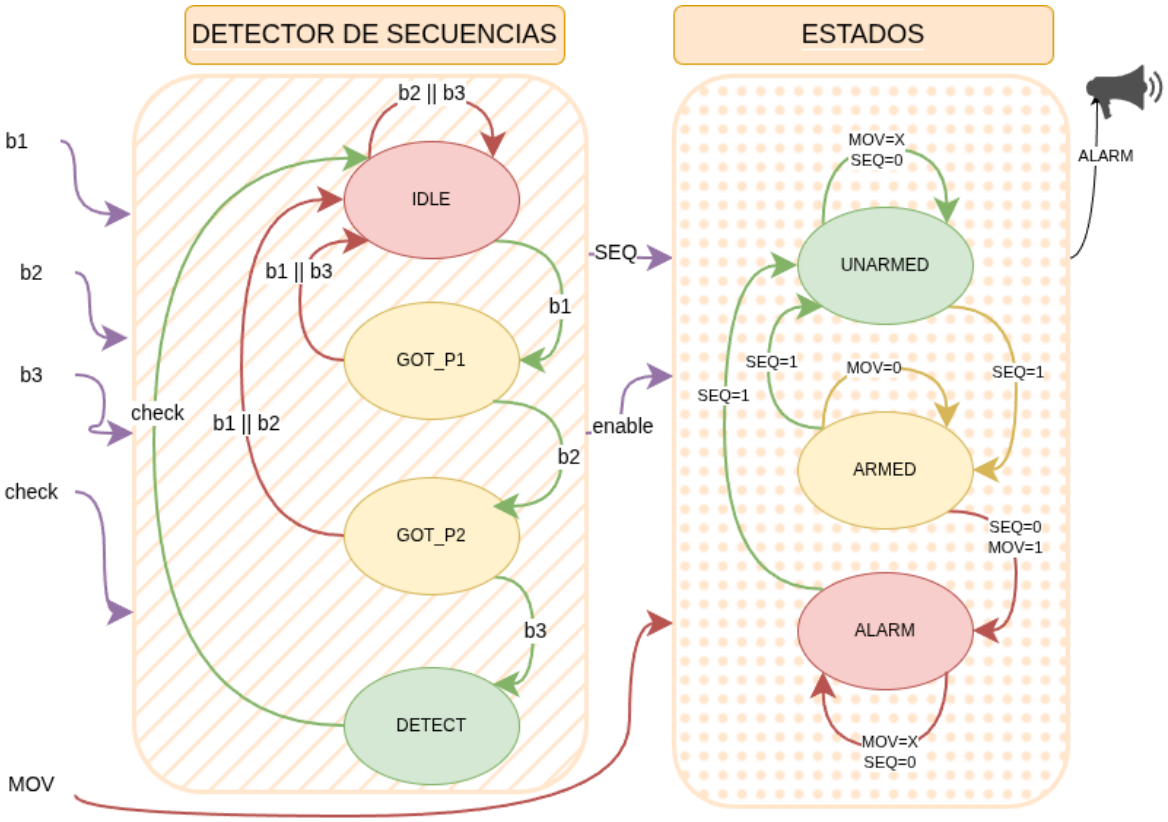

Figura 3: Diagrama de flujo del sistema de alarma

Se trata de two máquinas de estados, una para el detector de secuencias y otra para el cambio entre los estados.

0.4 seq_detecter

```
module seq_detecter(
  input wire b1,
  input wire b2,
  input wire b3,

  input wire check,
  input wire clk,

  output reg seq,
  output wire enable
);
```

Listing 4: Módulo seq_detecter

El detector de secuencias detecta la secuencia b1 –> b2 –> b3 la cual está "hard-coded" en la máquina de estados. Queda a futuro implementar que la secuencia se pase como parámetro. Una vez ingresada la secuencia de manera correcta, la máquina se mantiene en el estado DETECT y seq queda en alto. Cuando se presióna la entrada check, la salida enable vale 1 durante un pulso de reloj. Luego enable y seq se resetean a 0 y se vuelve al estado inicial En la siguiente imagen se puede ver el testbench. STATE_CHECK es un reg auxiliar, usado para este tesbench.

