Taller 08: Controlador SVGA y visualización de objetos en pantalla.

Nicolás Andrés Gómez Ramírez
Departamento de ingeniería electrónica
Pontificia Universidad Javeriana
Bogotá, Colombia
nandres-gomez@javeriana.edu.co

Luis Alberto Munóz Rodríguez
Departamento de ingeniería electrónica
Pontificia Universidad Javeriana
Bogotá, Colombia
munoz_la@javeriana.edu.co

Ruslán Domínguez Ivanova
Departamento de ingeniería electrónica
Pontificia Universidad Javeriana
Bogotá, Colombia
ru-dominguez@javeriana.edu.co

***Abstract*— Este proyecto tiene como objetivo desarrollar un controlador de imagen en VGA mediante una arquitectura modular que integra la sincronización de señales (*ImageSync)* y la generación de colores en pantalla (*PixelGenerate)*. Así mismo, se realizó un código que muestra en pantalla un cuadrado que responde a movimientos leídos por un circuito externo, siendo un *Schmitt Trigger* para le lectura de un joystick y de un sistema anti rebotes para un botón. Finalmente, se logró mostrar tanto la imagen del cuadrado como visualizar la respuesta de este al movimiento del joystick.**

Keywords*— *VGA, RGB, ModelSim, TestBench, barrido horizontal, barrido vertical, píxel, paquetes.

I. INTRODUCCIÓN

Este taller busca desarrollar un controlador de imagen en VGA mediante una arquitectura modular que combina sincronización y generación de colores en pantalla. Los módulos principales son *ImageSync*, que gestiona las señales de sincronización y genera las coordenadas XY del píxel, y *PixelGenerate*, que asigna colores RGB a cada píxel según su posición. También se implementó un cuadrado rojo cuyo movimiento en pantalla se controla mediante un joystick.

El joystick se integra utilizando comparadores LM393 que gestionan los ejes horizontal y vertical del movimiento, permitiendo un control preciso del cuadrado. Adicionalmente, se utilizó un filtro pasa-bajos para eliminar rebotes en los botones y asegurar señales estables mediante un inversor con trigger Schmitt.

El diseño utiliza paquetes reutilizables como *BasicPackage* y *VgaPackage* para la gestión eficiente de datos y sincronización, junto con un contador parametrizable mediante *GralLimCounter*. La operación del sistema se sincroniza con un reloj de 50 MHz, permitiendo gestionar 1040 píxeles horizontalmente y 666 verticalmente, con ajustes en backporch y frontporch para mantener la estabilidad visual.

Finalmente, el sistema se validó mediante un testbench en ModelSim, confirmando que las señales y colores generados cumplen con las especificaciones VGA, garantizando un funcionamiento preciso y sin errores visuales.

II. DISEÑO DEL SISTEMA

Sabiendo que el objetivo de este taller es realizar un controlador de imagen en VGA, se presenta el diagrama de bloques de la *Figura 1*, el cual explica a grandes rasgos el comportamiento de los bloques más importantes del controlador y se muestran las señales de entrada y salida que contiene el sistema.

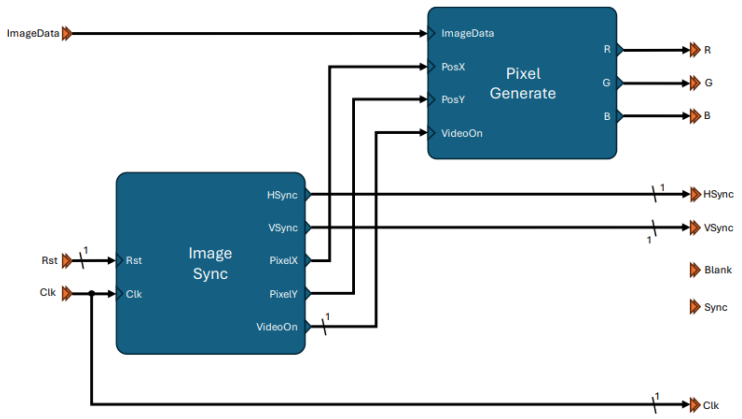

Figura 1. Diagrama en bloques general de un sistema digital

El bloque *ImageSync* se encarga de mantener la sincronización entre las señales horizontales y verticales, generando una coordenada XY que se mostrará en la pantalla. Por último, el bloque *PixelGenerate* toma como entrada la salida del anterior bloque y la convierte en color en formato RGB que tiene el píxel en la posición especificada anteriormente.

A. Descripción del funcionamiento de los códigos de cada bloque.

En primer lugar, se encuentra *BasicPackage (Figura 2)* el cual se encarga de contener en paquetes las señales que se utilizarán recurrentemente en el resto de los códigos. Junto con *BasicPackage* se crea el tipo de dato *ObjectT*, cuya función es contener las 3 señales de importancia (RGB) en un mismo grupo, para no tener que definirlas por separado. Cabe destacar que las señales R, G y B tendrán un tamaño de 8 bits acorde al estándar utilizado en este taller.

Además, se define la función *Int2sl* encargada de convertir un *Integer en un STD_LOGIC_VECTOR* con un tamaño de *size*. En el resto del código, se definen los subtipos que permitirán definir los vectores con un tamaño específico (desde 1 hasta 11 bits). Para finalizar, se implementa la función y se define la tarea que realizará (retornar un vector sin signo).

