


UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

PRÁCTICAS DE DISEÑO DIGITAL VLSI



M.I. Chávez Rodríguez Norma Elva
M.I. Flores Olvera Vicente
M.I. Fonseca Chávez Elizabeth
M.I. Guevara Rodríguez María del Socorro
M.I. Ibarra Carrillo Mario
M.I. Prieto Meléndez Rafael
M.I. Ramírez Chavarría Roberto Giovanni

PREFACIO

Este manual de prácticas es un testimonio del trabajo y compromiso de los profesores de la academia de Diseño Digital VLSI, en la búsqueda por elevar la calidad de la docencia y favorecer el proceso enseñanza aprendizaje.

Mediante estas prácticas se logró vincular las teorías del diseño digital moderno y los lenguajes de descripción de hardware de alto nivel.

Las prácticas de laboratorio propuestas pueden ser configuradas en cualquier dispositivo FPGA, empleando uno de los lenguajes de descripción de hardware más utilizados en la actualidad por los países líderes en el área: VHDL.

El antecedente para poder utilizar estas prácticas es un curso básico de diseño digital moderno, en donde se dan a conocer las bases de la lógica booleana, la lógica secuencial, y del diseño de dispositivos digitales básicos y manejo de lógica programable.

Los nombres de los autores están escritos en estricto orden alfabético.

Los Autores.

CONTENIDO

Práctica 1	Diseño de un reloj digital	4
Práctica 2	Diseño de registros de corrimiento en cascada	10
Práctica 3	Diseño del control de un tren eléctrico	15
Práctica 4	Diseño del control de servomotores	19
Práctica 5	Diseño del control de intensidad en leds	25
Práctica 6	Diseño del control de motores a pasos	30
Práctica 7	Diseño del control de sensores ultrasónicos	42
Práctica 8	Diseño de un transmisor para comunicación serial	46
Práctica 9	Diseño de un receptor para comunicación serial	52
Práctica 10	Diseño de un generador de video VGA	57
Práctica 11	Emulador de display de 7 segmentos en un monitor	65
Práctica 12	Emulador de contadores en un monitor	71
Práctica 13	Captura de imágenes de cámara digital	75
	Bibliografía	91
	Glosario	92

Práctica 1. DISEÑO DE UN RELOJ DIGITAL

OBJETIVO:

Demostrar a los estudiantes que en un FPGA las declaraciones concurrentes se efectúan al mismo tiempo (en paralelo). Implantar operaciones concurrentes mediante el diseño de un reloj digital, en donde el orden de escritura en las instrucciones concurrentes no afecta el resultado de síntesis o de simulación.

ESPECIFICACIONES:

Utilizando un FPGA y 4 displays de 7 segmentos, diseñar un reloj digital, el cual visualice en los dos primeros displays las horas y en los siguientes dos, los minutos. Cada vez que se llegue a 23 horas con 59 minutos, se reiniciará el conteo de horas y minutos. La figura 1.1 muestra el diagrama del bloque de este sistema.

DIAGRAMA DE BLOQUES:

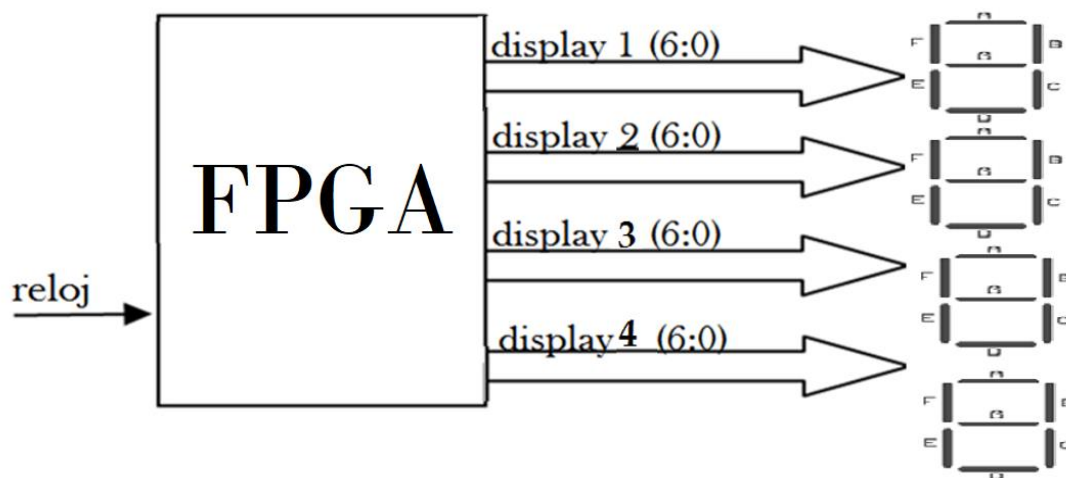


Figura 1.1. Diagrama de bloques del sistema reloj digital^[1]

Un FPGA puede configurarse con muchos bloques funcionales en lenguaje VHDL que estén ejecutando acciones a la vez. A estas acciones ejecutándose al mismo tiempo se le llama ejecución concurrente.

Las señales son declaraciones necesarias cuando se ejecutan instrucciones concurrentes, debido a que ellas unen los bloques funcionales.

La figura 1.2 muestra los bloques funcionales del sistema reloj digital donde las señales se muestran con flechas de color azul.

BLOQUES FUNCIONALES:

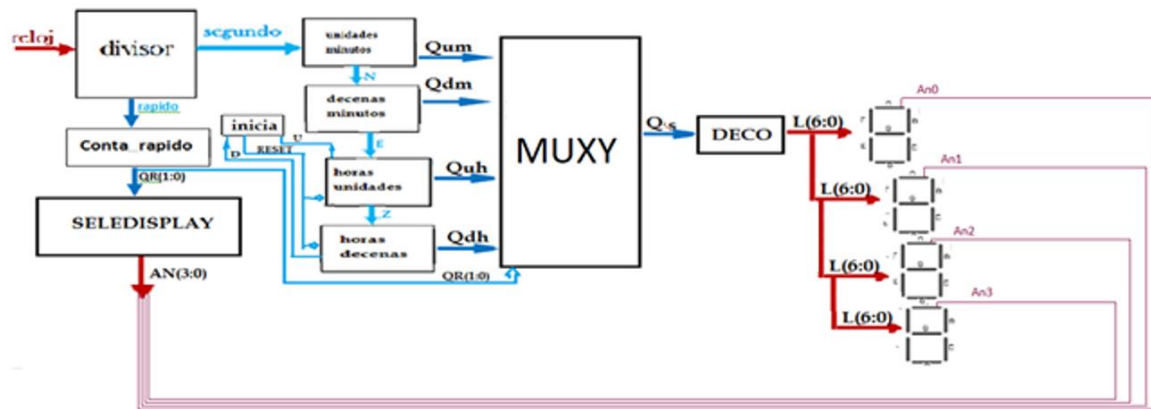


Figura 1.2. Bloques funcionales del sistema reloj digital ^[1]

La figura 1.3 muestra la entidad del sistema reloj digital.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity reldig is
Port (reloj: in std_logic;
      AN: out std_logic_vector (3 downto 0);
      L: out std_logic_vector (6 downto 0));
end reldig;
```

Figura 1.3. Entidad del sistema reloj digital

La figura 1.4 muestra la parte declaratoria de la arquitectura en el sistema reloj digital.

```
architecture behavioral of reldig is
    signal segundo: std_logic;
    signal rapido: std_logic;
    signal n: std_logic;
    signal Qs: std_logic_vector(3 downto 0);
    signal Qum: std_logic_vector(3 downto 0);
    signal Qdm: std_logic_vector(3 downto 0);
    signal e: std_logic;
    signal Qr: std_logic_vector(1 downto 0);
    signal Quh: std_logic_vector(3 downto 0);
    signal Qdh: std_logic_vector(3 downto 0);
    signal z: std_logic;
    signal u: std_logic;
    signal d: std_logic;
    signal reset: std_logic;
```

Figura 1.4. Parte declaratoria en la arquitectura del sistema reloj digital

La figura 1.5 muestra la parte operatoria de la arquitectura en el sistema reloj digital.

```

begin
  divisor: process (reloj)
    variable cuenta: std_logic_vector(27 downto 0) := X"00000000";
  begin
    if rising_edge (reloj) then
      if cuenta=X"48009E0" then
        cuenta:= X"00000000";
      else
        cuenta:= cuenta +1;
      end if;
    end if;
    segundo <= cuenta(22);
    rapido <= cuenta(10);
  end process;

  unidades: process (segundo)
    variable cuenta: std_logic_vector(3 downto 0) := "0000";
  begin
    if rising_edge (segundo) then
      if cuenta ="1001" then
        cuenta:="0000";
        n <= '1';
      else
        cuenta:= cuenta +1;
        n <= '0';
      end if;
    end if;
    qum <= cuenta;
  end process;

```

Figura 1.5. Parte operatoria de la arquitectura del sistema reloj digital

```

decenas: process (n)
    variable cuenta: std_logic_vector(3 downto 0) := "0000";
begin
    if rising_edge (n) then
        if cuenta ="0101" then
            cuenta:="0000";
            e <= '1';
        else
            cuenta:= cuenta +1;
            e<= '0';
        end if;
    end if;
    Qdm <= cuenta;
end process;

HoraU: Process (E,reset)
    variable cuenta: std_logic_vector(3 downto 0):="0000";
begin
    if rising_edge(E) then
        if cuenta="1001" then
            cuenta:= "0000";
            Z<='1';
        else
            cuenta:=cuenta+1;
            Z<='0';
        end if;
    end if;
    if reset='1' then
        cuenta:="0000";
    end if;
    Quh<=cuenta;
    U<=cuenta(2);
end Process;

HoraD: Process (Z, reset)
    variable cuenta: std_logic_vector(3 downto 0):="0000";
begin
    if rising_edge(Z) then
        if cuenta="0010" then
            cuenta:= "0000";
        else
            cuenta:=cuenta+1;
        end if;
    end if;
    if reset='1' then
        cuenta:="0000";
    end if;
    Qdh<=cuenta;
    D <=cuenta(1);
end Process;

```

Figura 1.5. (continuación) Parte operatoria de la arquitectura del sistema reloj digital


```

incia: process (U,D)
begin
    reset <= (U and D);
end process;

Contrapid: process (rapido)
    variable cuenta: std_logic_vector(1 downto 0) := "00";
begin
    if rising_edge (rapido) then
        cuenta:= cuenta +1;
    end if;
    Qr <= cuenta;
end process;

muxy: process (Qr)
begin
    if Qr = "00" then
        Qs<= Qum;
    elsif Qr = "01" then
        Qs<= Qdm;
    elsif Qr = "10" then
        Qs<= Quh;
    elsif Qr = "11" then
        Qs<= Qdh;
    end if;
end process;

seledisplay: process (Qr)
begin
    case Qr is
        when "00" =>
            AN<= "1110";
        when "01" =>
            AN<= "1101";
        when "10" =>
            AN<= "1011";
        when others =>
            AN<= "0111";
    end case;
end process;

with Qs select
    L <= "1000000" when "0000",    --0
        "1111001" when "0001",    --1
        "0100100" when "0010",    --2
        "0110000" when "0011",    --3
        "0011001" when "0100",    --4
        "0010010" when "0101",    --5
        "0000010" when "0110",    --6
        "1111000" when "0111",    --7
        "0000000" when "1000",    --8
        "0010000" when "1001",    --9
        "1000000" when others;    --F
end Behavioral;

```

Figura 1.5. (continuación) Parte operatoria de la arquitectura del sistema reloj digital

ACTIVIDADES COMPLEMENTARIAS:

- 1.- El alumno diseñará un reloj digital con alarma, en el cual se active una señal sonora cuya intensidad vaya aumentando que el usuario apague el sistema.

- 2.- El alumno diseñará un reloj digital que trabaje en sentido contrario a las manecillas del reloj, lo que significa que va a descontar tiempo empezando en un valor preseleccionado por el usuario.

Práctica 2.

DISEÑO DE REGISTROS DE CORRIMIENTO EN CASCADA

OBJETIVO:

Demostrar a los estudiantes que las declaraciones secuenciales requieren de un orden para ser ejecutadas. Diseñar registros de corrimiento en cascada utilizando las estructuras de control *if-then-else* o *case* dentro de un proceso.

ESPECIFICACIONES:

Utilizando un FPGA y 8 displays de 7 segmentos, diseñar un sistema digital que despliegue un mensaje que se desplace en los displays.

La figura 2.1 muestra el diagrama de bloques del sistema registros de corrimiento en cascada.

DIAGRAMA DE BLOQUES:

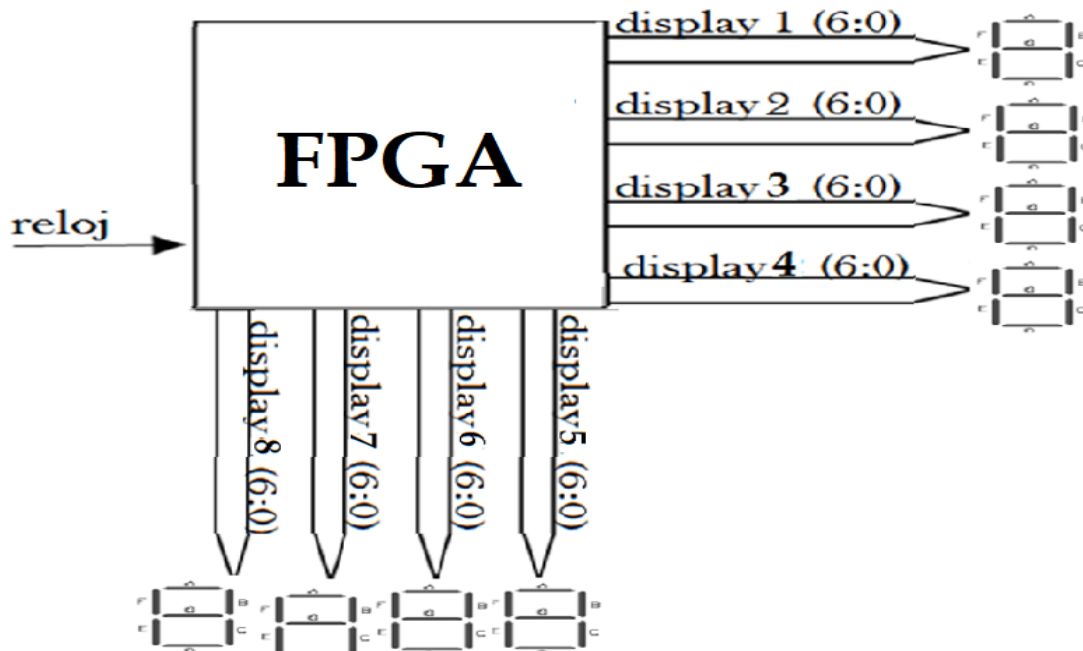


Figura 2.1. Diagrama de bloques del sistema registros de corrimiento en cascada. ^[2]

Dentro del sistema digital registros de corrimiento en cascada se tienen varios bloques funcionales, los cuales internamente ejecutan instrucciones en forma secuencial. La figura 2.2 muestra los bloques funcionales del sistema.

BLOQUES FUNCIONALES:

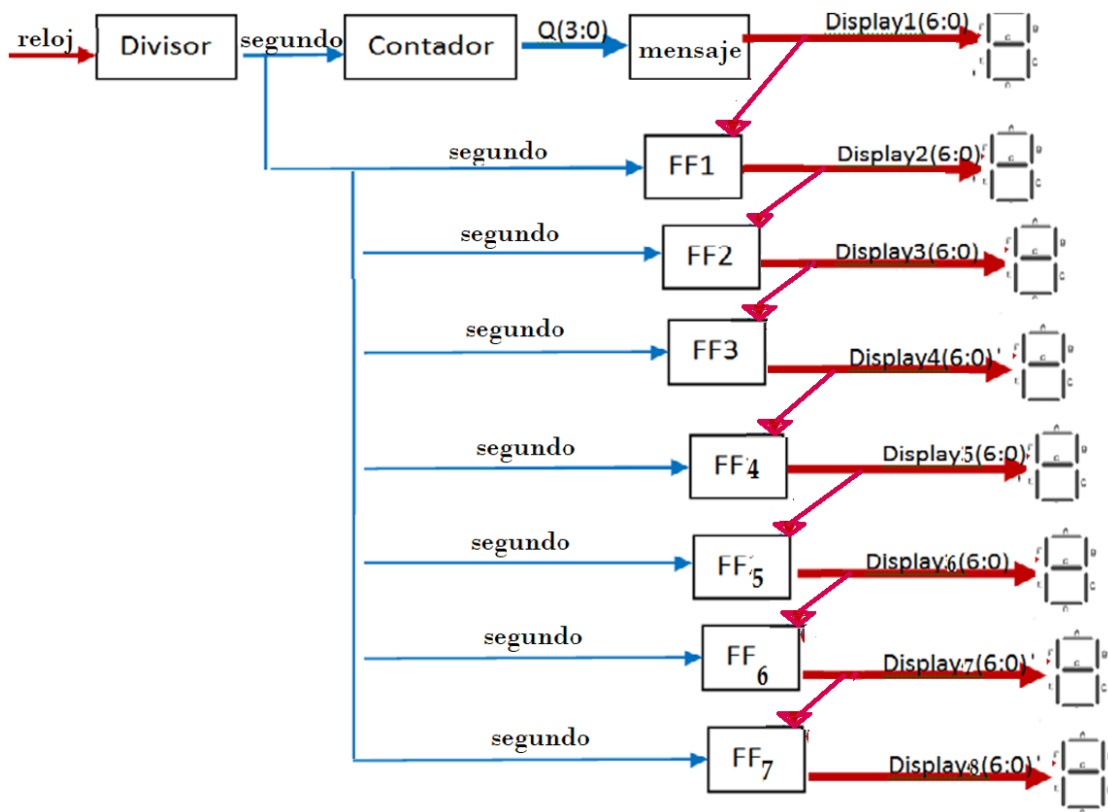


Figura 2.2. Diagrama de bloques funcionales del sistema registros de corrimiento en cascada

La figura 2.3 muestra la entidad del sistema registros de corrimiento en cascada.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity corri is
    Port (reloj: in std_logic;
          display1, display2, display3, display4, display5, display6,
          display7, display8: buffer std_logic_vector (6 downto 0));
end corri;
```

Figura 2.3. Entidad del sistema registros de corrimiento en cascada

La figura 2.4 muestra la parte declaratoria de la arquitectura del sistema registros de corrimiento en cascada.

```
architecture Behavioral of corri is
    signal segundo: std_logic;
    signal Q: std_logic_vector(3 downto 0) := "0000";
```

Figura 2.4. Parte declaratoria de la arquitectura del sistema registros de corrimiento en cascada

La figura 2.5 muestra la parte operatoria de la arquitectura del sistema registros de corrimiento en cascada.

```
begin
    divisor: process (reloj)
        variable cuenta: std_logic_vector(27 downto 0) := X"00000000";
    begin
        if rising_edge (reloj) then
            if cuenta=X"48009E0" then
                cuenta:= X"00000000";
            else
                cuenta:=cuenta+1;
            end if;
        end if;
        segundo <=cuenta(22);
    end process;

    contador:process (segundo)
    begin
        if rising_edge (segundo) then
            Q <= Q +1;
        end if;
    end process;
```

Figura 2.5. Parte operatoria de la arquitectura del sistema registros de corrimiento en cascada

```
with Q select
    display1 <= "0000110" when "0000", -- E
              "0101011" when "0001", -- n
              "1111111" when "0010", -- espacio
              "1000111" when "0011", -- L
              "0001000" when "0100", -- A
              "1111111" when "0101", -- espacio
              "1000000" when "0110", -- O
              "1000111" when "0111", -- L
              "0001000" when "1000", -- A
              "1111111" when others; -- espacios

FF1 : process (segundo)
begin
    if rising_edge (segundo) then
        display2 <= display1;
    end if;
end process;

FF2 : process (segundo)
begin
    if rising_edge (segundo) then
        display3 <= display2;
    end if;
end process;

FF3 : process (segundo)
begin
    if rising_edge (segundo) then
        display4 <= display3;
    end if;
end process;

FF4 : process (segundo)
begin
    if rising_edge (segundo) then
        display5 <= display4;
    end if;
end process;

FF5 : process (segundo)
begin
    if rising_edge (segundo) then
        display6 <= display5;
    end if;
end process;

FF6 : process (segundo)
begin
    if rising_edge (segundo) then
        display7 <= display6;
    end if;
end process;
```

Figura 2.5. (continuación) Parte operatoria de la arquitectura del sistema registros de corrimiento en cascada

```
FF7 : process (segundo)
begin
  if rising_edge (segundo) then
    display8 <= display7;
  end if;
end process;
end behavioral;
```

Figura 2.5. (continuación) Parte operatoria de la arquitectura del sistema registros de corrimiento en cascada

ACTIVIDAD COMPLEMENTARIA:

Diseñar un sistema que realice la venta de bebidas de 4 diferentes sabores, cada bebida vale \$15, se aceptan billetes de \$100, \$50, \$20 y monedas de \$1, \$2 \$5 y \$10 y da cambio. Cuando el sistema esté encendido y nadie esté comprando se activará una grabación invitando a consumir esas bebidas.

