

**UNIVERSIDAD TECNOLÓGICA DE LA HABANA**

**JOSÉ ANTONIO ECHEVERRÍA**



**FACULTAD DE INGENIERÍA AUTOMÁTICA Y BIOMÉDICA**

**MAESTRÍA EN SISTEMAS DIGITALES**

# **DESARROLLO DE UN MÓDULO VGA PARA BUS PLB DE MICROBLAZE**

**AUTORES:**

Ing. Frank Hernández Castro

Ing. Alejandro J. Pino Moya

**LA HABANA**

Año 2019

## RESUMEN

Este texto aborda el proceso de diseño de un controlador VGA para luego ser empleado como un módulo IP más interconectado a un sistema empotrado sobre FPGA de Xilinx. Para el trabajo, se realiza el diseño de un sistema híbrido hardware/software basado en el procesador *MicroBlaze v8.00.b* empleando variedad de módulos IP que ofrece el fabricante además del controlador VGA desarrollado manualmente como periféricos del procesador, con ayuda de las herramientas que ofrece el entorno de EDK (*Embedded Development Kit*) de Xilinx. El texto aborda el desarrollo particular del módulo VGA, sus partes y funcionalidades a partir de su integración al flujo de diseño y su posterior utilización dentro de una aplicación validada sobre la placa de desarrollo *Spartan 3A Starter Kit* de Xilinx.

# TABLA DE CONTENIDOS

RESUMEN.....	II
TABLA DE CONTENIDOS .....	III
INTRODUCCIÓN .....	I
MODULO IP VGA DESARROLLADO .....	1
1.1 Descripción general de la interfaz VGA.....	1
1.1.1. Capa física .....	2
1.1.2. Teoría de operación .....	5
1.1.3. Formato VGA 640x480 a 60Hz .....	8
1.2. Módulo IP VGA desarrollado .....	9
1.2.1. Características Generales .....	9
1.2.2. Estructura interna y operación.....	12
1.2.2.1. Bloque VGA_sync_unit.....	14
1.2.2.2. Bloque VGA_module_self_test .....	20
1.2.2.3. Bloque VGA_rotary_encoder .....	22
1.2.2.4. Bloque VGA_outputMux .....	28
1.2.3. Funcionalidades .....	29
1.2.3.1. Modo manual .....	34
1.2.3.2. Modo automático .....	37
1.2.3.3. Comandos de operación.....	37
1.3. Utilización de periféricos para MicroBlaze .....	39
1.4. Programa de Aplicación .....	43
1.4.1. Características Generales .....	43
1.4.2. Tareas del programa principal .....	44
1.4.3. Atención a comandos .....	48
1.4.3.1. Comandos simples .....	50
1.4.3.2. Comandos compuestos.....	52
1.4.3.3. Comandos de imagen .....	55
1.5. Rutinas de Interrupción .....	58
1.5.1. ISR switches .....	58
1.5.2. ISR Timer .....	61
1.5.3. ISR de UART .....	61
CONCLUSIONES .....	65
REFERENCIAS BIBLIOGRAFICAS.....	66

<b>ANEXOS .....</b>	<b>67</b>
---------------------	-----------

## INTRODUCCIÓN

VGA (video graphics array) constituye un estándar para la visualización de imágenes y video introducido a finales de la década de 1980 en el diseño del hardware para la visualización en los primeros ordenadores IBM PCs y que actualmente se encuentra muy difundido a nivel mundial en los gráficos de cualquier PC y monitores actuales. La *Fig. 1* muestra el aspecto del conector DB15 asociado.

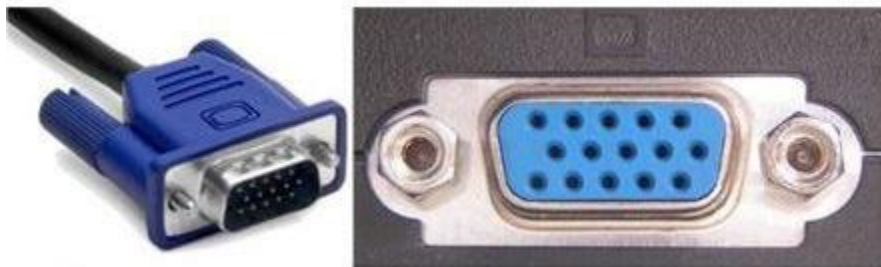


Fig. 1 Aspecto del conector físico DB15 empleado para VGA.

