Universidad ORT

OBLIGATORIO 2 - DISEÑO DIGITAL

FACULTAD DE INGENIERÍA

Agustín Costábile (281288)

Mateo Silva (282489)

Luca Stefanoli (285859)

Profesores: Diego Sáez, Gonzalo Díaz

30 de noviembre de 2023

Índice

1. Introducción

En este archivo se encuentra la documentación de la materia de Diseño Digital. Dicho obligatorio consiste en el diseño en *VHDL* de un sistema de control de un estacionamiento. Se simula un sensor en cada cubículo, siendo 10 en total, siendo en este trabajo cada sensor igual a un *switch* de la placa Altera, desde SW0 a SW9, que simboliza un estado ocupado con la posición alta de valor lógico '1', y la posición baja representa un estado libre. Además, se utilizarán los LEDs presentes en las placas para facilitar la nevegación entre menúes, así como se utilizarán los cuatros displays de siete segmentos disponibles. Para la realización del diseño, se planteó una solución que divida el problema en pequeños bloques, que trabajen de forma interconectada, derivando de esta forma las tareas de una forma controlada, buscando siempre la mayor eficiencia posible.

2. Especificaciones del sistema

El sistema implementado se realizó para que funcione de la siguiente manera: Los dos displays de derecha de la placa (*HEX_0* y *HEX_1*), muestran en cada instante de funcionamiento la cantidad de lugares libres, con un máximo de 10, y un mínimo de 0. Por otra parte, los dos displays de la izquierda(*HEX_2* y *HEX_3*) se ven delegados a la muestra de numerosas funciones. Estas funciones son accesibles mediante la navegación de un menú, dicha navegación esta controlada por medio del botón SEL (*BUTTON1*). Además, la posición del menú a la cual se esta accediendo esta dada por los LEDs en la placa, desde *LED9* hasta *LED0*, aunque cabe destacar que la posición undécima esta representada por el encendido de los LEDs 9,8,7,6 y 5, mientras que la duodécima se representa con el encendido de los LEDs 4,3,2,1 y 0. Las posiciones 9 a 0 del menú corresponden a la muestra de la función de tiempo, que contabiliza el tiempo que cada switch ha estado en alto, correspondiendo el número de switch con el numero de LED. Además, la función 11 corresponde al registro de la cantidad de vehículos que han ingresado al comercio desde que se encendió la alimentación, una observación pertinente es que en este contador no están considerados los vehículos que ya estaban en el estacionamiento, es decir, si ya hay 3 espacios ocupados al momento del encendido, esta función no los contemplará. Por otra parte, la duodécima y última función, es la encargada del sistema de alarmas, esto consiste de un valor variante entre 0 y 99 interpretado como el umbral de alarma. Dicho umbral puede ser aumentado o disminuido con el uso de los botones INCR (*BUTTON0*) y DECR(*BUTTON2*). Otro aspecto constructivo a considerar es la activación de la alarma, dicho estado esta representado por el punto decimal del más extremo de los displays de siete segmentos, siendo este el *HEX_3* También es necesario destacar que la navegación del

menú es cíclica, es decir, una vez se llega al último estado, la próxima iteración dirige al primero. Por último una pulsación prolongada del botón SEL (*BUTTON1*) funciona como un reset de la placa, es decir, todos los contadores de uso vuelven a 0, así como el contador de visitas; dejando el umbral de alarma a su estado por defecto, 45.

3. Diagrama de caja negra

Si se busca abstraer el problema planteado lo máximo posible, se puede llegar a lo mostrado en la figura XX, siendo esto el diagrama de caja negra. Dicho diagrama facilita la visualización de los parámetros de entrada o *inputs* así como los de salida o *outputs*. En el caso discutido, se puede observar que la todos los datos a manejar estan dados por... A su vez, se buscó implementar una resolución que cumpla todos los aspectos pedidos por letra minimizando la cantidad de salidas, ya que esto facilitaria hipoteticas conexiones.

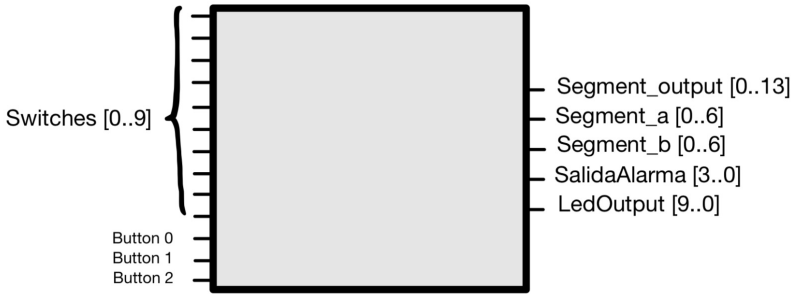

Figura 1: Diagrama de caja negra

4. Diagrama de bloques

Debido al gran número de funciones requeridas al sistema, se buscó dividir el problema en distintos bloques, delegando de esta forma las responsabilidades. Si bien cada bloque puede en si mismo contener múltiples instancias, se busca una agrupación coherente, que sea abarcativa de la función de debe cumplir, esto con el fin de mantener el orden y la eficiencia.