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

PACKAGE BasicPackage IS
    -- Declaración de la función Int2slv
    FUNCTION Int2slv (val : INTEGER; size : INTEGER) RETURN STD_LOGIC_VECTOR;

    -- Declaración del tipo ObjectT
    TYPE ObjectT IS RECORD
        R : STD_LOGIC_VECTOR(7 DOWNTO 0);
        G : STD_LOGIC_VECTOR(7 DOWNTO 0);
        B : STD_LOGIC_VECTOR(7 DOWNTO 0);
    END RECORD ObjectT;

    -- Definición de subtipos para vectores
    SUBTYPE uint01 IS STD_LOGIC;
    SUBTYPE uint02 IS STD_LOGIC_VECTOR(1 DOWNTO 0);
    SUBTYPE uint03 IS STD_LOGIC_VECTOR(2 DOWNTO 0);
    SUBTYPE uint04 IS STD_LOGIC_VECTOR(3 DOWNTO 0);
    SUBTYPE uint05 IS STD_LOGIC_VECTOR(4 DOWNTO 0);
    SUBTYPE uint06 IS STD_LOGIC_VECTOR(5 DOWNTO 0);
    SUBTYPE uint07 IS STD_LOGIC_VECTOR(6 DOWNTO 0);
    SUBTYPE uint08 IS STD_LOGIC_VECTOR(7 DOWNTO 0);
    SUBTYPE uint09 IS STD_LOGIC_VECTOR(8 DOWNTO 0);
    SUBTYPE uint10 IS STD_LOGIC_VECTOR(9 DOWNTO 0);
    SUBTYPE uint11 IS STD_LOGIC_VECTOR(10 DOWNTO 0);

END BasicPackage;

PACKAGE BODY BasicPackage IS
    -- Definición de la función Int2slv
    FUNCTION Int2slv (val : INTEGER; size : INTEGER) RETURN STD_LOGIC_VECTOR IS
    BEGIN
        RETURN STD_LOGIC_VECTOR(TO_UNSIGNED(val, size));
    END Int2slv;
END PACKAGE BODY;
```

Figura 2. Código en VHDL de BasicPackage

Posteriormente, se encuentra el código *VgaPackage* mostrado en la *Figura 3* se encarga de importar los tipos y subtipos definidos en el paquete del código anterior. Se crea el paquete *VgaPackage* que contiene los siguientes records:

- ***ColorT:*** Agrupa los canales de color de la VGA (rojo, verde y azul) definidos como vectores de 8 bits utilizando los subtipos de *BasicPackage*.
- ***VgaCtrlT*:** Agrupa las señales que controlan el funcionamiento de la sincronización de la VGA (Clk, Blank, Sync, Hsync y Vsync).

- ***SVGADataT***: Agrupa las variables que controlan el barrido horizontal y vertical de los píxeles en pantalla. En este *record*, se definen *HData* y *VData* que manejan los tiempos de espera horizontales y verticales respectivamente de las variables creadas en *SvgaDataT*.

Más precisamente, *HData* debe tardarse 1056 ciclos aproximadamente en recorrer de forma horizontal una línea, mientras que *VData* determina cuántas líneas se recorren para formar un recuadro o imagen (623 líneas) y, finalmente, ambas señales se convertirán en vectores utilizando la función *Int2slv*. Más adelante, se comprobará mediante *TestBenchs* si los tiempos de recorrido para *HData* y *VData* se cumplen.

```
 6  PACKAGE VgaPackage IS
 7
 8      TYPE ColorT IS RECORD
 9          R : uint08;
10          G : uint08;
11          B : uint08;
12      END RECORD ColorT;
13
14      TYPE VgaCtrlT IS RECORD
15          Clk     : uint01;
16          Blank   : uint01;
17          Sync    : uint01;
18          Hsync   : uint01;
19          Vsync   : uint01;
20      END RECORD VgaCtrlT;
21
22      TYPE SvgaDataT IS RECORD
23          Display : INTEGER;
24          FrontP  : INTEGER;
25          Retrace : INTEGER;
26          BackP   : INTEGER;
27      END RECORD SvgaDataT;
28
29      CONSTANT HData : SvgaDataT := (Display => 799, FrontP => 16, Retrace => 80, BackP => 160);
30      CONSTANT VData : SvgaDataT := (Display => 599, FrontP => 1, Retrace => 2, BackP => 21);
31
32      TYPE TimeStampT IS RECORD
33          Display    : uint11;
34          FrontPorch : uint11;
35          Retrace    : uint11;
36          FullScan   : uint11;
37      END RECORD TimeStampT;
38
39      CONSTANT HTime : TimeStampT := (Display    => (Int2slv((HData.Display), 11)),
40                                      FrontPorch => (Int2slv((HData.Display + HData.FrontP), 11)),
41                                      Retrace    => (Int2slv((HData.Display + HData.FrontP + HData.Retrace), 11)),
42                                      FullScan   => (Int2slv((HData.Display +
43                                                              HData.FrontP +
44                                                              HData.Retrace +
45                                                              HData.BackP), 11)));
46
47      CONSTANT VTime : TimeStampT := (Display    => (Int2slv((VData.Display), 11)),
48                                      FrontPorch => (Int2slv((VData.Display + VData.FrontP), 11)),
49                                      Retrace    => (Int2slv((VData.Display + VData.FrontP + VData.Retrace), 11)),
50                                      FullScan   => (Int2slv((VData.Display +
51                                                              VData.FrontP +
52                                                              VData.Retrace +
```

Figura 3. Código en VHDL de VgaPackage.

Después, hallamos el módulo *PixelGenerate* que genera el color de un píxel en función de los datos de entrada y las coordenadas del píxel. Utiliza los valores RGB proporcionados por la entrada *ImageData*, así como las coordenadas X (PosX) e Y (PosY) del píxel en la pantalla. La señal *VideoOn* controla si los colores se muestran o se apagan.

Cuando *VideoOn* está activa ('1'), los canales RGB se asignan con su valor máximo (x"FF"), generando un píxel blanco. Si *VideoOn* está inactiva ('0'), los valores de los canales RGB se ponen en cero, apagando el color del píxel. El módulo utiliza tipos definidos en los paquetes *BasicPackage* y *VgaPackage*. La entrada *ImageData* es de tipo *ObjectT*, que agrupa los canales RGB, mientras que la salida RGB utiliza el tipo *ColorT*. Estos tipos mejoran la claridad y modularidad del código.

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE WORK.BasicPackage.ALL;
USE WORK.VgaPackage.ALL;

ENTITY PixelGenerate IS
    PORT (
        ImageData : IN ObjectT;   -- Datos de la imagen con componentes R, G y B
        PosX      : IN uint11;      -- Coordenada X del pixel
        PosY      : IN uint11;      -- Coordenada Y del pixel
        VideoOn   : IN STD_LOGIC;   -- Señal que indica si el video está activo
        RGB       : OUT ColorT      -- Salida con el color del pixel
    );
END ENTITY PixelGenerate;

ARCHITECTURE Behavioral OF PixelGenerate IS
BEGIN
    PROCESS (ImageData, PosX, PosY, VideoOn)
    BEGIN
        IF VideoOn = '1' THEN
            -- Asignamos los colores correspondientes de la entrada ImageData
            RGB.R <= x"FF";
            RGB.G <= x"FF";
            RGB.B <= x"FF";
        ELSE
            -- Si VideoOn está en '0', apagamos los colores
            RGB.R <= (OTHERS => '0');
            RGB.G <= (OTHERS => '0');
            RGB.B <= (OTHERS => '0');
        END IF;
    END PROCESS;
END ARCHITECTURE Behavioral;
```