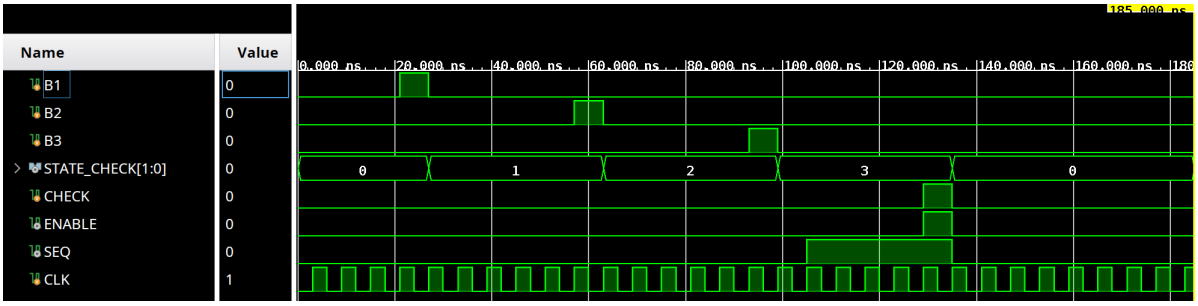

Figura 4: Waveform del detector de secuencias

Se puede ver cómo al presionar check (que dura un pulso por estar controlado por anti_bounce, detallado más adelante) enable se activa y luego se reinicia junto con seq a 0, al igual que el estado que vuelve a 0.

0.5 state_change

```
module state_change(
  input    seq,
  input    mov,
  input    enable,
  input    clk,

  output wire [1:0] state
);
```

Listing 5: Módulo state_change

siguiendo la lógica de la imagen al inicio de este ejercicio, cambia de estados según las entradas. Una secuencia incorrecta en el estado `ARMED` dispara la alarma mientras que una secuencia correcta la desactiva tanto en `ARMED` como en `ALARM`. En el gráfico las transiciónes son `seq=1` o `seq=0` pero en verdad se necesita presionar check para que se habilite enable en la máquina de estados y se hagan las transiciónes. En el waveform del testbench podemos ver los cambios de estado:

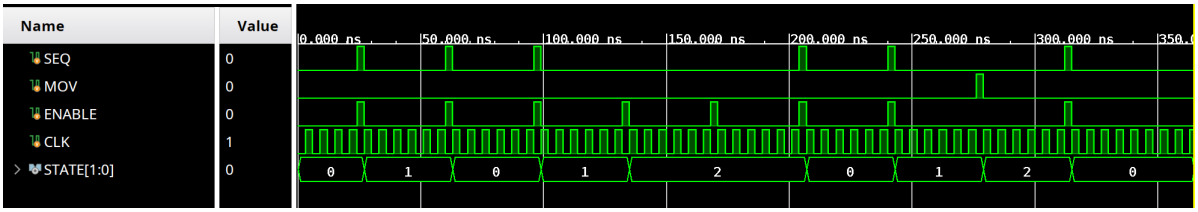

Figura 5: Waveform del cambio de estados

- Se inicia en estado IDLE. Se coloca la secuencia y se activa enable. Se pasa al estado ARMED (1). - Se ingresa la secuencia nuevamente y se activa check. La alarma pasa a DESARMED (0) - Se activa nuevamente y luego se simula un ingreso de secuencia incorrecto (enable=1, seq=0). Se pasa al estado ALARM (2) - Luego se desactiva la alarma haciendose la transición ALARM ->IDLE. Se vuelve a activar y se activa el detector de movimiento disparandose nuevamente la alarma - Finalmente se desactiva la alarma

0.6 anti_bounce

El sistema anti-rebote usado en el ejercicio anterior se desacopla y se genera un módulo a parte para poder usarlo en la posterioridad.

```
module anti_bounce(
  input btn_in,
  input tick_mf,
  input clk,

  output btn_out,
);
```

Listing 6: Módulo anti_bounce

Notese que se hace uso de tick_mf. Este módulo hace que se necesite presionar el botón durante 20 ms para que se considere su entrada. Su disparo es un solo pulso sincrono con el reloj que dura un pulso. Se activa cuando se suelta el botón, no cuando se presiona. Este módulo está instanciado para cada botón del FPGA

0.7 hw_control

```
module hw_control(
  input  wire[1:0] state,

  output alarm,
  output armed_led
);
```

Listing 7: Módulo hw_control

Simplemente ordena control de hardware, asignando los estados de state_change a salidas físicas del FPGA, activandose armed_led cuando se está en estado armado y alarm cuando la alarma se dispara (simulada con un led)

Parte III
Ejercicio 3

El proyecto utiliza una ***Nexys − 4 (XC7A100T-1CSG324C)*** en donde se busca implementar el sensor de temperatura integrado en la plaqueta, mostrando el valor en su display 7-Segmentos.