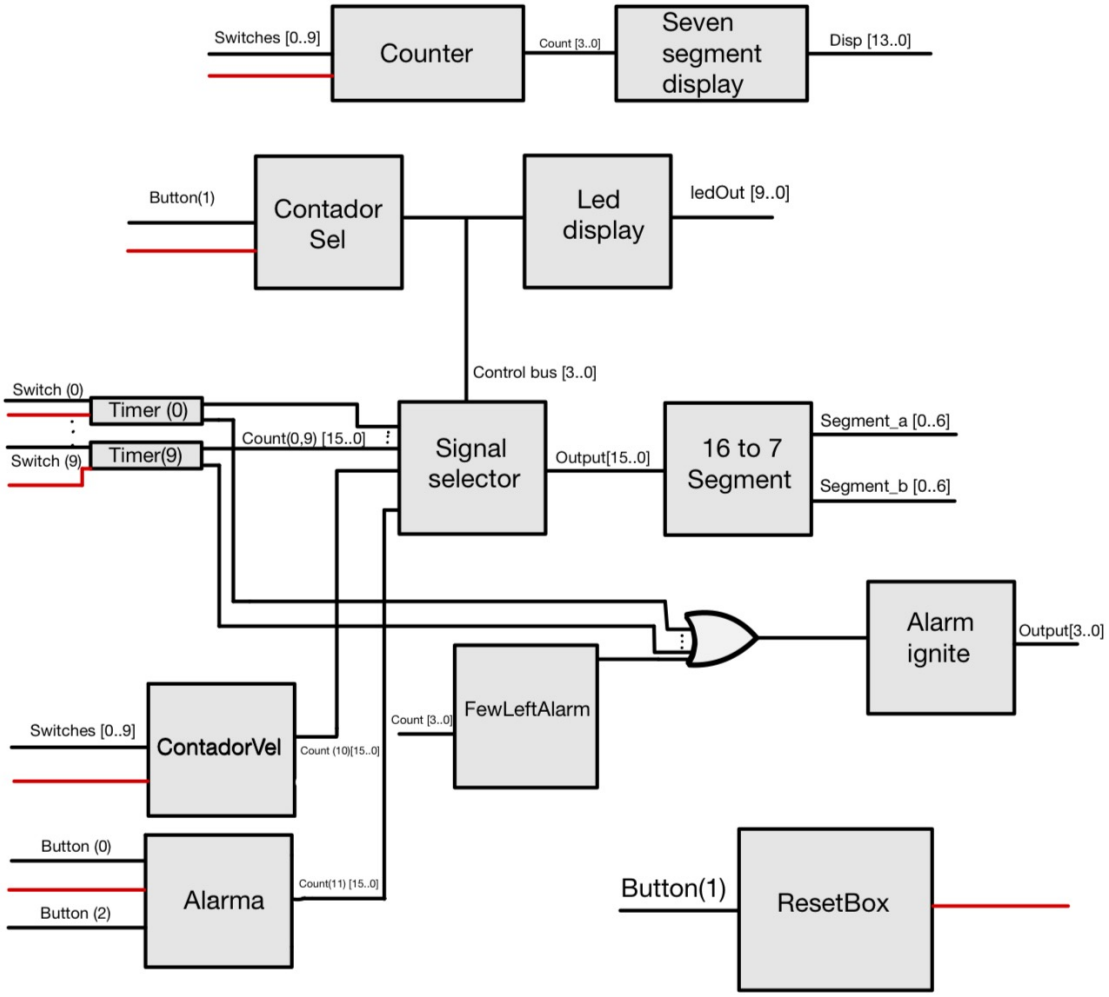

Figura 2: Diagrama de bloques

5. Esquemático

Lo mostrado en el diagrama de bloques(Figura 2) corresponde a una simplificación, ya que cada uno de estos bloques se vieron implementados usando múltiples símbolos. El esquemático completo refiere a lo mostrado en la figura 3.

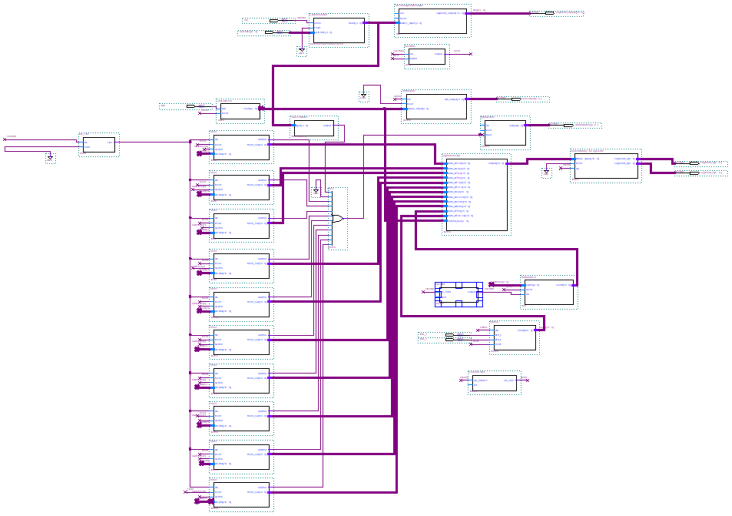

Figura 3: Esquemático completo

6. Explicación

Con el fin de explicar el funcionamiento completo, se va a describir la operación de cada bloque mostrado en el diagrama de bloques, expandiendo individualmente en los símbolos que se ven involucrados en su funcionamiento.

6.1. *Counter*

El símbolo *Counter* toma como entrada una señal de `clk`, en este caso la nativa de la placa correspondiente a $50MHz$, y un *array* de 10 posiciones llamado `Switches[0..9]` que esta asignado a cada uno de los *switches* de la placa. El símbolo esta encargado de enviar un numero binario, `count[3..0]` que corresponda a la cantidad de posiciones altas que tenga el *array* previamente mencionado, cada actualización ocurre en el pulso `clk`.