Figura 4. Código en VHDL de PixelGenerate.

El módulo *GralLimCounter* es un contador parametrizable que cuenta hacia arriba o hacia abajo según las señales de control *Up* y *Dwn*. Funciona con un tamaño configurable de bits *Nbits* y utiliza un reloj *Clk* para sincronizar las actualizaciones. Puede habilitarse o detenerse mediante la señal *Ena* y cuenta con *resets* sincrónico (MR) y asíncrono (SR) para reiniciarse a cero. El contador se detiene si alcanza el valor máximo definido por el parámetro *Limit*, activando la señal *MaxCount*; de manera similar, si llega a cero, activa la señal *MinCount*. La salida del contador se convierte en un *STD_LOGIC_VECTOR* para ser utilizada en otros módulos.

```vhdl
ENTITY GralLimCounter IS
    GENERIC (Nbits : INTEGER := 11); -- Se ajusta el tamaño del contador a 11 bits
    PORT (
        Clk       : IN STD_LOGIC;
        MR        : IN STD_LOGIC;  -- Reset sincrónico
        SR        : IN STD_LOGIC;  -- Reset asincrono
        Ena       : IN STD_LOGIC;  -- Enable del contador
        Up        : IN STD_LOGIC;  -- Dirección del contador
        Dwn       : IN STD_LOGIC;
        Limit     : IN STD_LOGIC_VECTOR(Nbits-1 DOWNTO 0); -- Límite del contador
        MaxCount  : OUT STD_LOGIC;
        MinCount  : OUT STD_LOGIC;
        Count     : OUT STD_LOGIC_VECTOR(Nbits-1 DOWNTO 0) -- Valor del contador
    );
END ENTITY;

ARCHITECTURE rtl OF GralLimCounter IS
    CONSTANT ONES  : UNSIGNED(Nbits-1 DOWNTO 0) := (OTHERS => '1');
    CONSTANT ZEROS : UNSIGNED(Nbits-1 DOWNTO 0) := (OTHERS => '0');

    SIGNAL count_s       : UNSIGNED(Nbits-1 DOWNTO 0);
    SIGNAL count_next    : UNSIGNED(Nbits-1 DOWNTO 0);
    SIGNAL maxCount_s    : STD_LOGIC; -- Señal intermedia para MaxCount
BEGIN

    -- Lógica de siguiente estado del contador
    count_next <= (OTHERS => '0') WHEN SR = '1' ELSE
                  (OTHERS => '0') WHEN MR = '1' ELSE
                  (OTHERS => '0') WHEN (Ena = '1' AND Up = '1' AND maxCount_s = '1') ELSE
                  count_s + 1     WHEN (Ena = '1' AND Up = '1') ELSE
                  count_s - 1     WHEN (Ena = '1' AND Up = '0') ELSE
                  count_s;
```

```vhdl
    -- Proceso secuencial para actualizar el valor del contador
    PROCESS(Clk, MR)
    BEGIN
        IF (MR = '1') THEN
            count_s <= ZEROS;
        ELSIF rising_edge(Clk) THEN
            IF (Ena = '1') THEN
                count_s <= count_next;
            END IF;
        END IF;
    END PROCESS;

    -- Asignación de la salida del contador
    Count <= STD_LOGIC_VECTOR(count_s);

    -- Señales de max_tick y min_tick
    maxCount_s <= '1' WHEN count_s = UNSIGNED(Limit) ELSE '0';
    MaxCount <= maxCount_s;
    MinCount <= '1' WHEN count_s = ZEROS ELSE '0';

END ARCHITECTURE;
```

Figura 5. Código en VHDL de GranLimCounter.

Luego, se encuentra el módulo *ImageSync* que utiliza contadores horizontales y verticales, *HCount* y *VCount*, para determinar las coordenadas actuales del píxel en la pantalla, que se asignan a las señales *PixelX* y *PixelY* cuando la señal *VideoOn* está activa. Esta señal se activa solo si los contadores se encuentran dentro del rango de visualización. Las señales de sincronización *HSync* y VSync se generan comparando los contadores con los valores de tiempo definidos, como *FrontPorch* y *Retrace*, asegurando una transición correcta entre las fases del ciclo VGA. Además, se emplean dos instancias del módulo *GralLimCounter* para gestionar los ciclos horizontal y vertical, manteniendo la sincronización precisa con el reloj del sistema y permitiendo que cada línea y cuadro de la pantalla se dibujen en el momento adecuado.

