TRABAJO SED VHDL
JUEGO ‘APLASTA LOS TOPOS’

Grupo: A404
Realizado por:
Marcos Espinoza Pino (56351)
Mario García López (56379)
Manu Hidalgo (56416)

ÍNDICE

INTRODUCCIÓN..... 3
DESCRIPCIÓN DEL CÓDIGO EN VHDL.....3
LFSR_GENERATOR (PSEUDOALEATORIEDAD).....3
FSM_CONTROLLER..... 7
DISPLAY_CONTROLLER.....14
PWM_RGB.....23
TOPO_TIMER..... 26
GAME_TIMER.....29
SYNCHRONIZER.....33
EDGETCTR (DETECTOR DE FLANCO).....35
FUNCIONAMIENTO DEL DISEÑO.....37

INTRODUCCIÓN

En este trabajo se ha diseñado e implementado el juego “Aplasta los topos” utilizando la placa Nexys4 DDR. El juego consiste en reaccionar rápidamente para aplastar los topos que aparecene de forma aleatoria en la pantalla obteniendo la mayor puntuación posible antes de que se termine la partida. El juego cuenta con dos modos, uno facil y otro díficil.

Para implementar este juego en la placa los topos son representados por los leds y el mazo son los switches correspondientes a cada led.

DESCRIPCIÓN DEL CÓDIGO EN VHDL

Para llevar a cabo el código en VHDL para la implementación del juego se van a describir la funcionalidad de cada uno de los módulos que conforman el diseño. El generador de pseudoaleatoriedad para determina la aparición de los topos, una FSM para gestionar la lógica del juego, el módulo display para mostrar la puntuación y tiempo transcurrido en la partida. También el módulo pwm_rgb para controlar los colores según aciertos o fallos, topo_timer y game_timer para la temporización de eventos, y los módulos synchronizer que garantiza la sincronización y el edgetctr que garantiza la detección de flancos en las señales de entrada. Todos estos son componentes instanciados en una entidad top. Además, se ha comprobado el correcto funcionamiento de cada entidad de forma individual mediante sus correspondientes testbenches.

LFSR_GENERATOR (PSEUDOALEATORIEDAD)

Como idea inicial se quería que se encendieran los leds de forma aleatoria pero tras investigar, no se puede generar una secuencia aleatoria como tal en VHDL, lo que sí se puede es generar una secuencia pseudoaleatoria, es decir , que variando el valor inicial (seed) se cambia la secuencia.

Proceso evolutivo:

En primer lugar, se abordó la idea de construir un LFSR el cúal permitiera esa pseudoaleatoriedad. Dió muchos problemas ya que no se entendia bien el concepto de pseudoaleatoriedad. Los problemas que surgieron:

-Los leds se quedaban undefined (U) durante toda la simulación.

-Aparecían estados X que no sabíamos muy bien que eran.

- Aunque cambiáramos la semilla, se repetía la misma secuencia.

Debido a invertir una gran cantidad de tiempo y no avanzar se decantó por un programa el cuál siguiera una secuencia.

Para la simulación se ha escogido un modelo simplificado, con unos tiempos adaptados a la herramienta de simulación de vivado. Los tiempos de encendido de cada led serán de 10 ns, el tiempo de apagado entre led y led es de 6 ns. La simulación dura 300 ns.

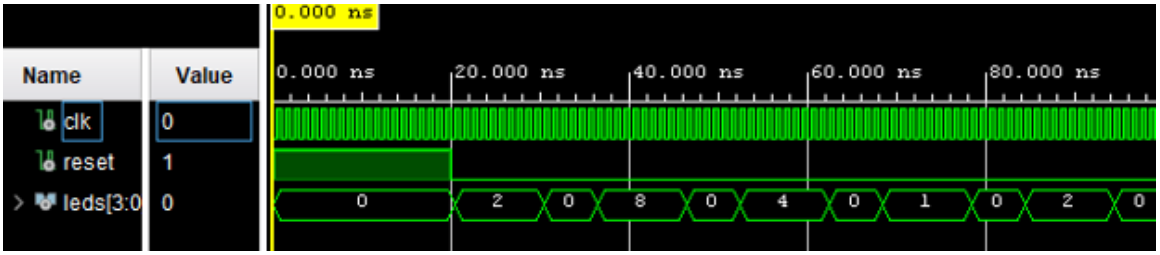
A través,de la siguiente línea se podía informar del orden encendido de leds.

```
type led_index is (LED1, LED3, LED2, LED0);  -- Orden de LEDs
```

Pero para que verdaderamente se modifique el encendido, se deberá también modificar el case de current_step con la nueva secuencia.

Para la simulación se ha escogido un modelo simplificado, con unos tiempos adaptados a la herramienta de simulación de vivado. Los tiempos de encendido de cada led serán de 10 ns, el tiempo de apagado entre led y led es de 6 ns. La simulación dura 300 ns.

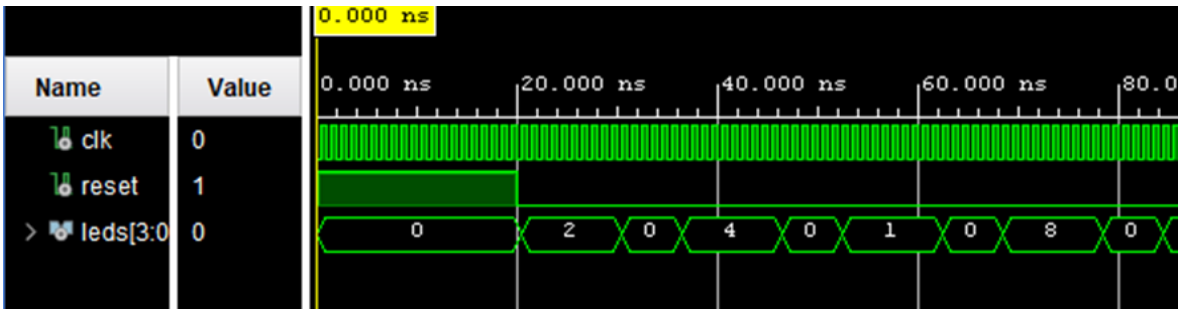
Como se puede observar el orden de encendido de leds en el preestablecido en type led_index



Si se modifica y simula:

```
type led_index is (LED1, LED2, LED0, LED3);

 case current_step is
     when LED1 => current_step <= LED2;
     when LED3 => current_step <= LED1;
     when LED2 => current_step <= LED0;
     when LED0 => current_step <= LED3;
 end case;
```

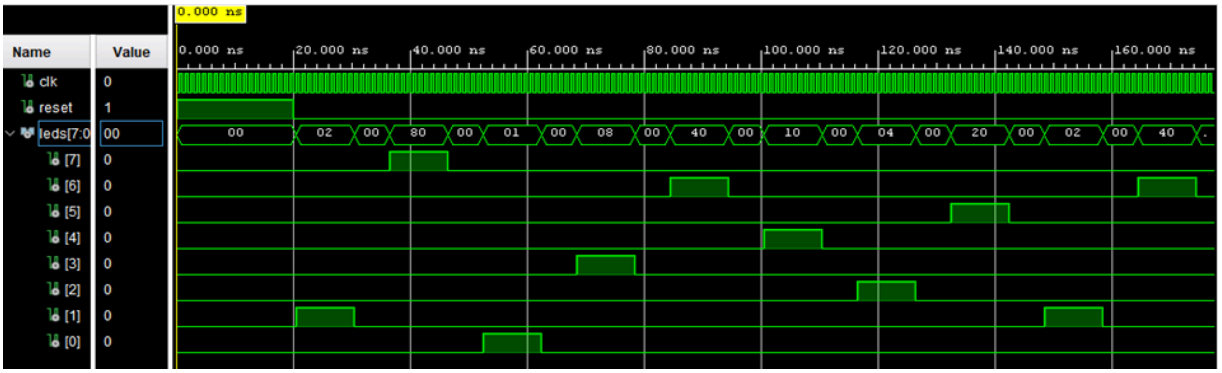
Se Observa que se ha modificado el orden de encendido, tal y como se quería, secuencia 1,2,0,3:



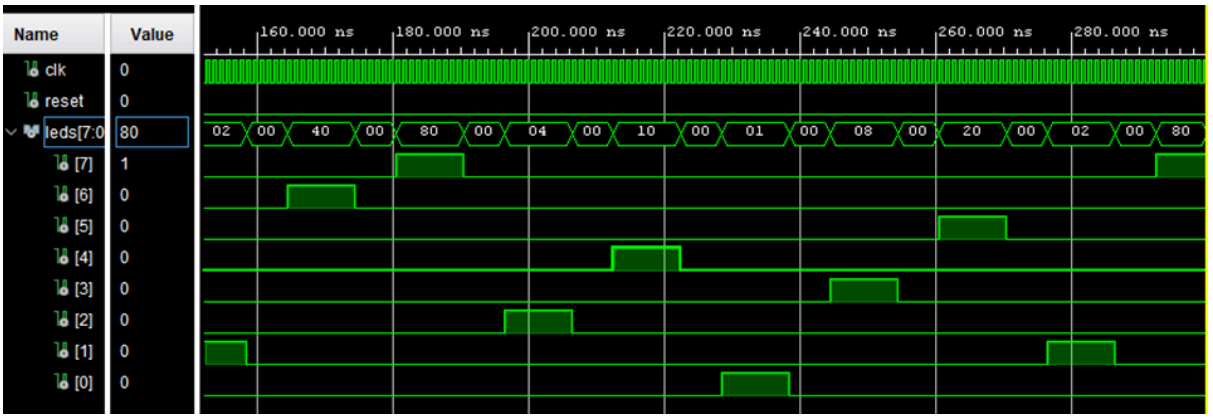
A continuación, se amplia el código a 8 leds. Se ha añadido una variable en el proceso llamada sec, que permite que haya dos secuencias distintas, una cuando sec=0 y otra cuando sec =1.

Cuando sec=0 la secuencia de leds es: 1,7,0,3,6,4,2,5. Cuando sec=1 la secuencia será:1,6,7,2,4,0,3,5

Sec=0:



Sec=1:



A los pocos días se tuvo una reunión con nuestro tutor del trabajo. Y se le explicó que no funcionaba la pseudoaleatoriedad porque no se entendía muy bien el concepto. Tras su explicación y con los conceptos más claros se trabajó en ello y finalmente se conseguió que funcionara nuestro testbench.

Se implementó una variable seed, la cual es un vector de 8 elementos, que permite hacer 256 secuencias diferentes, permitiendo así lograr la pseudoaleatoriedad.

La operación de pseudoaleatoriedad es: a través de feedback. Genero el valor de realimentación con la línea: `feedback <= lfsr(7) xor lfsr(5) xor lfsr(4) xor lfsr(3);`

Se gestiona el tiempo de encendido de los leds a través de cuentaTiempo, cuando hay 10 flancos de subida de CLK, se desplaza el registro lfsr a la derecha y se añade feedback como primer elemento.