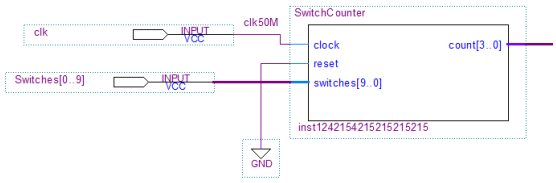

Figura 4: Símbolo *Counter*

El código utilizado es el siguiente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity SwitchCounter is
  Port (
    clock : in STD_LOGIC;
    reset : in STD_LOGIC;
    switches : in STD_LOGIC_VECTOR(9 downto 0);
    count : out integer range 0 to 15
  );
end entity;
architecture Behavioral of SwitchCounter is
  type State is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9,S10);
  signal next_state: State;

  signal switch_count : integer range 0 to 15;

begin
  process (clock, reset)
  begin
    if reset = '1' then
      next_state <= S0;
      switch_count <= 0;
```

```
26      elsif rising_edge(clock) then
27        case next_state is
28          when S0 =>
29            if switches(0) = '0' then
30              switch_count <= switch_count + 1;
31            end if;
32            next_state <= S1;
33
34          when S1 =>
35            if switches(1) = '0' then
36              switch_count <= switch_count + 1;
37            end if;
38            next_state <= S2;
39
40          when S2 =>
41            if switches(2) = '0' then
42              switch_count <= switch_count + 1;
43            end if;
44            next_state <= S3;
45
46          when S3 =>
47            if switches(3) = '0' then
48              switch_count <= switch_count + 1;
49            end if;
50            next_state <= S4;
51
52          when S4 =>
53            if switches(4) = '0' then
54              switch_count <= switch_count + 1;
55            end if;
56            next_state <= S5;
57
58          when S5 =>
59            if switches(5) = '0' then
60              switch_count <= switch_count + 1;
61            end if;
62            next_state <= S6;
63
64          when S6 =>
65            if switches(6) = '0' then
66              switch_count <= switch_count + 1;
67            end if;
68            next_state <= S7;
```

```
69
70          when S7 =>
71            if switches(7) = '0' then
72              switch_count <= switch_count + 1;
73            end if;
74            next_state <= S8;
75
76          when S8 =>
77            if switches(8) = '0' then
78              switch_count <= switch_count + 1;
79            end if;
80            next_state <= S9;
81
82          when S9 =>
83            if switches(9) = '0' then
84              switch_count <= switch_count + 1;
85            end if;
86            next_state <= S10;
87
88          when S10 =>
89            count<=switch_count;
90            switch_count <= 0;
91            next_state <= S0;
92          end case;
93        end if;
94      end process;
95  end architecture Behavioral;
```

6.2. *Seven segment display (1)*

Este símbolo es encargado del pasaje de un numero binario de 4 dígitos, en este caso el saliente del bloque anteriormente descripto, y expulsar una señal binaria de 14 dígitos que contenga la codificación de dos módulos de 7 segmentos. A fines prácticos, se asigna cada valor de este *array* de salida a su correspondiente segmento del *display*.

Figura 5: Símbolo *Seven segment display(1)*

El código utilizado es el siguiente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity SevenSegmentDisplay is
  Port(
    clk : in STD_LOGIC;
    reset : in STD_LOGIC;
    binary_input : in STD_LOGIC_VECTOR(3 downto 0);
    segment_output : out STD_LOGIC_VECTOR(13 downto 0)

  );
end SevenSegmentDisplay;

architecture Behavioral of SevenSegmentDisplay is
  signal segment_data : STD_LOGIC_VECTOR(13 downto 0);
begin

  process(clk, reset, binary_input)
    begin
          case binary_input is
          when "0000" => segment_data<= "01111110000000";
          when "0001" => segment_data<= "00001100000000";
          when "0010" => segment_data<= "10110110000000";
          when "0011" => segment_data<= "10011110000000";
          when "0100" => segment_data<= "11001100000000";
          when "0101" => segment_data<= "11011010000000";
          when "0110" => segment_data<= "11111010000000";
          when "0111" => segment_data<= "00001110000000";
          when "1000" => segment_data<= "11111110000000";
          when "1001" => segment_data<= "11001110000000";
          when "1010" => segment_data<= "01111110000110";
          when others => segment_data<= "11111111111111";
          end case;
    end process;
    segment_output<=not(segment_data);

end Behavioral;
```

6.3. *ContadorSel*

Este símbolo es el encargado de llevar registro de las pulsaciones del botón de navegación, debido a la naturaleza cíclica del problema, cuando llega a un numero determinado, este

comienza a contar nuevamente. Su salida, que es un binario conteniendo dicha información, se ve utilizada en **Led display**, con el fin de ser visualizada; y también se utiliza como *control bus* en el bloque **Signal selector**.

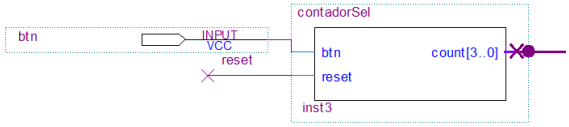

Figura 6: Símbolo *ContadorSel*

El código es el siguiente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity contadorSel is
  Port(
    btn : in STD_LOGIC;
    count : out STD_LOGIC_VECTOR(3 downto 0);
    reset : in STD_LOGIC
  );
end contadorSel;

architecture Behavioral of contadorSel is
  signal counter_internal : STD_LOGIC_VECTOR(3 downto 0) := "0000";

begin
  process(btn)
  begin
    if rising_edge(btn) then
      if counter_internal = "1011" or reset = '1' then
        counter_internal <= "0000";
      else
        counter_internal <= counter_internal+1;
      end if;
    end if;
  end process;