Práctica 3.

DISEÑO DEL CONTROL DE UN TREN ELÉCTRICO

OBJETIVO:

Demostrar a los estudiantes la forma de declarar tipos de datos diferentes a los definidos en el lenguaje VHDL mediante el diseño del sistema de control de un tren eléctrico

ESPECIFICACIONES:

Diseñar un sistema digital que moverá un tren de derecha a izquierda y viceversa, sobre una línea, deteniéndose en cada estación por 2 minutos. En cada una hay sensores que detectan cuando un tren entra a la estación. Existe un botón de emergencia en los vagones que hará que el tren se detenga por un minuto de más en la estación, si así se requiere.

En la figura 3.1 se muestra el diagrama de bloques del sistema Tren Eléctrico.

DIAGRAMA DE BLOQUES:

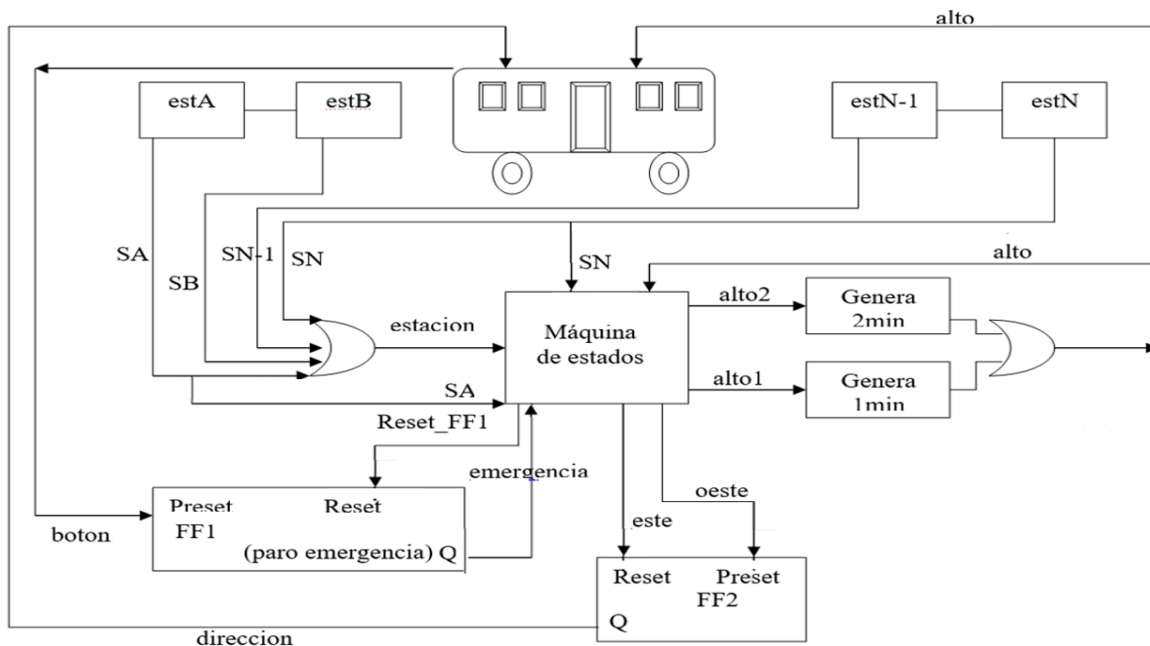


Figura 3.1. Diagrama de bloques del sistema tren eléctrico^[3]

La figura 3.2 muestra los bloques funcionales dentro del FPGA del sistema tren eléctrico y en la figura 3.3 se muestra su carta ASM.

BLOQUES FUNCIONALES:

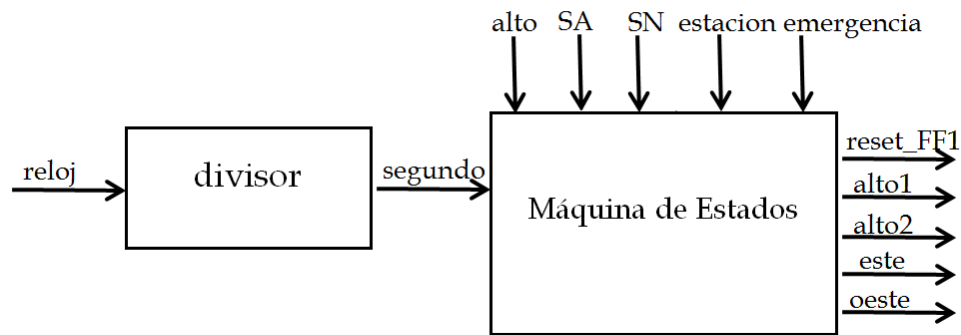


Figura 3.2. Bloques funcionales del sistema tren eléctrico ^[3]

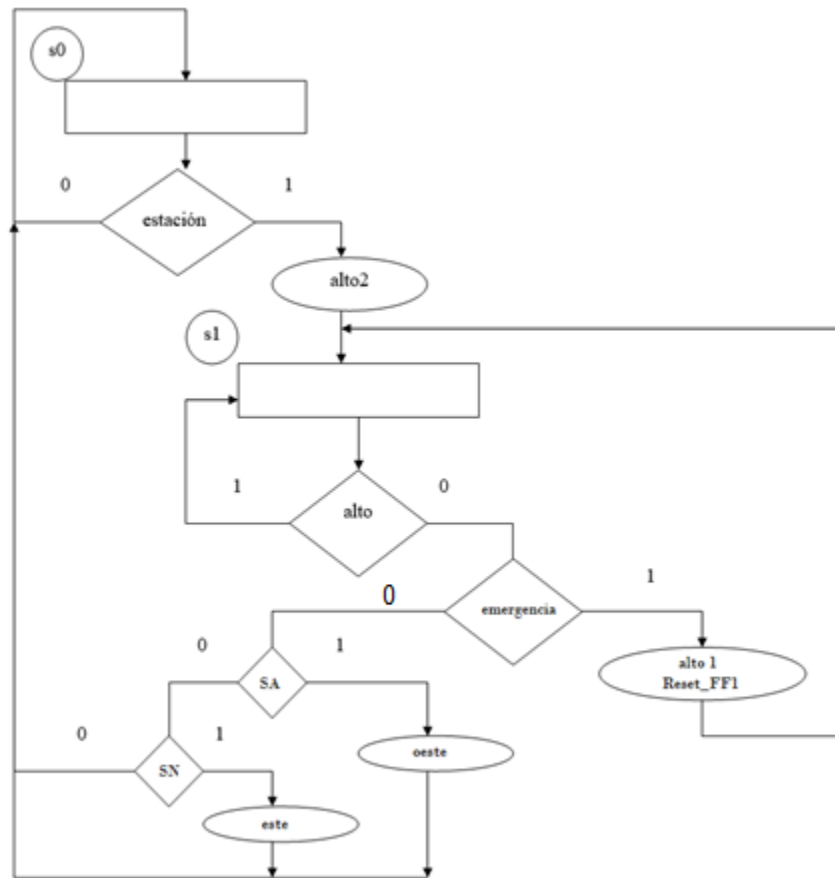


Figura 3.3. Carta ASM del sistema tren eléctrico ^[3]

La figura 3.4 muestra la entidad del sistema tren eléctrico.

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

entity tren is
    Port (reloj, alto, SA, SN, emergencia, estacion : in std_logic;
          alto1, alto2, este, oeste, reset_FF1: out std_logic);
end tren;
```

Figura 3.4. Entidad del sistema tren eléctrico

La figura 3.5 muestra la parte declaratoria de la arquitectura en el sistema tren eléctrico.

```

architecture Behavioral of tren is
    signal segundo: std_logic;
    type estados is (s0, s1);
    signal epresente, esiguiente: estados;

```

Figura 3.5. Parte declaratoria de la arquitectura del sistema tren eléctrico

La figura 3.6 muestra la parte operatoria en la arquitectura en el sistema tren eléctrico.

```

begin
    process (reloj)
    begin
        if rising_edge(segundo) then
            epresente <= esiguiente;
        end if;
    end process;

    process (epresente, estacion, alto, emergencia, SA, SN)
    begin
        case epresente is
            when s0 =>
                if estacion = '1' then
                    alto2 <= '1';
                    esiguiente <= s1;

```

Figura 3.6. Parte operatoria de la arquitectura del sistema tren eléctrico

```

        else
            esiguiente <=s0;
        end if;
    when s1 =>
        if alto = '1' then
            esiguiente <=s1;
        elsif emergencia = '1' then
            reset_FF1 <= '1';
            altol <= '1';
            esiguiente <=s1;
        elsif SA = '1' then
            oeste <= '1';
            esiguiente <=s0;
        elsif SN = '1' then
            este <= '1';
            esiguiente <=s0;
        elsif esiguiente <=s0;
        end if;
    end process;
end Behavioral;

```

Figura 3.6. (continuación) Parte operatoria de la arquitectura del sistema tren eléctrico

ACTIVIDAD COMPLEMENTARIA:

El alumno diseñará un sistema que controle la apertura y cierre de un puente en el cruce de barcos que van de norte a sur y viceversa, y de autos que van de este a oeste y viceversa. Los barcos tienen preferencia, por lo que se requiere que el sistema manipule sensores con el fin de que cuando se detecte un barco, se envíe una señal a unos semáforos que pasen de la luz verde, a la amarilla y luego a la roja. Cuando el barco ya no se encuentre cerca del puente, la luz roja se apagará y se encenderá la verde.

Al mismo tiempo que se active el detector de barcos, se activará una señal sonora para que los conductores distraídos pongan atención al cambio de luces en los semáforos y se empiece a abrir el puente dando paso a los barcos.

Práctica 4.

DISEÑO DEL CONTROL DE SERVOMOTORES

OBJETIVO:

El alumno aprenderá la manera de organizar un proyecto de manera modular y separarlo en diferentes archivos, con la finalidad de que vaya construyendo su propia biblioteca de módulos funcionales, y pueda reutilizar los módulos generados en otros proyectos.

ESPECIFICACIONES:

Diseñar el control de un servomotor de modelismo utilizando en un FPGA, en el cual, por medio de cuatro interruptores de presión tipo *push-boton*, se pueda controlar la posición del eje del motor. Dos de los interruptores permitirán llevar al eje a cada una de las posiciones extremas, mientras que los otros permitirán que el motor gire en cada dirección avanzando paso a paso a través de 12 posiciones definidas cada vez que el interruptor es presionado.

La determinación de la posición se hará por medio de una señal PWM. La figura 4.1 muestra el diagrama del bloque de este sistema.

DIAGRAMA DE BLOQUES:

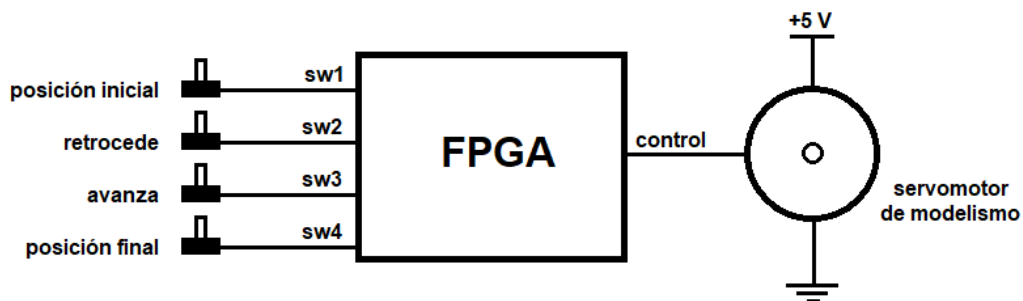


Figura 4.1. Diagrama de bloques del control de un servomotor de modelismo

En la elaboración de un proyecto basado en un FPGA, comúnmente se desarrollan gran cantidad de módulos funcionales para manejar las tareas necesarias en esa aplicación. Una buena práctica de diseño es la de utilizar cada uno de esos módulos de manera

independiente, ya que esto simplifica el proceso de diseño y permite distribuir las diferentes tareas entre varios grupos de trabajo. Además, si se hace una buena división de tareas, al final se contará con un conjunto de módulos funcionales que eventualmente podrán ser reutilizados en otros proyectos. De esta manera, al aplicar esta metodología de diseño, el alumno podrá ir construyendo su propia biblioteca de módulos funcionales, lo que en el futuro le permitirá reducir los tiempos de diseño al reutilizar estos módulos. Esto implica que cada módulo funcional debe estar contenido en un archivo diferente.

Para el desarrollo de esta práctica se aplicará este concepto de división en módulos funcionales, cada uno de ellos contenidos en un archivo diferente, que posteriormente son integrados en un solo proyecto al ser instanciados en el módulo principal. La figura 4.2 muestra los bloques funcionales que componen al control de servomotor.

BLOQUES FUNCIONALES:

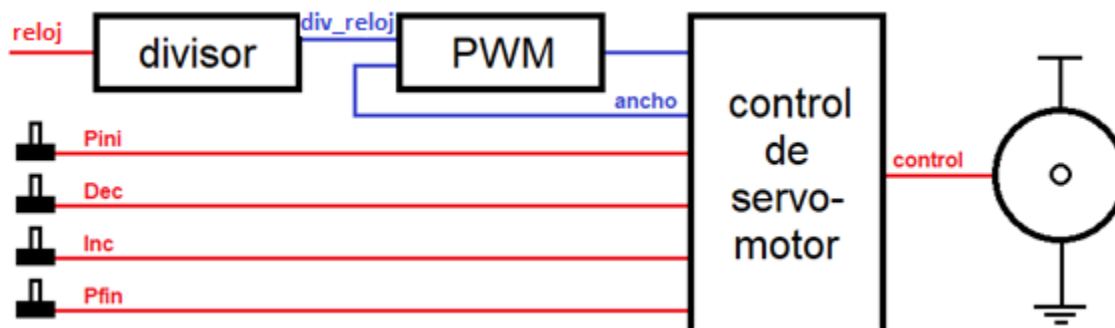


Figura 4.2. Bloques funcionales del control de servomotor

Para la elaboración de este proyecto, se diseñarán dos módulos funcionales de aplicación genérica, el módulo Divisor y el módulo PWM, que podrán ser los dos primeros módulos funcionales de la biblioteca del alumno, además del módulo principal dedicado a la aplicación específica del control del servomotor controlado por cuatro interruptores, en donde se instanciarán los dos módulos de uso general.

El primer módulo para diseñar es el correspondiente al divisor, el cual generará, a partir de la señal de reloj de 50 MHz de la tarjeta de desarrollo, una señal de salida cuya frecuencia corresponde a dividir la señal de entrada, entre una potencia de 2. La frecuencia de salida estará definida por el valor de la constante N. En la figura 4.3 muestra el código para este módulo, el cual estará contenido en el archivo divisor.

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

entity divisor is
    Port (reloj: in std_logic;
          div_reloj: out std_logic);
end divisor;

architecture behavioral of divisor is
begin
    process (reloj)
        constant N: integer:= 11;
        variable cuenta: std_logic_vector (27 downto 0):= X"00000000";
    begin
        if rising_edge (reloj) then
            cuenta:= cuenta + 1;
        end if;
        div_reloj <= cuenta (N);
    end process;
end Behavioral;

-- Periodo de la señal de salida en funcion del valor N para reloj=50 MHz:
-- 27 ~ 5.37s,    26 ~ 2.68s,    25 ~ 1.34s,    24 ~ 671ms,    23 ~ 336 ms
-- 22 ~ 168 ms,  21 ~ 83.9 ms,  20 ~ 41.9 ms,  19 ~ 21 ms,    18 ~ 10.5 ms
-- 17 ~ 5.24 ms, 16 ~ 2.62 ms,  15 ~ 1.31 ms,  14 ~ 655 us,   13 ~ 328 us
-- 12 ~ 164 us,  11 ~ 81.9 us,  10 ~ 41 us,    9 ~ 20.5 us,   8 ~ 10.2 us
```

Figura 4.3. Código para el módulo divisor

El siguiente módulo es el que se encargará de generar una señal PWM. El ciclo de trabajo de la señal generada estará definido por el valor D, el cual tiene una resolución de 256 niveles, con una frecuencia correspondiente a 256 ciclos de su reloj de entrada. La figura 4.4 muestra el código para el módulo PWM, el cual estará contenido en el archivo PWM.

```

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

entity PWM is
    Port ( reloj_pwm : in  STD_LOGIC;
          D : in  STD_LOGIC_VECTOR (7 downto 0);
          S : out  STD_LOGIC);
end PWM;

architecture Behavioral of PWM is
begin
    process (reloj_pwm)
        variable cuenta : integer range 0 to 255 := 0;
    begin
        if reloj_pwm = '1' and reloj_pwm 'event then
            cuenta := (cuenta + 1) mod 256;
            if cuenta < D then
                S <= '1';
            else
                S <= '0';
            end if;
        end if;
    end process;
end behavioral;

```

Figura 4.4. Código para el módulo PWM

Los dos módulos anteriores formarán parte de la biblioteca de módulos funcionales del alumno, los cuales pueden ser utilizados en cualquier otro proyecto en donde se requiera hacer una división de frecuencia o donde se requiera una señal PWM.