0.8 Arquitectura del Sistema

La fuente principal del diseño temp_display.v , [7:0] an es el control de anodos para el 7 segmentos y la siguiente salida es el control de segmentos para el display

```
module temp_display(
      input  wire clk,
      input  wire reset,
      output wire [7:0] an,
      output wire [7:0] seg
);
```

Figura 6: Diagrama de bloques del sistema de temperatura

Inicialmente, se intentó implementar la lectura de temperatura mediante el sensor externo **ADT7420** de la placa, utilizando un módulo de comunicación **I²C** desarrollado (i2c_temp_reader).
Pero el diseño presentó problemas de funcionamiento debido a la complejidad del protocolo I²C y a la sincronización entre las señales SDA y SCL.

Como alternativa, se decidió reemplazar este módulo por el **IP Core "XADC Wizard"** de Xilinx, que permite leer directamente la **temperatura interna del FPGA** a través de su conversor analógico–digital integrado.

Esta decisión simplificó el diseño, eliminó la necesidad de manejo del bus I²C y garantizó una lectura de temperatura confiable, permitiendo concentrar el desarrollo en la visualización de los datos mediante el display de 7 segmentos.

El temp_data tiene los datos crudos desde el sensor. El data_ready indica cuándo hay nuevos datos disponibles. El registro tiene el valor de display procesado para ser mostrado.

```
10        // Señales para XADC
11        wire [15:0] temp_data;
12        wire data_ready;
13        reg [15:0] display_val = 0;
```

Figura 7: Implementación con XADC Wizard

```
15    // Instancia del XADC - SOLO PUERTOS ESENCIALES
16    xadc_wiz_0 xadc_inst (
17        .dclk_in(clk),          // Reloj del sistema
18        .reset_in(reset),       // Reset
19        .daddr_in(8'h00),       // Dirección 00h = temperatura interna del FPGA
20        .den_in(1'b1),          // Siempre habilitar lectura
21        .dwe_in(1'b0),          // Solo lectura
22        .di_in(16'h0000),       // Datos de entrada (no usado)
23        .do_out(temp_data),     // Datos de temperatura
24        .drdy_out(data_ready)   // Datos listos
25        // ELIMINAR vauxp4, vauxn4, vp_in, vn_in - NO SON NECESARIOS
26    );
```

Figura 8: Proceso de conversión XADC

El XADC convierte la temperatura analógica interna a digital ->Cuando la conversión está lista, activa data_ready ->Los datos de temperatura (12-16 bits) salen por temp_data

Es el **traductor** que toma el número binario de temperatura y lo convierte en el formato adecuado para que el display físico muestre los dígitos correctamente.

Esta instancia completa la cadena: **XADC → Procesamiento → Driver → Display Físico.**

0.9 Modulo XADC

Sensor de Temperatura: Se utilizó el XADC interno del FPGA configurado en modo continuo para lectura automática del sensor térmico.

0.9.1. Parámetros principales:

- Canal: Sensor interno (00h)
- Resolución: 12 bits
- Reloj: 100 MHz con divisor a 25 MHz.

0.10 Procesamiento de datos

El bloque secuencial que captura los datos de temperatura cuando el XADC indica que están listos (data_ready), tomando los 12 bits más significativos del registro de temperatura y formateándolos para su visualización.

```
28        // Instancia del driver del display
29        display_driver disp (
30            .clk(clk),
31            .number(display_val),
32            .an(an),
33            .seg(seg)
34        );
35
```

Figura 9: Traductor de datos de temperatura

```
36    // Procesar temperatura cuando los datos estén listos
37    always @(posedge clk) begin
38        if (data_ready) begin
39            // Los 12 bits de temperatura están en temp_data[15:4]
40            display_val <= {4'b0000, temp_data[15:4]}; // Valor raw
41        end
42    end
```