  count <= counter_internal;
end Behavioral;
```

6.4. *Led display*

Este símbolo es el encargado de tomar el binario dado por *ContadorSel* y expulsar un binario con una cantidad de dígitos igual al numero de LEDs, de la primera a la décima configuración cada dígito corresponde a su misma configuración, mientras que en los casos décimo-primero y décimo-segundo se prenden los cinco primeros y últimos respectivamente.

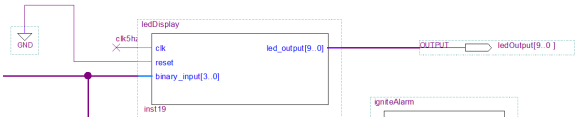

Figura 7: Símbolo *Led display*

El codigo utilizado es el siguiente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity ledDisplay is
  Port(
    clk : in STD_LOGIC;
    reset : in STD_LOGIC;
    binary_input : in STD_LOGIC_VECTOR(3 downto 0);
    led_output : out STD_LOGIC_VECTOR(9 downto 0)

  );
end ledDisplay;

architecture Behavioral of ledDisplay is
  signal segment_data : STD_LOGIC_VECTOR(9 downto 0);

begin
  process(clk, reset, binary_input)
    begin
          case binary_input is
          when "0000" => segment_data <= "1000000000";
          when "0001" => segment_data <= "0100000000";
          when "0010" => segment_data <= "0010000000";
          when "0011" => segment_data <= "0001000000";
          when "0100" => segment_data <= "0000100000";
          when "0101" => segment_data <= "0000010000";
```

```
28          when "0110" => segment_data <= "0000001000";
29          when "0111" => segment_data <= "0000000100";
30          when "1000" => segment_data <= "0000000010";
31          when "1001" => segment_data <= "0000000001";
32          when "1010" => segment_data <= "1111100000";
33          when "1011" => segment_data <= "0000011111";
34          when others => segment_data <= "1111111111";
35          end case;
36    end process;
37  led_output <= segment_data;
38 end Behavioral;
```

6.5. *Timer*

El bloque *Timer* es utilizado un total de 10 veces. Dicho bloque toma como entrada uno de los *switches*, así como una señal de `clk`, y un binario dado por **Alarma**. Esta encargado de contabilizar el tiempo en segundos que su entrada *switch* esta alto, expulsando este resultado como un binario de 16 dígitos (`timer_out`). También pone en alto a la salida `alarma` si el binario dado por la entrada (`umbral`) es menor al contador.

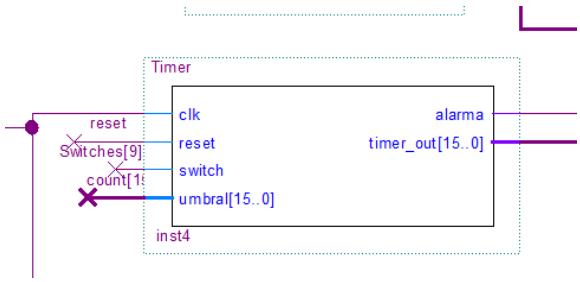

Figura 8: Símbolo *Timer*

El código es el siguiente:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  entity Timer is
7    Port( clk : in STD_LOGIC;
8        reset : in STD_LOGIC;
9        switch : in STD_LOGIC;
10       umbral : in STD_LOGIC_VECTOR(15 downto 0);
11       alarma : out boolean;
```

```
12      timer_out : out STD_LOGIC_VECTOR(15 downto 0)
13  );
14 end entity;
15 architecture Behavioral of Timer is
16   signal count : STD_LOGIC_VECTOR(15 downto 0) := "0000000000000000";
17   type StateType is (IDLE, COUNTING);
18   signal state : StateType := IDLE;
19   signal alarma_internal : boolean := false;
20
21 begin
22   process(clk, reset, switch)
23   begin
24     if reset = '1' then
25       state <= IDLE;
26       count <= "0000000000000000";
27     elsif rising_edge(clk) then
28       case state is
29         when IDLE =>
30           if switch = '1' then
31             state <= COUNTING;
32             count <= "0000000000000000";
33           end if;
34         when COUNTING =>
35             if switch = '0' then
36               state <= IDLE;
37               count <= "0000000000000000";
38             else
39               count <= count +1;
40             end if;
41       end case;
42       if count = umbral  or count > umbral then
43         alarma_internal <= true;
44       else
45         alarma_internal <= false;
46       end if;
47     end if;
48   end process;
49   timer_out <= count;
50   alarma <= alarma_internal;
51
52 end Behavioral;
```

6.6. *ContadorVel*

Este símbolo es el encargado de contar el número de altos de todos los *switches* desde el momento de encendido. Para esto, toma como entrada el mismo *array* que **Counter**, así como una señal de `clk`.

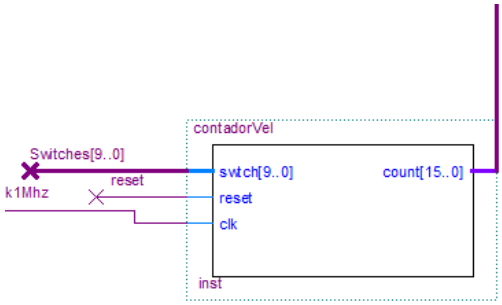

Figura 9: Símbolo *ContadorVel*

El código es el siguiente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity contadorVel is
  Port(
    swtch : in STD_LOGIC_VECTOR(9 downto 0);
    count : out STD_LOGIC_VECTOR(15 downto 0);
    reset : in STD_LOGIC;
    clk   : in STD_LOGIC
  );
end contadorVel;

architecture Behavioral of contadorVel is
    signal counter_internal : STD_LOGIC_VECTOR(15 downto 0) :=
   "0000000000000000";
    signal prevStates : STD_LOGIC_VECTOR(9 downto 0) := (others => '0');
   signal initialized : BOOLEAN := FALSE;
begin
    process(clk, reset)
    begin
        if reset = '1' then
            counter_internal <= "0000000000000000";
        elsif rising_edge(clk) then
```

```
25          if not initialized then
26            for i in swtch'range loop
27              prevStates(i) <= swtch(i);
28            end loop;
29            initialized <= TRUE;
30          else
31            for i in swtch'range loop
32              if swtch(i) = '1' and prevStates(i) = '0' then
33                counter_internal <= counter_internal + 1;
34              end if;
35              prevStates(i) <= swtch(i);
36            end loop;
37          end if;
38          end if;
39      end process;
40
41      count <= counter_internal;
42  end Behavioral;
```

6.7. *Alarma*

Este símbolo toma 3 entradas, una señal de `clk` y dos *inputs* asignados a los botones de control. Por cada *rising_edge* de la señal `clk`, verifica si uno esta pulsado, y si esto ocurre aumenta o disminuye. Si el contador llega a 99 vuelve de manera cíclica al 0, esto también ocurre en el caso inverso.