Finalmente, el módulo principal de esta aplicación se encargará de detectar la actividad en los interruptores y a partir de ello definir el ciclo de trabajo de la señal PWM. Hay que recordar que en un servomotor de modelismo típico se requiere que la señal de control tenga un período de 20 ms, y que el ancho del pulso varíe en el rango de 1 a 2 ms, en donde el ancho del pulso determina la posición del eje del servomotor; este módulo debe asegurar que esto se cumpla. Por ello se eligió el bit 11 en el divisor de frecuencia, para tener en 256 ciclos aproximadamente los 20 ms. El ancho del pulso de salida variará en el rango de 13 a 24 ciclos para tener el rango de 1 a 2 ms. Con esto el servomotor tendrá 12 posiciones que

podrá adoptar en su recorrido. La figura 4.5 muestra el código para el módulo Servomotor, el cual estará contenido en el archivo servomotor.

```

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

entity servomotor is
    Port ( reloj_sv : in  STD_LOGIC;
          Pini : in  STD_LOGIC;
          Pfin : in  STD_LOGIC;
          Inc : in  STD_LOGIC;
          Dec : in  STD_LOGIC;
          control : out  STD_LOGIC);
end Servomotor;

architecture Behavioral of Servomotor is
    component divisor is
        Port ( reloj : in std_logic;
              div_reloj : out std_logic);
    end component;
    component PWM is
        Port ( reloj_pwm : in  STD_LOGIC;
              D : in  STD_LOGIC_VECTOR (7 downto 0);
              S : out  STD_LOGIC);
    end component;
    signal reloj_serv : STD_LOGIC;
    signal ancho : STD_LOGIC_VECTOR (7 downto 0) := X"0F";
begin
    U1: divisor port map (reloj_sv, reloj_serv);
    U2: PWM port map (reloj_serv, ancho, control);

    process (reloj_serv, Pini, Pfin, Inc, Dec)
        variable valor : STD_LOGIC_VECTOR (7 downto 0) := X"0F";
        variable cuenta : integer range 0 to 1023 := 0;
    begin
        if reloj_serv='1' and reloj_serv'event then
            if cuenta>0 then
                cuenta := cuenta -1;
            else
                if Pini='1' then
                    valor := X"0D";
                elsif Pfin='1' then
                    valor := X"18";
                elsif Inc='1' and valor<X"18" then
                    valor := valor + 1;
                elsif Dec='1' and valor>X"0D" then
                    valor := valor - 1;
                end if;
                cuenta := 1023;
                ancho <= valor;
            end if;
        end if;
    end process;
end Behavioral;

```

Figura 4.5. Código para el módulo servomotor.

ACTIVIDADES COMPLEMENTARIAS:

1. Siguiendo la metodología de diseño presentada, el alumno elaborará un módulo funcional genérico para controlar un servomotor de modelismo, que complementará la biblioteca de módulos del alumno.
2. Utilizando el módulo genérico para controlar un servomotor diseñado en el punto anterior, construir un sistema que haga que dos servomotores de modelismo se muevan de forma complementaria, es decir, se moverán de la misma forma, pero girando en la dirección opuesta.
3. Utilizando el módulo genérico para controlar un servomotor diseñado en el primer punto, construir un sistema que haga que dos servomotores de modelismo se muevan independientemente, cada uno de ellos controlado por dos interruptores de presión tipo *push-boton*, que al presionarlos harán avanzar o retroceder al eje del motor.

Práctica 5.

DISEÑO DEL CONTROL DE INTENSIDAD EN LEDS

OBJETIVO:

El alumno aprenderá a diseñar módulos con parámetros genéricos, lo que permitirá crear un proyecto con varias instancias de un mismo elemento pero con diferentes características de operación, con el fin de dar una mayor versatilidad a los módulos diseñados por el alumno.

ESPECIFICACIONES:

Diseñar un circuito utilizando un FPGA que se encargue de controlar el encendido de cuatro LEDs, cada uno de los cuales encenderá con diferente intensidad. La intensidad de cada LED será controlada por medio del ciclo de trabajo de una señal PWM. Las luces en los LEDs irán cambiando siguiendo una secuencia determinada. La figura 5.1 muestra el diagrama del bloque de este sistema.

DIAGRAMA DE BLOQUES:

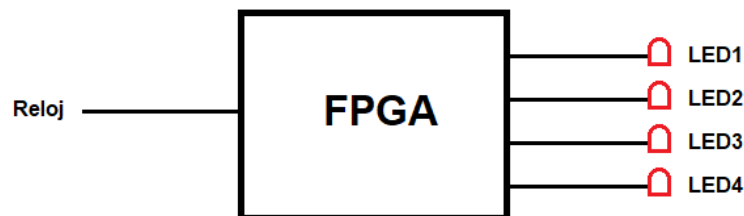


Figura 5.1. Diagrama de bloques del control de intensidad de encendido de LEDs

Al realizar un diseño utilizando un FPGA es común que se requiera tener funcionando concurrentemente varias copias de un mismo objeto, y en muchas ocasiones cada copia del objeto deberá tener características de operación diferente. Por ejemplo, en un mismo diseño se puede requerir utilizar varios registros similares, pero de diferente tamaño. No sería práctico tener en la biblioteca una versión del mismo registro para cada tamaño posible. Lo conveniente en este caso, es tener una sola definición del registro en el que se pueda definir

el tamaño del mismo cuando sea creada una instancia de él. Esto se puede lograr con el uso de parámetros genéricos.

En esta práctica se utilizarán los bloques funcionales diseñados en la práctica anterior, creando varias instancias de cada uno, pero se modificará uno de estos módulos para que utilice un parámetro genérico. En la figura 5.2 muestran los bloques funcionales que integran este diseño.

BLOQUES FUNCIONALES:

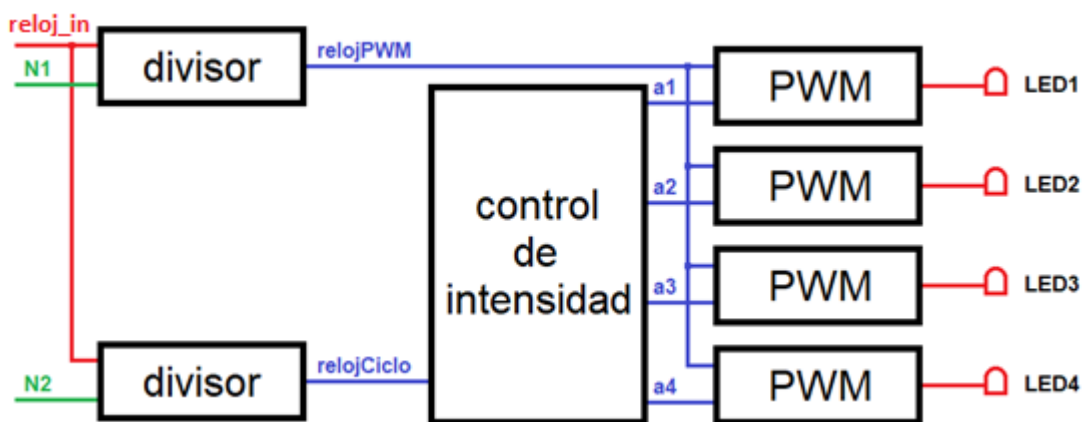


Figura 5.2. Bloques funcionales del control de intensidad de encendido de LEDs

Como se observa en el diagrama, se utilizarán cuatro instancias del módulo PWM y dos del módulo Divisor. Hay que notar que se requiere utilizar dos divisores de frecuencia, ya que se tienen dos procesos que utilizan relojes con frecuencia diferente, uno de frecuencia alta para generar la señal PWM que se utilizará para encender los LEDs, y otro de frecuencia menor que señalará los tiempos en que cambia el estado de la secuencia que se observará en cada LED. La figura 5.3 muestra el código para el módulo divisor, que estará contenido en el archivo divisor, en el que ha cambiado la definición del valor N, siendo ahora un parámetro genérico en lugar de una constante.

```

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

entity divisor is
  Generic ( N : integer := 24);
  Port      ( reloj : in std_logic;
              div_reloj : out std_logic);
end Divisor;

architecture Behavioral of Divisor is
begin
  process (reloj)
    variable cuenta: std_logic_vector (27 downto 0) := X"00000000";
  begin
    if rising_edge (reloj) then
      cuenta := cuenta + 1;
    end if;
    div_reloj <= cuenta (N);
  end process;
end Behavioral;
-- Periodo de la señal de salida en funcion del valor N para reloj=50 MHz:
-- 27 ~ 5.37s,   26 ~ 2.68s,   25 ~ 1.34s,   24 ~ 671ms,   23 ~ 336 ms
-- 22 ~ 168 ms,  21 ~ 83.9 ms, 20 ~ 41.9 ms, 19 ~ 21 ms,   18 ~ 10.5 ms
-- 17 ~ 5.24 ms, 16 ~ 2.62 ms, 15 ~ 1.31 ms, 14 ~ 655 us,  13 ~ 328 us
-- 12 ~ 164 us,  11 ~ 81.9 us, 10 ~ 41 us,   9 ~ 20.5 us,  8 ~ 10.2 us

```

Figura 5.3. Código para el módulo divisor

Para el caso del módulo PWM no es necesario realizar modificación alguna al código de la práctica anterior, por lo que aquí se reutilizará directamente el módulo previamente construido y que forma parte de la biblioteca de módulos del alumno.

Ahora sólo falta construir el módulo principal, el cual se encargará de generar la secuencia que se observará en los LEDs. La figura 5.4 muestra el código para el módulo Leds, el cual irá contenido en el archivo leds. Para probar el funcionamiento de esta práctica se utilizarán los LEDs de la tarjeta de desarrollo.

Es importante notar en el código que para cambiar la asignación de intensidades no es necesario utilizar una variable auxiliar para evitar la pérdida de los valores, ya que aquí se está describiendo hardware.

```

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

entity leds is
    Port ( reloj_in : in  STD_LOGIC;
          led1 : out  STD_LOGIC;
          led2 : out  STD_LOGIC;
          led3 : out  STD_LOGIC;
          led4 : out  STD_LOGIC);
end Leds;

architecture Behavioral of Leds is
    component divisor is
        Generic ( N : integer := 24);
        Port ( relo : in std_logic;
              div_reloj : out std_logic);
    end component;
    component PWM is
        Port ( reloj_pwm : in  STD_LOGIC;
              D : in  STD_LOGIC_VECTOR (7 downto 0);
              S : out  STD_LOGIC);
    end component;
    signal relojPWM : STD_LOGIC;
    signal relojCiclo : STD_LOGIC;
    signal a1 : STD_LOGIC_VECTOR (7 downto 0) := X"08";
    signal a2 : STD_LOGIC_VECTOR (7 downto 0) := X"20";
    signal a3 : STD_LOGIC_VECTOR (7 downto 0) := X"60";
    signal a4 : STD_LOGIC_VECTOR (7 downto 0) := X"F8";
begin
    N1: divisor generic map (10) port map (reloj, relojPWM);
    N2: divisor generic map (23) port map (reloj, relojCiclo);
    P1: PWM port map (relojPWM, a1, led1);
    P2: PWM port map (relojPWM, a2, led2);
    P3: PWM port map (relojPWM, a3, led3);
    P4: PWM port map (relojPWM, a4, led4);

    process (relojCiclo)
        variable Cuenta : integer range 0 to 255 := 0;
    begin
        if relojCiclo='1' and relojCiclo'event then
            a1 <= a4;
            a2 <= a1;
            a3 <= a2;
            a4 <= a3;
        end if;
    end process;
end Behavioral;

```

Figura 5.4. Código para el módulo leds

ACTIVIDADES COMPLEMENTARIAS:

1. Hacer que la misma secuencia de 4 LEDs encendidos usada en esta práctica, ahora recorra los 8 LEDs de la tarjeta de desarrollo, y al llegar al final vaya de regreso, siempre con el LED de mayor intensidad al inicio y el de menor intensidad al final.
2. Utilizando un LED RGB, controlar la intensidad de encendido de cada color de tal manera que la luz resultante vaya mostrando los colores del arcoíris.

Práctica 6.

DISEÑO DEL CONTROL DE MOTORES A PASOS

OBJETIVO:

El alumno aprenderá a diseñar el controlador de un motor a pasos mediante el uso e implantación de máquinas de estado.

ESPECIFICACIONES:

Diseñar el circuito de control utilizando un FPGA, el cual se encargue de activar un motor a pasos bipolar con 4 líneas de control. Los movimientos que debe realizar el motor son en sentido a las manecillas del reloj, viceversa y detenido por medio de tres botones que controlan estos movimientos.

La figura 6.1 muestra el diagrama a bloques del sistema.

DIAGRAMA DE BLOQUES:

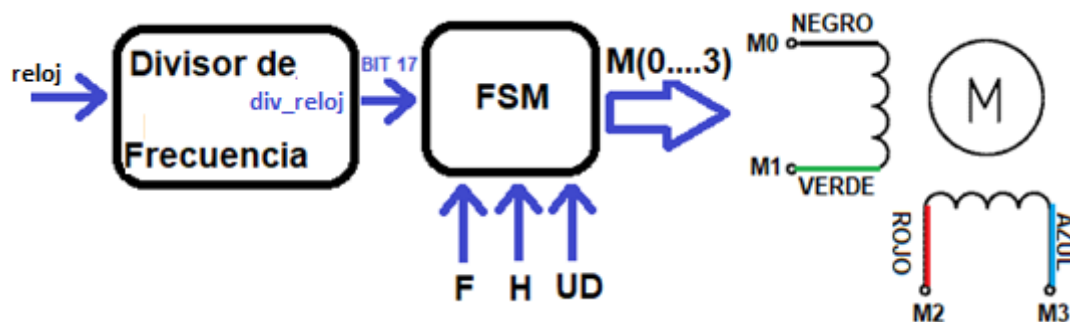


Figura 6.1. Diagrama a bloques del control de motor a pasos

TABLA DE ESTADOS:

La figura 6.2, muestra la tabla de estados, la cual está diseñada con 11 estados que inician en el estado SM0 hasta el estado SM10. Por la cantidad de condiciones de entrada y estados está expresada por colores para cada estado, para una mejor comprensión. En la figura 6.2.A se observa el Estado 0, Estado 1, Estado 2 y Estado 3. En la figura 6.2.B se observa

el Estado 4, Estado 5, Estado 6 y Estado 7. En la figura 6.2.C se observa el Estado 8, Estado 9 y Estado 10.

Estado Presente		Condiciones			Estado Siguiete
Estado 0	SM0	F	H	UD	
		0	0	0	SM0
		0	0	1	SM0
		0	1	0	SM0
	Salidas	0	1	1	SM0
		1	0	0	SM0
	M3 M2 M1 M0	1	0	1	SM0
	0 0 0 0	1	1	0	SM0
		1	1	1	SM0
Estado 1	SM1	F	H	UD	Estado Siguiete
		0	0	0	SM7
		0	0	1	SM3
		0	1	0	SM8
	Salidas	0	1	1	SM2
		1	0	0	SM8
	M3 M2 M1 M0	1	0	1	SM2
	1 0 0 0	1	1	0	SM4
		1	1	1	SM1
Estado 2	SM2	F	H	UD	Estado Siguiete
		0	0	0	SM7
		0	0	1	SM1
		0	1	0	SM8
	Salidas	0	1	1	SM4
		1	0	0	SM1
	M3 M2 M1 M0	1	0	1	SM3
	1 0 0 0	1	1	0	SM4
		1	1	1	SM9
Estado 3	SM3	F	H	UD	Estado Siguiete
		0	0	0	SM1
		0	0	1	SM5
		0	1	0	SM8
	Salidas	0	1	1	SM2
		1	0	0	SM2
	M3 M2 M1 M0	1	0	1	SM4
	0 1 0 0	1	1	0	SM4
		1	1	1	SM9

Figura 6.2.A. Estado 0, Estado 1, Estado 2 y Estado 3

Estado Presente		Condiciones			
Estado 4	SM4	F	H	UD	Estado Sigiente
		0	0	0	SM7
		0	0	1	SM1
	Salidas	0	1	0	SM2
		0	1	1	SM6
		1	0	0	SM3
	M3 M2 M1 M0	1	0	1	SM5
	0 1 1 0	1	1	0	SM10
		1	1	1	SM9
Estado 5	SM5	F	H	UD	Estado Sigiente
		0	0	0	SM3
		0	0	1	SM7
	Salidas	0	1	0	SM8
		0	1	1	SM2
		1	0	0	SM6
	M3 M2 M1 M0	1	0	1	SM4
	0 0 1 0	1	1	0	SM4
		1	1	1	SM9
Estado 6	SM6	F	H	UD	Estado Sigiente
		0	0	0	SM7
		0	0	1	SM1
	Salidas	0	1	0	SM4
		0	1	1	SM8
		1	0	0	SM5
	M3 M2 M1 M0	1	0	1	SM7
	0 0 1 1	1	1	0	SM4
		1	1	1	SM9
Estado 7	SM7	F	H	UD	Estado Sigiente
		0	0	0	SM5
		0	0	1	SM1
	Salidas	0	1	0	SM8
		0	1	1	SM2
		1	0	0	SM6
	M3 M2 M1 M0	1	0	1	SM8
	0 0 0 1	1	1	0	SM4
		1	1	1	SM9

Figura 6.2.B. Estado 4, Estado 5, Estado 6 y Estado 7

Estado Presente		Condiciones			Estado Sigiente
Estado 8	SM8	F	H	UD	Estado Sigiente
		0	0	0	SM7
		0	0	1	SM1
		0	1	0	SM6
	Salidas	0	1	1	SM2
		1	0	0	SM7
	M3 M2 M1 M0	1	0	1	SM1
	1 0 0 1	1	1	0	SM9
		1	1	1	SM10
Estado 9	SM9	F	H	UD	Estado Sigiente
		0	0	0	SM7
		0	0	1	SM1
		0	1	0	SM8
	Salidas	0	1	1	SM2
		1	0	0	SM8
	M3 M2 M1 M0	1	0	1	SM1
	1 0 1 0	1	1	0	SM4
		1	1	1	SM8
Estado 10	SM10	F	H	UD	Estado Sigiente
		0	0	0	SM7
		0	0	1	SM1
		0	1	0	SM8
	Salidas	0	1	1	SM2
		1	0	0	SM8
	M3 M2 M1 M0	1	0	1	SM1
	0 1 0 1	1	1	0	SM8
		1	1	1	SM4

Figura 6.2.C. Estado 8, Estado 9 y Estado 10

Las siguientes figuras muestran el código del control para el motor a pasos, el cual estará contenido en el archivo llamado MotPasos. En la figura 6.3 se observa el código de la entidad y las señales dentro de la arquitectura.

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity MotPasos is
    port ( reloj  : in STD_LOGIC;
          UD      : in STD_LOGIC;
          rst      : in STD_LOGIC;
          FH       : in STD_LOGIC_VECTOR(1 downto 0);
          led       : out STD_LOGIC_VECTOR(3 downto 0);
          MOT       : out STD_LOGIC_VECTOR(3 downto 0) );
end MotPasos;

architecture behavioral of MotPasos is
    signal div : std_logic_vector(17 downto 0);
    signal clks : std_logic;
    type estado is (sm0, sm1, sm2, sm3, sm4, sm5, sm6, sm7, sm8,
                   sm9, sm10);
    signal pres_S, next_s : estado;
    signal motor : std_logic_vector(3 downto 0);
begin

```

Figura 6.3. Código para la entidad y parte de la arquitectura de MotPasos

En la Figura 6.4 se observa el código del bloque Divisor de Frecuencia.

```

process (reloj, rst)
begin
    if rst='0' then
        div <= (others=>'0');
    elsif reloj'event and reloj='1' then
        div <= div + 1;
    end if;
end process;
clks <= div(17);

```

Figura 6.4. Código del bloque Divisor de Frecuencia

En la Figura 6.5 se observa el código de las transiciones de estados.