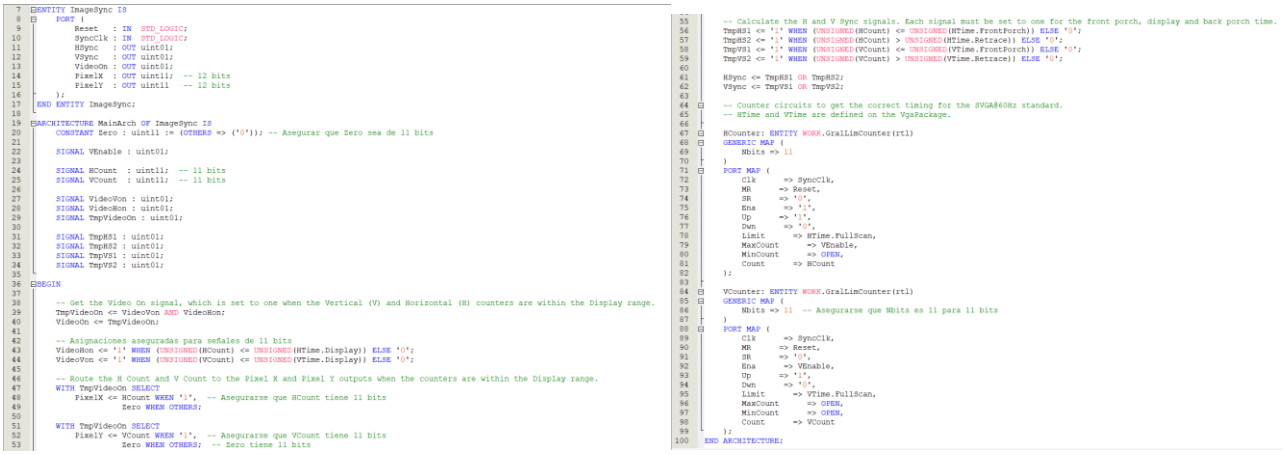

Figura 6. Código en VHDL de ImageSync.

El módulo *TopLevel* es la entidad principal del sistema que integra los módulos *ImageSync* y *PixelGenerate* para gestionar la sincronización y generación de colores en un sistema VGA. Utiliza señales internas como *PixelX* y *PixelY* para indicar las coordenadas del píxel en la pantalla, y la señal *VideoOn* para controlar si el video está activo. La señal de reloj *SyncClk* y el *Reset* sincronizan los módulos internos. *ImageSync* se encarga de generar las señales de sincronización horizontal *HSync* y vertical *VSync*, así como las coordenadas del píxel actual. Luego, el módulo PixelGenerate utiliza estas coordenadas y los datos de la imagen ImageData para definir el color del píxel a través de la salida RGB. Este diseño modular permite una gestión eficiente del video mediante la conexión ordenada de los submódulos en una arquitectura clara y estructurada.

```vhdl
ENTITY TopLevel IS
    PORT (
        Reset     : IN  STD_LOGIC;
        SyncClk   : IN  STD_LOGIC;
        ImageData : IN  ObjectT;    -- Datos de la imagen con componentes R, G y B
        HSync     : OUT uint01;
        VSync     : OUT uint01;
        RGB       : OUT ColorT
    );
END ENTITY TopLevel;

ARCHITECTURE Behavioral OF TopLevel IS

    -- Señales internas para conectar los módulos
    SIGNAL VideoOn   : uint01;
    SIGNAL PixelX    : uint11;  -- Coordenada X del píxel
    SIGNAL PixelY    : uint11;  -- Coordenada Y del píxel

BEGIN

    -- Instancia del módulo ImageSync
    ImageSync_inst : ENTITY WORK.ImageSync
    PORT MAP (
        Reset   => Reset,
        SyncClk => SyncClk,
        HSync   => HSync,
        VSync   => VSync,
        VideoOn => VideoOn,
        PixelX  => PixelX,    -- Salida PixelX
        PixelY  => PixelY     -- Salida PixelY
    );

    -- Instancia del módulo PixelGenerate
    PixelGenerate_inst : ENTITY WORK.PixelGenerate
    PORT MAP (
        ImageData => ImageData, -- Imagen de entrada con datos de color
        PosX      => PixelX,    -- Entrada de coordenada X del píxel desde ImageSync
        PosY      => PixelY,    -- Entrada de coordenada Y del píxel desde ImageSync
        VideoOn   => VideoOn,   -- Entrada VideoOn desde ImageSync
        RGB       => RGB         -- Salida componente azul
    );

END ARCHITECTURE Behavioral;
```

Figura 7. Código en VHDL de TopLevel.

Así mismo, se tiene *TopLevel_tb* como se puede ver en la Figura 8. Este código, declara las señales necesarias para conectar con el módulo mayor *TopLevel* y simular su comportamiento, incluyendo un reloj de sincronización con un período de 20 ns. El proceso de estímulo aplica un reset al sistema al inicio (por 40 ns) para reiniciar el estado del módulo, seguido de una ejecución continua del sistema durante 24,000 ns para observar su comportamiento. Este *testbench* permite evaluar la respuesta del sistema bajo condiciones

controladas, asegurando que el módulo *TopLevel* sincronice las señales y genere los colores esperados en cada momento de la simulación.