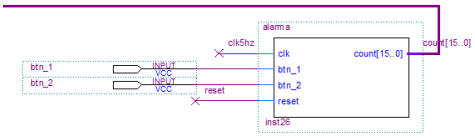

Figura 10: Símbolo *Alarma*

El código es el siguiente:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  entity alarma is
7    Port(
8      clk : in STD_LOGIC;
```

```
    btn_1 : in STD_LOGIC;
    btn_2 : in STD_LOGIC;
    reset : in STD_LOGIC;
    count : out STD_LOGIC_VECTOR(15 downto 0)
  );
end alarma;

architecture Behavioral of alarma is
  signal counter_internal : STD_LOGIC_VECTOR(15 downto 0) :=
    "0000000000101101";

begin
   process(clk)
    begin
        if rising_edge(clk) then
        if reset = '1' then
          counter_internal <= "0000000000101101";
        end if;
            if btn_1 = '0' then
                if counter_internal = "0000000000000000" then
                    counter_internal <= "0000000001100011";
                else
                    counter_internal <= counter_internal - 1;
                end if;
            elsif btn_2 = '0' then
                if counter_internal = "0000000001100011" then
                    counter_internal <= "0000000000000000";
                else
                    counter_internal <= counter_internal + 1;
                end if;
            end if;
        end if;
    end process;

  count <= counter_internal;
end Behavioral;
```

6.8. *Signal selector*

Este símbolo funciona como un multiplexor, teniendo como entrada de control a la salida de **ContadorSel**, que selecciona entre las diez salidas de **Timer**, asi como la salida de **ContadorVel** y la de **Alarma**.

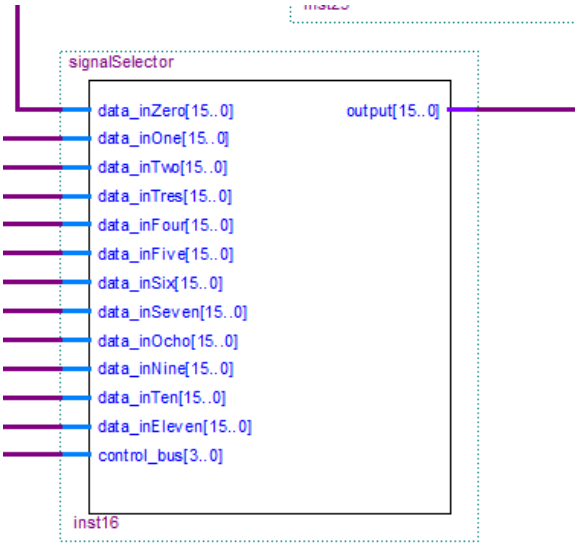

Figura 11: Símbolo *Signal selector*

El código es el siguiente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity signalSelector is
  Port ( data_inZero : in STD_LOGIC_VECTOR(15 downto 0);
    data_inOne : in STD_LOGIC_VECTOR(15 downto 0);
    data_inTwo : in STD_LOGIC_VECTOR(15 downto 0);
    data_inTres : in STD_LOGIC_VECTOR(15 downto 0);
    data_inFour : in STD_LOGIC_VECTOR(15 downto 0);
    data_inFive : in STD_LOGIC_VECTOR(15 downto 0);
    data_inSix : in STD_LOGIC_VECTOR(15 downto 0);
    data_inSeven : in STD_LOGIC_VECTOR(15 downto 0);
    data_inOcho : in STD_LOGIC_VECTOR(15 downto 0);
    data_inNine : in STD_LOGIC_VECTOR(15 downto 0);
    data_inTen : in STD_LOGIC_VECTOR(15 downto 0);
    data_inEleven : in STD_LOGIC_VECTOR(15 downto 0);
    control_bus : in STD_LOGIC_VECTOR(3 downto 0);
    output : out STD_LOGIC_VECTOR(15 downto 0)
  );
end signalSelector;

architecture Behavioral of signalSelector is
```

```
25 begin
26   process(control_bus)
27     begin
28       case control_bus is
29         when "0000" => output <= data_inZero;
30         when "0001" => output <= data_inOne;
31         when "0010" => output <= data_inTwo;
32         when "0011" => output <= data_inTres;
33         when "0100" => output <= data_inFour;
34         when "0101" => output <= data_inFive;
35         when "0110" => output <= data_inSix;
36         when "0111" => output <= data_inSeven;
37         when "1000" => output <= data_inOcho;
38         when "1001" => output <= data_inNine;
39         when "1010" => output <= data_inTen;
40         when "1011" => output <= data_inEleven;
41         when others => output <= "0000000000000000";
42       end case;
43   end process;
44 end Behavioral;
```

6.9. *16 to 7 Segment*

Este símbolo cumple una función similar a **Seven segment diplay(1)**, pero toma como entrada la salida de **Signal selector**, que es un binario de 16 dígitos. Además, su salida esta dividida en dos señales, cada una contiendo la información necesaria para cada uno de los dos 7 segmentos asignados.