```

process (clks,rst)
begin
    if rst='0' then
        pres_S <= sm0;
    elsif clks'event and clks='1' then
        pres_S <= next_s;
    end if;
end process;

process (pres_S,UD,rst,FH)
begin
    case(pres_S) is
        when sm0 =>                                -- Estado 0
            next_s <= sm1;
        when sm1 =>                                -- Estado 1
            if FH="00" then ---motor bipolar
                if UD='1' then
                    next_s <= sm3;
                else
                    next_s <= sm7;
                end if;
            elsif FH="01" then
                if UD='1' then
                    next_s <= sm2;
                else
                    next_s <= sm8;
                end if;
            elsif FH="10" then
                if UD='1' then
                    next_s <= sm2;
                else
                    next_s <= sm8;
                end if;
            elsif FH="11" then
                if UD='1' then
                    next_s <= sm9;
                else
                    next_s <= sm4;
                end if;
            else
                next_s <= sm1;
            end if;
        when sm2 =>                                -- Estado 2
            if FH="00" then
                if UD='1' then
                    next_s <= sm1;
                else
                    next_s <= sm7;
                end if;
            end if;
    end case;
end process;

```

Figura 6.5. Transiciones de estados


```

    elsif FH="01" then
        if UD='1' then
            next_s <= sm4;
        else
            next_s <= sm8;
        end if;
    elsif FH="10" then
        if UD='1' then
            next_s <= sm3;
        else
            next_s <= sm1;
        end if;
    elsif FH="11" then
        if UD='1' then
            next_s <= sm9;
        else
            next_s <= sm4;
        end if;
    else
        next_s <= sm2;
    end if;
when sm3 =>                                -- Estado 3
    if FH="00" then
        if UD='1' then
            next_s <= sm5;
        else
            next_s <= sm1;
        end if;
    elsif FH="01" then
        if UD='1' then
            next_s <= sm2;
        else
            next_s <= sm8;
        end if;
    elsif FH="10" then
        if UD='1' then
            next_s <= sm4;
        else
            next_s <= sm2;
        end if;
    elsif FH="11" then
        if UD='1' then
            next_s <= sm9;
        else
            next_s <= sm4;
        end if;
    else
        next_s <= sm3;
    end if;

```

Figura 6.5. (continuación) Transiciones de estados

```

when sm4 =>                                -- Estado 4
    if FH="00" then
        if UD='1' then
            next_s <= sm1;
        else
            next_s <= sm7;
        end if;
    elsif FH="01" then
        if UD='1' then
            next_s <= sm6;
        else
            next_s <= sm2;
        end if;
    elsif FH="10" then
        if UD='1' then
            next_s <= sm5;
        else
            next_s <= sm3;
        end if;
    elsif FH="11" then
        if UD='1' then
            next_s <= sm9;
        else
            next_s <= sm10;
        end if;
    else
        next_s <= sm4;
    end if;
when sm5 =>                                -- Estado 5
    if FH="00" then
        if UD='1' then
            next_s <= sm7;
        else
            next_s <= sm3;
        end if;
    elsif FH="01" then
        if UD='1' then
            next_s <= sm2;
        else
            next_s <= sm8;
        end if;
    elsif FH="10" then
        if UD='1' then
            next_s <= sm6;
        else
            next_s <= sm4;
        end if;
    elsif FH="11" then
        if UD='1' then
            next_s <= sm9;
        else
            next_s <= sm4;
        end if;

```

Figura 6.5. (continuación) Transiciones de estados

```

    else
        next_s <= sm3;
    end if;
when sm6 =>                                -- Estado 6
    if FH="00" then
        if UD='1' then
            next_s <= sm1;
        else
            next_s <= sm7;
        end if;
    elsif FH="01" then
        if UD='1' then
            next_s <= sm8;
        else
            next_s <= sm4;
        end if;
    elsif FH="10" then
        if UD='1' then
            next_s <= sm7;
        else
            next_s <= sm5;
        end if;
    elsif FH="11" then
        if UD='1' then
            next_s <= sm9;
        else
            next_s <= sm4;
        end if;
    else
        next_s <= sm7;
    end if;
when sm7 =>                                -- Estado 7
    if FH="00" then
        if UD='1' then
            next_s <= sm1;
        else
            next_s <= sm5;
        end if;
    elsif FH="01" then
        if UD='1' then
            next_s <= sm2;
        else
            next_s <= sm8;
        end if;
    elsif FH="10" then
        if UD='1' then
            next_s <= sm8;
        else
            next_s <= sm6;
        end if;

```

Figura 6.5. (continuación) Transiciones de estados

```

        elsif FH="11" then
            if UD='1' then
                next_s <= sm9;
            else
                next_s <= sm4;
            end if;
        else
            next_s <= sm7;
        end if;
when sm8 =>                                -- Estado 8
    if FH="00" then
        if UD='1' then
            next_s <= sm1;
        else
            next_s <= sm7;
        end if;
    elsif FH="01" then
        if UD='1' then
            next_s <= sm2;
        else
            next_s <= sm6;
        end if;
    elsif FH="10" then
        if UD='1' then
            next_s <= sm1;
        else
            next_s <= sm7;
        end if;
    elsif FH="11" then
        if UD='1' then
            next_s <= sm10;
        else
            next_s <= sm9;
        end if;
    else
        next_s <= sm8;
    end if;
when sm9 =>                                -- Estado 9
    if FH="00" then
        if UD='1' then
            next_s <= sm1;
        else
            next_s <= sm7;
        end if;
    elsif FH="01" then
        if UD='1' then
            next_s <= sm2;
        else
            next_s <= sm8;
        end if;

```

Figura 6.5. (continuación) Transiciones de estados

```

        elsif FH="10" then
            if UD='1' then
                next_s <= sm1;
            else
                next_s <= sm8;
            end if;
        elsif FH="11" then
            if UD='1' then
                next_s <= sm8;
            else
                next_s <= sm4;
            end if;
        else
            next_s <= sm9;
        end if;
    when sm10 =>                                -- Estado 10
        if FH="00" then
            if UD='1' then
                next_s <= sm1;
            else
                next_s <= sm7;
            end if;
        elsif FH="01" then
            if UD='1' then
                next_s <= sm2;
            else
                next_s <= sm8;
            end if;
        elsif FH="10" then
            if UD='1' then
                next_s <= sm1;
            else
                next_s <= sm8;
            end if;
        elsif FH="11" then
            if UD='1' then
                next_s <= sm4;
            else
                next_s <= sm8;
            end if;
        else
            next_s <= sm10;
        end if;
    when others => next_s <= sm0;
end case;
end process;

```

Figura 6.5. (continuación) Transiciones de estados

En la Figura 6.6 se observa el código de las salidas de estados al motor.

```
process (pres_S)
begin
  case pres_S is
    when sm0 => motor <= "0000";
    when sm1 => motor <= "1000";
    when sm2 => motor <= "1100";
    when sm3 => motor <= "0100";
    when sm4 => motor <= "0110";
    when sm5 => motor <= "0010";
    when sm6 => motor <= "0011";
    when sm7 => motor <= "0001";
    when sm8 => motor <= "1001";
    when sm9 => motor <= "1010";
    when sm10 => motor <= "0101";
    when others => motor <= "0000";
  end case;
end process;

MOT<=motor;
led<=motor;

end behavioral;
```

Figura 6.6. Salidas de estados

ACTIVIDAD COMPLEMENTARIA:

El alumno deberá realizar las modificaciones pertinentes para poder girar el motor las vueltas necesarias que representen los dígitos de su número de cuenta, se deben combinar los giros horario, anti horario y detenido.

Práctica 7.

DISEÑO DEL CONTROL DE SENSORES ULTRASÓNICO

OBJETIVO:

El alumno aprenderá a diseñar mediante la utilización de atributos a señales ('HIGH) y tipos de variables (UNSIGNED) el control de un sensor ultrasónico (HC-SR04).

ESPECIFICACIONES:

Diseñar un circuito controlador utilizando un FPGA que se encargue de calcular la distancia de un obstáculo por medio de un sensor ultrasónico (HC-SR04), y observar los resultados de distancia por medio de 2 displays de 7 segmentos. La figura 7.1 muestra el diagrama a bloques del sistema.

DIAGRAMA DE BLOQUES:

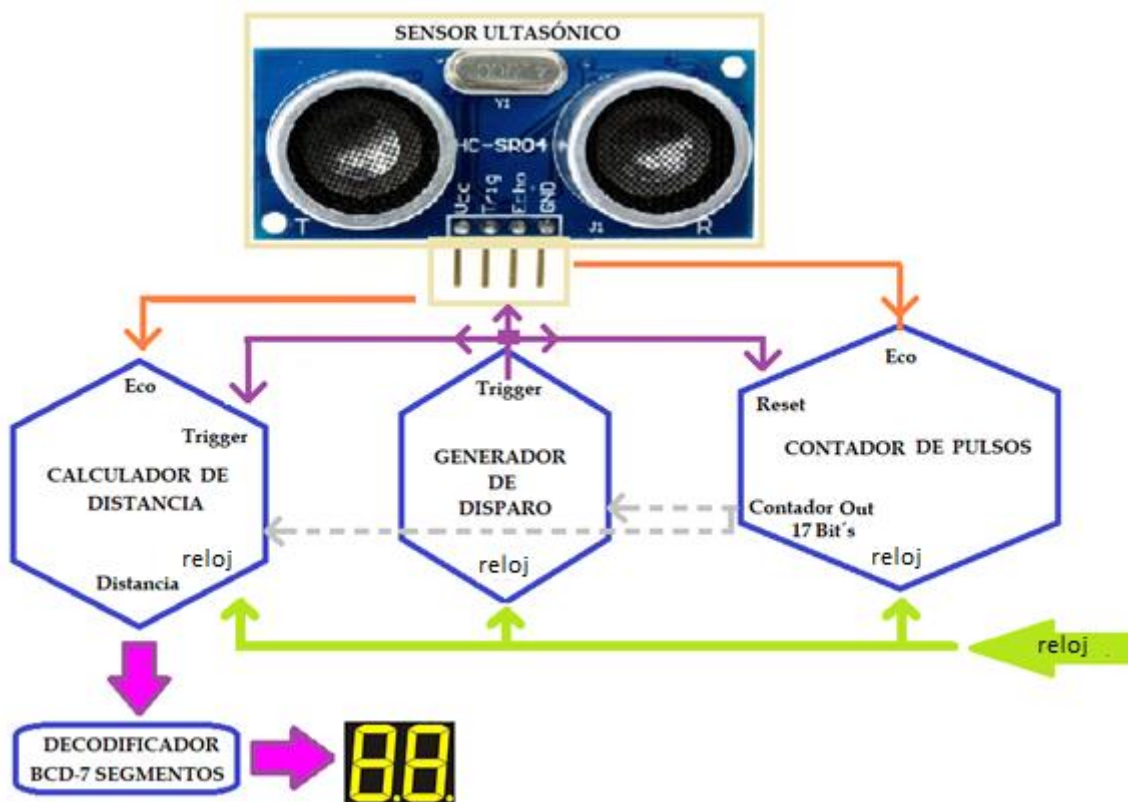


Figura 7.1. Diagrama a bloques del control para el sensor ultrasónico

Las siguientes figuras muestran el código del control para el sensor ultrasónico, que estará contenido en el archivo `sonicos`. El código fue separado, para su mejor comprensión, de acuerdo con el diagrama a bloques mostrado.

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

entity sonicos is
    Port (clk: in STD_LOGIC;
          sensor_disp: out STD_LOGIC;
          sensor_eco: in STD_LOGIC;
          anodos: out STD_LOGIC_VECTOR (3 downto 0);
          segmentos: out STD_LOGIC_VECTOR (7 downto 0));
end sonicos;

architecture Behavioral of sonicos is
    signal cuenta: unsigned(16 downto 0) := (others => '0');
    signal centimetros: unsigned(15 downto 0) := (others => '0');
    signal centimetros_unid: unsigned(3 downto 0) := (others => '0');
    signal centimetros_dece: unsigned(3 downto 0) := (others => '0');
    signal sal_unid: unsigned(3 downto 0) := (others => '0');
    signal sal_dece: unsigned(3 downto 0) := (others => '0');
    signal digito: unsigned(3 downto 0) := (others => '0');
    signal eco_pasado: std_logic := '0';
    signal eco_sinc: std_logic := '0';
    signal eco_nsinc: std_logic := '0';
    signal espera: std_logic := '0';
    signal siete_seg_cuenta: unsigned(15 downto 0) := (others => '0');
begin
    anodos(1 downto 0) <= "11";

    siete_seg: process(clk)
    begin
        if rising_edge(clk) then
            if siete_seg_cuenta(siete_seg_cuenta'high) = '1' then
                digito <= sal_unid;
                anodos(3 downto 2) <= "01";
            else
                digito <= sal_dece;
                anodos(3 downto 2) <= "10";
            end if;
            siete_seg_cuenta <= siete_seg_cuenta + 1;
        end if;
    end process;
end Behavioral;
```

Figura 7.2. Código para la entidad y arquitectura de sonicos

La Figura 7.3 se observa el código de la señal Trigger.

```
Trigger:process(clk)
begin
    if rising_edge(clk) then
        if espera = '0' then
            if cuenta = 500 then
                sensor_disp <= '0';
                espera <= '1';
                cuenta <= (others => '0');
            else
                sensor_disp <= '1';
                cuenta <= cuenta+1;
            end if;
        end if;
    end if;
end process;
```

Figura 7.3. Código del bloque generador de disparo (Trigger)

La Figura 7.4 se observa el código de los bloques calculador de distancia y contador de pulsos.

```

elseif eco_pasado = '0' and eco_sinc = '1' then
    cuenta <= (others => '0');
    centimetros <= (others => '0');
    centimetros_unid <= (others => '0');
    centimetros_dece <= (others => '0');
elseif eco_pasado = '1' and eco_sinc = '0' then
    sal_unid <= centimetros_unid;
    sal_dece <= centimetros_dece;
elseif cuenta = 2900-1 then
    if centimetros_unid = 9 then
        centimetros_unid <= (others => '0');
        centimetros_dece <= centimetros_dece + 1;
    else
        centimetros_unid <= centimetros_unid + 1;
    end if;
    centimetros <= centimetros + 1;
    cuenta <= (others => '0');
    if centimetros = 3448 then
        espera <= '0';
    end if;
else
    cuenta <= cuenta + 1;
end if;
    eco_pasado <= eco_sinc;
    eco_sinc <= eco_nsinc;
    eco_nsinc <= sensor_eco;
end if;
end process;

```

Figura 7.4 Código del bloque calculador de distancia y del bloque contador de pulsos

La figura 7.5 muestra el código para la decodificación de datos a dos displays de siete segmentos; este código está diseñado para utilizar displays de ánodo común.

```

Decodificador: process (digito)
begin
  if      digito=X"0" then segmentos <= X"81";
  elsif  digito=X"1" then segmentos <= X"F3";
  elsif  digito=X"2" then segmentos <= X"49";
  elsif  digito=X"3" then segmentos <= X"61";
  elsif  digito=X"4" then segmentos <= X"33";
  elsif  digito=X"5" then segmentos <= X"25";
  elsif  digito=X"6" then segmentos <= X"05";
  elsif  digito=X"7" then segmentos <= X"F1";
  elsif  digito=X"8" then segmentos <= X"01";
  elsif  digito=X"9" then segmentos <= X"21";
  elsif  digito=X"a" then segmentos <= X"11";
  elsif  digito=X"b" then segmentos <= X"07";
  elsif  digito=X"c" then segmentos <= X"8D";
  elsif  digito=X"d" then segmentos <= X"43";
  elsif  digito=X"e" then segmentos <= X"0D";
  else
    segmentos<= X"1D";
  end if;
end process;
end Behavioral;

```

Figura 7.5 Código del Bloque Decodificador BCD-7 Segmentos.

ACTIVIDAD COMPLEMENTARIA:

El Alumno deberá realizar las modificaciones pertinentes para poder detectar una distancia exacta propuesto por el profesor de un objeto, cuando sea detectada deberá poner la letra S (Stop) en un display de 7 segmentos, la cual indica que no puede acercarse más o chocara con el objeto.

Práctica 8.

DISEÑO DE UN TRANSMISOR PARA COMUNICACIÓN SERIAL

OBJETIVO:

Demostrar a los estudiantes mediante el diseño de un módulo transmisor (TX), empleado en comunicaciones de tipo serial UART (*Universal Asynchronous Receiver Transmitter*), la utilidad de este módulo, así como la importancia de su presencia en la arquitectura de un procesador para aplicaciones electrónicas en envío de información.

ESPECIFICACIONES:

Utilizando un FPGA y un switch de 4 posiciones, diseñar un módulo Transmisor serial, el cual sea capaz de leer el valor binario del switch, procesarlo en el FPGA y posteriormente enviarlo a una computadora personal, en donde el dato deberá estar en formato hexadecimal. La conexión entre el FPGA y la computadora deberá realizarse empleando un circuito convertidor USB TTL-Serial. La figura 8.1 muestra el diagrama de bloques del sistema.

DIAGRAMA DE BLOQUES:

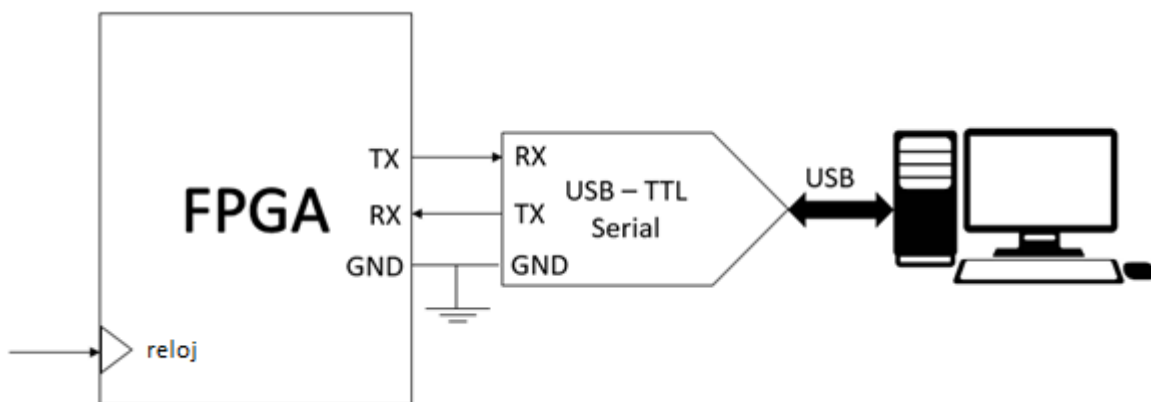


Figura 8.1. Diagrama de bloques para la comunicación serial

Un FPGA es un dispositivo lógico programable, el cual posee la característica de no contar con una arquitectura fija como en el caso de un procesador. Dicha característica trae consigo la posibilidad de diseñar arquitecturas reconfigurables en donde la cantidad de puertos o módulos periféricos puede ser establecida de acuerdo a las especificaciones de diseño. Así, el diseño de un módulo TX de comunicación UART puede ser elaborado y configurado para realizar tareas específicas consumiendo el mínimo de recursos posible. La figura 8.2 muestra los bloques funcionales del sistema Transmisor, donde las señales se muestran como flechas de color azul, mientras que las terminales físicas se muestran en color rojo. Cada bloque funcional corresponde a un proceso que deberá ejecutarse dentro de la arquitectura.