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE WORK.BasicPackage.ALL;
USE WORK.VgaPackage.ALL;

ENTITY TopLevel_tb IS
END ENTITY TopLevel_tb;

ARCHITECTURE behavior OF TopLevel_tb IS

    -- Declaración de señales para conectar con el TopLevel
    SIGNAL Reset     : STD_LOGIC := '0';
    SIGNAL SyncClk   : STD_LOGIC := '0';
    SIGNAL ImageData : ObjectT;            -- Datos de imagen RGB de entrada
    SIGNAL HSync     : uint01;
    SIGNAL VSync     : uint01;
    SIGNAL RGB       : ColorT;


    -- Constante para definir el color blanco (RGB máximo)
    CONSTANT White : ColorT := (R => X"FF", G => X"FF", B => X"FF");

    -- Reloj de sincronización (50 MHz, periodo 20 ns)
    CONSTANT SyncClkPeriod : TIME := 20 ns;

BEGIN

    -- Instancia del DUT (Device Under Test)
    DUT: ENTITY WORK.TopLevel
    PORT MAP (
        Reset     => Reset,
        SyncClk   => SyncClk,
        ImageData => ImageData,
        HSync     => HSync,
        VSync     => VSync,
        RGB       => RGB
    );

    -- Generación del reloj de sincronización
    SyncClk_Process: PROCESS
    BEGIN
        WHILE TRUE LOOP
            SyncClk <= '0';
            WAIT FOR SyncClkPeriod / 2;
            SyncClk <= '1';
            WAIT FOR SyncClkPeriod / 2;
        END LOOP;
    END PROCESS;

    -- Proceso para aplicar estímulos
    Stimulus_Process: PROCESS
    BEGIN
        -- Inicialización de las señales
        Reset <= '1';              -- Aplicar reset
        WAIT FOR 40 ns;            -- Espera durante 2 ciclos de reloj
        Reset <= '0';              -- Liberar reset

        -- Simulación durante 24,000 ns
        WAIT FOR 24000 ns;

        -- Finaliza la simulación
        WAIT;
    END PROCESS;