Por último, ha sido necesario añadir un process el cúal permite que solo se encienda un led a la vez, sin este process se encenderían varios leds de forma simultánea. El índice de led que inicializa depende de los tres primeros bits del lfsr. Ejemplos, si los tres primeros bits fueran “010” se encendería el led[2].

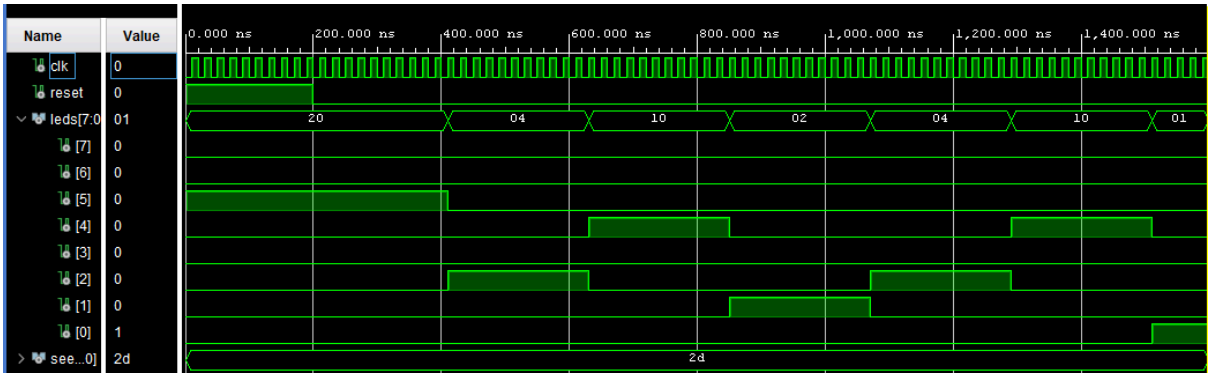
```vhdl
process (lfsr)
variable led_index : integer range 0 to 7;
begin
    -- Inicializa todos los LEDs apagados
    leds <= (others => '0');

    -- Determina el índice del LED a encender según el valor de lfsr
    led_index := to_integer(unsigned(lfsr(2 downto 0))); -- Usa los 3 bits menos significativos
    if led_index <= 7 then
        leds(led_index) <= '1'; -- Enciende el LED correspondiente
    end if;
end process;
```

Testbench:

Se simula durante 1600 ns y cambia un led cada 200 ns.

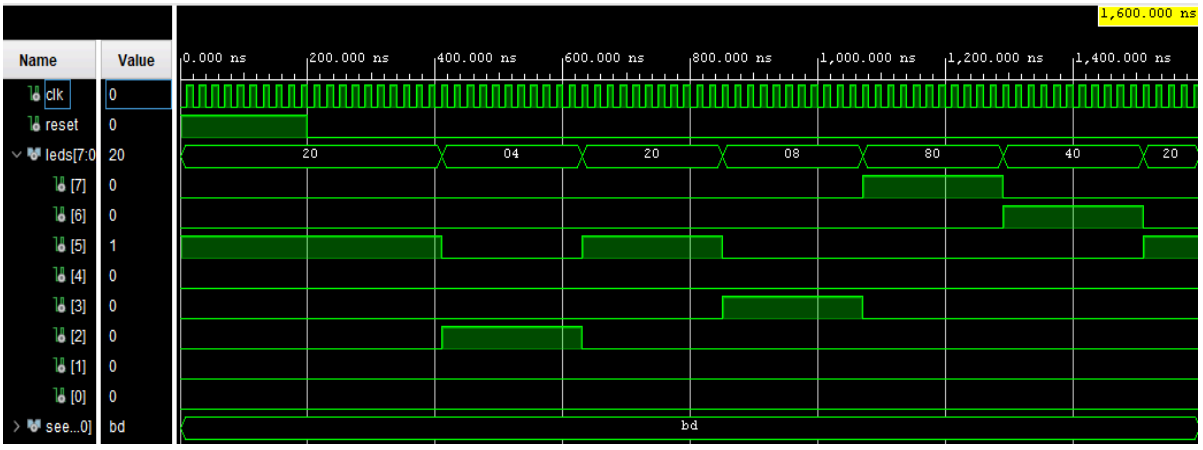
Si se prueba la semilla: `signal seed :std_logic_vector(7 downto 0) := "00101101";`

Se obtiene la siguiente secuencia:



Por otro lado, si se modifica dicha semilla

`signal seed :std_logic_vector(7 downto 0) := "10111101";` , obtendremos:



Se ha verificado, que al cambiar de semilla se cambia de secuencia. Obteniendo así un total de 256 combinaciones.

FSM_CONTROLLER

Para realizar la FSM, se ha seguido como modelo de ejemplo la troqueladora vista en clases de teoría.

La FSM es una máquina de Mealy, ya que las salidas dependen de los estados actuales y de las entradas.

Entradas y salidas:

```
entity fsm_controller is
    Port (
        clk          : in std_logic;                     -- Reloj principal
        rst          : in std_logic;                     -- Señal de reinicio global
        sync_btn     : in std_logic;                     -- Botón sincronizado
        sw           : in std_logic_vector(7 downto 0);  -- Interruptores del jugador
        random_led   : in std_logic_vector(7 downto 0);  -- LED activo del LFSR
        topo_time_up : in std_logic;                     -- Tiempo agotado para el LED actual
        game_time_up : in std_logic;                     -- Tiempo total del juego agotado
        current_score: out unsigned(7 downto 0);         -- Puntuación del jugador
        led_active   : out std_logic;                    -- Señal de activación de LEDs
        rgb_red      : out unsigned(7 downto 0);         -- Ciclo de trabajo del canal rojo
        rgb_green    : out unsigned(7 downto 0)          -- Ciclo de trabajo del canal verde
    );
end fsm_controller;
```

- clk: es la señal de reloj que sincroniza el proceso.
- rst: es la señal de reinicio del juego, si está a 0 el circuito vuelve a su estado inicial (INIT en la FSM)
- sync_btn: es el botón que inicia el juego.
- sw: es el vector de swicthes que indican la posición de los que están activos, son con los que se juega.
- random_led: es el vector de leds que indican la posición de los que están activos.
- topo_time_up: es la señal que indica que el tiempo del led actual se ha agotado.
- game_time_up: es la señal que indica que el tiempo de la partida actual se ha agotado.
- current_score: es la puntuación que lleva obtenida el jugador en la partida.
- led_active: es la señal que indica que un está activo.
- rgb_red: controla el led rgb rojo el cual está encendido durante el tiempo que no se haya acertado ningún “topo”.
- rgb_green: controla el led rgb verde que se enciende al acertar un “topo”.

En la architecture se tienen cuatro estados y las siguientes señales:

```
type state_type is (INIT, LED_ON, CHECK, FINAL);            -- Estados de la FSM
signal current_state, next_state : state_type := INIT;      -- Estado actual y próximo
signal score_internal : unsigned(7 downto 0) := (others => '0'); -- Puntuación interna
signal next_score     : unsigned(7 downto 0) := (others => '0'); -- Puntuación temporal
signal last_result    : std_logic := '0';                   -- Resultado anterior (para RGB)
signal next_last_result: std_logic := '0';                  -- Resultado temporal
signal match_detected : std_logic := '0';                   -- Detecta coincidencia en LED_ON
signal next_match_detected: std_logic := '0';               -- Coincidencia temporal
```

Se tienen cuatro estados:

- INIT: Es el estado inicial, el juego no se ha iniciado, por tanto, el temporizador, contador y más están inactivos. Este estado espera a que el botón sync_btn sea activado para iniciar el juego.
- LED_ON: Activa un led pseudoaleatoriamente mediante el lfsr y detecta coincidencias entre los sw y el led activo llamado random_led.
- CHECK: Comprueba si han coincidido random_led con su switch correspondiente e incrementa la puntuación.
- FINAL: Indica el final de la partida, tras haberse transcurrido el tiempo. En este estado se muestra la puntuación final obtenida por el jugador.

Proceso de transición de estados:

Este proceso actualiza los valores de las señales de estado y las variables internas sincronizadas con el reloj. Al resetarse, es decir, rst =’0’ se vuelve al estado inicial y todos los contadores se reinician. Sin embargo, si el reloj detecta que hay flanco de subida el estado actual pasa al siguiente estado y se actualizan las señales internas.

```
process(clk, rst)
begin
    if rst = '0' then
        current_state <= INIT;                      -- Reinicia al estado inicial
        score_internal <= (others => '0');          -- Reinicia la puntuación
        last_result <= '0';                         -- Reinicia el resultado
        match_detected <= '0';
    elsif rising_edge(clk) then
        current_state <= next_state;                -- Actualiza el estado
        score_internal <= next_score;               -- Actualiza la puntuación
        last_result <= next_last_result;            -- Actualiza el resultado anterior
        match_detected <= next_match_detected;      -- Registro de la coincidencia
    end if;
end process;
```

Proceso de lógica combinacional:

Este proceso implementa la lógica combinacional de la FSM. En el estado INIT espera a que el botón sync_btn sea activado para iniciar el juego. Una vez el botón se ha activado, pasa al estado LED_ON activandose un led pseudoaleatoriamente mediante el lfsr_generator y detecta coincidencias entre los switches y el led activo llamado random_led. Al apagarse el led pasa al estado CHECK, que comprueba si han coincidido random_led con su switch correspondiente e incrementa la puntuación. Por último, una vez ha finalizado el tiempo de juego pasa al estado FINAL que termina el juego y muestra en el display la puntuación final.

```
process(current_state, sync_btn, topo_time_up, game_time_up, sw, random_led, score_internal, last_result, match_detected)
begin
    -- Valores predeterminados
    next_state <= current_state;
    next_score <= score_internal;
    next_last_result <= last_result;
    next_match_detected <= match_detected;
    led_active <= '0';
    rgb_red <= (others => '0');
    rgb_green <= (others => '0');

    case current_state is
        -- Estado Inicial
        when INIT =>
            next_score <= (others => '0');
            next_last_result <= '0';
            next_match_detected <= '0';
            led_active <= '0';
            if sync_btn = '1' then
                next_state <= LED_ON;
            end if;

        -- Estado LED_ON: Verificación en tiempo real
        when LED_ON =>
            led_active <= '1';
            -- Detecta coincidencia si el SW coincide con el LED encendido
            if sw = random_led then
                next_match_detected <= '1'; -- Se detecta coincidencia
            else
                next_match_detected <= '0'; -- No hay coincidencia
            end if;

              -- Muestra el resultado anterior
              if last_result = '1' then
                  rgb_green <= "11111111"; -- Verde si la respuesta anterior fue correcta
              else
                  rgb_red <= "11111111";    -- Rojo si fue incorrecta o tiempo agotado
              end if;

              if topo_time_up = '1' then
                  next_state <= CHECK; -- Transición al estado CHECK
              end if;

         -- Estado CHECK: Actualiza la puntuación y resultado
         when CHECK =>
              led_active <= '0';
              -- Actualiza la puntuación si hubo coincidencia detectada
              if match_detected = '1' then
                  next_score <= score_internal + 1;
                  next_last_result <= '1'; -- Correcto
              else
                  next_last_result <= '0'; -- Incorrecto
              end if;

              -- Transición a LED_ON o FINAL según el tiempo del juego
              if game_time_up = '1' then
                  next_state <= FINAL; -- Transición al estado FINAL
              else
                  next_state <= LED_ON; -- Regresa a LED_ON
                  next_match_detected <= '0'; -- Reinicia coincidencia detectada
              end if;
```

```
            -- Estado FINAL: Muestra el score final
            when FINAL =>
                led_active <= '0';
                rgb_red <= "11111111";         -- LED rojo encendido indicando el fin del juego
                next_score <= score_internal; -- Mantiene el score final

            -- Estado por defecto
            when others =>
                next_state <= INIT;
        end case;
    end process;
```