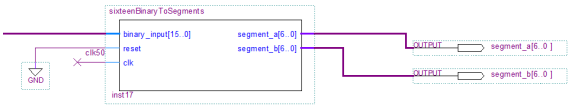

Figura 12: Símbolo *16 to 7 Segment*

El código es el siguiente:

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_ARITH.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5 use IEEE.NUMERIC_STD.ALL;
6
```

```
entity sixteenBinaryToSegments is
  Port ( binary_input : in integer range 0 to 65535;
    segment_a, segment_b : out STD_LOGIC_VECTOR(6 downto 0);
    reset : in STD_LOGIC;
    clk : in STD_LOGIC
  );
end sixteenBinaryToSegments;

architecture Behavioral of sixteenBinaryToSegments is
  signal decimal_value : integer range 0 to 99 := 0;
  signal display_value : integer range 0 to 99 := 0;
  signal segment_pattern_a, segment_pattern_b : STD_LOGIC_VECTOR(6 downto
    0);

  type segment_array is array(0 to 9) of STD_LOGIC_VECTOR(6 downto 0);
  constant seven_segment_mapping : segment_array := ("0111111", "0000110",
    "1011011", "1001111", "1100110", "1101101", "1111101", "0000111",
    "1111111", "1100111");

begin
  process(clk, reset)
    begin
      if reset = '1' then
        decimal_value <= 0;
        display_value <= 0;
      elsif rising_edge(clk) then
        decimal_value <= binary_input;
        if decimal_value < 100 then
          display_value <= decimal_value;
        else
          display_value <= 99;
        end if;
      end if;
  end process;

  process(display_value)
    begin
      if display_value < 10 then
        segment_pattern_a <= seven_segment_mapping(display_value);
        segment_pattern_b <= "0000000";
      else
        segment_pattern_a <= seven_segment_mapping(display_value mod 10);
        segment_pattern_b <= seven_segment_mapping(display_value / 10);
```

```
47        end if;
48      end process;
49
50    segment_a <= not(segment_pattern_a);
51    segment_b <= not(segment_pattern_b);
52
53  end Behavioral;
54
```

6.10. *FewLeftAlarm*

Este símbolo toma como entrada a la salida de **ContadorVel**, y devuelve alto si esta entrada es menor o igual a 2.

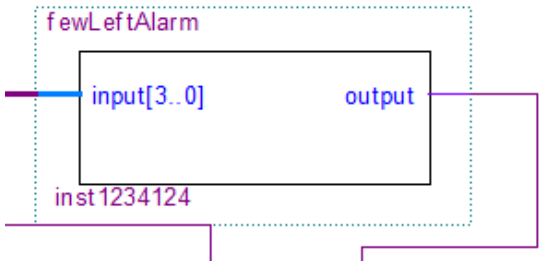

Figura 13: Símbolo *FewLeftAlarm*

El código es el siguiente:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  entity fewLeftAlarm is
7    Port(   input : in STD_LOGIC_VECTOR(3 downto 0);
8          output : out boolean
9    );
10 end fewLeftAlarm;
11 architecture Behavioural of fewLeftAlarm is
12   signal output_internal : boolean := false;
13
14 begin
15   process(input)
16     begin
17       if input < 2 or input = 2 then
```

```
        output_internal <= true;
      else
        output_internal <= false;
      end if;
    end process;
    output <= output_internal;
end Behavioural;
```

6.11. *Alarm ignite*

Este símbolo tiene como entrada una señal unificada que llega desde una compuerta *OR* de 12 entradas. A esta compuerta se encuentran conectadas las salidas `alarma` de todos los **Timer**, así como la salida de **FewLeftAlarm** (son 11 en total, una esta conectada a `GND`). Si esta señal unificada es alta, el símbolo se encarga de enviar un código, en binario, asignado a los puntos decimales de los dos segmentos de la izquierda. Por tanto, si la alarma se enciende, ambos puntos decimales estarán encendidos

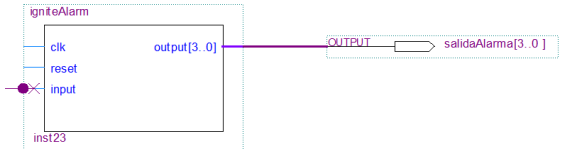

Figura 14: Símbolo *Alarm ignite*

El código es el siguiente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity igniteAlarm is
  Port(
    clk : in STD_LOGIC;
    reset : in STD_LOGIC;
    output : out STD_LOGIC_VECTOR(3 downto 0);
    input : in STD_LOGIC
  );
end igniteAlarm;

architecture Behavioural of igniteAlarm is
  signal output_internal : STD_LOGIC_VECTOR(3 downto 0) := "0000";

begin
```

```
18   process(input,reset)
19     begin
20       if reset = '1' then
21         output_internal <= "0000";
22       end if;
23       if input = '1' then
24         output_internal <= "1100";
25       else
26         output_internal <= "0000";
27       end if;
28     end process;
29     output <= not(output_internal);
30 end Behavioural;
```

6.12. *ResetBox*

Este símbolo es encargado de enviar una señal alta si el botón *Sel* es presionado por mas de dos segundos. Para eso, toma como entrada una señal de `clk`, así como un *input* asignado al botón en cuestión. La señal de salida esta conectada a todas los bloques pertinentes con el fin de asegurar un correcto *reset*.

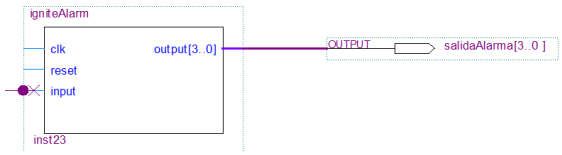

Figura 15: Símbolo *ResetBox*

El código es el siguiente:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  entity resetBox is
7      Port ( clk : in STD_LOGIC;
8             button : in STD_LOGIC;
9             output : out STD_LOGIC);
10 end resetBox;
11
12 architecture Behavioral of resetBox is
13
```

```
    type State_Type is (IDLE, TIMING);
    signal state : State_Type := IDLE;
   signal reset : STD_LOGIC := '0';

    signal counter : integer := 0;


    constant LONG_PRESS_TIME : integer := 2000000;

begin
    process(clk)
    begin
         if rising_edge(clk) then
        case state is
            when IDLE =>
                if button = '0' then
                    counter <= 0;
                    state <= TIMING;
                else
                    state <= IDLE;
                end if;

            when TIMING =>
                if button = '0' then
                    counter <= counter + 1;
                    if counter >= LONG_PRESS_TIME then
                        reset <= '1';
                        state <= IDLE;
                    end if;
                else
                    counter <= 0;
              reset <= '0';
                    state <= IDLE;
                end if;
        end case;
    end if;
    end process;

   output <= '1' when reset = '1' else '0';
end Behavioral;
```