BLOQUES FUNCIONALES:

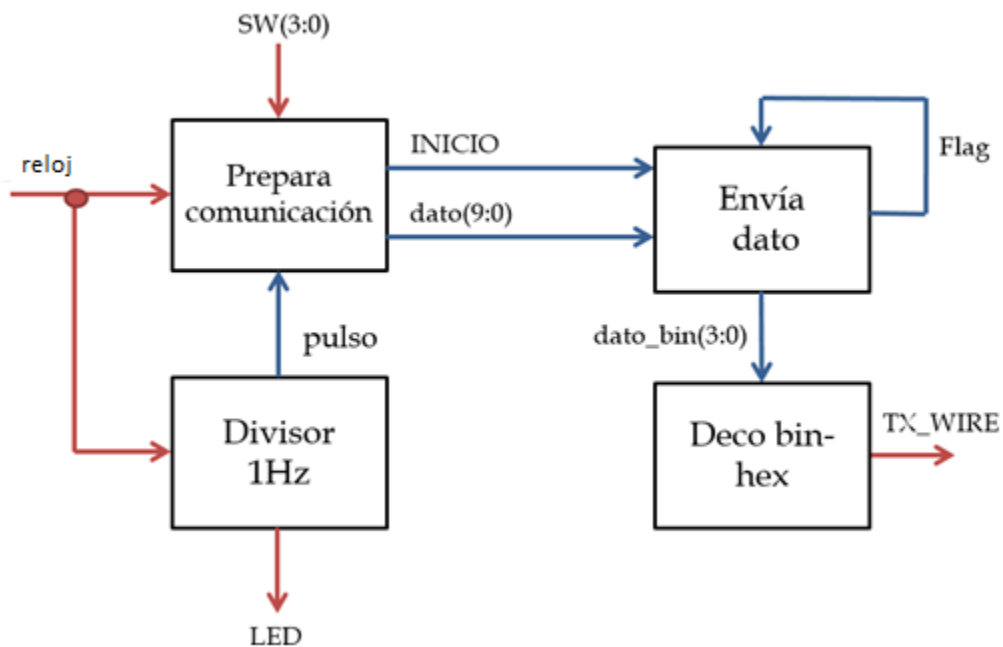


Figura 8.2. Bloques funcionales del sistema transmisor serial

La figura 8.3 muestra la parte entidad del sistema transmisor de comunicación serial. Las terminales físicas corresponden al reloj maestro del FPGA de 50 MHz, cuatro bits de un switch, un LED testigo y la línea de transmisión (TX_WIRE).

```

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

entity TX is
  port( reloj      : IN  STD_LOGIC;
        SW         : IN  STD_LOGIC_VECTOR(3 downto 0);
        LED        : OUT STD_LOGIC;
        TX_WIRE    : OUT STD_LOGIC);
end entity;

```

Figura 8.3. Entidad del sistema Transmisor Serial TX

La figura 8.4 muestra la parte declaratoria de la arquitectura del módulo **TX**. en donde se declaran todas las señales involucradas en los diferentes procesos del sistema de transmisión.

```

architecture behaivoral OF TX IS
  signal conta : INTEGER := 0;
  signal valor : INTEGER := 70000;
  signal INICIO: STD_LOGIC;
  signal dato  : STD_LOGIC_VECTOR(7 DOWNT0 0);
  signal PRE   : INTEGER RANGE 0 TO 5208 := 0;
  signal INDICE: INTEGER RANGE 0 TO 9 := 0;
  signal BUFF  : STD_LOGIC_VECTOR(9 DOWNT0 0);
  signal Flag  : STD_LOGIC := '0';
  signal PRE_val: INTEGER range 0 to 41600;
  signal baud  : STD_LOGIC_VECTOR(2 DOWNT0 0);
  signal i     : INTEGER range 0 to 4;
  signal pulso : STD_LOGIC:='0';
  signal conta2: integer range 0 to 499999999 := 0;
  signal dato_bin: STD_LOGIC_VECTOR(3 DOWNT0 0);
  signal hex_val: STD_LOGIC_VECTOR(7 DOWNT0 0):= (others => '0');

```

Figura 8.4. Parte declaratoria en la arquitectura del sistema transmisor serial

La figura 8.5 muestra el proceso “TX_divisor” asociado al bloque funcional “Divisor 1 Hz”. Éste se encarga de generar una señal denominada “pulso”, la cual indica al siguiente proceso cuando es que debe preparar el dato que será transmitido. La configuración mostrada envía un dato cada segundo.

```

begin
  TX_divisor : process(reloj)
  begin
    if rising_edge(reloj) then
      contador<=contador+1;
      if (contador < 140000) then
        pulso <= '1';
      else
        pulso <= '0';
      end if;
    end if;
  end process TX_divisor;

```

Figura 8.5. Proceso Tx_divisor del sistema transmisor serial

La figura 8.6 muestra el proceso “Tx_prepara”, en él se implementa al bloque “Prepara comunicación”. Este proceso se encarga de generar un arreglo que contiene 2 datos a transmitir en formato ASCII. Por default se envía el carácter ‘0’, seguido de un salto de línea.

```

TX_prepara : process(reloj, pulso)
  type arreglo is array (0 to 1) of STD_LOGIC_VECTOR(7 downto 0);
  variable asc_dato : arreglo := (X"30",X"0A");
begin
  asc_dato(0):=hex_val;
  if (pulso='1') then
    if rising_edge(reloj) then
      if (conta=valor) then
        conta <= 0;
        INICIO <= '1';
        Dato <= asc_dato(i)
        if (i = 1) then
          i <= 0;
        else
          i <= i + 1;
        end if;
      else
        conta <= conta+1;
        inicio <= '0';
      end if;
    end if;
  end if;
end process TX_prepara;

```

Figura 8.6. Proceso TX_prepara del sistema transmisor serial

La figura 8.7 presenta el código del proceso “TX_envia”, correspondiente a la descripción del bloque funcional “Envía dato”. Dicho proceso es el encargado de generar la velocidad de transmisión “*Baudrate*” y colocar los datos previamente preparados para ser enviados a través de la línea de transmisión.

```
TX_envia : process(reloj, inicio, dato)
begin
    if(reloj'EVENT and reloj = '1') then
        if(Flag = '0' and INICIO = '1') then
            Flag<= '1';
            BUFF(0) <= '0';
            BUFF(9) <= '1';
            BUFF(8 DOWNT0 1) <= dato;
        end if;
        if(Flag = '1') then
            if(PRE < PRE_val) then
                PRE <= PRE + 1;
            else
                PRE<= 0;
            end if;
            if(PRE = PRE_val/2) then
                TX_WIRE <= BUFF(INDICE);
                if(INDICE < 9) then
                    INDICE <= INDICE + 1;
                else
                    Flag <= '0';
                    INDICE <= 0;
                end if;
            end if;
        end if;
    end if;
end process TX_envia;
```

Figura 8.7. Proceso TX_envia del sistema transmisor serial

Finalmente, la figura 8.8 muestra la última parte de la arquitectura del sistema transmisor serial, en donde se realiza la lectura y decodificación del valor binario leído en el switch, para su correspondiente transformación al código ASCII que será transmitido. Así mismo, se presenta la selección de la velocidad de transmisión mediante la señal “baud” dentro de una lista sensible.


```

LED <= pulso;
dato_bin<=SW;
baud<="011";

with(dato_bin) select
    hex_val <= X"30" when "0000",
                X"31" when "0001",
                X"32" when "0010",
                X"33" when "0011",
                X"34" when "0100",
                X"35" when "0101",
                X"36" when "0110",
                X"37" when "0111",
                X"38" when "1000",
                X"39" when "1001",
                X"41" when "1010",
                X"42" when "1011",
                X"43" when "1100",
                X"44" when "1101",
                X"45" when "1110",
                X"46" when "1111",
                X"23" when others;

with (baud) select
    PRE_val <= 41600 when "000", -- 1200 bauds
                20800 when "001", -- 2400 bauds
                10400 when "010", -- 4800 bauds
                5200 when "011", -- 9600 bauds
                2600 when "100", -- 19200 bauds
                1300 when "101", -- 38400 bauds
                866 when "110", -- 57600 bauds
                432 when others; --115200 bauds

end architecture behaivoral;

```

Figura 8.8. Código para manipulación de periféricos y selector de velocidad dentro de la arquitectura del sistema transmisor serial

ACTIVIDAD COMPLEMENTARIA:

El alumno diseñará un sistema capaz de enviar el valor del switch en forma binaria, es decir cuatro caracteres, uno por bit leído. La forma en que la secuencia de texto que deberá ser visualizado en la computadora es: **Valor binario=XXXX**, donde XXXX representa el número de 4 bits.

Práctica 9.

DISEÑO DE UN RECEPTOR PARA COMUNICACIÓN SERIAL

OBJETIVO:

Demostrar a los estudiantes mediante el diseño de un módulo receptor (RX), usado en comunicaciones de tipo serial UART (*Universal Asynchronous Receiver Transmitter*), la utilidad de este módulo, así como la importancia de su presencia en la arquitectura de un procesador para aplicaciones electrónicas de recepción de información. Mostrar su aplicación en el control de dispositivos periféricos desde una terminal remota.

ESPECIFICACIONES:

Utilizando un FPGA y 8 LEDS, diseñar un sistema receptor serial, el cual sea capaz de recibir un carácter ASCII del teclado de una computadora, procesarlo en el FPGA y posteriormente mostrar su código binario en los 8 LEDS. La conexión entre el FPGA y la computadora deberá realizarse empleando un circuito convertidor USB TTL-Serial. La figura 9.1 muestra el diagrama de bloques del sistema.

DIAGRAMA DE BLOQUES:

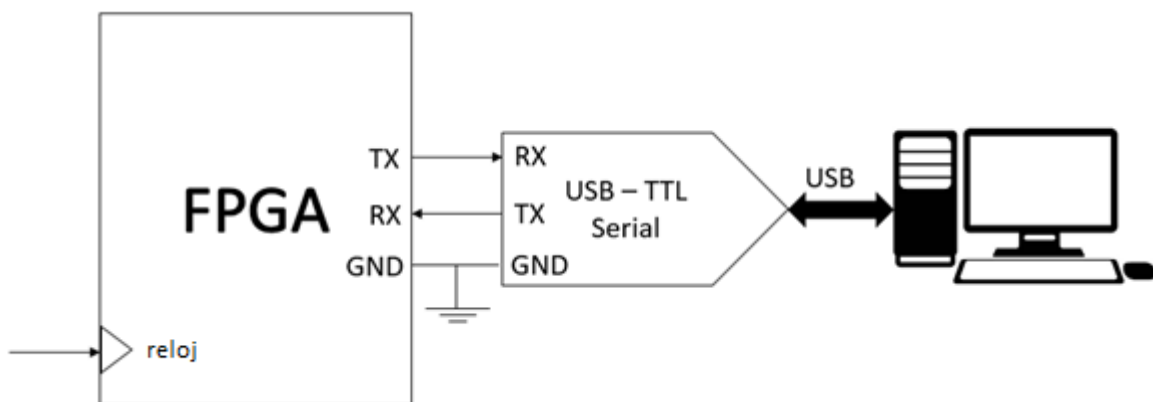


Figura 9.1. Diagrama de bloques para la comunicación serial.

Al igual que el sistema transmisor, su contraparte receptora resulta útil de ser implementada en un dispositivo FPGA, dadas las características de reconfiguración de éste. Aunado a ello y a la capacidad de emplear recursos de hardware mínimos, un módulo RX en la comunicación UART permite a su vez una amplia gama de aplicaciones electrónicas y de cómputo.

Es importante resaltar que en este punto, será posible observar que la implementación de este sistema será más simple que en el caso del transmisor, en donde gran parte de la lógica que establece la velocidad de transmisión es idéntica.

La figura 9.2 muestra los bloques funcionales del sistema Receptor, donde las señales se muestran como flechas de color azul, mientras que las terminales físicas se muestran en color rojo.

BLOQUES FUNCIONALES:

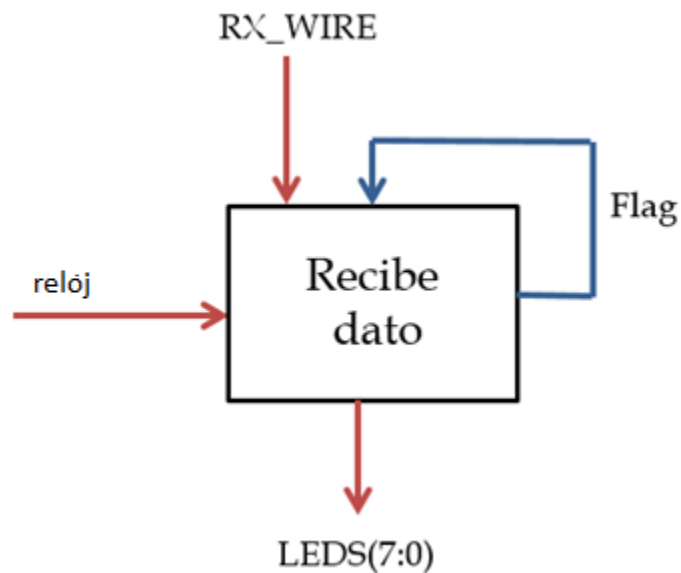


Figura 9.2. Bloque funcional del sistema receptor serial

La figura 9.3 muestra la parte entidad del sistema receptor de comunicación serial. Las terminales físicas corresponden al reloj maestro del FPGA de 50 MHz, 8 LEDS y la línea de recepción (RX_WIRE).

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

entity RX is
  port( reloj  : IN  STD_LOGIC;
        LEDS   : OUT STD_LOGIC_VECTOR(7 downto 0);
        RX_WIRE : IN  STD_LOGIC);
end entity;
```

Figura 9.3. Entidad del sistema receptor serial RX

La figura 9.4 muestra la parte declaratoria de la arquitectura del módulo **RX**, en donde se declaran todas las señales involucradas en el proceso de recepción del dato proveniente de la computadora.

```
architecture behaivoral OF RX IS
  signal BUFF: STD_LOGIC_VECTOR(9 downto 0);
  signal Flag: STD_LOGIC := '0';
  signal PRE: INTEGER RANGE 0 TO 5208 := 0;
  signal INDICE: INTEGER RANGE 0 TO 9 := 0;
  signal PRE_val: INTEGER range 0 to 41600;
  signal baud: STD_LOGIC_VECTOR(2 downto 0);
```

Figura 9.4. Parte declaratoria en la arquitectura del sistema receptor serial

La figura 9.5 presenta el código del proceso “RX_dato”, correspondiente a la descripción del bloque funcional “Recibe dato”. Dicho proceso es el encargado de generar la velocidad de transmisión “*Baudrate*” y recibir los bits asociados al dato proveniente de la terminal física RX_WIRE, para posteriormente ser transferido a los 8 LEDS.

```

begin
  RX_dato : process(reloj)
  begin
    if (reloj'EVENT and reloj = '1') then
      if (Flag = '0' and RX_WIRE = '0') then
        Flag<= '1';
        INDICE <= 0;
        PRE <= 0;
      end if;
      if (Flag = '1') then
        BUFF(INDICE)<=RX_WIRE;
        if(PRE < PRE_val) then
          PRE <= PRE + 1;
        else
          PRE <= 0;
        end if;
        if(PRE = PRE_val/2) then
          if(INDICE < 9) then
            INDICE <= INDICE + 1;
          else
            if(BUFF(0) = '0' and BUFF(9)= '1') then
              LEDS <= BUFF(8 DOWNT0 1);
            else
              LEDS <= "000000000";
            end if;
            Flag <= '0';
          end if;
        end if;
      end if;
    end if;
  end process RX_dato;

```

Figura 9.5. Proceso RX_Dato del sistema receptor serial

Finalmente, la figura 9.6 muestra la última parte de la arquitectura del sistema receptor serial, en donde se selecciona la velocidad de recepción.

```

baud<="011";
with (baud) select
  PRE_val <= 41600 when "000",  -- 1200 bauds
    20800 when "001",  -- 2400 bauds
    10400 when "010",  -- 4800 bauds
    5200 when "011",  -- 9600 bauds
    2600 when "100",  -- 19200 bauds
    1300 when "101",  -- 38400 bauds
    866 when "110",  -- 57600 bauds
    432 when others; --115200 bauds