Testbench:

Código:

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

entity tb_fsm_controller is
end tb_fsm_controller;

architecture Behavioral of tb_fsm_controller is
    -- Declaración de señales internas
    signal clk          : std_logic := '0';
    signal rst          : std_logic := '0';
    signal sync_btn     : std_logic := '0';
    signal sw           : std_logic_vector(7 downto 0) := (others => '0');
    signal random_led   : std_logic_vector(7 downto 0) := (others => '0');
    signal topo_time_up : std_logic := '0';
    signal game_time_up : std_logic := '0';
    signal current_score: unsigned(7 downto 0);
    signal led_active   : std_logic;
    signal rgb_red      : unsigned(7 downto 0);
    signal rgb_green    : unsigned(7 downto 0);

    -- Instancia de la unidad bajo prueba (UUT)
    component fsm_controller
        Port (
            clk          : in std_logic;
            rst          : in std_logic;
            sync_btn     : in std_logic;
            sw           : in std_logic_vector(7 downto 0);
            random_led   : in std_logic_vector(7 downto 0);
            topo_time_up : in std_logic;
            game_time_up : in std_logic;
```

```
            current_score: out unsigned(7 downto 0);
            led_active   : out std_logic;
            rgb_red      : out unsigned(7 downto 0);
            rgb_green    : out unsigned(7 downto 0)
        );
    end component;

begin
    -- Instanciación de la FSM
    uut: fsm_controller
        Port map (
            clk          => clk,
            rst          => rst,
            sync_btn     => sync_btn,
            sw           => sw,
            random_led   => random_led,
            topo_time_up => topo_time_up,
            game_time_up => game_time_up,
            current_score => current_score,
            led_active   => led_active,
            rgb_red      => rgb_red,
            rgb_green    => rgb_green
        );

    -- Generación del reloj
    clk_process: process
    begin
        while true loop
            clk <= '0';
            wait for 10 ns;

            clk <= '1';
            wait for 10 ns;
        end loop;
    end process;

    -- Proceso de estímulos
    stim_process: process
    begin
        -- Reset inicial
        rst <= '1';
        wait for 50 ns;
        rst <= '0';
        wait for 50 ns;

        -- Activación del botón sincronizado para iniciar el juego
        sync_btn <= '1';
        wait for 20 ns;
        sync_btn <= '0';
        wait for 100 ns;

        -- Simular el encendido de 6 LEDs y switches
        -- LED 1: Switch correcto
        random_led <= "00000001";
        sw <= "00000001";
        wait for 100 ns;
        topo_time_up <= '1';
        wait for 20 ns;
        topo_time_up <= '0';
        wait for 100 ns;
```

```
        -- LED 2: Switch incorrecto
        random_led <= "00000010";
        sw <= "00000100";
        wait for 100 ns;
        topo_time_up <= '1';
        wait for 20 ns;
        topo_time_up <= '0';
        wait for 100 ns;

        -- LED 3: Switch correcto
        random_led <= "00000100";
        sw <= "00000100";
        wait for 100 ns;
        topo_time_up <= '1';
        wait for 20 ns;
        topo_time_up <= '0';
        wait for 100 ns;

        -- LED 4: Switch incorrecto
        random_led <= "00001000";
        sw <= "00010000";
        wait for 100 ns;
        topo_time_up <= '1';
        wait for 20 ns;
        topo_time_up <= '0';
        wait for 100 ns;

        -- LED 5: Switch incorrecto
        random_led <= "00100000";
        sw <= "01000000";

        wait for 100 ns;
        topo_time_up <= '1';
        wait for 20 ns;
        topo_time_up <= '0';
        wait for 100 ns;

        -- LED 6: Switch incorrecto
        random_led <= "01000000";
        sw <= "10000000";
        wait for 100 ns;
        topo_time_up <= '1';
        wait for 20 ns;
        topo_time_up <= '0';
        wait for 100 ns;

        -- Finalizar el juego
        game_time_up <= '1';
        wait for 50 ns;

        -- Final de la simulación
        wait;
    end process;

end Behavioral;
```

El testbench consiste en la simulación de seis leds y switches en los cuales hay casos de aciertos y fallos, así se comprueba mediante los resultados de la simulación su funcionamiento.

En la configuración inicial, se configuraron estímulos específicos, incluyendo un ciclo de reloj de 20 ns, señales de reinicio global (rst) y botón de sincronización (sync_btn) para iniciar el juego. Además, se genera una secuencia de estímulos que simulan la activación de LEDs y la interacción del usuario mediante switches.

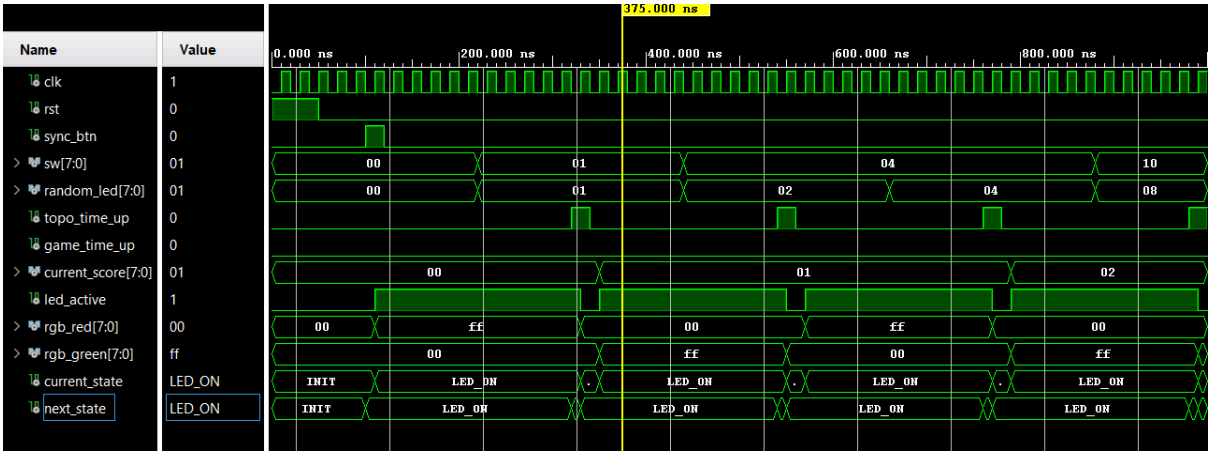
En el estado INIT, la FSM espera que se active el botón de sincronización (sync_btn) para iniciar el juego.

En el estado LED_ON, la FSM activa un led aleatorio (indicado por random_led) y permite que el usuario interactúe mediante los switches (sw).

En el estado CHECK, la FSM actualiza la puntuación y muestra el resultado del intento previo. El resultado es correcto cuando hay coincidencia entre random_led y sw, el indicador RGB muestra color verde (rgb_green = FF) y se incrementa la puntuación (current_score). En cambio, el resultado es incorrecto si no hay coincidencia, el indicador RGB muestra color rojo (rgb_red = FF) y la puntuación se mantiene.

Finalmente, al concluir el tiempo total del juego (game_time_up = 1), la FSM transiciona al estado FINAL. En este estado el led rojo permanece encendido como indicación del fin del juego (rgb_red = FF).

Resultados:



Como se puede observar la simulación mostró un comportamiento consistente con el diseño de la FSM. A continuación, se explica el comportamiento de cada estado en la simulación:

En el estado INIT, la FSM permanece en este estado hasta que sync_btn se activa (aproximadamente en 100 ns), momento en el cual la FSM pasa al estado LED_ON.

En el estado LED_ON, la simulación confirma la detección de aciertos. Por ejemplo, transcurridos 200 ns, se activa el led 1 (random_led = 01) y el switch correspondiente (sw = 01). Entonces la FSM detecta el acierto, por lo tanto, se reflejará posteriormente en el estado CHECK. También, transcurridos 400 ns, se activa el led 2 (random_led = 02), pero el switch no es su correspondiente (sw = 04). Entonces la FSM no detecta que haya un acierto, ya que se ha fallado. Este comportamiento se repite para los siguientes leds, alternando entre aciertos y fallos.

En el estado CHECK se realizan las actualizaciones de la puntuación y los indicadores RGB mediante los estímulos aplicados. Por ejemplo, transcurridos 300 ns, después de un acierto en el estado LED_ON, la puntuación incrementa a 1 y el

indicador RGB se enciende en color verde. Sin embargo, transcurridos 500 ns, tras un intento fallido, el indicador RGB se enciende en color rojo. Al finalizar la simulación, se aprecia que han habido 2 aciertos (random_led y sw= 01 después de 300 ns y 04 antes de 700 ns) ya que el valor de current_score es 2.

El estado FINAL no se observa en la simulación ya que la señal game_time_up no se activa dentro del tiempo de simulación establecido.

En conclusión, se puede concluir que la FSM funciona correctamente, realizando las transiciones de estado según los estímulos aplicados, aunque no se observe en la simulación la transición FINAL. También, se detectan los aciertos entre random_led y sw adecuadamente, actualizando los indicadores y la puntuación. Por último, el comportamiento de los RGB refleja de forma precisa los resultados de cada intento.

DISPLAY_CONTROLLER

Para representar la puntuación se optó por representarla en los displays de siete segmentos que contiene la fpga. En primer lugar, se empleó el componente codificador empleado en el laboratorio, pero este no cumplia con las caráteristicas que se necesitaban ya que el objetivo era utilizar una parte de los displays para representar el temporizador de segundos y otra parte para mostrar la puntuación, todo ello actualizado al instante. Por ello, se creo el componente display_controller. Este controla la visualización de dos tipos de información en un conjunto de displays de 7 segmentos: puntuación y tiempo. Además, utiliza multiplexación para alternar entre los diferentes displays rápidamente, generando la percepción de que todos están activos simultáneamente.

Entradas y salidas:

```
entity display_controller is
    Port (
        clk        : in std_logic;                          -- Reloj principal
        rst        : in std_logic;                          -- Señal de reinicio
        score      : in unsigned(7 downto 0);               -- Puntaje (0-255)
        seconds    : in unsigned(13 downto 0);              -- Tiempo en segundos (0-9999)
        anodes     : out std_logic_vector(6 downto 0);      -- Control de los anodos de los displays
        segments   : out std_logic_vector(6 downto 0)       -- Señales para los segmentos del display
    );
end display_controller;
```

- clk (Reloj): Señal de reloj para controlar las operaciones.
- rst (Reinicio): Reinicia el controlador.
- score (Puntuación): Número de 8 bits (0-255) para mostrar en los primeros 3 displays.
- seconds (Tiempo): Número de 14 bits (0-9999) para mostrar en los últimos 4 displays.
- anodes: Controla qué display está activo (lógica activa baja).
- segments: Define los segmentos del display activo para formar los números.

En la architecture se tienen tres señales:

```
signal digit_select : unsigned(2 downto 0) := (others => '0'); -- Selección del display activo
signal current_digit : std_logic_vector(3 downto 0);          -- Dígito actual para el display
signal clk_div : unsigned(15 downto 0) := (others => '0');    -- Divisor de frecuencia para multiplexado
```

- digit_select: Selecciona cuál de los displays está activo (índice del display).
- current_digit: Dígito actual a mostrar en el display activo.
- clk_div: Divisor de frecuencia para multiplexar los displays.

Además, en la arquitectura se tiene la siguiente constante:

```
-- Constantes para los displays
constant MAX_DISPLAYS : integer := 7;
```

- MAX_DISPLAYS: es la constante que representa el número total de segmentos del display, siendo 7 segmentos utilizados en el sistema.

Función de decodificación de dígitos:

Esta función convierte el valor de current_digit a los segmentos del display, activando o apagando según corresponda.

```
-- Decodificación de dígitos a segmentos
function decode_digit(digit : std_logic_vector(3 downto 0)) return std_logic_vector is
begin
    case digit is
        when "0000" => return "0000001"; -- 0
        when "0001" => return "1001111"; -- 1
        when "0010" => return "0010010"; -- 2
        when "0011" => return "0000110"; -- 3
        when "0100" => return "1001100"; -- 4
        when "0101" => return "0100100"; -- 5
        when "0110" => return "0100000"; -- 6
        when "0111" => return "0001111"; -- 7
        when "1000" => return "0000000"; -- 8
        when "1001" => return "0000100"; -- 9
        when others => return "1111111"; -- Apagado
    end case;
end function;
```

Proceso de divisor de frecuencia:

Este proceso reduce la frecuencia del reloj para controlar el ritmo del multiplexado. A través del clk_div (15), el controlador alterna rápidamente entre los displays creando la ilusión de que todos están activos.

```
-- Divisor de frecuencia para multiplexado
process(clk, rst)
begin
    if rst = '0' then
        clk_div <= (others => '0');
    elsif rising_edge(clk) then
        clk_div <= clk_div + 1;
    end if;
end process;

-- Selección del dígito y anodo activo
process(clk_div(15), rst)
begin
    if rst = '0' then
        digit_select <= (others => '0');
    elsif rising_edge(clk_div(15)) then
        if digit_select = to_unsigned(MAX_DISPLAYS - 1, 3) then
            digit_select <= (others => '0');
        else
            digit_select <= digit_select + 1;
        end if;
    end if;
end process;
```

Proceso de selección de display activo:

Este proceso cambia el display activo mediante un contador cada pulso de reloj, que se reinicia al alcanzar el número máximo de displays.

```
-- Selección del dígito y anodo activo
process(clk_div(15), rst)
begin
    if rst = '0' then
        digit_select <= (others => '0');
    elsif rising_edge(clk_div(15)) then
        if digit_select = to_unsigned(MAX_DISPLAYS - 1, 3) then
            digit_select <= (others => '0');
        else
            digit_select <= digit_select + 1;
        end if;
    end if;
end process;
```

Proceso del mapeo del contenido de los displays:

Este proceso asigna qué contenido se muestra en cada display alternando entre la puntuación y el tiempo.

```
-- Mapeo del contenido de los displays
process(digit_select, score, seconds)
begin
    case digit_select is
        when "000" => -- Display 1 (puntaje, centenas)
            current_digit <= std_logic_vector(to_unsigned(to_integer(score) / 100, 4));
            anodes <= "1111110"; -- Activa el display 1
        when "001" => -- Display 2 (puntaje, decenas)
            current_digit <= std_logic_vector(to_unsigned(to_integer(score / 10) mod 10, 4));
            anodes <= "1111101"; -- Activa el display 2
        when "010" => -- Display 3 (puntaje, unidades)
            current_digit <= std_logic_vector(to_unsigned(to_integer(score) mod 10, 4));
            anodes <= "1111011"; -- Activa el display 3
        when "011" => -- Display 4 (tiempo, millares)
            current_digit <= std_logic_vector(to_unsigned(to_integer(seconds) / 1000, 4));
            anodes <= "1110111"; -- Activa el display 4
        when "100" => -- Display 5 (tiempo, centenas)
            current_digit <= std_logic_vector(to_unsigned(to_integer(seconds / 100) mod 10, 4));
            anodes <= "1101111"; -- Activa el display 5
        when "101" => -- Display 6 (tiempo, decenas)
            current_digit <= std_logic_vector(to_unsigned(to_integer(seconds / 10) mod 10, 4));
            anodes <= "1011111"; -- Activa el display 6
        when "110" => -- Display 7 (tiempo, unidades)
            current_digit <= std_logic_vector(to_unsigned(to_integer(seconds) mod 10, 4));
            anodes <= "0111111"; -- Activa el display 7
        when others =>
            current_digit <= "0000";
            anodes <= "1111111"; -- Todos los displays apagados
    end case;
end process;
```

Tras los procesos, se asigna la señal current_digit de la funció decode_digit a los segmentos del display.

```
-- Decodificación de dígito a segmentos
segments <= decode_digit(current_digit);
```

Además, la asignación de información a los displays está formada por la puntuación y el tiempo. Consecuentemente, los 3 primeros displays muestran las centenas, decenas y unidades de la puntuación, y los últimos 4 displays muestran los millares, centenas, decenas y unidades del tiempo.

Testbench:

Para realizar la prueba del display controller se han empleado dos testbench, uno de 1000 ns para observar la puntuación y el tiempo y otro de 500ms para identificar los anodos y segmentos.

- 1000ns

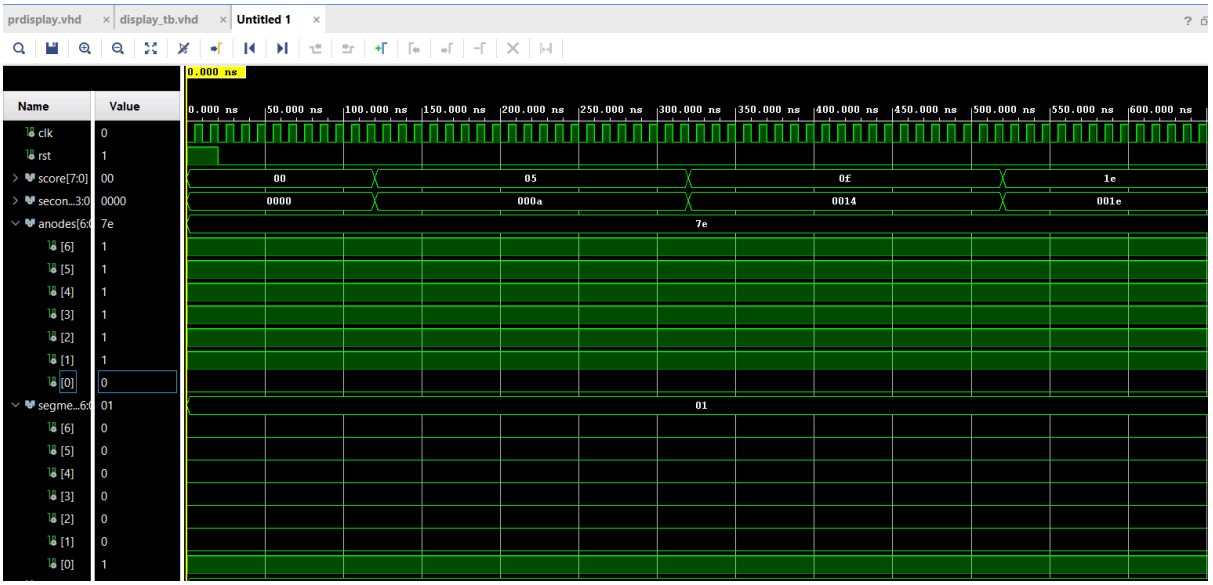
  Código:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity tb_display_controller is
-- Testbench no tiene puertos
end tb_display_controller;

architecture Behavioral of tb_display_controller is

    -- Componentes y señales internas
    component display_controller
        Port (
            clk         : in std_logic;
            rst         : in std_logic;
            score       : in unsigned(7 downto 0);
            seconds     : in unsigned(13 downto 0);
            anodes      : out std_logic_vector(6 downto 0);
            segments    : out std_logic_vector(6 downto 0)
        );
    end component;

    -- Señales para la simulación
    signal clk         : std_logic := '0';
    signal rst         : std_logic := '0';
    signal score       : unsigned(7 downto 0) := (others => '0');
    signal seconds     : unsigned(13 downto 0) := (others => '0');
    signal anodes      : std_logic_vector(6 downto 0);
    signal segments    : std_logic_vector(6 downto 0);


    -- Parámetro para el reloj
    constant CLK_PERIOD : time := 10 ns; -- Período del reloj (100 MHz)

begin

    -- Instancia del módulo display_controller
    uut: display_controller
        Port map (
            clk         => clk,
            rst         => rst,
            score       => score,
            seconds     => seconds,
            anodes      => anodes,
            segments    => segments
        );

    -- Generación del reloj
    clk_process : process
    begin
        while True loop
            clk <= '0';
            wait for CLK_PERIOD / 2;
            clk <= '1';
            wait for CLK_PERIOD / 2;
        end loop;
    end process;

    -- Proceso de simulación
    stim_process : process
    begin
```

```
    -- Proceso de simulación
    stim_process : process
    begin
        -- Inicialización
        rst <= '1';               -- Activa el reinicio
        wait for 20 ns;           -- Espera dos ciclos de reloj
        rst <= '0';               -- Desactiva el reinicio

        -- Simula diferentes valores de score y seconds dentro del rango 0-30
        wait for 100 ns;
        score <= to_unsigned(5, 8);   -- Configura un puntaje de 5
        seconds <= to_unsigned(10, 14); -- Configura un tiempo de 10 segundos

        wait for 200 ns;
        score <= to_unsigned(15, 8);  -- Configura un puntaje de 15
        seconds <= to_unsigned(20, 14); -- Configura un tiempo de 20 segundos

        wait for 200 ns;
        score <= to_unsigned(30, 8);  -- Configura el puntaje máximo (30)
        seconds <= to_unsigned(30, 14); -- Configura el tiempo máximo (30 segundos)

        wait for 200 ns;
        score <= to_unsigned(0, 8);   -- Configura el puntaje mínimo (0)
        seconds <= to_unsigned(5, 14); -- Configura un tiempo de 5 segundos

        -- Finaliza la simulación
        wait for 500 ns;
        assert false report "Fin de la simulación" severity failure;
    end process;

end Behavioral;
```

El testbench simula el funcionamiento del display. A través del proceso clk_process genera un reloj con un período de 10 ns (100 MHz). Al inicio la señal rst se activa, y se desactiva después de dos ciclos de reloj para simular un reinicio. Para comprobar el correcto comportamiento del display se simulan diferentes valores de score (puntuación) y seconds (tiempo) para ver si se muestran en el display. Estos valores varian entre 0 y 30, como un ejemplo de score: 5, 15, 30, 0, o como un ejemplo de seconds: 10, 20, 30, 5.
Además, se incluyen periodos de espera entre cambios de valores para observar el comportamiento del módulo durante cada configuración.