Hoy en día VGA se encuentra posicionado como el primer estándar que debe incluir todo hardware para procesar cualquier información de video. Se trata de una conexión analógica a través de un conector DB15 que cuenta con 15 pines (aunque no siempre se empleen todos) para transmitir hacia la pantalla los colores rojo, verde y azul, además de otras señales para la sincronización igualmente importantes, pero sin permitir la transmisión de ninguna señal de audio o video hacia el visualizador. No obstante, el estándar ofrece disímiles ventajas, entre ellas, la más significativa quizás, es precisamente su alto grado de difusión en comparación con otros estándares como HDMI o DVI, dado que cualquier dispositivo compatible con el estándar es mucho más barato que sus similares más contemporáneos. Esto permite incluso que, aun teniendo una tarjeta gráfica anticuada en nuestro ordenador, sea posible la visualización mediante cualquier televisor o monitor que lo integre sin problemas.

Sin embargo, dado el carácter analógico del mismo, la calidad de la imagen siempre dependerá de la calidad del cable y la longitud del mismo, por lo que su calidad no es precisamente del mayor estándar actual. No obstante, el estándar en cuestión y su interfaz física independientemente de lo anticuada que pueda parecer, debido a su acelerada evolución en la actualidad se emplea en resoluciones de video de hasta 1080p. Precisamente, mientras que el ancho de banda de transmisión de VGA permite lograr resoluciones aún mayores, la limitación radica en el compromiso que existe entre la frecuencia de refrescamiento de las señales que componen la interfaz y la resistividad total del cable VGA.

Dada la naturaleza propia del estándar VGA y de los modernos visualizadores existentes, uno de los problemas más comunes en el diseño de los circuitos digitales requeridos para la visualización es precisamente las altas frecuencias de operación necesarias según la cantidad de píxeles a manejar en correspondencia con la resolución a la que se desee visualizar. Esto provoca que, a partir de cierto punto los algoritmos de cómputo empleados para la operación requieran una plataforma hardware más potente para poder llevar a cabo el refrescamiento necesario para visualizar adecuadamente todo el conjunto de píxeles que conforma la imagen; lo que incita al diseño de hardware completamente dedicado a estas funciones independientemente del sistema de control general existente en la aplicación.

Aunque el tema en cuestión ha sido ampliamente abordado en la actualidad, empleando diferentes enfoques, el presente trabajo aborda precisamente la implementación de un controlador VGA digital basado en hardware reconfigurable (FPGA) compatible con un sistema de procesamiento empotrado basado en *MicroBlaze* en su versión 8.00.b para bus PLB de Xilinx, específicamente validado en la placa de desarrollo de FPGA *Spartan-3A Starter Kit* del mismo fabricante. El desarrollo consiste en un módulo IP que emplea una resolución de 640x480 píxeles y una tasa de refrescamiento de 60 Hz para la visualización, además de algunos registros internos para su configuración y control.

## MODULO IP VGA DESARROLLADO

### 1.1 Descripción general de la interfaz VGA

La interfaz VGA permite la manipulación de los visualizadores compatibles con solo 5 hilos fundamentales que manejan siempre señales netamente analógicas, lo cual no ha variado mucho desde su creación. De estos 5 hilos, 3 se encuentran asociados a cada color que compone un pixel, dígase el Rojo, el Verde y el Azul; y los otros 2 manejan las señales de sincronismo horizontal y vertical para la pantalla -véase **Teoría de operación**-. Para el caso de los cables asociados a los colores, cada uno por separado indicará al visualizador la tonalidad del color de determinado píxel dentro de la componente correspondiente a cada color básico, según el nivel de tensión asociado. Esta correspondencia entre los rangos de tensión admitidos y el color que representa dentro de cada color básico se encuentra ya estandarizada y varía en función de la resolución con la que se puedan representar cada color. Normalmente, para establecer determinado nivel de tensión dentro del rango deseado se emplean conversores Analógicos-Digitales o bien divisores resistivos. En ambos casos, aunque la conexión final ocupa un solo pin para cada color, para lograr el nivel de tensión requerido, se necesitan de varias salidas (o bits) desde el controlador en cuestión que van conectadas a cualquiera de las dos variantes.