end architecture behaivoral;

```

Figura 9.6. Código para selección de velocidad del sistema receptor serial

ACTIVIDAD COMPLEMENTARIA:

El alumno diseñará un sistema capaz de realizar el control de acciones sobre periféricos conectados al FPGA. El control deberá realizarse seleccionando cuatro diferentes caracteres del teclado de la computadora para ejecutar las siguientes tareas:

- 1) Corrimiento de LEDS
- 2) Leer el estado del dipswitch y mostrarlo en 4 leds
- 3) Contador binario de 0 a 9
- 4) PWM en un LED

Práctica 10.

DISEÑO DE UN GENERADOR DE VIDEO VGA

OBJETIVO:

El alumno aprenderá los principios de la señalización para generar video en formato VGA, así como su implantación en un FPGA.

ESPECIFICACIONES:

Utilizando un FPGA, un cable y pantalla VGA, se programará el controlador de video VGA, con la finalidad de proyectar una imagen estática.

Como se observa en el diagrama de bloques de la figura 10.1, el sistema tiene una entrada de reloj, y cinco salidas h_sync, v_sync, R, G y B.

DIAGRAMA DE BLOQUES:

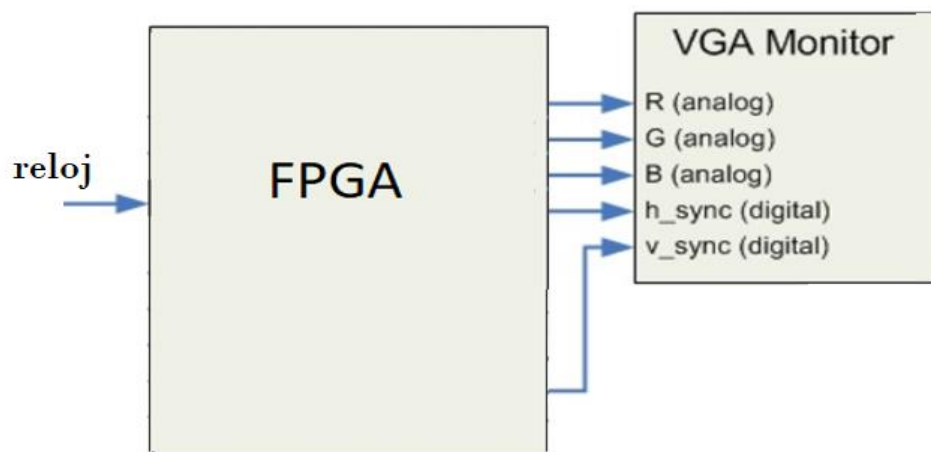


Figura 10.1. Diagrama de bloques del sistema adaptador de video VGA

Como se observa en el diagrama de bloques funcionales de la figura 10.2, el sistema cuenta con cuatro bloques funcionales.

DIAGRAMA DE BLOQUES FUNCIONALES:

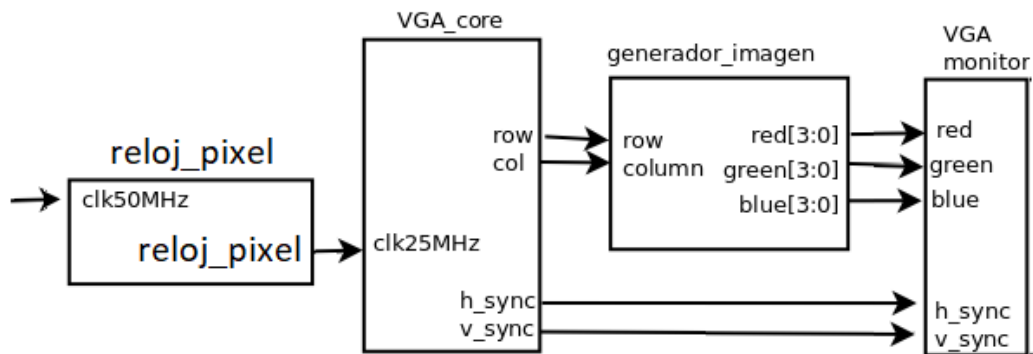


Figura 10.2. Diagrama de bloques funcionales del sistema controlador de video VGA

INTRODUCCIÓN.

El VGA es un estándar de gráficos hecho por IBM en la década de los 80s. VGA es un adaptador gráfico de video, con una resolución de 640x480.

Una señal de vídeo VGA contiene 5 señales activas:

- Dos para sincronizar video: Sincronización horizontal (h_sync) y Sincronización vertical (v_sync).
- Tres para asignar color: Rojo (R), Verde (G), Azul (B).

El formato VGA permite que se vean imágenes y video en un monitor, el video desplegará la emulación de movimiento con imágenes mostradas a una determinada velocidad. Las imágenes deben estar contenidas en un cuadro visible de 640x480 pixeles, que a su vez debe estar dentro de otro cuadro más grande (un margen invisible de derecha a izquierda y de arriba hacia abajo) de 800 pixeles x 525 líneas.

En la figura 10.3 se muestra gráficamente cómo se compone un cuadro de imagen VGA.

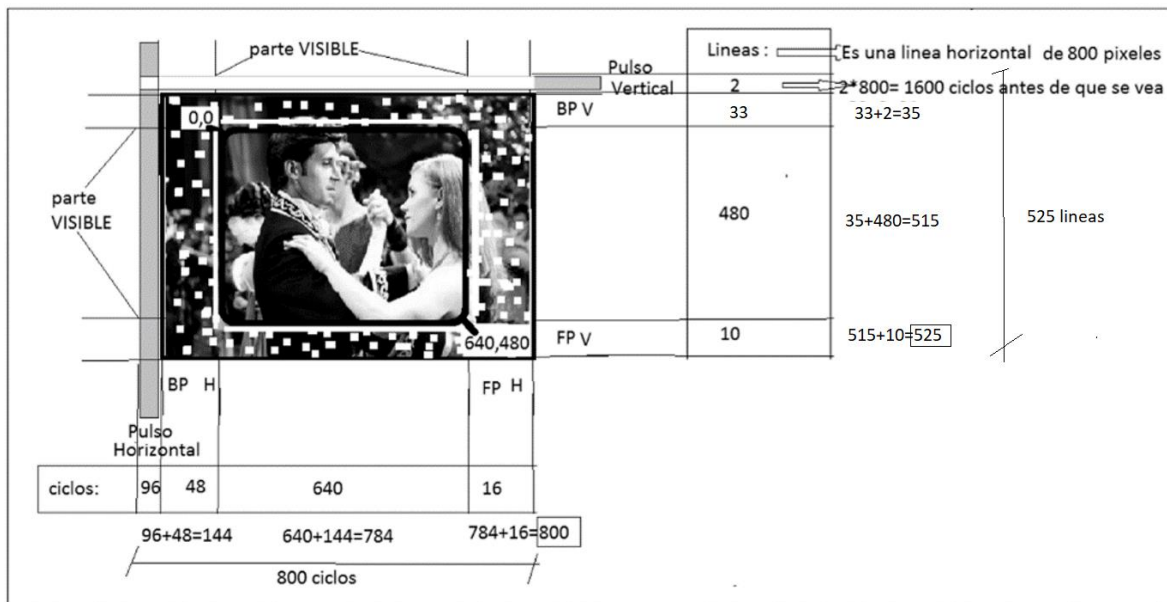


Figura 10.3. Ciclos y líneas de formato VGA 640x480

Los datos que se requieren para programar el controlador son:

- Frecuencia de actualización; 60hz. Resolución: 640x480 píxeles. Reloj: 25MHz.
- Parámetros Horizontales=> PulsoH=96, BPH=48, PH=640, FPH=16 (píxeles)
- Parámetros Verticales=> PulsoV=2, BPV=33, PV=480, FPV=10 (líneas)

DESARROLLO.

La figura 10.4 muestra la entidad del sistema de señalización VGA.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity vga is
  port ( clk50MHz:  in std_logic;
         red: out std_logic_vector (3 downto 0);
         green: out std_logic_vector (3 downto 0);
         blue: out std_logic_vector (3 downto 0);
         h_sync: out std_logic;
         v_sync: out std_logic );
end entity vga;

```

Figura 10.4. Entidad del sistema de adaptador de video VGA

Se requiere generar una frecuencia de 25 MHz, la cual se obtendrá a partir del reloj principal de la tarjeta de desarrollo. En la figura 10.5 se observa el código de un divisor para obtener la frecuencia anteriormente mencionada a partir de un reloj de 50 MHz.

```

relojpixel: process (clk50MHz) is
begin
  if rising_edge(clk50MHz) then
    reloj_pixel <= not reloj_pixel;
  end if;
end process relojpixel;  -- 25mhz

```

Figura 10.5. Divisor de frecuencia del sistema adaptador de video VGA

Para controlar los tiempos horizontales y las líneas verticales, se requiere de dos contadores, uno horizontal y el otro vertical. El primero va de 0 a 800 y el segundo de 0 a 525. La figura 10.6 muestra el código correspondiente a los procesos requeridos.

```

contadores : process (reloj_pixel)  -- H_perodo=800, V_perodo=525
begin
    if rising_edge(reloj_pixel) then
        if h_count<(h_perodo-1) then
            h_count<=h_count+1;
        else
            h_count<=0;
            if v_count<(v_perodo-1) then
                v_count<=v_count+1;
            else
                v_count<=0;
            end if;
        end if;
    end if;
end process contadores;

```

Figura 10.6. Procesos de contadores del sistema Adaptador de Video VGA

```

senial_hsync : process (reloj_pixel)  --h_pixel+h_fp+h_pulse= 784
begin
    if rising_edge(reloj_pixel) then
        if h_count>(h_pixels + h_fp) or
            h_count>(h_pixels + h_fp + h_pulse) then
            h_sync<='0';
        else
            h_sync<='1';
        end if;
    end if;
end process senial_hsync;

senial_vsync : process (reloj_pixel)  --vpixels+v_fp+v_pulse=525
begin
    --chechar si se en parte visible es 1 o 0
    if rising_edge(reloj_pixel) then
        if v_count>(v_pixels + v_fp) or
            v_count>(v_pixels + v_fp + v_pulse) then
            v_sync<='0';
        else
            v_sync<='1';
        end if;
    end if;
end process senial_vsync;

coords_pixel: process(reloj_pixel)
begin
    --asignar una coordenada en parte visible
    if rising_edge(reloj_pixel) then
        if (h_count < h_pixels) then
            column <= h_count;
        end if;
        if (v_count < v_pixels) then
            row <= v_count;
        end if;
    end if;
end process coords_pixel;

```

Figura 10.6. (continuación) Procesos de contadores del sistema adaptador de video VGA

Para visualizar el cuadro de imagen en el monitor VGA, se requiere programar en que renglón y columna inicia y finaliza.

La figura 10.7, muestra las dos condiciones que se requieren para el despliegue de la imagen:

- 1.- Que el habilitador de pintura esté en el espacio visible.
- 2.- Colocar en la coordenada, el color asignado.

Si la cuenta horizontal (h_count) es menor que 640 y si al mismo tiempo el contador horizontal (v_count) es menor que 480, significa que estamos en el espacio visual y activamos la bandera de habilitación de despliegue (display_ena=1).

```
generador_imagen: process(display_ena, row, column)
begin
  if(display_ena = '1') then
    if ((row > 300 and row <350) and
        (column>350 and column<400)) then
      red <= (others => '1');
      green<=(others => '0');
      blue<=(others => '0');
    elsif ((row > 300 and row <350) and
            (column>450 and column<500)) then
      red <= (others => '0');
      green<=(others => '1');
      blue<=(others => '0');
    elsif ((row > 300 and row <350) and
            (column>550 and column<600)) then
      red <= (others => '0');
      green<=(others => '0');
      blue<=(others => '1');
    else
      red <= (others => '0');
      green <= (others => '0');
      blue <= (others => '0');
    end if;
  else
    red<= (others => '0');
    green <= (others => '0');
    blue<= (others => '0');
  end if;
end process generador_imagen;
```

Figura 10.7. Proceso de generación de imagen del sistema Adaptador de Video VGA

Para pintar se tiene 3 colores rojo, verde y azul (en inglés Red, Green, Blue). Cuando: RGB= 1,0,0 el color es Rojo, si RGB= 0,1,0 el color que desplegará será verde, si RGB=0,0,1 será Azul, si RGB=1,1,1 el color es blanco y si RGB es 0,0,0 el color será negro.

La figura 10.8 muestra el proceso habilitador de visualización del sistema Adaptador de Video VGA.

```
display_enable: process(reloj_pixel) --- h_pixels=640; y_pixeles=480
begin
    if rising_edge(reloj_pixel) then
        if (h_count < h_pixels AND v_count < v_pixels) THEN
            display_ena <= '1';
        else
            display_ena <= '0';
        end if;
    end if;
end process display_enable;
```

Figura 10.8. Proceso habilitador de visualización del sistema adaptador de video VGA

Se requiere dar valores a las constantes para el manejo del formato VGA, que se declaran dentro de un “generic”, la figura 10.8, muestra un ejemplo de estos valores.

```
generic(      --Constantes para monitor VGA en 640x480
    constant h_pulse   : integer := 96;
    constant h_bp      : integer := 48;
    constant h_pixels  : integer := 640;
    constant h_fp      : integer := 16;
    constant v_pulse   : integer := 2;
    constant v_bp      : integer := 33;
    constant v_pixels  : integer := 480;
    constant v_fp      : integer := 10
);
```

Figura 10.9. Constantes dentro de un *GENERIC* del sistema adaptador de video VGA

Las declaraciones y operaciones de las constantes tipo señal adicionales, se muestra en figura 10.10.

```
--Contadores
constant h_period : integer := h_pulse + h_bp + h_pixels + h_fp;
constant v_period : integer := v_pulse + v_bp + v_pixels + v_fp;
signal h_count    : integer range 0 to h_period - 1 := 0;
signal v_count    : integer range 0 to v_period - 1 := 0;
```

Figura 10.10. Declaraciones y operaciones de las constantes tipo señal

ACTIVIDAD COMPLEMENTARIA:

El alumno unirá los distintos procesos en uno solo y mostrará sus resultados en un monitor.

Práctica 11.

EMULADOR DE DISPLAY 7 SEGMENTOS EN MONITOR

OBJETIVO:

El alumno diseñara un emulador de display de 7 segmentos empleando un FPGA y un monitor VGA.

ESPECIFICACIONES:

Utilizando un FPGA, un cable y un monitor con entrada VGA, se diseñará un sistema digital en el que su entrada sea un número binario de cuatro bits y su salida sea la visualización de ese número en un display de 7 segmentos en un monitor.

La figura 11.1 muestra el diagrama de bloques del sistema y la figura 11.2 muestra los bloques funcionales requeridos en el sistema Emulador de Display 7 Segmentos en Monitor.

DIAGRAMA DE BLOQUES:

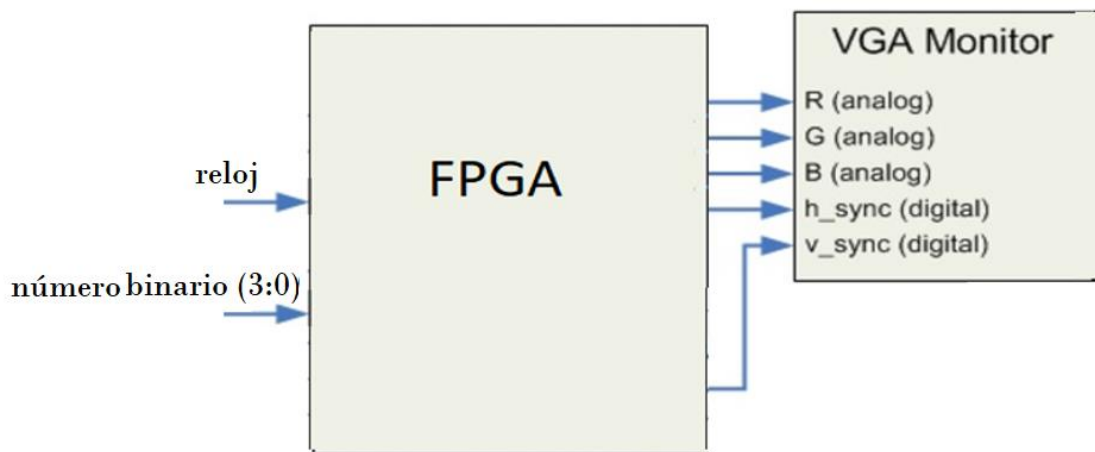


Figura 11.1. Diagrama de bloques del sistema emulador de display 7 segmentos en monitor

DIAGRAMA DE BLOQUES FUNCIONALES:

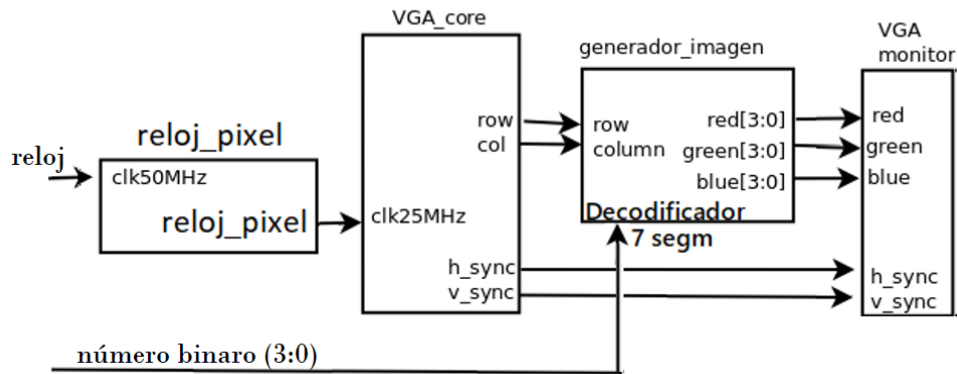


Figura 11.2. Bloques funcionales del sistema emulador de display 7 segmentos en monitor

La entidad del sistema emulador de display 7 segmentos en monitor se muestra en la figura 11.3.

```
port( clk50MHz:  in std_logic;
      red: out std_logic_vector (3 downto 0);      -- al monitor
      green: out std_logic_vector (3 downto 0);
      blue: out std_logic_vector (3 downto 0);
      h_sync: out std_logic;
      v_sync: out std_logic;
      dipsw: in std_logic_vector(3 downto 0); -- numeros para
      A,B,C,D,E,F,G: out std_logic );          -- decodificador
end entity mivga;
```

Figura 11.3. Entidad del sistema emulador de display 7 segmentos en monitor

Se requiere un caso por cada número que se desee visualizar en el monitor, cada caso corresponderá a un cuadro de imagen diferente. Todos los casos se deben declarar en el proceso generador de imagen. La figura 11.4 muestra la declaración de constantes.

```

constant cero:      std_logic_vector(6 downto 0) := "0111111"; --GFEDCBA
constant uno:       std_logic_vector(6 downto 0) := "0000110";
constant dos:       std_logic_vector(6 downto 0) := "1011011";
constant tres:      std_logic_vector(6 downto 0) := "1001111";
constant cuatro:    std_logic_vector(6 downto 0) := "1100110";
constant cinco:     std_logic_vector(6 downto 0) := "1101101";
constant seis:      std_logic_vector(6 downto 0) := "1111101";
constant siete:     std_logic_vector(6 downto 0) := "0000111";
constant ocho:      std_logic_vector(6 downto 0) := "1111111";
constant nueve:     std_logic_vector(6 downto 0) := "1110011";
constant r1:std_logic_vector(3 downto 0):=(others => '1');
constant r0:std_logic_vector(3 downto 0):=(others => '0');
constant g1:std_logic_vector(3 downto 0):=(others => '1');
constant g0:std_logic_vector(3 downto 0):=(others => '0');
constant b1:std_logic_vector(3 downto 0):=(others => '1');
constant b0:std_logic_vector(3 downto 0):=(others => '0');
-- variable a,b,c,d,e,f: std_logic;
signal conectornum:std_logic_vector(6 downto 0); -- coneccion del
-- decodificador con image_gen

```

Figura 11.4. Declaración de constantes del sistema emulador de display 7 segmentos en monitor

El decodificador BCD a 7 segmentos se declara dentro de la arquitectura, como se muestra en la figura 11.5.

```

with dipsw select conectornum <= --decodificador para los números
    "0111111" when "0000",
    "0000110" when "0001",
    "1011011" when "0010",
    "1001111" when "0011",
    "1100110" when "0100",
    "1101101" when "0101",
    "1111101" when "0110",
    "0000111" when "0111",
    "1111111" when "1000",
    "1110011" when "1001",
    "0000000" when others;

```

Figura 11.5. Código decodificador de 7 segmentos del sistema emulador de display 7 segmentos en monitor

Respecto al display de 7 segmentos, la figura 11.6 muestra la asignación de cada segmento con sus respectivas coordenadas.