END ARCHITECTURE behavior;
```

Figura 8. Código en VHDL de TopLevel_tb.

III. Pruebas de Funcionamiento

Con base en la simulación ejecutada del TopLevel_tb en ModelSim, es posible evaluar el comportamiento preciso de las señales críticas del sistema, en especial HSync y VSync, verificando si las fases del ciclo de sincronización horizontal (Display, Front Porch, Retrace y Back Porch) se ejecutan correctamente y conforme a los tiempos esperados. Estas fases son esenciales para la estabilidad visual, ya que controlan cuándo deben mostrarse los píxeles y cuándo la pantalla se prepara para un nuevo ciclo de dibujo. A partir de las ondas generadas durante la simulación, se capturaron los tiempos de cada fase y se convirtieron en ciclos de reloj para compararlos con los valores predeterminados, asegurando así que el diseño cumpla con las especificaciones de temporización del protocolo VGA o cualquier estándar correspondiente. La sincronización precisa de estas fases evita defectos visuales, como saltos o distorsiones, y permite validar el comportamiento de la señal RGB, garantizando que los colores de los píxeles se generen correctamente según las coordenadas proporcionadas por ImageSync y los datos de entrada de ImageData.

Para asegurar un correcto proceso de impresión, es fundamental comprender los tiempos que la señal HSync requiere en cada una de sus fases. Esto implica realizar la conversión entre los ciclos de reloj y la duración de la señal en nanosegundos. En esta primera simulación, se utilizaron los valores de HSync definidos en ciclos de reloj como: `CONSTANT HData : SvgaDataT := (Display => 799, FrontP => 16, Retrace => 80, BackP => 160);`. Con base en estos datos, es posible calcular el tiempo esperado para cada fase. En el primer caso ilustrado en la Figura 9, se midió la duración de las fases **Display** y **Front Porch**, con 799 y 16 ciclos de reloj, respectivamente. Sumando estos valores (815 ciclos) y multiplicándolos por su equivalente en tiempo por ciclo (20 ns), se obtiene:

$$815(Ciclos\ de\ reloj) \cdot 20ns = 16300ns$$

Como se puede ver, este tiempo es coherente al tiempo diferencial entre ambos cursores, por lo que se puede afirmar que esta primera fase de la señal Hsync funciona correctamente.

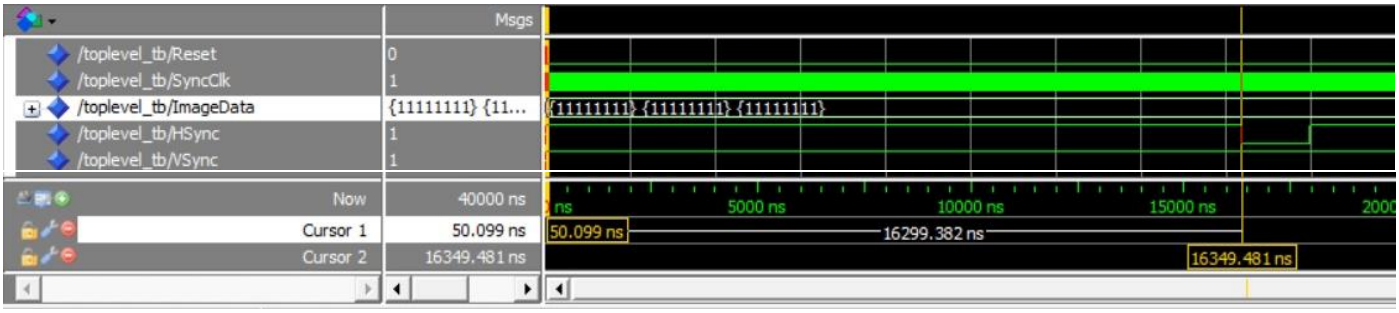

Figura 9. Medición de tiempos con TestBench en ModelSim para Display y FrontP.

Aplicamos el mismo proceso para evaluar el tiempo de duración de *Retrace* y de un periodo completo de *Hsync* en las Figuras 10 y 11 respectivamente, obteniendo así:

$$Tiempo\ de\ Retrace => 80(Ciclos\ de\ reloj) \cdot 20ns = 1600ns$$

$$Tiempo\ de\ Hsync => (799 + 16 + 80 + 160)(Ciclos\ de\ reloj) \cdot 20ns = 21100ns$$

Como se puede ver en las simulaciones, los tiempos y el conteo de las señales, así como el cambio en el color de la pantalla, funcionan correctamente y respetan la lógica del programa.

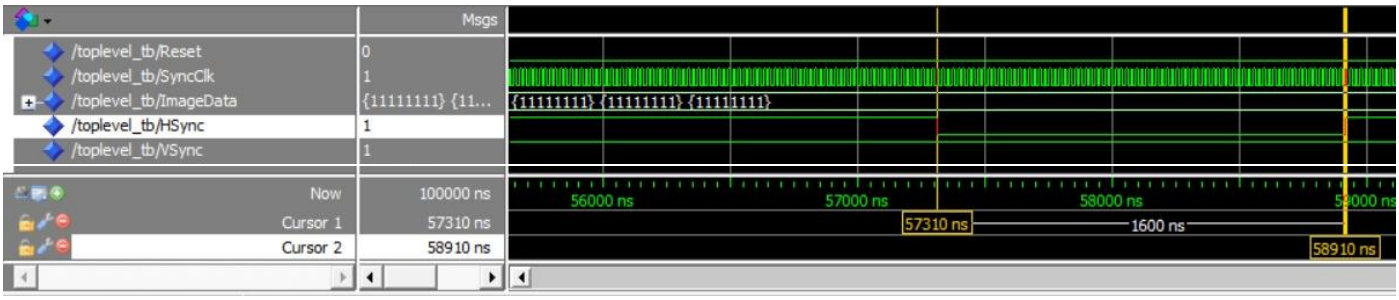

Figura 10. Medición de tiempos con TestBench en ModelSim para Retrace.

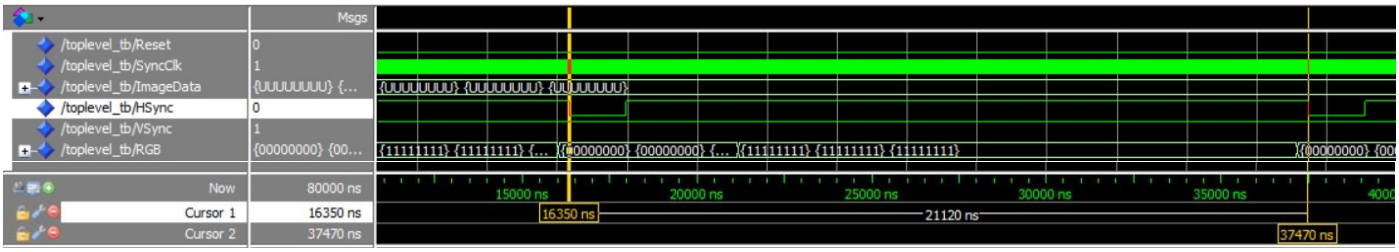

Figura 11. Medición de tiempos con TestBench en ModelSim para un periodo de Hsync.

IV. OBJETOS EN PANTALLA

A. Hardware

Con la finalidad de capturar un buen movimiento del cuadrado como respuesta a un control (Joystick) movimiento del cuadrado en pantalla, se realizaron los siguientes ajustes en hardware y sus posteriores comprobaciones mediante el osciloscopio.

la configuración de la *Figura 12*, utiliza dos comparadores del circuito integrado LM393, los cuales evalúan el voltaje proveniente de un joystick. El LM393 es un comparador doble que compara las señales de sus entradas (+ y -) y genera un cambio en el estado del voltaje de salida, dependiendo de si la entrada no inversora (+) es mayor o menor que la entrada inversora (-). Cada comparador controla un eje del joystick: uno para el eje horizontal (izquierda-derecha) y otro para el eje vertical (arriba-abajo). Las resistencias R1, R2, R4 y R5 forman divisores de voltaje que fijan los niveles de referencia (VCompH y VCompL). Dependiendo de la posición del joystick, las salidas de los comparadores (HOutputH y LOutputL) cambian su estado, permitiendo controlar el movimiento del objeto en pantalla. Esto asegura una respuesta precisa según la dirección del joystick en ambos ejes.

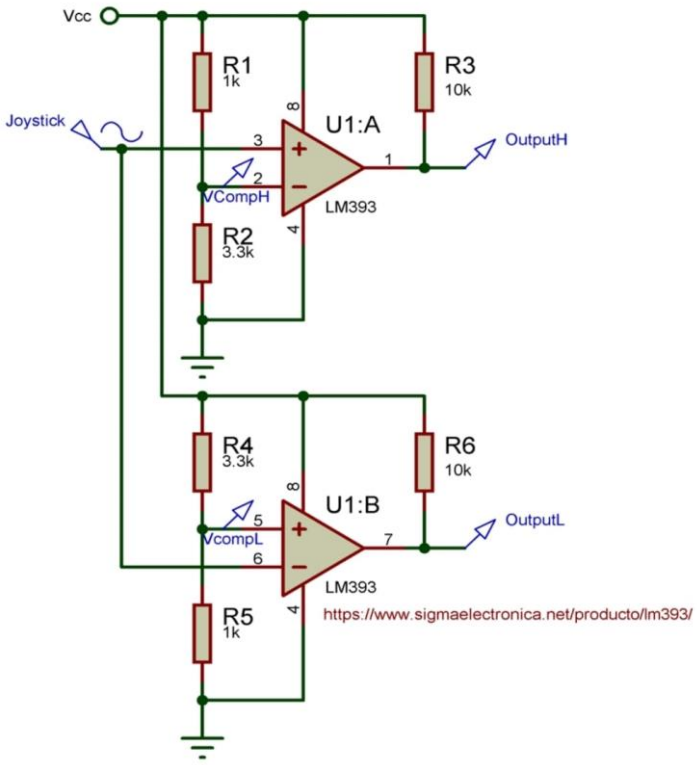

Figura 12. Circuito de comparadores para la toma de datos de un joystick.

El circuito de la *Figura 13*, es un filtro pasa-bajos diseñado para eliminar los rebotes de un botón, utilizando una resistencia R7 y un condensador C1 que forman una red RC. Al presionar un botón mecánico, es común que la señal generada tenga fluctuaciones rápidas (rebotes), lo que podría causar lecturas erróneas en sistemas digitales. Este filtro atenúa las señales de alta frecuencia no deseadas y deja pasar únicamente los cambios lentos y estables, es decir, los pulsos reales. El integrado CD40106, que es un inversor con trigger Schmitt, se utiliza para limpiar y estabilizar la señal filtrada. Esto garantiza que la salida (FinalButton) tenga transiciones claras y sin ruido al detectar las pulsaciones del botón.

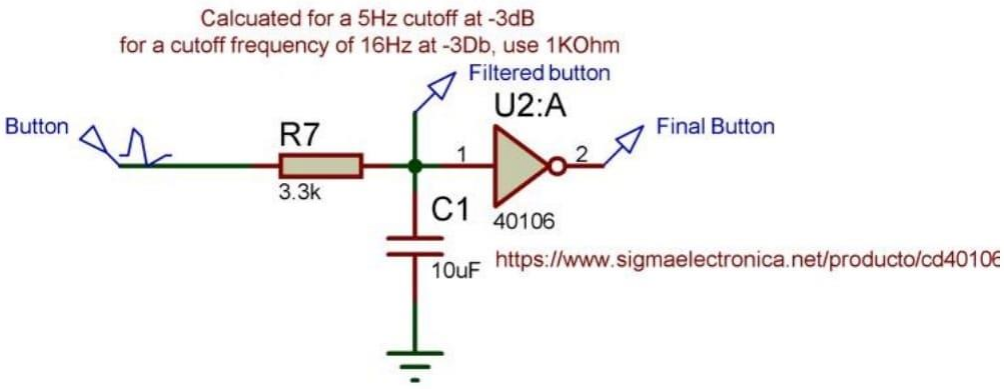

Figura 13. Filtro pasa-bajos para eliminar rebotes en un botón.

Para hacer la comprobación del circuito comparador del Joystick como se muestra en la *Figura 14*, se comprobó desde el osciloscopio que la señal del comparador se fuera a alto o a bajo a partir de un punto en específico definido por el voltaje de comparación en RX y en RY. Este circuito muestra como el comparador funciona correctamente para la salida de alto. Es importante mencionar que este circuito debe ser duplicado si se quieren tener valores tanto de alto como de bajo y como de izquierda y de derecha, aplicando esta comprobación 4 veces para cada caso.

Figura 13. Comprobación de los comparadores para la toma de datos de un joystick.

B. Software

Se tuvo problemas con el código anteriormente implementado, pues al subirlo a la FPGA e intentar mostrarlo por medio del puerto VGA a la pantalla, mostraba una pantalla negra (inclusive esperando una blanca) y al aplicar el botón de reset anteriormente descrito, se notó un leve cmbio en la coloración de la pantalla, por lo cual, mostraba algo, pero no era cercano a lo que se esperaba.

Dados los anteriores problemas, el equipo decidió mirar la referencia [3] que grosso modo hacía lo mismo que de la anterior implementación siguiendo la guía solo que de una manera un tanto más compacta.

- Proceso principal de sincronización y dibujo.

Este proceso se basa en el seguimiento de los flancos de bajada y de subida del reloj de 50Mhz (20ns) de la FPGA.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  ENTITY VGA IS
7     PORT(
8        CLOCK_50          : IN STD_LOGIC; --reloj fpga
9        K_Down            : IN STD_LOGIC_VECTOR(3 DOWNTO 0); --pulsadores
10
11       VGA_HS,VGA_VS     :OUT STD_LOGIC; -- senales de sincronia horizontal y vertical
12       VGA_R,VGA_B,VGA_G: OUT STD_LOGIC_VECTOR(7 downto 0); -- senales para los colores RGB
13
14       VGA_SYNC_N        : OUT STD_LOGIC:='0';
15       VGA_BLANK_N       : OUT STD_LOGIC:='1';
16       VGA_CLK           : OUT STD_LOGIC       -- reloj para el controlador VGA
17
18    );
19 END VGA;
20
21
22 ARCHITECTURE MAIN OF VGA IS
23 -- definir arquitectura
24
25   COMPONENT SYNC IS
26     PORT(
27        CLK  : IN STD_LOGIC;
28        HSYNC: OUT STD_LOGIC;   --sincronia horizontal
29        VSYNC: OUT STD_LOGIC;   --sincronia vertical
30        R:     OUT STD_LOGIC_VECTOR(7 downto 0); --rojo 8 bits
31        G:     OUT STD_LOGIC_VECTOR(7 downto 0); -- verde
32        B:     OUT STD_LOGIC_VECTOR(7 downto 0);  --azul
33        KEYS: IN STD_LOGIC_VECTOR(3 DOWNTO 0) --pulsadores
34     );
35   END COMPONENT SYNC;
```

```
37       C1: SYNC PORT MAP (CLOCK_50,VGA_HS,VGA_VS,VGA_R,VGA_G,VGA_B,K_Down);
38
39       VGA_CLK<=CLOCK_50;
40
41   END MAIN;
```