6.13. *Clocks*

Como se mencionó anteriormente, existen distintas señales de `clk` en el circuito, cada una dependiendo del problema a resolver por el bloque. Para esto, se crearon los siguientes divisores de frecuencia, considerando la entrada de $50MHz$ del pin `G_21`. Se crearon divisores que dejen como salida valores de 1Hz, 1MHz y 5Hz. Su funcionamiento es igual en todos los casos, cambiando solamente el valor al que se quiere contar(en el caso del código posterior, dicha variable es llamada *counter*). Este es el código utilizado para el símbolo que divide a 1Mhz.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity clk1Mhz is
    Port ( clk_50M : in STD_LOGIC;
           reset    : in STD_LOGIC;
           output : out STD_LOGIC);
end clk1Mhz;

architecture Behavioral of clk1Mhz is
    signal counter : integer range 0 to 50 := 0;
    signal out_internal : STD_LOGIC := '0';
begin
    process(clk_50M, reset)
    begin
        if reset = '1' then
            counter <= 0;
            out_internal <= '0';
        elsif rising_edge(clk_50M) then
            if counter = 50 then
                counter <= 0;
                out_internal <= not out_internal;
            else
                counter <= counter + 1;
            end if;
        end if;
    end process;

    output <= out_internal;
end Behavioral;
```

7. Asignaciones

A continuación, se detalla la tabla de asignaciones de todas las señales usadas en el proyecto.

	tatu	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
1	✓		Switches[0]	Location	PIN_J6	Yes			
2	✓		Switches[1]	Location	PIN_H5	Yes			
3	✓		Switches[2]	Location	PIN_H6	Yes			
4	✓		Switches[3]	Location	PIN_G4	Yes			
5	✓		Switches[4]	Location	PIN_G5	Yes			
6	✓		Switches[5]	Location	PIN_J7	Yes			
7	✓		Switches[6]	Location	PIN_H7	Yes			
8	✓		Switches[7]	Location	PIN_E3	Yes			
9	✓		Switches[8]	Location	PIN_E4	Yes			
10	✓		Switches[9]	Location	PIN_D2	Yes			
11	✓		clk	Location	PIN_G21	Yes			
12	✓		segme...ut[1]	Location	PIN_B13	Yes			
13	✓		segme...ut[2]	Location	PIN_C13	Yes			
14	✓		segme...ut[3]	Location	PIN_A14	Yes			
15	✓		segme...ut[4]	Location	PIN_B14	Yes			
16	✓		segme...ut[5]	Location	PIN_E14	Yes			
17	✓		segme...ut[6]	Location	PIN_A15	Yes			
18	✓		segme...ut[7]	Location	PIN_E11	Yes			
19	✓		segme...ut[8]	Location	PIN_F11	Yes			
20	✓		segme...ut[9]	Location	PIN_H12	Yes			
21	✓		segme...t[10]	Location	PIN_H13	Yes			
22	✓		segme...t[11]	Location	PIN_G12	Yes			
23	✓		segme...t[12]	Location	PIN_F12	Yes			
24	✓		segme...t[13]	Location	PIN_F13	Yes			
25	✓		segme...ut[0]	Location	PIN_A13	Yes			
26	✓		btn	Location	PIN_G3	Yes			
27	✓		test	Location	PIN_B2	Yes			
28	✓		segm...a[1]	Location	PIN_A16	Yes			
29	✓		segm...a[2]	Location	PIN_B16	Yes			
30	✓		segm...a[3]	Location	PIN_E15	Yes			
31	✓		segm...a[4]	Location	PIN_A17	Yes			

Figura 16: Asignaciones, parte primera

32	✓		segm...a[5]	Location	PIN_B17	Yes			
33	✓		segm...a[6]	Location	PIN_F14	Yes			
34	✓		segm...b[0]	Location	PIN_B18	Yes			
35	✓		segm...b[1]	Location	PIN_F15	Yes			
36	✓		segm...b[2]	Location	PIN_A19	Yes			
37	✓		segm...b[3]	Location	PIN_B19	Yes			
38	✓		segm...b[4]	Location	PIN_C19	Yes			
39	✓		segm...b[5]	Location	PIN_D19	Yes			
40	✓		segm...b[6]	Location	PIN_G15	Yes			
41	✓		segm...a[0]	Location	PIN_D15	Yes			
42	✓		ledOutput[1]	Location	PIN_J2	Yes			
43	✓		ledOutput[2]	Location	PIN_J3	Yes			
44	✓		ledOutput[3]	Location	PIN_H1	Yes			
45	✓		ledOutput[4]	Location	PIN_F2	Yes			
46	✓		ledOutput[5]	Location	PIN_E1	Yes			
47	✓		ledOutput[6]	Location	PIN_C1	Yes			
48	✓		ledOutput[7]	Location	PIN_C2	Yes			