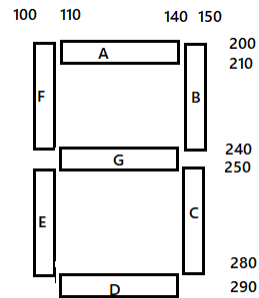


Figura 11.7. Asignación de cada segmento con sus respectivas coordenadas

Por ejemplo, para que aparezca el numero '1' deberá activarse el segmento B (color: verde) y C (color: rojo). La figura 11.8, muestra el código para visualizar los números uno y dos.

```
when uno=>
  if ((row > 210 and row <240) and
      (column>140 and column<150)) then    -- B verde
    red   <= (others => '0');
    green <= (others => '1');
    blue  <= (others => '0');
  elsif ((row > 250 and row <280) and
          (column>140 and column<150)) then -- C rojo
    red   <= (others => '1');
    green <= (others => '0');
    blue  <= (others => '0');
  else
    red   <= (others => '0');
    green <= (others => '0');
    blue  <= (others => '0');
  end if;

when dos=>
  if ((row > 200 and row <210) and
      (column>110 and column<140)) then    -- A azul
    red   <= (others => '0');
    green <= (others => '0');
    blue  <= (others => '1');
  elsif ((row > 210 and row <240) and
          (column>140 and column<150)) then -- B verde
    red   <= (others => '0');
    green <= (others => '1');
    blue  <= (others => '0');
```

Figura 11.8. Código para visualizar los números uno y dos en un monitor

```

elseif ((row > 280 and row <290) and
        (column>110 and column<140)) then -- D blanco
    red   <= (others => '1');
    green <= (others => '1');
    blue  <= (others => '1');
elseif ((row > 250 and row <280) and
        (column>100 and column<110)) then -- E cian
    red   <= (others => '0');
    green <= (others => '1');
    blue  <= (others => '1');
elseif ((row > 240 and row <250) and
        (column>110 and column<140)) then -- G violeta
    red   <= (others => '1');
    green <= (others => '0');
    blue  <= (others => '1');
else
    red   <= (others => '0');
    green <= (others => '0');
    blue  <= (others => '0');
end if;

```

Figura 11.8. (continuación) Código para visualizar los números uno y dos en un monitor

Para visualizar el número nueve se deben activar los bloques A, B, C, F y G. La figura 11.9 muestra el código requerido.

```

when nueve=>
    if ((row > 200 and row <210) and
        (column>110 and column<140)) then -- A azul
        red   <= (others => '0');
        green <= (others => '0');
        blue  <= (others => '1');
    elsif ((row > 210 and row <240) and
           (column>140 and column<150)) then -- B verde
        red   <= (others => '0');
        green <= (others => '1');
        blue  <= (others => '0');
    elsif ((row > 250 and row <280) and
           (column>140 and column<150)) then -- C rojo
        red   <= (others => '1');
        green <= (others => '0');
        blue  <= (others => '0');
    elsif ((row > 210 and row <240) and
           (column>100 and column<110)) then -- F amarillo
        red   <= (others => '1');
        green <= (others => '1');
        blue  <= (others => '0');
    end if;
end when;

```

Figura 11.9. Código para visualizar el número nueve en un monitor

```
elseif ((row > 240 and row <250) and
        (column>110 and column<140)) then -- G   violeta
    red   <= (others => '1');
    green <= (others => '0');
    blue  <= (others => '1');
else
    red   <= (others => '0');
    green <= (others => '0');
    blue  <= (others => '0');
end if;
```

Figura 11.9. (continuación) Código para visualizar el número nueve en un monitor

ACTIVIDAD COMPLEMENTARIA:

En esta práctica se mostró como codificar para que se emulen los números 1, 2 y 9 en la pantalla VGA. El alumno implementará además los números 0, 3, 4, 5, 6, 7 y 8.

Práctica 12.

EMULADOR DE CONTADORES EN UN MONITOR

OBJETIVO:

El alumno aprenderá el diseño de contadores mediante un FPGA y con visualización en un monitor VGA.

ESPECIFICACIONES:

Utilizando un FPGA, un cable VGA y un monitor, diseñar un contador que cuente del cero al nueve. Cuando el conteo llegue a su límite, el contador deberá reiniciarse. La figura 12.1 muestra el diagrama de bloques y la figura 12.2 muestra los bloques funcionales del sistema emulador de contadores en un monitor.

DIAGRAMA DE BLOQUES:

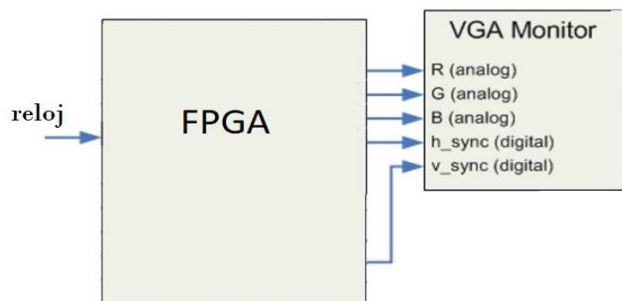


Figura 12.1. Diagrama de bloques del sistema emulador de contadores en un monitor

DIAGRAMA DE BLOQUES FUNCIONALES:

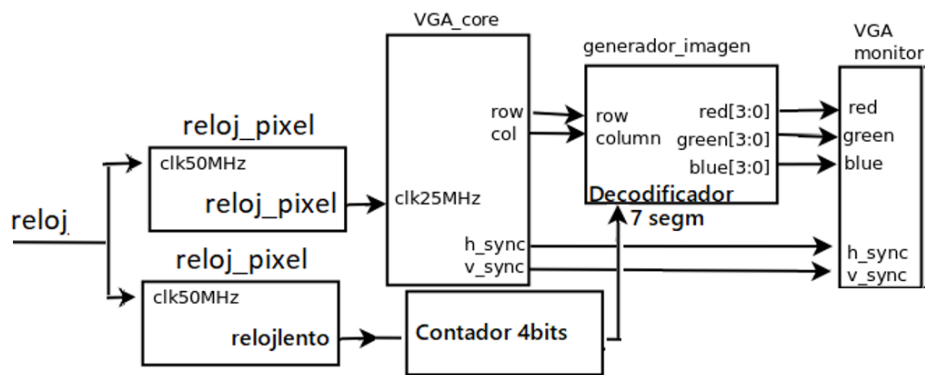


Figura 12.2 Diagrama de bloques del sistema emulador de contadores en un monitor

El proceso del divisor de frecuencia se muestra en la figura 12.3, donde puede observarse la entrada de reloj de 50 mHz y a su salida la señal de reloj “clklow” a muy baja frecuencia.

```
process (clk50mhz)
begin
    if (clk50mhz'event and (clk50mhz ='1')) then
        conteo<=conteo+1;
        if (conteo=1) then
            conteo<=0;
            clklow<=not (clklow);
        end if;
    end if;
end process;
```

Figura 12.3. Divisor de frecuencia del sistema emulador de contadores en un monitor

La figura 12.4 muestra la declaración de constantes y la máquina de estados requerida para la obtención del contador de cuatro bits.

```
subtype state is std_logic_vector (3 downto 0);
signal present_state, next_state: state;
constant state0: state:= "0000";
constant state1: state:= "0001";
constant state2: state:= "0010";
constant state3: state:= "0011";
constant state4: state:= "0100";
constant state5: state:= "0101";
constant state6: state:= "0110";
constant state7: state:= "0111";
constant state8: state:= "1000";
constant state9: state:= "1001";
constant state10: state:= "1010";
constant state11: state:= "1011";
constant state12: state:= "1100";
constant state13: state:= "1101";
constant state14: state:= "1110";
constant state15: state:= "1111";
begin
```

Figura 12.4. Máquina de estados para contador de 4 bits

```

contal: process(clklow)
begin
    if rising_edge(clklow) then
        if (reset='1') then
            present_state <= state0;
        else
            present_state<= next_state;
        end if;
    end if;
end process;

conta2: process(present_state)
begin
    case present_state is
        when state0=>
            next_state<= state1;
        when state1=>
            next_state<= state2;
        when state2=>
            next_state<= state3;
        when state3=>
            next_state<= state4;
        when state4=>
            next_state<= state5;
        when state5=>
            next_state<= state6;
        when state6=>
            next_state<= state7;
        when state7=>
            next_state<= state8;
        when state8=>
            next_state<= state9;
        when state9=>
            next_state<= state10;
        when state10=>
            next_state<= state11;
        when state11=>
            next_state<= state12;
        when state12=>
            next_state<= state13;
        when state13=>
            next_state<= state14;
        when state14=>
            next_state<= state15;
        when state15=>
            next_state<= state0;
        when others=>
            next_state<= state0;
    end case;
    count <= present_state;
end process;

```

Figura 12.4. (continuación) Máquina de estados para contador de 4 bits

ACTIVIDAD COMPLEMENTARIA:

Diseñar un contador binario descendente con visualización en un monitor VGA. Cuando el contador llegue a su límite de cuenta, éste deberá reiniciarse.

Práctica 13.

CAPTURA DE IMÁGENES DE CÁMARA DIGITAL

OBJETIVOS:

El alumno aplicará los conocimientos y las habilidades obtenidas en el manejo de la señalización VGA, para definir una unidad de control de una cámara digital. Aprenderá además la señalización requerida en el almacenamiento de imágenes digitales en un FPGA.

INTRODUCCIÓN.

La cámara digital OV7670 captura imágenes de 640x480 píxeles. Opera a 3.3 V, aunque cuenta con un regulador que permite polarización de hasta 5V. El formato de salida de video, por defecto es el YUV (4:2:2), aunque puede generar RGB 4:2:2 y RGB565/555/444. El protocolo de comunicación con la cámara es el SCCB, compatible con el protocolo de comunicación I2C (*Inter Integrated Circuits*). La cámara incluye un módulo para el control del color, de la saturación, del tinte, de gama, y de realzado de bordes, entre otros. Éstos deben ser configurados escribiendo los valores adecuados en los registros correspondientes.

La cámara opera por default en formato YUV 4:2:2 de 640x480. De la señal entregada por la cámara, solamente se recupera la componente de luminancia y esta componente alimenta a un monitor con entrada VGA, y dependiendo de la capacidad del FPGA es la cantidad de bits que define la componente de luminancia.

La imagen que entrega la cámara es almacenada en una memoria de doble puerto (escritura y lectura) dentro del FPGA. La figura 13.1 muestra la cámara OV7670 y el kit de desarrollo.



Figura 13.1. Fotografía de la Cámara OV7670 [8]

ESPECIFICACIONES:

Utilizando una cámara digital, un FPGA y un monitor con entrada VGA, almacenar las imágenes dentro del FPGA con el fin de mostrarlas en un monitor. La figura 13.2 muestra el diagrama de bloques del sistema de Captura de Imágenes de Cámara Digital.

DIAGRAMA DE BLOQUES:

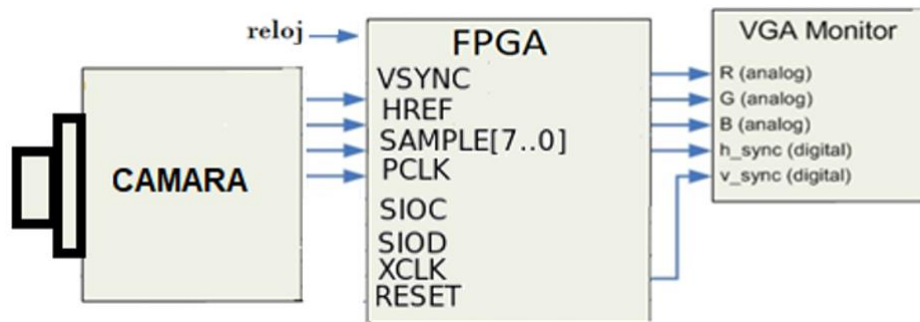


Figura 13.2. Diagrama de bloques de sistema captura de imágenes de cámara digital

El diagrama a bloques funcionales del sistema captura de imágenes de cámara digital es mostrado en la figura 13.3.

BLOQUES FUNCIONALES:

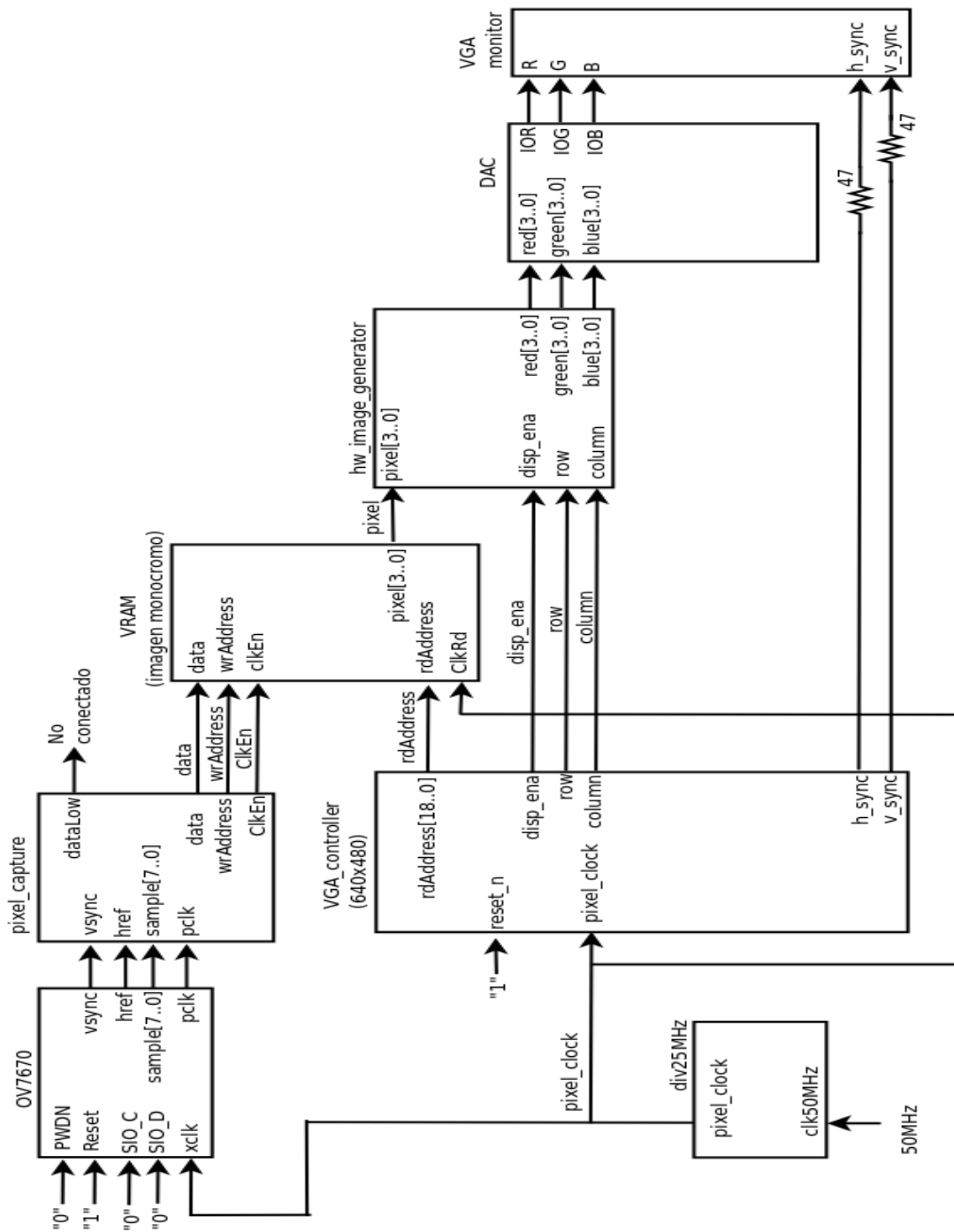


Figura 13.3. Diagrama a bloques del sistema captura de imágenes de cámara digital

Como puede observarse en el diagrama, se requiere diseñar cinco bloques funcionales dentro del FPGA. A cada uno lo llamaremos módulo, siendo los más importantes los de Captura_pixel y VGA_controller.

El módulo Captura_pixel, se encarga de capturar la información YUV de cada pixel. Entonces, se separa la componente de luminancia. Se calcula una dirección de memoria para almacenar únicamente esta componente de luminancia correspondiente a cada pixel.

El módulo llamado VGA_controller, se encarga de generar las señales de sincronía para el monitor VGA. Genera las direcciones de memoria para lectura de los valores de pixel: sólo luminancia. Los valores de pixel son enviados al hw_image_generator el cual genera la señal RGB para el monitor VGA.

La figura 13.4 muestra las terminales de la cámara, su tipo y descripción de cada una de ellas.

Pin	Type	Description
VDD	Supply	Power supply
GND	Supply	Ground level
SIOC	input	SCCB clock
SIOD	input/output	SCCB data
VSYNC	output	Vertical synchronization
HREF	output	Horizontal synchronization
PCLK	output	Pixel clock
XCLK	input	System clock
D0-D7	output	Video parallel output
RESET	input	Reset (active low)
PWDN	input	Power down (active high)

Figura 13.4. Definición de terminales de la cámara OV7670 [8]

La figura 13.5 muestra el diagrama de tiempos de las señales de sincronización recibidas por la cámara digital.

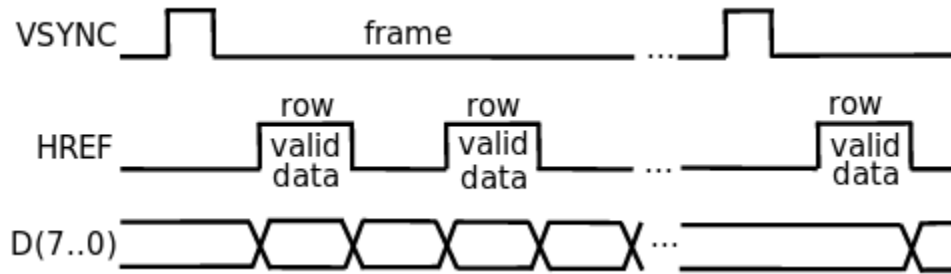


Figura 13.5. Diagrama de tiempos de las señales de sincronización recibidas por la cámara [8]

La señal “VSYNC” es indicativa de cada cuadro de imagen. La señal “HREF” enmarca la información de cada pixel.

La cámara entrega, por defecto, una señal YCbCr en formato 4.2.2. Este formato implica que se envía completo el plano “Y” en tanto que los planos de color Cb y Cr se envían submuestreados en un factor de dos. La figura 13.6 ilustra este formato. En esta figura se observa que por cada pixel es enviada una pareja de componentes CbY ó una la pareja CrY. Cada componente requiere de un byte para su representación.

La frecuencia de reloj con la cual, la cámara entrega datos, en formato YCbCr, es el doble de la frecuencia con la que se alimenta la cámara.

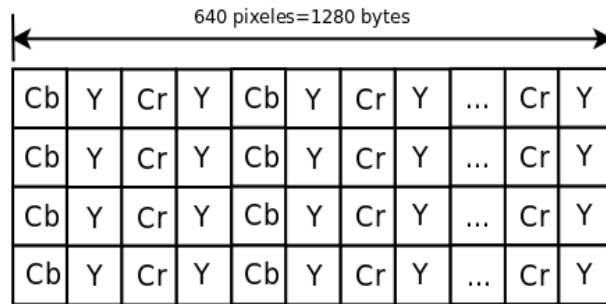


Figura 13.6. Formato de señales entregada por la cámara

La figura 13.7 muestra a detalle el diagrama de tiempos con la cual la cámara envía bytes de datos. Nótese que la cámara opera en el flanco negativo de la señal de reloj.

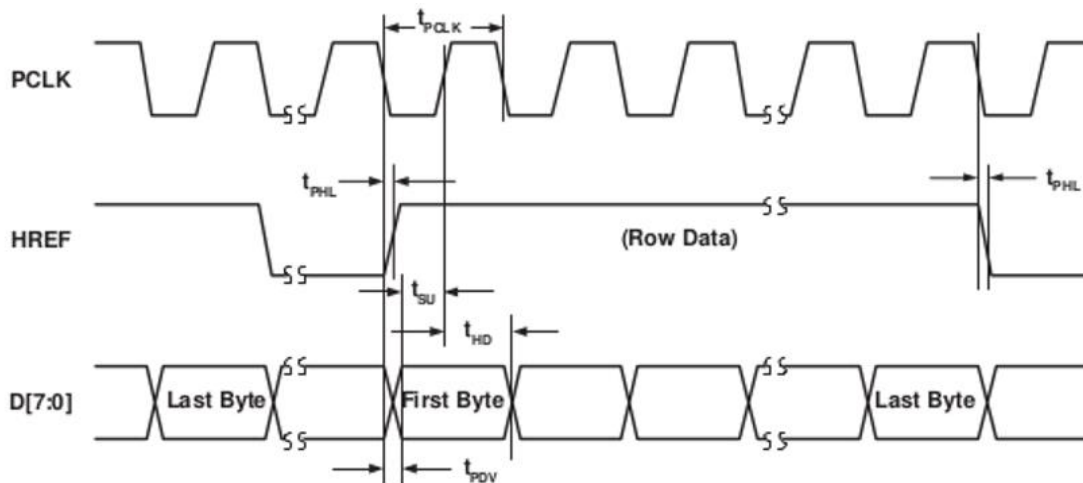


Figura 13.7. Diagrama de tiempos de las señales de salida de la cámara OV7670 [8]

Adicionalmente, se debe mencionar que la cámara debe alimentarse con una señal de reloj de 25MHz, debido a que provee 30 cuadros por segundo.