Figura 14. Cuerpo de la VGA

```
44  PROCESS(CLK)
45    BEGIN
46    IF (clk'event and clk ='1') THEN
47       ----
48
49       ----
50       IF (HPOS<1040) THEN
51        HPOS<=HPOS+1;
52       ELSE
53           HPOS<=0;
54           IF(VPOS<666)THEN
55              VPOS<=VPOS+1;
56           ELSE
57              VPOS<=0;
58           END IF;
59       END IF;
60       -- sincronia horizontal
61       IF(HPOS>56 AND HPOS<176)THEN----HSYNC
62          HSYNC<='0';
63       ELSE
64          HSYNC<='1';
65       END IF;
66       -- sincronia vertical
67       IF(VPOS>37 AND VPOS<43)THEN-----vsync
68          VSYNC<='0';
69       ELSE
70          VSYNC<='1';
71       END IF;
72       -- rgb por fuera del dysplay
```

```
74       IF((HPOS>0 AND HPOS<240) OR (VPOS>0 AND VPOS<66))THEN
75          R<=(others=>'0');
76          G<=(others=>'0');
77          B<=(others=>'0');
78       END IF;
79       ---
80       -- vamos a dibujar el cuadro cuando draw=1
81       IF(DRAW='1')THEN
82          R<=(others=>'1');
83          G<=(others=>'0');
84          B<=(others=>'0');
85       ELSE
86          R<=(others=>'0');
87          G<=(others=>'0');
88          B<=(others=>'0');
89       END IF;
90       --mover el cuadro solo si estamos en (0,0)
91       IF(VPOS=0 AND HPOS=0) THEN
92          IF(KEYS(0)='0')THEN
93             SQ_X<=SQ_X+5;
94          END IF;
95
96          IF(KEYS(1)='0')THEN
97             SQ_X<=SQ_X-5;
98          END IF;
99
100         IF(KEYS(2)='0')THEN
101            SQ_Y<=SQ_Y-5;
102         END IF;
103
104         IF(KEYS(3)='0')THEN
105            SQ_Y<=SQ_Y+5;
106         END IF;
107      END IF;
```