Sin embargo, independientemente de la resolución que emplee el controlador de VGA, el sistema de colores siempre será retro compatible aún con los adaptadores EGA<sup>1</sup> y CGA<sup>2</sup> antiguos. Desde los primeros controladores básicos de VGA para microcomputadoras, existen varios modos gráficos estandarizados de los cuales el de 640x480 a 16 colores o monocromático resulta el más popular y es aún en la actualidad frecuentemente empleado en aplicaciones donde no se requiere una alta calidad de imagen, sino más bien la visualización de patrones y otro tipo de información alfanumérica. Para el caso de los otros 2 hilos restantes, se refieren a las señales de sincronismo horizontal y vertical denominadas **hsync** y **vsync** respectivamente. Estas señales digitales de 0V/5V son de vital importancia para el funcionamiento de todo el sistema de visualización en general y sin ellas, aunque las señales de color mantengan un adecuado funcionamiento, el visualizador simplemente no reaccionará. Dado que el refrescamiento de la pantalla en general se realiza barriendo el área completa de visualización en un orden específico pixel a pixel en un tiempo específico, suministrando en cada caso las señales de color correspondientes -véase **Teoría de operación**-, es necesario una señal que indique la temporización necesaria para el paso de un pixel a otro, en función de la

---

<sup>1</sup> Siglas de **Enhanced Graphics Adapter**. Constituye la especificación estándar de IBM PC para visualización de gráficos situada entre CGA y VGA en términos de rendimiento gráfico (amplitud de colores y resolución).

<sup>2</sup> Siglas de **Color Graphics Adapter**. Primera tarjeta gráfica en color de IBM y el primer estándar gráfico para IBM PC

cantidad de píxeles que conformen la pantalla, tanto de largo como de ancho. Para ello **hsync** controlará el tiempo de cada pixel en cada línea horizontal y **vsync** lo hará de forma similar, pero en la vertical, indicando el tiempo que dura un recorrido completo de la pantalla desde su borde superior al inferior.

Finalmente, para lograr la visualización completa solo se necesita una memoria de video en la que pueda almacenarse la representación digital de la imagen. En esta memoria deberá contener la información de color necesaria para cada pixel a visualizar según la resolución que se utilice. Luego, el hardware deberá ser capaz de manejar correctamente las señales de control descritas anteriormente y proveer los valores de color a la velocidad necesaria.

### 1.1.1. Capa física

Tal como se había mencionado anteriormente, el carácter analógico del estándar VGA hace que sean necesarios ciertos componentes antes de acoplar directamente las salidas del controlador VGA y la entrada del visualizador. En algunos casos, para la conversión de los códigos binarios de color a señales de tensión entendibles por el visualizador se emplean conversores DAC escogidos según la profundidad de color que se requiera, con la ventaja de que se tienen valores de tensión más estables y precisos que con la contraparte resistiva, pero sin embargo, conllevan mucha mayor complejidad circuital y componentes externos para su operación, como referencias de tensión -en los casos que no estén integradas en el propio conversor o que necesiten valores de tensión específicos-, redes de compensación, entre otros. La Fig. 2 representa de manera general un diagrama típico para esta configuración.

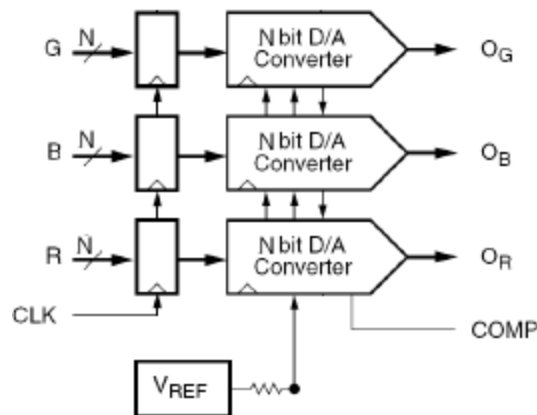


Fig. 2. Representación esquemática genérica de acoplamiento de señales VGA con convertidores DAC. Fuente: Elaboración propia.