49	✓		ledOutput[8]	Location	PIN_B2	Yes			
50	✓		ledOutput[9]	Location	PIN_B1	Yes			
51	✓		ledOutput[0]	Location	PIN_J1	Yes			
52	✓		btn_1	Location	PIN_F1	Yes			
53	✓		btn_2	Location	PIN_H2	Yes			
54	✓		salid...ma[1]	Location	PIN_B15	Yes			
55	✓		salid...ma[2]	Location	PIN_A18	Yes			
56	✓		salid...ma[3]	Location	PIN_G16	Yes			
57	✓		salid...ma[0]	Location	PIN_D13	Yes			
58	✓		Buzzer	Location	PIN_AA13	Yes			

Figura 17: Asignaciones, parte segunda

8. Opcionales

8.1. Opcional A

Como primer ejercicio opcional, se concreta la implementación de una memoria no volátil. El objetivo es que el programa compilado se guarde en una unidad *EPROM* y no en una unidad *RAM*, de esta forma, al apagar y prender la placa, esta volverá a tener la configuración ya ingresada. Para lograr esto, primeramente se configura el dispositivo mediante la ventana *Device and pin options*.

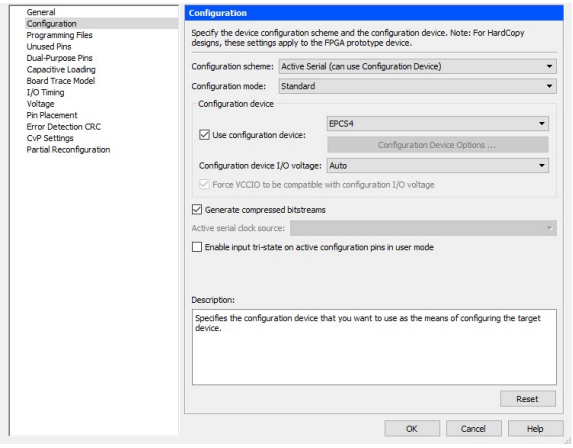

Figura 18: Ventana *Device and pin options*

En la parte de configuración, se tilda *Use configuration device* y se selecciona la opción *EPCS4*. En segunda instancia, es necesario recompilar el programar, y a la hora de cargarlo se cambia el modo a *Active Serial Programming*.

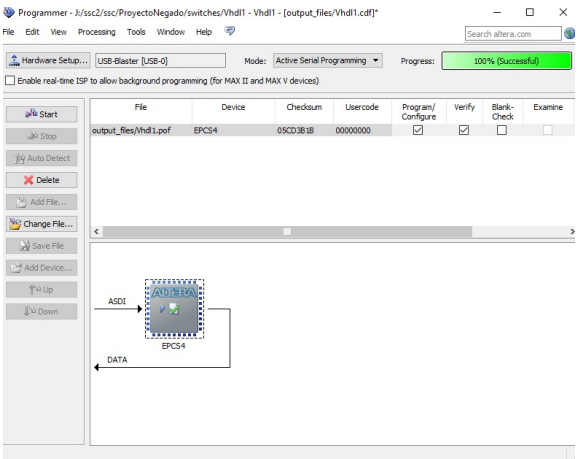

Figura 19: Ventana *Programmer*

Ahora es necesario seleccionar el archivo a cargar (.pof), por medio de *Add file*. Luego es

necesario tildar *Program/Configure* y *Verify*. Antes de cargar la placa, es necesario apagarla y mover el *switch* al modo *PROG*; de esta forma, el programa puede ser cargado de forma persistente. Como muestra de su correcto funcionamiento, ver Anexo 8.1.

8.2. Opcional B

Como segundo ejercicio opcional, se elije la implementación lógica y el armado físico de un sistema sonoro que funcione como alerta cuando una alarma es encendida. Se agrega un símbolo nuevo. Este toma como entrada una señal de `clk`, y otra de `alarma`. El código se ejecuta cuando la señal de `alarma` es alta. Se selecciona la frecuencia del *buzzer* y del `clk`, mediante una relación dada por las mismas se consigue enviar una frecuencia audible para el oído humano, reproducida por el *buzzer*.

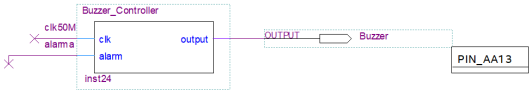

Figura 20: Símbolo *BuzzerController*

El codigo es el siguiente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Buzzer_Controller is
  Port ( clk : in STD_LOGIC;
      alarm : in STD_LOGIC;
         output : out STD_LOGIC);
end Buzzer_Controller;

architecture Behavioral of Buzzer_Controller is
  constant clockFreq : integer := 50_000_000;
  constant buzzerFreq : integer := 1500;

  signal counter : integer range 0 to clockFreq / buzzerFreq / 2 := 0;
  signal buzzerSignal : STD_LOGIC := '0';

begin
  process(clk)
```

```
21   begin
22     if rising_edge(clk) and alarm = '1' then
23       if counter = 0 then
24         buzzerSignal <= not buzzerSignal;
25         counter <= clockFreq / buzzerFreq / 2 - 1;
26       else
27         counter <= counter - 1;
28       end if;
29     end if;
30   end process;
31
32   output <= buzzerSignal;
33
34 end Behavioral;
```

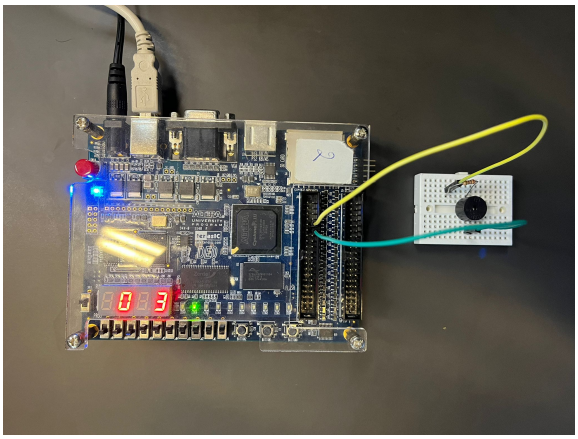

Figura 21: Conexiones en protoboard

Se determina un pin de salida para la secuencia de salida, elegido convenientemente, el cual se conecta a la protoboard y por tanto, al *buzzer*. Como muestra de su correcto funcionamiento, ver Anexo 8.2.

9. Anexos

9.1. Enlace al vídeo, Opcional A

https://www.youtube.com/shorts/zFkUcAv1_8A

9.2. Enlace al vídeo, Opcional B

https://www.youtube.com/watch?v=lHbx9J_3sS0