Dado que no se va enviar datos a la cámara, las señales de comunicación **SIOD** y **SIOC** pueden dejarse abiertas o bien, en “1”.

La cámara viene pre configurada para proveer una salida en formato YUV 4:2:2, en particular, cada línea de imagen se suministra en la secuencia Y, V, Y, U.

Cada componente de color está compuesto por un byte. Se sigue el formato “little endian”, es decir, el bit menos significativo D(0) es enviado primero y el bit más significativo D(7) es enviado al final.

La figura 13.8 muestra la entidad del sistema Captura de imágenes de cámara digital en un FPGA.

```
port( clk50MHz:  in std_logic;  --for this example is 50MHz
      red: out std_logic_vector (3 downto 0);
      green: out std_logic_vector (3 downto 0);
      blue: out std_logic_vector (3 downto 0);
      n_sync: out std_logic;
      n_blank: out std_logic;
      h_sync: out std_logic;
      v_sync: out std_logic;
      sio_c: out std_logic:='0';
      sio_d: out std_logic:='0';
      pwn: out std_logic:='0';
      resetcamera: out std_logic:='1';
      xclk: out std_logic;
      pclk:  in std_logic;
      vsync:  in std_logic;
      href:  in std_logic;
      sample:in std_logic_vector (7 downto 0) );
```

Figura 13.8. Entidad del sistema captura de imágenes de cámara digital

Las constantes usadas en el programa corresponden a las constantes que se requieren para manipular el monitor con entrada de puerto VGA. La figura 13.9 muestra la declaración de dichas constantes.


```

generic(      --constantes para monitor vga en 640x480
  constant h_pulse : integer:=96; --horizontal sync pulse width in pixels
  constant h_bp    : integer:=48; --horizontal back porch width in pixels
  constant h_pixels: integer:=640;--horizontal display width in pixels
  constant h_fp    : integer:=16;--horizontal front porch width in pixels
  constant v_pulse : integer:=2;  --vertical sync pulse width in rows
  constant v_bp    : integer:=33; --vertical back porch width in rows
  constant v_pixels: integer:=480;--vertical display width in rows
  constant v_fp    : integer:=10  --vertical front porch width in rows
);

```

Figura 13.9. Declaración de constantes.

Se diseñará un módulo VRAM, con el fin de tener dos puertos síncronos de acceso. Ambos puertos tienen relojes independientes. La finalidad de tener dos puertos es para evitar el diseño de una cola para las peticiones de acceso (lectura y escritura) a un chip RAM. El código de la memoria es mostrado en la figura 13.10.

```

wrvram:process (clken)      -- sección de escritura
begin
  if rising_edge(clken) then
    memory(wraddr)<=data;
  end if;
end process wrvram;

rdvram: process (reloj_pixel) --sección de lectura
begin
  if falling_edge(reloj_pixel) then
    pixel<=memory(rdaddress);
  end if;
end process rdvram;

```

Figura 13.10. Código del módulo de memoria VRAM

La razón de esta memoria está en que la cámara maneja su propia señalización para enviar datos. Esta señalización es diferente de la que requiere el monitor VGA, por lo tanto, se requiere de un búfer que reciba datos de la cámara y que luego, pase los datos al monitor. El búfer diseñado para este proyecto tiene un doble puerto, cada uno con su propia señalización de reloj. Así, hay un puerto de escritura y hay un puerto de lectura.

La figura 13.11 muestra la carta ASM con la información de la cámara y como se generan direcciones de memoria para almacenar los datos y la figura 13.12 muestra el código del módulo Captura_pixel.

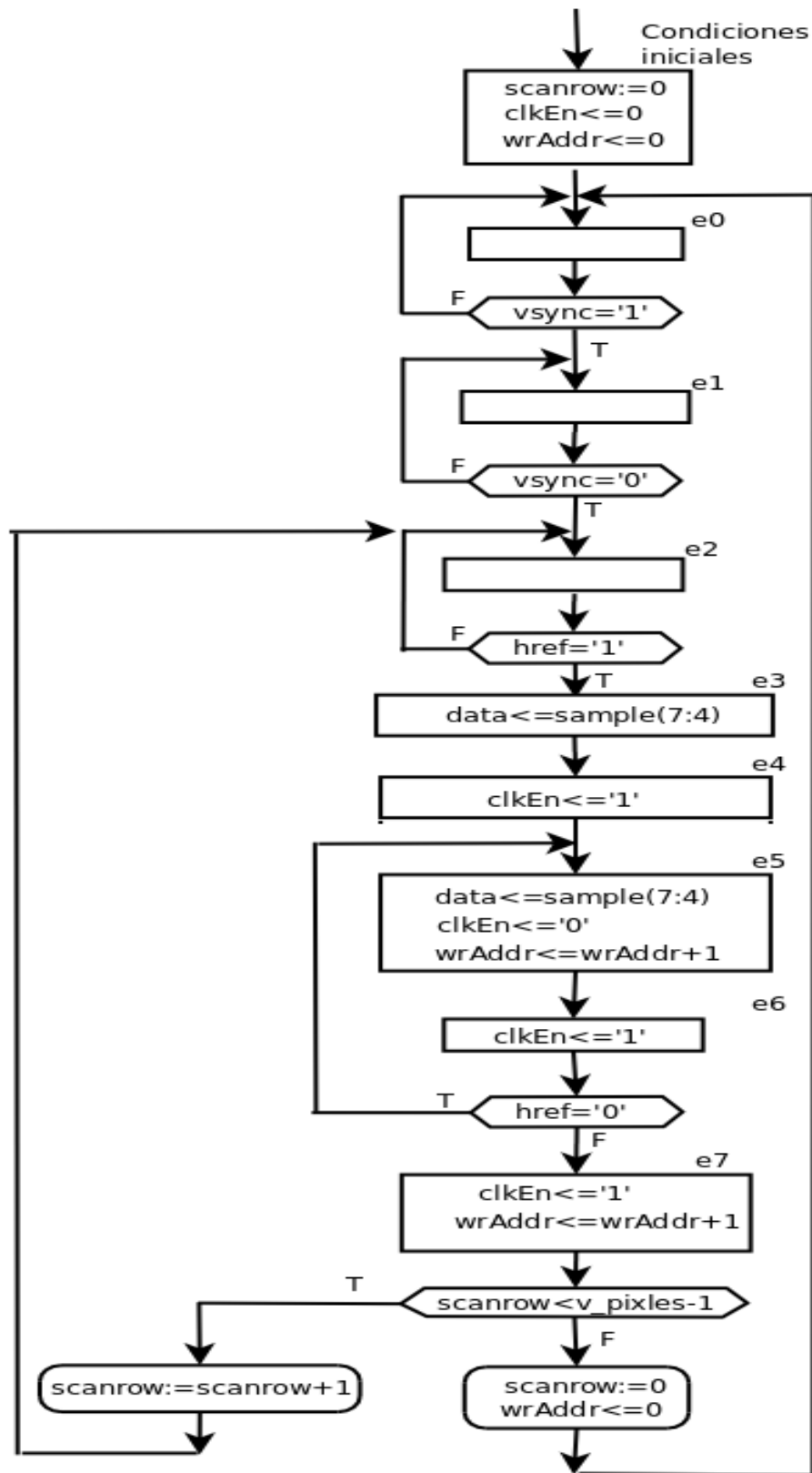


Figura 13.11. Carta ASM del módulo captura_pixel

```
captura_pixel:process (pclk)
  variable scanrow : integer range 0 to v_pixels:=0;
begin
  if rising_edge(pclk) then
    case state is
      when e0=>
        if vsync='1' then
          state<=e1;
        end if;
      when e1=>
        if vsync='0' then
          state<=e2;
        end if;
      when e2=>
        if href='1' then
          state<=e3;
        end if;
      when e3=>
        data<=sample(7 downto 4);
        state<=e4;
      when e4=>
        clkEn<='1';
        state<=e5;
      when e5=>
        data<=sample(7 downto 4);
        clkEn<='0'; wrAddr<=wrAddr+1;
        state<=e6;
      when e6=>
        clkEn<='1';
        if href='1' then
          state<=e5;
        else
          state<=e7;
        end if;
      when e7=>
        clkEn<='0';
        wrAddr<=wrAddr+1;
        if scanrow<v_pixels-1 then
          scanrow:=scanrow+1;
          state<=e2;
        else
          scanrow:=0;
          wrAddr<=0;
          state<=e0;
        end if;
      end case;
    end if;
  end process captura_pixel;
```

Figura 13.12. Código del módulo captura_pixel

La figura 13.13 muestra el código requerido para el generador de imagen.

```
generador_imagen: process(display_ena, row, column, pixel)
begin
  if(display_ena = '1') then
    if ((row > 300 and row <350) and
        (column>350 and column<400)) then
      red <= (others => '1');
      green<=(others => '0');
      blue<=(others => '0');
    elsif ((row > 300 and row <350) and
            (column>450 and column<500)) then
      red <= (others => '0');
      green<=(others => '1');
      blue<=(others => '0');
    elsif ((row > 300 and row <350) and
            (column>550 and column<600)) then
      red <= (others => '0');
      green<=(others => '0');
      blue<=(others => '1');
    elsif (row<v_pixels and column<h_pixels) then
      red<= pixel;
      green<= pixel;
      blue<= pixel;
    else
      red <= (others => '0');
      green<= (others => '0');
      blue <= (others => '0');
    end if;
  else
    red<= (others => '0');
    green <= (others => '0');
    blue<= (others => '0');
  end if;
end process generador_imagen;
```

Figura 13.13. Código del módulo generador de imagen

Finalmente, la unión de todos los códigos antes mencionados se muestra en la figura 13.14.

```

library ieee;
use ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity OV7670 is
  generic(
    --constantes para monitor vga en 640x480
    constant h_pulse : integer:=96; --horiztonal sync pulse width in pixels
    constant h_bp    : integer:=48; --horiztonal back porch width in pixels
    constant h_pixels: integer:=640;--horiztonal display width in pixels
    constant h_fp    : integer:=16;--horiztonal front porch width in pixels
    constant v_pulse : integer:=2;  --vertical sync pulse width in rows
    constant v_bp    : integer:=33; --vertical back porch width in rows
    constant v_pixels: integer:=480;--vertical display width in rows
    constant v_fp    : integer:=10  --vertical front porch width in rows
  );
  port( clk50MHz:  in std_logic; --for this example is 50MHz
        -- reset      : in std_logic;  -- to monitor
        --pixel_clk: out std_logic;      --for monitoring frequency
        red: out std_logic_vector (3 downto 0);
        green: out std_logic_vector (3 downto 0);
        blue: out std_logic_vector (3 downto 0);
        n_sync: out std_logic;
        n_blank: out std_logic;
        h_sync: out std_logic;
        v_sync: out std_logic;
        sio_c: out std_logic:='0';      -- to camera
        sio_d: out std_logic:='0';
        pwrn: out std_logic:='0';
        resetcamera: out std_logic:='1';
        xclk: out std_logic;
        pclk:  in std_logic;              -- from camera
        vsync: in std_logic;
        href:  in std_logic;
        sample:in std_logic_vector (7 downto 0) );
  end entity OV7670;
architecture behavioral of OV7670 is
  --Contadores
  signal h_period: integer := h_pulse + h_bp + h_pixels + h_fp;
  signal v_period: integer := v_pulse + v_bp + v_pixels + v_fp;
  signal h_count: integer range 0 to h_period := 0;
  signal v_count: integer range 0 to v_period - 1 := 0;
  --vram
  type matrix is array (0 to 307199) of std_logic_vector (3 downto 0);
  signal memory: matrix;
  signal memorysize : integer:=h_pixels*v_pixels;
  signal pixel: std_logic_vector( 3 downto 0);
  --pixel_capture - vram
  signal data: std_logic_vector( 3 downto 0);
  signal wrclk: std_logic;
  --pixel_capture: internal signals
  type states is (e0,e1,e2,e3,e4,e5,e6,e7);
  signal state: states:=e0;
  signal wraddr: integer range 0 to h_pixels*v_pixels:=0;
  signal clken: std_logic:='0';

```

Figura 13.14. Código final del sistema captura de imágenes de cámara digital

```
--Reloj de pixel
signal reloj_pixel: STD_LOGIC:='0';
--vga_controller - vram
signal rdaddress: integer:=0;
--vga_controller - hw_image_generator
signal display_ena: std_logic; --display enable ('1' = display time,
'0' = blanking time)
signal column: integer;      --horizontal pixel coordinate
signal row: integer;         --vertical pixel coordinate
begin
  div25MHz: process (clk50MHz) is
  begin
    if rising_edge(clk50MHz) then
      reloj_pixel <= not reloj_pixel;
    end if;
  end process div25MHz;

  -- Reloj a la camara
  XCLK<=reloj_pixel;

  -- Controlador del monitor
  Contadores : process (reloj_pixel)
  begin
    if rising_edge(reloj_pixel) then
      if h_count<(h_period-1) then
        h_count<=h_count+1;
      else
        h_count<=0;
        if v_count<(v_period-1) then
          v_count<=v_count+1;
        else
          v_count<=0;
        end if;
      end if;
    end if;
  end process Contadores;

  rdAddress <= column + (row * h_pixels);

  senial_hsync : process (h_count)
  begin
    --if rising_edge(reloj_pixel) then
      if h_count>(h_pixels + h_fp) or
        h_count>(h_pixels + h_fp + h_pulse) then
        h_sync<='0';
      else
        h_sync<='1';
      end if;
    --end if;
  end process senial_hsync;
```

Figura13.14. (continuación) Código final del sistema captura de imágenes de cámara digital

```

senial_vsync : process (v_count)
begin
    --if rising_edge(reloj_pixel) then
        if v_count>(v_pixels + v_fp) or
           v_count>(v_pixels + v_fp + v_pulse) then
            v_sync<='0';
        else
            v_sync<='1';
        end if;
    --end if;
end process senial_vsync;

coords_pixel_col: process(h_count)
begin
    if (h_count < h_pixels) then
        column <= h_count;
    end if;
end process coords_pixel_col;

coords_pixel_row: process(v_count)
begin
    if (v_count < v_pixels) then
        row <= v_count;
    end if;
end process coords_pixel_row;

generador_imagen: process(display_ena, row, column,pixel)
begin
    if(display_ena = '1') THEN          --display time
        if ((row > 300 and row <350) and
            (column>350 and column<400)) THEN
            red <= (others => '1');
            green<=(others => '0');
            blue<=(others => '0');
        elsif ((row > 300 and row <350) and
            (column>450 and column<500)) then
            red <= (others => '0');
            green<=(others => '1');
            blue<=(others => '0');
        elsif ((row > 300 and row <350) and
            (column>550 and column<600)) then
            red <= (others => '0');
            green<=(others => '0');
            blue<=(others => '1');
        elsif (row<v_pixels and column<h_pixels) then
            --yuv: solo luminancia
            red <= pixel;
            green <= pixel;
            blue <= pixel;
        else

```

Figura 13.14. (continuación) Código final del sistema captura de imágenes de cámara digital


```

        red <= (others => '0'); --es el fondo
        green <= (others => '0');
        blue <= (others => '0');
    end if;
else
    --blanking time
    red <= (others => '0');
    green <= (others => '0');
    blue <= (others => '0');
end if;
end process generador_imagen;

captura_pixel: process (pclk)
    --variable memorysize : integer := h_pixels*v_pixels;
    variable scanrow      : integer range 0 to v_pixels:=0;
begin
    if rising_edge(pclk) then
        case state is
            when e0 =>
                if vsync='1' then
                    state <=e1;
                end if;
            when e1 =>
                if vsync='0' then
                    state <=e2;
                end if;
            when e2 =>
                if href='1' then
                    state <=e3;
                end if;
            when e3 =>
                data<=sample(7 downto 4);
                state <=e4;
            when e4 =>
                clkEn<='1';
                state<=e5;
            when e5 =>
                data<=sample(7 downto 4);
                clkEn<='0';
                wrAddr<=wrAddr+1;
                state<=e6;
            when e6 =>
                clkEn<='1';
                if href='1' then
                    state<=e5;
                else
                    state<=e7;
                end if;
            when e7 =>
                clkEn<='0';
                wrAddr<=wrAddr+1;
                if scanrow<v_pixels-1 then
                    scanrow:=scanrow+1;
                    state<=e2;
                end if;
            end case;
        end if;
    end process captura_pixel;
end process generador_imagen;

```

Figura 13.14. (continuación) Código final del sistema captura de imágenes de cámara digital

```
        else
            scanrow:=0;
            wrAddr<=0;
            state<=e0;
        end if;
    end case;
end if;
end process Captura_pixel;
end architecture behavioral;
```

Figura 13.14. (continuación) Código final del sistema captura de imágenes de cámara digital

ACTIVIDADES COMPLEMENTARIAS:

El alumno investigará:

1. El funcionamiento de los sistemas de captura de imágenes BAYER, FOVEON X3.
2. El modelo de color YUV.
3. Cuántos detectores de luz de color rojo, de color verde y de color azul hay en el ojo humano.
4. El alumno investigará el tamaño de imagen que captura la cámara de su teléfono celular.
5. El significado de binarización de imágenes.

BIBLIOGRAFÍA

- 1.- NE. Chávez.
Tutorial para prácticas en lenguaje VHDL
Facultad de Ingeniería, UNAM. 2013.
- 2.- NE. Chávez, MS. Guevara, V. Flores.
Prácticas de Diseño Digital Moderno
Facultad de Ingeniería, UNAM. 2017.
- 3.- J. Savage, G. Vázquez, NE. Chávez.
Diseño de Microprocesadores
Facultad de Ingeniería, UNAM. 2016.
- 4.-Pong. P.
FPGA prototyping by VHDL examples.
Wiley Interscience, pp. 163-182. 2015.
- 5.-Wilson. P.
Design Recipes for FPGAs.
Newnes, Oxford, pp. 209-228. 2015
- 6.-Pong P. Chu.
VGA Controller I: Graphic
Wiley Online Library. 2015
- 7.-Ultrasonic Ranging Module HC - SR04
ElecFreaks
Tech Support: services@elecfreaks.com.
<https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf>.
- 8.-Cámara. sensor OV7670
Omni Vision Technology Inc.
Advanced Information. Preliminary Datasheet. OV7670/OV7171CMOS VGA
(640x480) CAMERA CHIP TM with Omni Pixel® Technology. USA: Omni Vision
Technologies, Inc. (2005-10) [2016-07].
- 9.-Altera Corporation
User configurable logic data book
Altera Corporation. Santa Clara.2015.

10.-Xilinx ISE (Integrated Synthesis Environment)

ISE Design Suite: WebPACK Edition.

Xilinx Corporation. 2015.

11.-Coelho, David R.

The VHDL handbook

Kluwer Academic Publishers. Boston. 2011.

12.-Pardo, F. y Boluda, J. A.

VHDL. Lenguaje para síntesis y modelado de circuitos. 3ª edición.

Alfaomega. México. 2011.

13.-Rodríguez Andina, J. J., de la Torre Aranz, E. y Valdés Peña, M. D.

FPGAs Fundamentals, Advanced Features, and Applications in Industrial Electronics

CRC Press. Boca Raton. 2017