En el diagrama, las salidas **O<sub>G</sub>**, **O<sub>B</sub>** y **O<sub>R</sub>** representan las salidas de los convertidores DAC para cada color respectivamente, las cuales ya se conectan directamente al terminal VGA. Típicamente, estos convertidores DAC ya vienen integrados en un solo chip con la lógica y la potencia necesarias para manejar satisfactoriamente las señales de color de VGA, a partir de otras señales para el control del



propio chip. Ejemplo de lo anterior son los integrados ADV7123 de ANALOG DEVICES ® y FMS3818KRC de ON Semiconductor ®, entre otros. El resto de las señales de sincronismo de VGA se conectan directamente desde el controlador de VGA al visualizador. La Fig. 3, a modo de ejemplo, muestra el diagrama de bloques de un sistema basado en PLD donde se emplean este tipo de circuito integrado para el manejo de las componentes de color de la señal de video. Como puede observarse, el conexionado es bastante sencillo, solo que se necesitan más señales desde el controlador de VGA para el control del DAC, lo que aumenta naturalmente su complejidad circuital. Las características específicas de estas señales pueden variar de un conversor a otro, sin embargo, su funcionalidad prácticamente es la misma en casi todos los casos.

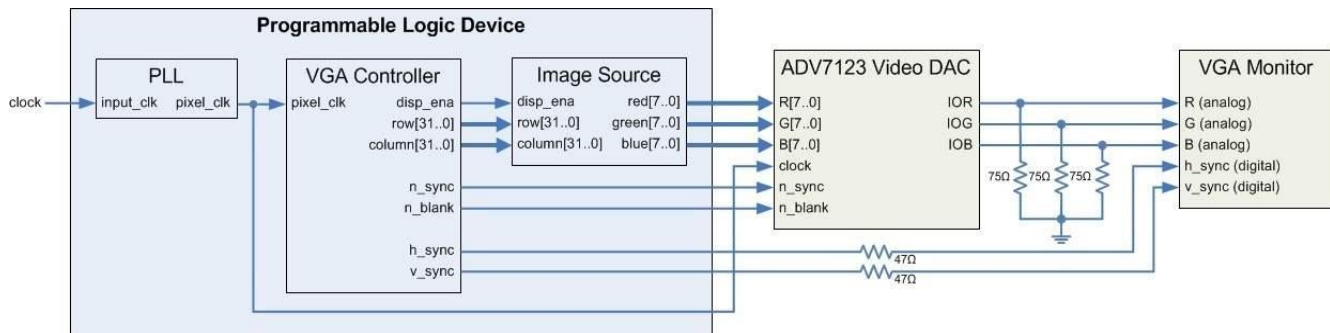


Fig. 3. Ejemplo de controlador VGA basado en PLD empleando DAC para el acoplamiento. Tomado de [1]

Otro enfoque para el acoplamiento de las señales de color de VGA es el de emplear un divisor resistivo en los pines del conector VGA. Este es el caso del conector de placa de desarrollo de FPGA *Spartan 3A Starter Kit*. A diferencia del caso anterior, aquí solo se emplean resistencias conectadas directamente a cada bit de las salidas de las señales de color del controlador VGA, unidas todas a un nodo común que finalmente se conecta al terminal VGA. Estas resistencias poseen valores calculados en cada caso, lo que posibilita dar en el nodo común un valor de tensión según el código binario que se escriba. El resto de las señales de sincronismo horizontal y vertical se encontrarán entonces conectadas directamente al conector VGA empleando solo una resistencia limitadora de corriente para cada una. La representación esquemática del caso anterior se puede apreciar en la Fig. 4, la cual coincide con la circuitería implementada en la placa de desarrollo de FPGA empleada para la realización de este trabajo.