Resultados:



Se observa claramente cómo score y seconds cambian en el tiempo, por ejemplo, en 300 ns, el score es 0F y en decimal equivale a 15. En 500 ns, el score es 1E y en decimal equivale a 30.

En este caso, el divisor de frecuencia está configurado para cambiar anodes solo cuando clk_div(15) genera un pulso. Si el reloj principal tiene una frecuencia de 100 MHz, clk_div se incrementa a razón de 100 millones de ciclos por segundo.

`clk_div(15)` produce un pulso cada 2^{15} ciclos (32,768 ciclos), lo que equivale a:

$$\text{Tiempo por pulso} = \frac{32,768}{100\ \text{MHz}} = 0.32768\ \text{ms}$$

Si la simulación no dura lo suficiente para que clk_div(15) genere pulsos, el valor de anodes no cambiará. En este caso, el valor parece permanecer en 7E, se podría deber a que no se ha simulado el tiempo suficiente para que el multiplexado progrese.

- 500 ms

Extiende la simulación para que dure varios milisegundos. Esto permitirá observar cómo el anodes cambia cíclicamente.

Código:

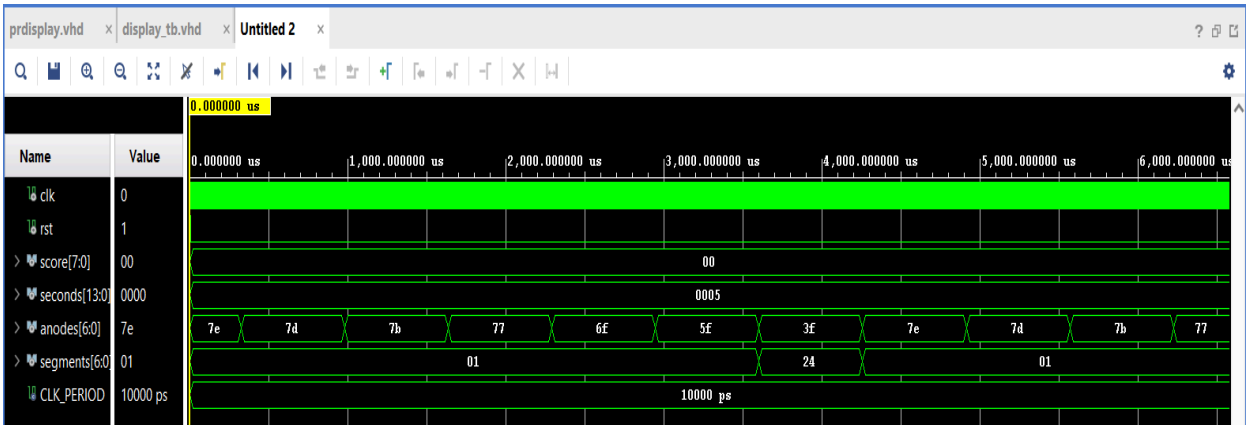
La configuración es la misma que para los 1000 ns.

```
    -- Proceso de simulación
stim_process : process
begin
    -- Inicialización
    rst <= '1';               -- Activa el reinicio
    wait for 20 ns;           -- Espera dos ciclos de reloj
    rst <= '0';               -- Desactiva el reinicio

    -- Simula diferentes valores de score y seconds dentro del rango 0-30
    wait for 100 ns;
    score <= to_unsigned(5, 8);    -- Configura un puntaje de 5
    seconds <= to_unsigned(10, 14); -- Configura un tiempo de 10 segundos

    wait for 200 ns;
    score <= to_unsigned(15, 8);   -- Configura un puntaje de 15
    seconds <= to_unsigned(20, 14); -- Configura un tiempo de 20 segundos

    wait for 200 ns;
    score <= to_unsigned(30, 8);   -- Configura el puntaje máximo (30)
    seconds <= to_unsigned(30, 14); -- Configura el tiempo máximo (30 segundos)

    wait for 200 ns;
    score <= to_unsigned(0, 8);    -- Configura el puntaje mínimo (0)
    seconds <= to_unsigned(5, 14); -- Configura un tiempo de 5 segundos

    -- Mantén la simulación activa por 500 ms
    wait for 500 ms;

    -- Finaliza la simulación
    assert false report "Fin de la simulación" severity failure;
end process;
```

Este testbench se asemeja al anterior ya que tiene la misma estructura pero varian ciertos aspectos. El reloj se genera igual pero con un periodo de 10 ns (100 MHz) alternando entre nivel alto y bajo cada 5 ns. El rst se activa por 20 ns al inicio para inicializar el módulo. La estimulación (stim_process) sigue configurando diferentes valores para score y seconds entre 0 y 30 pudiendose observar los resultados.

A diferencia del testbench anterior la simulación dura 500 ms, tiempo suficiente para ver cómo cambia la salida de anodes y segments debido al multiplexado.

Resultados:



Como se observa en la simulación los valores de anodes cambian cíclicamente entre 7E, 7D, 7B, 77, 6F, 5F, 3F. Esto muestra que el multiplexado de los displays está funcionando como se esperaba: 7E activa el display 0, 7D activa el display 1, 7B activa el display 2, y así sucesivamente, asegurando que cada display se activa en orden.

Por otro lado, los valores de segments cambian según el contenido de score y seconds. Esto indica que el decodificador está generando correctamente los patrones de segmentos para mostrar los números.

PWM_RGB

Generador de señales PWM, para controlar led RGB de la placa.

Funcionamiento: Cuando se acierte se encenderá el LD17 de color verde, en caso contrario se encenderá de color rojo. Aunque no se emplea el color azul en el trabajo, se ha decidido mantenerlo por si fuera necesario en futuras mejoras.

Entradas y salidas:

```
clk        : in std_logic;              -- Reloj principal
rst        : in std_logic;              -- Señal de reinicio
red_duty   : in unsigned(7 downto 0);   -- Ciclo de trabajo para el canal rojo
green_duty : in unsigned(7 downto 0);   -- Ciclo de trabajo para el canal verde
blue_duty  : in unsigned(7 downto 0);   -- Ciclo de trabajo para el canal azul
red_out    : out std_logic;             -- Salida PWM para el canal rojo
green_out  : out std_logic;             -- Salida PWM para el canal verde
blue_out   : out std_logic              -- Salida PWM para el canal azul
```

- clk: Sincroniza el proceso
- rst: Reinicio del juego, si está a 0 el circuito vuelve a su estado inicial (INIT en la FSM)
- red_duty, green_duty, blue_duty: Indica qué porcentaje de tiempo la salida PWM estará en estado ‘1’
- red:_out, green_out, blue_out: Salidas PWM.

Dentro de architecture sólo se tiene una señal:

```
signal pwm_counter : unsigned(7 downto 0) := (others => '0'); -- Contador de PWM
```

Es un contador de 8 bits que genera el patrón de modulación PWM, este contador se incrementa con cada ciclo de reloj.

```
process(clk, rst)
begin
    if rst = '0' then
        pwm_counter <= (others => '0');
    elsif rising_edge(clk) then
        pwm_counter <= pwm_counter + 1;
    end if;
end process;
```

Generación de las señales PWM:

```
red_out <= '1' when pwm_counter < red_duty else '0';
green_out <= '1' when pwm_counter < green_duty else '0';
blue_out <= '1' when pwm_counter < blue_duty else '0';
```

Cada salida PWM se basa en una comparación entre el pwm_counter y el ciclo de trabajo correspondiente. Si el valor del contador es menor que el ciclo de trabajo la salida se pondrá a ‘1’ en caso contrario se pondrá a 0.

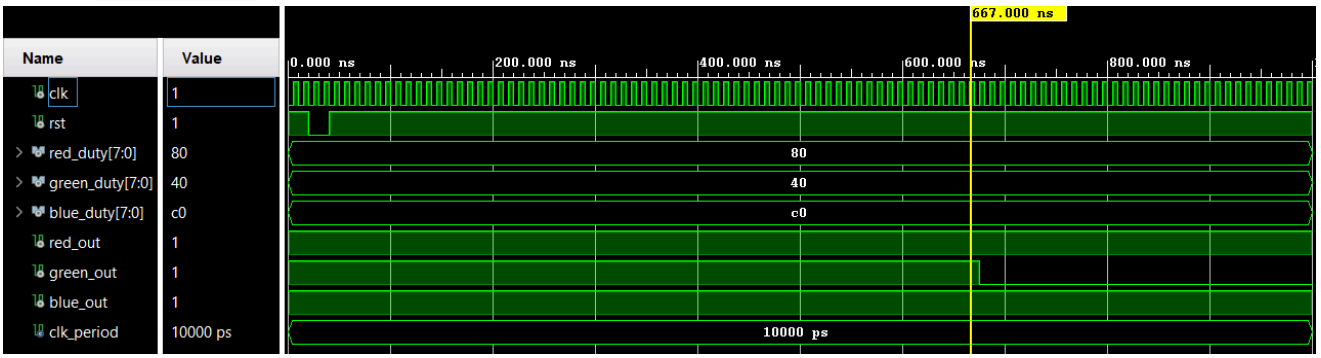
Testbench:

Código:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_TEXTIO.ALL;
use std.textio.ALL;

entity tb_pwm_rgb is
end tb_pwm_rgb;

architecture Behavioral of tb_pwm_rgb is
    -- Señales para conectar con la entidad pwm_rgb
    signal clk        : std_logic := '0';
    signal rst        : std_logic := '0';
    signal red_duty   : unsigned(7 downto 0) := (others => '0');
    signal green_duty : unsigned(7 downto 0) := (others => '0');
    signal blue_duty  : unsigned(7 downto 0) := (others => '0');
    signal red_out    : std_logic;
    signal green_out  : std_logic;
    signal blue_out   : std_logic;

    -- Período del reloj
    constant clk_period : time := 10 ns;

begin
    -- Instancia del módulo bajo prueba (UUT)
    uut: entity work.pwm_rgb port map (
        clk        => clk,
        rst        => rst,
        red_duty   => red_duty,
        green_duty => green_duty,
        blue_duty  => blue_duty,
        red_out    => red_out,
        green_out  => green_out,
        blue_out   => blue_out
    );
```

```
    -- Generador de reloj
    process
    begin
        while now < 1000 ns loop  -- Simulación hasta 1000 ns
            clk <= '0';
            wait for clk_period / 2;
            clk <= '1';
            wait for clk_period / 2;
        end loop;
        wait;
    end process;

    -- Proceso de prueba
    process
    begin
        -- Inicialización
        rst <= '1';
        red_duty   <= to_unsigned(128, 8);  -- 50% ciclo de trabajo
        green_duty <= to_unsigned(64, 8);   -- 25% ciclo de trabajo
        blue_duty  <= to_unsigned(192, 8);  -- 75% ciclo de trabajo
        wait for 20 ns;

        -- Liberamos el reset
        rst <= '0';
        wait for 20 ns;
        rst <= '1';

        -- Esperamos para observar comportamiento
        wait for 500 ns;

        -- Finalizar simulación
        wait;
    end process;

end Behavioral;
```

Resultados:



En la simulación se observa que el clk tiene un periodo de 10 ns sincronizando el incremento del contador pwm. Además, las señales red_duty, green_duty y blue_duty tienen valores de 80 (128 en decimal, 50%), 40 (64 en decimal, 25%) y C0 (192 en decimal, 75%) coincidiendo con los valores establecidos en el testbench. Las salidas red_out, green_out y blue_out reflejan estos ciclos de trabajo activandose mientras el contador pwm es menor al valor correspondiente de duty cycle y luego desactivandose. También se puede considerar que cada canal pwm genera pulsos con la relación correcta validando que el módulo regula adecuadamente la señal de salida para cada color dependiendo de su ciclo de trabajo. Tras analizar los resultados de la simulación se puede declarar que funciona correctamente.

TOPO_TIMER

Es un temporizador básico que opera basado en la frecuencia del reloj. Para su funcionamiento emplea un contador para determinar si ha pasado el tiempo configurado activandose una señal llamada time_up. Este temporizador se puede habilitar y resetear.

Proceso evolutivo:

En primer lugar, se creó este componente únicamente con un tiempo de duración encendido de 1 segundo, para comprobar que funcionaba correctamente el temporizador. Tras obtener el resultado final del juego en la FPGA, se añadió otro tiempo de encendido de los leds de 2 segundos. Esto se obtiene añadiendo otra constante y la selección del tiempo de encendido se elige mediante un switch llamado mode_sel.

Entradas y salidas:

```
entity topo_timer is
    Port (
        clk       : in std_logic;                    -- Reloj principal
        rst       : in std_logic;                    -- Señal de reinicio global
        enable    : in std_logic;                    -- Habilita el temporizador
        mode_sel  : in std_logic;                    -- Selección de modo (0: 1 segundo, 1: 2 segundos)
        time_up   : out std_logic                    -- Señal de "tiempo agotado"
    );
end topo_timer;
```

- clk: es la señal de reloj que sincroniza el proceso.
- rst: es la señal de reinicio del juego, si está a 0 el circuito vuelve a su estado inicial (INIT en la FSM)
- enable: es el habilitador para iniciar o parar el temporizador.
- mode_sel: es el switch de selección del modo de juego, si está en ‘0’ el temporizador es de 1 segundo, y si está en ‘1’ el temporizador es de 2 segundos.
- time_up: es la señal de salida que indica que el tiempo configurado ha pasado.

En la architecture se tienen dos constantes y dos señales:

```
    constant CLOCK_FREQ_1S : unsigned(26 downto 0) := to_unsigned(100_000_000, 27); -- 1 segundo
    constant CLOCK_FREQ_2S : unsigned(26 downto 0) := to_unsigned(200_000_000, 27); -- 2 segundos

    signal clk_counter : unsigned(26 downto 0) := (others => '0'); -- Contador de ciclos de reloj
    signal current_limit : unsigned(26 downto 0);                  -- Límite actual del contador
```

- CLOCK_FREQ_1s: es la constante que representa el número de ciclos de reloj necesarios para durar el temporizador 1 segundo, teniendo una frecuencia de 100 MHz.
- CLOCK_FREQ_2s: es la constante que representa el número de ciclos para 2 segundos.
- clk_counter: es la señal del contador que incrementa el temporizador en cada ciclo de reloj, si este está habilitado.
- current_limit: es la señal del límite dinámico del contador, es decir, determina el tiempo del temporizador dependiendo del valor de mode_sel.

Proceso de selección de modo de juego:

Hemos usado un proceso para realizar la selección del modo de juego permitiendo elegir el tiempo de encendido de los leds para variar la dificultad del juego. Para ello, hemos obtenido dicho resultado mediante un switch llamado mode_sel, el cual dependiendo de su valor cambia current_limit determinando el tiempo del temporizador de los leds.

```
process(mode_sel)
begin
    if mode_sel = '0' then
        current_limit <= CLOCK_FREQ_1S; -- 1 segundo
    else
        current_limit <= CLOCK_FREQ_2S; -- 2 segundos
    end if;
end process;
```

Proceso principal:

Este proceso implementa el temporizador de los leds, de forma que si rst = ‘0’ este se reinicia y la señal time_up pasa a ‘0’. En cambio si el reloj detecta un flanco de subida mientras la señal enable está activa, el temporizador incrementa el contador en cada ciclo comparandolo con el current_limit. Una vez alcanzado el valor límite, el contador se reinicia y time_up se activa indicando que ha pasado el tiempo. Por último, si enable está desactivado el temporizador se detiene reiniciando clk_counter y desactivando time_up.

```
process(clk, rst)
begin
    if rst = '0' then
        clk_counter <= (others => '0');
        time_up <= '0';
    elsif rising_edge(clk) then
        if enable = '1' then
            if clk_counter = current_limit - 1 then
                clk_counter <= (others => '0'); -- Reinicia el contador
                time_up <= '1';                 -- Indica que el tiempo ha terminado
            else
                clk_counter <= clk_counter + 1;
                time_up <= '0';                 -- Tiempo aún en curso
            end if;
        else
            clk_counter <= (others => '0');     -- Reinicia el contador si no está habilitado
            time_up <= '0';
        end if;
    end if;
end process;
```

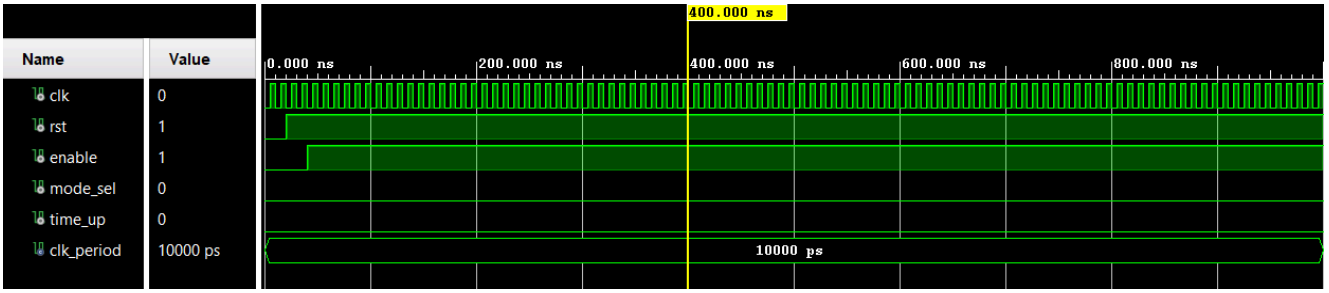
Testbench:

Código:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity topo_timer_tb is
end topo_timer_tb;

architecture behavior of topo_timer_tb is

    -- Señales para conectar con la UUT (Unidad Bajo Prueba)
    signal clk       : std_logic := '0';
    signal rst       : std_logic := '0';
    signal enable    : std_logic := '0';
    signal mode_sel  : std_logic := '0';
    signal time_up   : std_logic;

    -- Periodo de reloj de 10 ns (100 MHz)
    constant clk_period : time := 10 ns;

begin

    -- Instanciamos la UUT (Unidad Bajo Prueba)
    uut: entity work.topo_timer
        port map (
            clk       => clk,
            rst       => rst,
            enable    => enable,
            mode_sel  => mode_sel,
            time_up   => time_up
        );

    -- Generador de reloj
    clk_process: process
    begin
        clk <= '0';
        wait for clk_period / 2;
        clk <= '1';
        wait for clk_period / 2;
    end process;

    -- Estímulos del testbench
    stim_proc: process
    begin
        -- Reinicio y configuración inicial
        rst <= '0';
        mode_sel <= '0'; -- Modo de 1 segundo
        enable <= '0';
        wait for 20 ns;

        rst <= '1';  -- Desactivamos el reset
        wait for 20 ns;

        enable <= '1'; -- Habilitamos el temporizador
        wait for 10 ns;

        -- Comprobamos que time_up se activa después de 1 segundo
        wait for 1 sec;  -- Esperamos 1 segundo
        assert (time_up = '1') report "Error: El tiempo de 1 segundo no se agotó correctamente" severity error;
```

```
        -- Reiniciamos el temporizador y cambiamos a modo de 2 segundos
        rst <= '0';
        wait for 20 ns;
        rst <= '1';
        mode_sel <= '1'; -- Modo de 2 segundos
        wait for 10 ns;

        -- Comprobamos que time_up se activa después de 2 segundos
        wait for 2 sec;  -- Esperamos 2 segundos
        assert (time_up = '1') report "Error: El tiempo de 2 segundos no se agotó correctamente" severity error;

        -- Finalizamos la simulación
        wait;
    end process;

end behavior;
```

Resultados:



GAME_TIMER

Es un temporizador básico que opera basado en la frecuencia del reloj. Es el temporizador del juego. El funcionamiento es bastante similar al del temporizador de topo_timer ya que emplea un contador para determinar si ha pasado el tiempo configurado activando una señal llamada time_up.