Figura 15. Proceso principal de sincronización y dibujo para mostrar los objetos en pantalla.

La condición IF (clk'event AND clk = '1') garantiza que el proceso se ejecute en el flanco de subida del reloj. Esto asegura que la lógica se ejecute de forma sincrónica con el reloj de 50 MHz. De la línea 50 a la línea 58 se tiene el incremento de las posiciones horizontal y vertical anteriormente descritas, nótese que para este apartado se usó un total de 1040 píxeles de manera horizontal y otro total de 666 para el tope vertical y que los valores de backporche y frontporche también cambiaron, pero se sigue usando la misma lógica descrita en el primer apartado, (línea 661 y 67).

Para establecer el color negro fuera del área visible (línea 74 a la 78) fue tan simple como comparar si el valor de HPOS es mayor a 0 y menor a 240, lo mismo con VPOS entre 0 y 66, estableciendo que todos los bits estén en ‘0’. Para dibujar el cuadradito rojo (líneas 81 - 89) se estableció la señal DRAW en ‘1’ y se ajustaron los valores de RGB para que fuera rojo.

Para el control de movimiento del cuadrado (91 – 107) se verifica si la posición actual es (0,0), es decir, si se ha completado un cuadrado completo, Si se presiona uno de los pulsadores (cuando un bit de KEYS está en '0'), se actualizan las coordenadas SQ_X y SQ_Y: KEYS(0) = '0': Mueve el cuadrado 5 píxeles a la derecha. KEYS(1) = '0': Mueve el cuadrado 5 píxeles a la izquierda. KEYS(2) = '0': Mueve el cuadrado 5 píxeles hacia arriba KEYS(3) = '0': Mueve el cuadrado 5 píxeles hacia abajo. Claramente, estos botones serán el input del hoystick anteriormente descrito en la sección de hardware.

V. CONCLUSIONES

- La sincronización de las señales *HSync* y *VSync*, junto con la generación precisa de las coordenadas y el manejo adecuado de los datos de imagen, garantizan que el sistema funcione de manera estable. La validación realizada demuestra que el código es capaz de imprimir en pantalla un color blanco uniforme, confirmando así la integridad del diseño y su capacidad para gestionar correctamente la visualización de datos.

- El uso del circuito basado en comparadores LM393 permite evaluar y convertir las variaciones de voltaje del joystick en señales digitales claras para controlar con precisión el movimiento en ambos ejes (horizontal y vertical) de un objeto en pantalla. Cada comparador se configura con divisores de voltaje para establecer niveles de referencia adecuados, asegurando que las transiciones de la señal ocurran de manera estable y precisa según la posición del joystick.

- La implementación del filtro RC junto con el inversor Schmitt (CD40106) es fundamental para garantizar la detección correcta de pulsaciones de botones, eliminando los rebotes mecánicos que podrían generar señales erróneas.

- Los problemas iniciales con la visualización en la FPGA, como la pantalla negra inesperada y los cambios leves en la coloración tras el reset, llevaron al equipo a investigar y optimizar la implementación del código. Al adaptar la lógica siguiendo una referencia más compacta y eficiente, se logró mejorar la sincronización del dibujo en pantalla, utilizando un reloj de 50 MHz y ajustando los valores de backporch, frontporch y coordenadas. Este proceso permitió corregir la representación del cuadrado rojo y su movimiento en pantalla, alineando las funciones del joystick con la actualización de los píxeles, logrando así un control preciso del objeto dibujado.

REFERENCIAS

[1] Díaz R., “08 – Taller SVGA”

[2] Manual de uso de la FPGA “DE2_115_User_manual”

[3] Orlando M., “Controlador de VGA en VHDL: Sistemas Digitales II”

[4] Ru. (10-18-2024). Demostración del funcionamiento del taller de VGA [Video]. YouTube. https://youtu.be/fA4BiLyqKCU?si=9G0qyGzLgdXfi83s