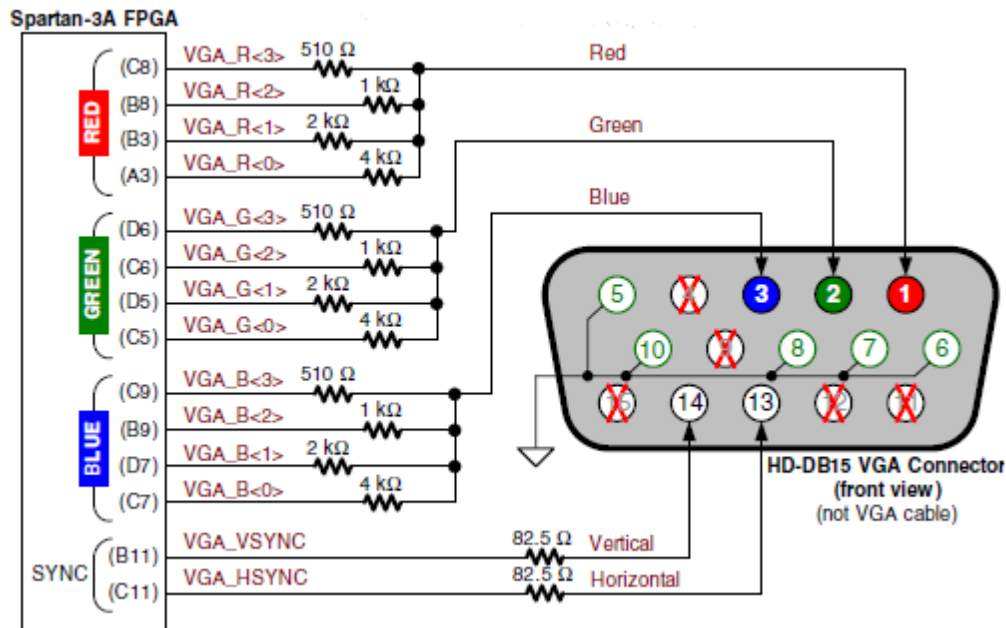


Fig. 4. Conexiones VGA empleando divisores resistivos. Tomado de [2].

La figura anterior muestra que es posible implementar sobre esta placa de desarrollo un controlador VGA con hasta 4bits de resolución por color, generando un color final definido por 12bits para un total de 4096 colores. Si se agrupan los 4bits de cada color para obtener solo tres buses independientes, se obtiene la gama de colores que aparece representada en la Tabla 1. Estos 8 colores básicos serán utilizados posteriormente en el desarrollo del módulo IP controlador de VGA para establecer patrones de prueba a fin de verificar el correcto sincronismo con la pantalla y el manejo de los códigos de colores.

Tabla 1. Códigos de colores básicos para las señales de Rojo, Verde y Azul de 4bits. Tomado de [2].

VGA_R[3:0]	VGA_G[3:0]	VGA_B[4:0]	Resulting Color
0000	0000	0000	Black
0000	0000	1111	Blue
0000	1111	0000	Green
0000	1111	1111	Cyan
1111	0000	0000	Red
1111	0000	1111	Magenta
1111	1111	0000	Yellow
1111	1111	1111	White

El aspecto de la paleta completa RGB de 12bits que surge de los 4096 colores posibles a lograr con estas salidas se muestra en la Fig. 5.

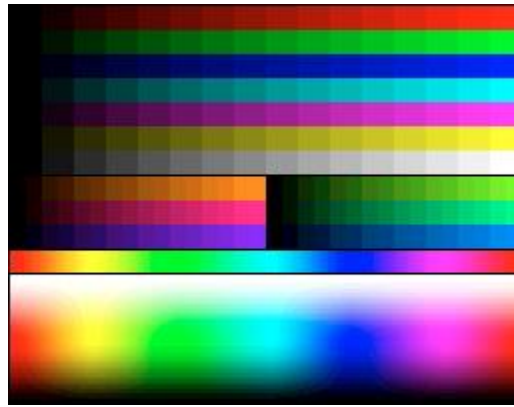


Fig. 5. Paleta RGB de 12bits

Este método tiene como principal ventaja la sencillez de su implementación, pero presenta varios inconvenientes. En primer lugar, ante un fallo en cualquier resistencia, evento improbable pero posible, la tensión resultante en el nodo común del color correspondiente quedará afectada, provocando en caso de que se abra la resistencia, la inhibición total de la influencia del bit correspondiente en el color final, y en el caso que quede cortocircuitada, que la tensión final solo tenga dos valores posibles, según el estado del bit afectado y provocando además, el sobrecalentamiento de los demás resistores (sobre todo los de menor valor).

En segundo lugar, aunque menos fatídico, está el inconveniente ligado a la naturaleza de todo resistor de variar su valor según la temperatura. Esto puede provocar que los valores de tensión requeridos para cada color se vean afectados; fenómeno que igualmente estará vinculado al valor de tolerancia de la misma. Sin embargo, este problema puede atenuarse escogiendo resistores de alta calidad y baja tolerancia.

En sentido general, la interfaz VGA no se encuentra diseñada para **Hot plug**<sup>3</sup>, aunque en la práctica esto puede ser realizado sin causarle daño alguno al hardware o al software. Sin embargo, no se recomienda dado que nada garantiza que la entrada VGA del dispositivo conectado disponga de las protecciones eléctricas y de software adecuadas para soportar cualquier interferencia, ruido u otra señal indeseada que pueda surgir en las señales del conector debido a la manipulación mecánica. Además, en dependencia del hardware o del software del visualizador, el proceso de identificación de la conexión puede no funcionar correctamente en todos los casos.