Figura 10: Procesamiento de datos de temperatura

0.11 Driver de Display:

Implementa multiplexación temporal para controlar 4 dígitos de 7 segmentos, convirtiendo el valor binario de temperatura a BCD y refrescando cada dígito a 1 kHz para persistencia visual.

```
1  `timescale 1ns / 1ps
2
3  module display_driver(
4      input  wire clk,              // reloj de 100 MHz
5      input  wire [15:0] number,    // número binario a mostrar (0-9999)
6      output reg  [7:0] an,         // ánodos (activos bajos)
7      output reg  [7:0] seg         // segmentos (activos bajos)
8  );
9
10     //-----------------------------------
11     // 1. Divisor de frecuencia (≈1 kHz)
12     //-----------------------------------
13     reg [16:0] refresh_counter = 0;
14     reg [1:0] digit_select = 0;  // Cambiado a 2 bits para 4 dígitos
15
16     always @(posedge clk) begin
17         refresh_counter <= refresh_counter + 1;
18         if (refresh_counter == 17'd100_000) begin // 100MHz / 100000 = 1kHz
19             refresh_counter <= 0;
20             digit_select <= digit_select + 1;
21         end
22     end
23
```

Figura 11: Driver de display - parte 1

```
24      //-----------------------------------
25      // 2. Conversión binario -> BCD (Double Dabble algorithm)
26      //-----------------------------------
27      wire [3:0] thousands, hundreds, tens, ones;
28
29      bin_to_bcd converter (
30          .bin(number),
31          .thousands(thousands),
32          .hundreds(hundreds),
33          .tens(tens),
34          .ones(ones)
35      );
36
37      //-----------------------------------
38      // 3. Selección de dígito activo
39      //-----------------------------------
40      reg [3:0] current_digit;
41
42      always @(*) begin
43          case (digit_select)
44              2'd0: begin an = 8'b11111110; current_digit = ones;      end
45              2'd1: begin an = 8'b11111101; current_digit = tens;      end
46              2'd2: begin an = 8'b11111011; current_digit = hundreds;  end
47              2'd3: begin an = 8'b11110111; current_digit = thousands; end
48              default: begin an = 8'b11111111; current_digit = 4'd0; end
49          endcase
50      end
51
```

Figura 12: Driver de display - parte 2

```
52      //----------------------------------
53      // 4. Decodificador 7 segmentos
54      //----------------------------------
55      always @(*) begin
56          case (current_digit)
57              4'd0: seg = 8'b11000000; // 0
58              4'd1: seg = 8'b11111001; // 1
59              4'd2: seg = 8'b10100100; // 2
60              4'd3: seg = 8'b10110000; // 3
61              4'd4: seg = 8'b10011001; // 4
62              4'd5: seg = 8'b10010010; // 5
63              4'd6: seg = 8'b10000010; // 6
64              4'd7: seg = 8'b11111000; // 7
65              4'd8: seg = 8'b10000000; // 8
66              4'd9: seg = 8'b10010000; // 9
67              default: seg = 8'b11111111; // apagado
68          endcase
69      end
70
71  endmodule
72
73  //------------------------------------------------------------
74  // Módulo separado para conversión binario a BCD (Double Dabble)
75  //------------------------------------------------------------
76  module bin_to_bcd(
77      input [15:0] bin,
78      output reg [3:0] thousands,
79      output reg [3:0] hundreds,
80      output reg [3:0] tens,
81      output reg [3:0] ones
82  );
```

Figura 13: Driver de display - parte 3

```
84      integer i;
85      reg [19:0] bcd; // 4 dígitos BCD de 4 bits cada uno
86
87      always @(*) begin
88          bcd = 20'd0;
89          for (i = 0; i < 16; i = i + 1) begin
90              // Ajustar dígitos BCD
91              if (bcd[3:0] >= 5) bcd[3:0] = bcd[3:0] + 3;
92              if (bcd[7:4] >= 5) bcd[7:4] = bcd[7:4] + 3;
93              if (bcd[11:8] >= 5) bcd[11:8] = bcd[11:8] + 3;
94              if (bcd[15:12] >= 5) bcd[15:12] = bcd[15:12] + 3;
95              if (bcd[19:16] >= 5) bcd[19:16] = bcd[19:16] + 3;
96
97              // Desplazar
98              bcd = bcd << 1;
99              bcd[0] = bin[15 - i];
100         end
101
102         // Asignar salidas
103         thousands = bcd[19:16];
104         hundreds = bcd[15:12];
105         tens = bcd[11:8];
106         ones = bcd[7:4];
107     end
108
109 endmodule
```

Figura 14: Driver de display - parte 4