Entradas y salidas:

```
entity game_timer is
    Port (
        clk       : in std_logic;                    -- Reloj principal
        rst       : in std_logic;                    -- Señal de reinicio global
        enable    : in std_logic;                    -- Habilita el temporizador
        start_btn : in std_logic;
        limit     : in unsigned(13 downto 0);        -- Límite en segundos
        seconds   : out unsigned(13 downto 0);       -- Tiempo transcurrido en segundos
        time_up   : out std_logic                    -- Señal de "tiempo agotado"
    );
end game_timer;
```

- clk: es la señal de reloj que sincroniza el proceso.
- rst: es la señal de reinicio del juego, si está a 0 el circuito vuelve a su estado inicial (INIT en la FSM)
- enable: es el habilitador para iniciar o parar el temporizador.
- start_btn: es el botón que inicia el juego, y por tanto, el temporizador de la partida del juego.
- limit: es el límite configurado en segundosque define cuando debe activarse time_up.

- seconds: es la salida que muestra el tiempo transcurrido en segundos.
- time_up: es la señal de salida que indica que el tiempo configurado ha pasado.

En la architecture se tienen una constante y tres señales:

```
constant CLOCK_FREQ : unsigned(26 downto 0) := to_unsigned(100_000_000, 27); -- Frecuencia del reloj (100 MHz)
signal clk_counter  : unsigned(26 downto 0) := (others => '0');              -- Contador de ciclos de reloj
signal second_count : unsigned(13 downto 0) := (others => '0');              -- Contador de segundos
signal timer_active : std_logic := '0';                                      -- Control interno para detener el temporizador
```

- CLOCK_FREQ: es la constante que representa el número de ciclos de reloj necesarios para 1 segundo, teniendo una frecuencia de 100 MHz.
- clk_counter: es la señal del contador que incrementa el temporizador en cada ciclo de reloj, si este está habilitado.
- second_count: es la señal del contador que lleva el tiempo transcurrido en segundos.
- timer_active: es la señal del control interno que indica si el temporizador está activo.

Proceso principal:

Este proceso implementa el temporizador del juego, que es síncrono controlado por un reloj. El funcionamiento de este consiste en que al estar rst =’0’ el temporizador se reinicia poniendo los contadores a cero y desactivando time_up y timer_active. Por otro lado, al presionar start_btn el juego comienza y el temporizador comienza a funcionar incrementando clk_counter. Además, mientras clk_counter se incrementa, también lo hace second_counter hasta alcanzar limit. Una vez se ha alcanzado limit se activa time_up y el temporizador se detiene. Finalmente, cuenta con enable de forma que solo funciona si está activado y así estando habilitado el reloj.

```
process(clk, rst,start_btn)
begin
    if rst = '0' then
        clk_counter <= (others => '0');          -- Reinicia el contador de ciclos
        second_count <= (others => '0');         -- Reinicia el contador de segundos
        time_up <= '0';                          -- Resetea la señal de tiempo agotado
       timer_active <= '0';                     -- Activa el temporizador tras reinicio
    elsif start_btn = '1' then
    timer_active <= '1';
    elsif rising_edge(clk) then
        if enable = '1' and timer_active = '1' then
            -- Incrementa el contador de ciclos
            if clk_counter = CLOCK_FREQ - 1 then
                clk_counter <= (others => '0');          -- Reinicia el contador de ciclos
                second_count <= second_count + 1;        -- Incrementa el contador de segundos
            else
                clk_counter <= clk_counter + 1;          -- Incrementa el contador de ciclos
            end if;

            -- Comprueba si se alcanzó el límite
            if second_count = limit then
                time_up <= '1';                          -- Indica que el tiempo ha terminado
                timer_active <= '0';                     -- Detiene el temporizador
            end if;
        end if;
    end if;
end process;
```

Tras el proceso se asigna la señal second_count a la salida seconds para proporcionar el tiempo transcurridos en segundos.

```
seconds <= second_count;
```

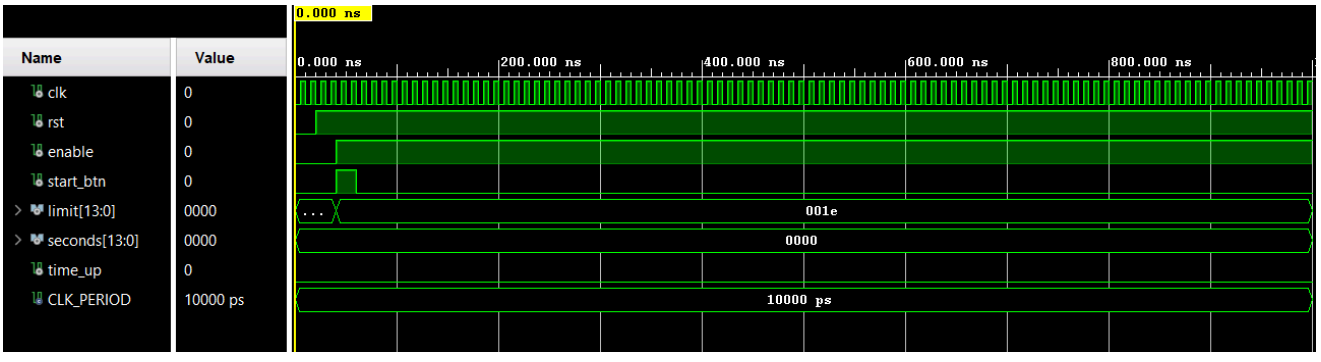
Testbench:

Código:

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

entity tb_game_timer is
-- No tiene puertos, ya que es un testbench
end tb_game_timer;

architecture Behavioral of tb_game_timer is
    -- Componentes del DUT (Device Under Test)
    component game_timer
        Port (
            clk       : in std_logic;
            rst       : in std_logic;
            enable    : in std_logic;
            start_btn : in std_logic;
            limit     : in unsigned(13 downto 0);
            seconds   : out unsigned(13 downto 0);
            time_up   : out std_logic
        );
    end component;

    -- Señales internas del testbench
    signal clk       : std_logic := '0';
    signal rst       : std_logic := '0';
    signal enable    : std_logic := '0';
    signal start_btn : std_logic := '0';
    signal limit     : unsigned(13 downto 0) := (others => '0');
    signal seconds   : unsigned(13 downto 0);

    signal time_up   : std_logic;

    constant CLK_PERIOD : time := 10 ns; -- Período del reloj (100 MHz)
begin
    -- Instancia del DUT
    uut: game_timer
        port map (
            clk       => clk,
            rst       => rst,
            enable    => enable,
            start_btn => start_btn,
            limit     => limit,
            seconds   => seconds,
            time_up   => time_up
        );

    -- Generador de reloj
    clk_process: process
    begin
        while true loop
            clk <= '0';
            wait for CLK_PERIOD / 2;
            clk <= '1';
            wait for CLK_PERIOD / 2;
        end loop;
    end process;
```

```
    -- Proceso de estímulos
    stimulus_process: process
    begin
        -- Reinicio global
        rst <= '0';
        wait for 20 ns;
        rst <= '1';
        wait for 20 ns;

        -- Configura límite a 30 segundos
        limit <= to_unsigned(30, 14);

        -- Prueba: Inicia el temporizador
        enable <= '1';
        start_btn <= '1';
        wait for 20 ns;
        start_btn <= '0';

        -- Espera hasta que el temporizador alcance el límite
        wait for 30 * 1 sec;

        -- Verifica que `time_up` esté activo después de 30 segundos
        assert time_up = '1'
        report "Test fallido: El temporizador no terminó correctamente en 30 segundos" severity error;

        -- Comprueba que la salida `seconds` haya llegado a 30
        assert seconds = to_unsigned(30, 14)
        report "Test fallido: El contador no llegó a 30 segundos" severity error;

        -- Reinicia y prueba otro caso
        rst <= '0';
        wait for 20 ns;
        rst <= '1';

        -- Prueba: Temporizador no debe contar con `enable` desactivado
        enable <= '0';
        start_btn <= '1';
        wait for 20 ns;
        start_btn <= '0';
        wait for 1 sec;

        -- Verifica que `seconds` no incremente
        assert seconds = to_unsigned(0, 14)
        report "Test fallido: El temporizador contó con enable desactivado" severity error;

        -- Final de la simulación
        wait;
    end process;
end Behavioral;
```

Resultados:



SYNCHRONIZER

Es el módulo de sincronización de nuestro programa. Se ha modificado el código visto en la práctica 2 de la asignatura.

Como se ha comentado, las señales de entrada a la FPGA pueden generar metaestabilidad. Esto ocurre cuando, por ejemplo, se conecta la entrada a un flip-flop y la señal de entrada cambia cerca del flanco de la señal de reloj, no respetando el tiempo de setup o de hold. Esto puede hacer que el flip-flop no sea capaz de determinar de manera confiable si debe capturar un 0 o un 1, entrando en un estado intermedio, donde la salida no es ni un 0 lógico ni un 1 lógico, o tarda un tiempo indeterminado en estabilizarse (ver Figura 10). Esto ocurre cuando se quiere detectar un flanco de la señal de entrada y usamos 'event'. Además, Vivado suele detectar estos problemas y da un error o warning.

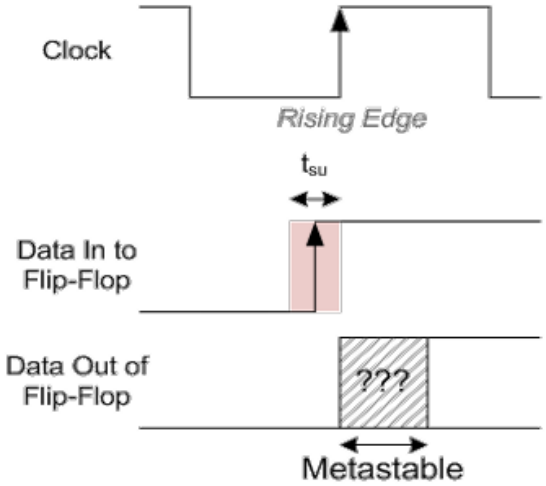


Figura 10: Ejemplo de metaestabilidad

Una de las posibles soluciones pasa por hacer la señal de entrada por varios Flip-Flops, de manera que se minimiza las probabilidades de que ocurra la metaestabilidad (ver Figura 11). El primer Flip-Flop captura la señal asíncrona en el flanco de subida del reloj. El segundo Flip-Flop toma la salida del primer Flip-Flop y actúa como un filtro para para asentar cualquier estado metaestable, y evitar su propagación al resto del sistema.

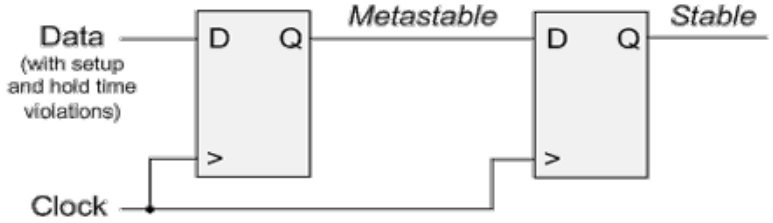


Figura 11: Ejemplo de solución a la metaestabilidad

Entradas y salidas:

```
entity synchronizer is
    Port (
        clk       : in std_logic;
        async_in  : in std_logic;
        sync_out  : out std_logic
    );
end synchronizer;
```

- clk: Es el reloj del sistema que sincroniza la señal.
- async_in: Es la señal asincrónica que llega al sincronizador.
- sync_out: Es la salida sincronizada a nuestro reloj clk.

Dentro del architecture tenemos una señal:

```
signal sync_reg : std_logic_vector(1 downto 0) := (others => '0');
```

Es un registro de dos bits que almacena dos etapas del sincronizador:

- sync_reg(0): guarda el primer valor sincronizado.
- sync_reg(1): guarda el segundo valor sincronizado.

Proceso que permite la sincronización:

```
process(clk)
begin
    if rising_edge(clk) then
        sync_reg(0) <= async_in;
        sync_reg(1) <= sync_reg(0);
    end if;
end process;
```

La señal asíncrona se captura en la primera etapa, el valor de la primera se transfiere a la segunda etapa. Este doble almacenamiento es clave para minimizar el riesgo de metaestabilidad.

Por último, se consigue la salida sincronizada sync_out toma el valor valor estable de la segunda etapa, sincronizándola al reloj del sistema:

```
sync_out <= sync_reg(1);
```

Ventajas:

Evita la metastabilidad: La señal pasa por dos registros antes de usarse, lo que aumenta las probabilidades de estabilización.

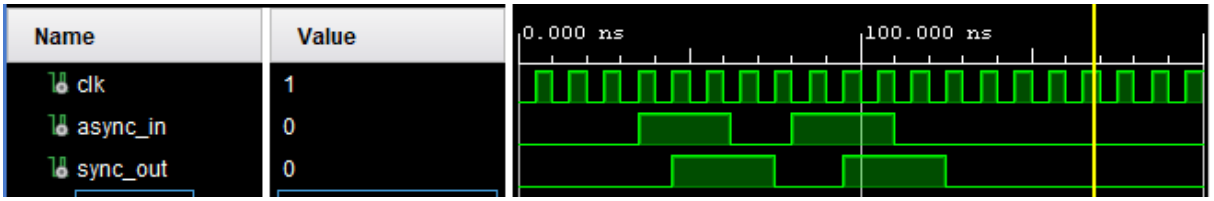
Sincronización confiable: Convierte señales asincrónicas en señales sincronizadas con el reloj del sistema.

Testbench:

Código:

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

entity synchronizer is
    Port (
        clk       : in std_logic;
        async_in  : in std_logic;
        sync_out  : out std_logic
    );
end synchronizer;

architecture Behavioral of synchronizer is
    signal sync_reg : std_logic_vector(1 downto 0) := (others => '0');
begin
    process(clk)
    begin
        if rising_edge(clk) then
            sync_reg(0) <= async_in;
            sync_reg(1) <= sync_reg(0);
        end if;
    end process;

    sync_out <= sync_reg(1);
end Behavioral;
```

Resultados:



EDGETCTR (DETECTOR DE FLANCO)

Cuando se pulsa el botón se genera un pulso de duración indeterminada (hasta que se suelta el botón). Esto es un inconveniente en circuitos síncronos, donde se quiere que todo esté bien medido.

Para solucionar esto se utiliza este módulo, el cual genera un solo pulso (de duración un período de la señal de reloj) cada vez que hay un flanco de bajada en la señal de entrada (botón). De esta forma, cada vez que se pulse el botón se generará una señal que dura un ciclo de reloj, y consecuentemente sólo habrá un flanco de subida en un período de reloj.

Entradas y salidas:

```
entity EDGEDTCTR is
    port (
        CLK       : in std_logic;
        SYNC_IN   : in std_logic;
        EDGE      : out std_logic
    );
end EDGEDTCTR;
```

- CLK: Es la señal del reloj de entrada. Controla las transiciones del proceso.
- SYNC_IN: Es la señal de entrada cuya transición (flanco) queremos detectar.
- EDGE: Es la señal de salida que indicará si se ha detectado un flanco ascendente en SYNC_IN.

Dentro de architecture tenemos una señal interna:

```
signal sreg : std_logic_vector(2 downto 0);
```

Es un registro de 3 bits, utilizado para almacenar los estados anteriores de SYNC_IN y detectar la transición del flanco. El tamaño de 3 bits es suficiente para recordar los dos valores previos de SYNC_IN y detectar un flanco ascendente.

Cuando se produce un flanco de subida de CLK:

```
sreg <= sreg(1 downto 0) & SYNC_IN;
```

El nuevo valor de SYNC_IN se coloca en el bit menos significativo de sreg, desplazando los otros valores hacia la izquierda. Esto permite almacenar los dos valores anteriores de SYNC_IN en los bits más significativos de sreg.

La salida EDGE se pone a '1' cuando sreg es igual a "100". Esto indica que ha ocurrido un flanco ascendente de la señal SYNC_IN, porque:

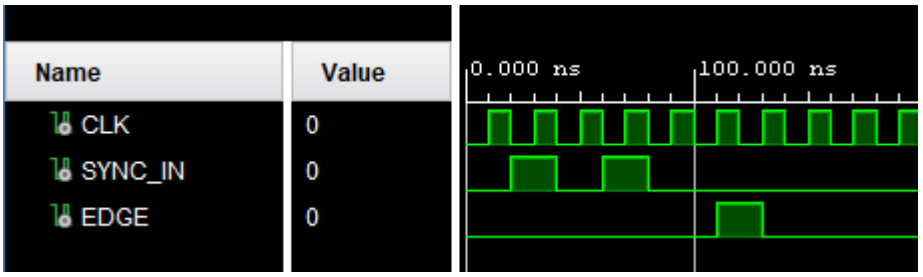
- En el ciclo anterior, SYNC_IN era '0' (bajo).
- En el ciclo actual, SYNC_IN es '1' (alto).
- El patrón "100" en sreg refleja una transición de bajo a alto, lo que indica un flanco ascendente de SYNC_IN

Testbench:

Código:

```
-- Proceso para estimular la señal de entrada SYNC_IN
stim_proc: process
begin
    -- Caso 1: Inicialización, no debe haber flanco (EDGE = '0')
    SYNC_IN <= '0';
    wait for 20 ns;  -- Esperamos 20 ns

    -- Caso 2: Cambio de SYNC_IN de '0' a '1' (flanco ascendente)
    SYNC_IN <= '1';
    wait for 20 ns;  -- Esperamos 20 ns

    -- Caso 3: Cambio de SYNC_IN de '1' a '0' (no hay flanco ascendente)
    SYNC_IN <= '0';
    wait for 20 ns;  -- Esperamos 20 ns

    -- Caso 4: Otro cambio de '0' a '1' (nuevo flanco ascendente)
    SYNC_IN <= '1';
    wait for 20 ns;  -- Esperamos 20 ns

    -- Caso 5: Volvemos a '0', sin flanco ascendente
    SYNC_IN <= '0';
    wait for 20 ns;  -- Esperamos 20 ns

    -- Finalizamos la simulación
    wait;
end process;
```

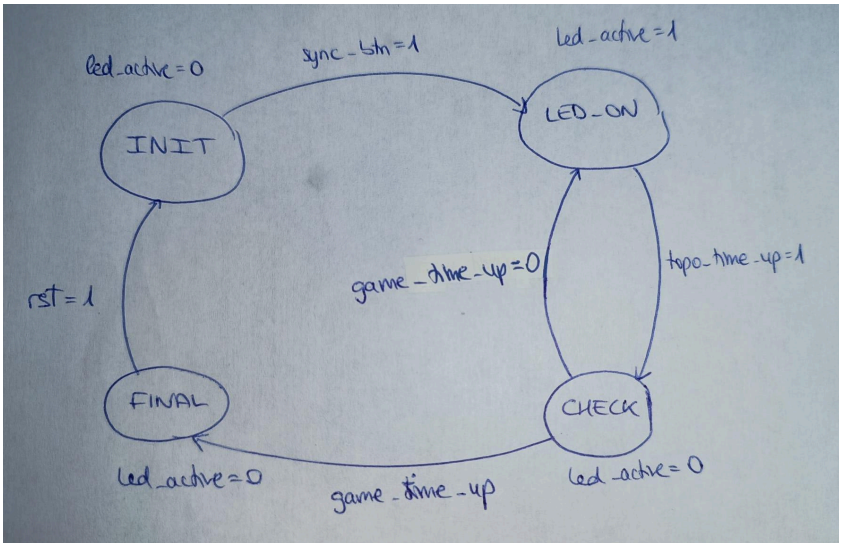
Resultados:



Como se puede observar en los resultados obtenidos en la simulación, inicialmente, sync_in se mantiene en 0, por lo que no se activa la señal EDGE. Al cambiar SYNC_IN de 0 a 1 se genera un flanco ascendente y EDGE se activa brevemente validando la detección del evento. Luego, al regresar SYNC_IN a 0, no se vuelve a generar ningún pulso EDGE porque solo detecta flancos ascendentes. Por lo tanto, se confirma que el funcionamiento de esta entidad es correcto a través del testbench.

FUNCIONAMIENTO DEL DISEÑO

Para el desarrollo del funcionamiento de la FSM se ha realizado el siguiente diagrama de estados:



A continuación, se obtiene la tabla de estados correspondiente a la FSM:

Estados actuales	rst	sync_btn	acierto (sw=random_led)	topo_time_up	game_time_up	Estados siguientes	led_active	rgb_red	rgb_green	current_score
INIT	1	0	X	X	X	INIT	0	0	0	0
	0	1	X	X	X	LED_ON	0	0	0	0
LED_ON	0	X	1	0	X	LED_ON	1	0	1	current_score
	0	X	0	0	X	LED_ON	1	1	0	current_score
	0	X	0	1	X	CHECK	0	0	0	current_score
CHECK	0	X	0	X	0	LED_ON	0	0	0	current_score
	0	X	1	X	0	LED_ON	0	0	0	current_score+1
	0	X	X	X	1	FINAL	0	0	0	current_score
FINAL	0	X	X	X	X	FINAL	0	1	0	current_score
	1	X	X	X	X	INIT	0	0	0	0

Tras haber obtenido la tabla de estados se obtienen las siguientes ecuaciones de las salidas, comprobando así que se trata de una máquina de Mealy debido a que las salidas dependen de las entradas y los estados.

led_active= LED_ON + rst’

rgb_red = FINAL + LED_ON · acierto’

rgb_green= LED_ON · acierto · rst’

current_score= current_score+1 si CHECK · acierto · rst’