### 1.1.2. Teoría de operación

Existe gran variedad de estándares VGA, cada uno con su resolución y frecuencias de refrescamiento específicas -ver Anexo 1-. Estos estándares hoy en día, además de otros relacionados con monitores, televisores y equipos similares en general, son regidos por la Asociación de Estándares Electrónicos

<sup>3</sup>Así se le denomina a la capacidad de un periférico para ser conectado o desconectado sin tener que apagar la PC y poder funcionar correctamente. No debe confundirse con **Plug-and-Play**.

de Video, VESA, por sus siglas de en inglés. Sin embargo, independientemente del estándar que se emplee, la teoría de operación siempre es la misma.

En esencia, todo visualizador puede ser tratado como una matriz plana donde cada elemento constituye un pixel cuya ubicación es única y se encuentra definida por una determinada fila y columna dentro de la matriz. Esta misma se encuentra seccionada en distintas zonas que cumplen un propósito específico en la pantalla. De esta forma, se tiene el área de visualización, donde se muestra la información de color correspondiente a cada pixel que la conforma, y las regiones correspondientes a los bordes izquierdo, derecho, superior e inferior, además de otra pequeña zona de retraso<sup>4</sup>. De forma esquemática, la Fig. 6 muestra una representación de las secciones mencionadas.

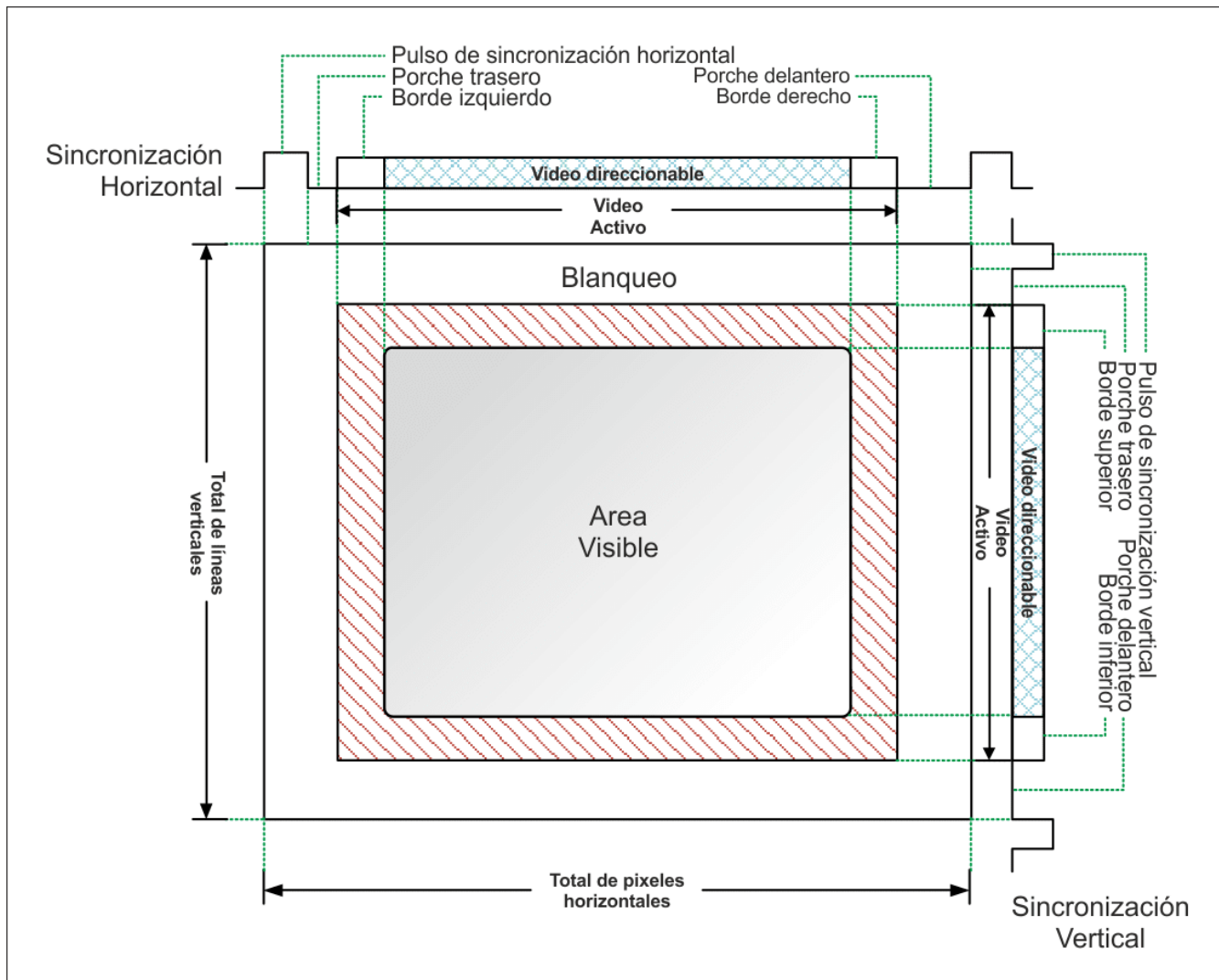


Fig. 6. Secciones que componen el área total de la pantalla en el estándar VGA. Fuente: Elaboración propia.

<sup>4</sup> El término se refiere a una de las zonas de la pantalla denominada así por la función que cumple, asociada al tiempo que se demora el barrido de pixeles al retornar del borde derecho de la pantalla al izquierdo al llegar al final de una fila de pixeles. No confundir con *retraso*.

Como se puede apreciar en la figura anterior, las secciones definidas anteriormente poseen determinadas características que las distinguen entre sí. El área visible o área activa, constituye aquella zona en la que puede visualizarse satisfactoriamente la información de color correspondiente a cada pixel que la compone. Al conjunto de píxeles ubicados en la misma fila se le denomina línea, y el conjunto de líneas determinan el tamaño de la pantalla, denominada “frame”. Para que la imagen se haga visible, la pantalla se debe barrer periódicamente, mostrando solo la información de color de un píxel a la vez durante determinado tiempo, y luego la del siguiente, y así sucesivamente. Este proceso se repite una y otra vez, barriendo la pantalla completa típicamente 60 veces por segundo, o 60Hz, aunque dicho valor puede variar siempre que esté por encima de los 40Hz mínimo, frecuencia a partir de la cual el ojo humano dejar de percibir el parpadeo. El tiempo de activación de cada píxel en una línea se rige por la activación de la señal **hsync** -la polaridad de la misma depende del estándar que se emplee-; mientras que el tiempo de barrido de la pantalla completa, o tiempo de frame, se indica mediante la activación de la señal **vsync**.

La Fig. 7 muestra otro enfoque similar al de la Fig. 6, pero en este caso, representando la correspondencia adecuada entre las señales de sincronismo horizontal y vertical y el refrescamiento de la pantalla. En este esquema, puede verse claramente el sentido de barrido de los píxeles que componen la pantalla, así como su disposición espacial en el plano del visualizador. Nótese en esta misma figura, que han sido representadas a la vez, las secciones que componen la pantalla señaladas con sus siglas correspondientes, junto a su duración en píxeles, para el caso de las secciones delimitadas horizontalmente, y en líneas para el caso de las que se encuentran enmarcadas de manera vertical. Esta duración se corresponde al estándar de 640x480x4 @60Hz, cuyas especificaciones pueden consultarse en el Anexo 1, y del cual se tratará más adelante -véase **Formato VGA 640x480 a 60Hz**-. Puede verse, además, su relación con la forma de onda que adquieren las señales de sincronismo.

Como sugiere la Fig. 7, las siglas **RT**, **BP** y **FP** para el caso de las secciones horizontales, se refieren al **retrazo**, **back porch** (o borde izquierdo) y **front porch** (o borde derecho), respectivamente. Para el caso del **retrazo**, esta sección constituye el tiempo que demora el mecanismo de posicionamiento de pixel del visualizador en retornar al principio de una nueva línea luego de haber llegado al final de la anterior. En los visualizadores TRC clásicos, esto podría entenderse como la región en la que los haces de electrones que definen la componente de cada color de un mismo pixel retornan al borde izquierdo de la pantalla. En correspondencia con el estándar VGA, la señal de video en esta zona debe estar inhibida. La duración de cada una de ellas resulta un parámetro distintivo en cada estándar.

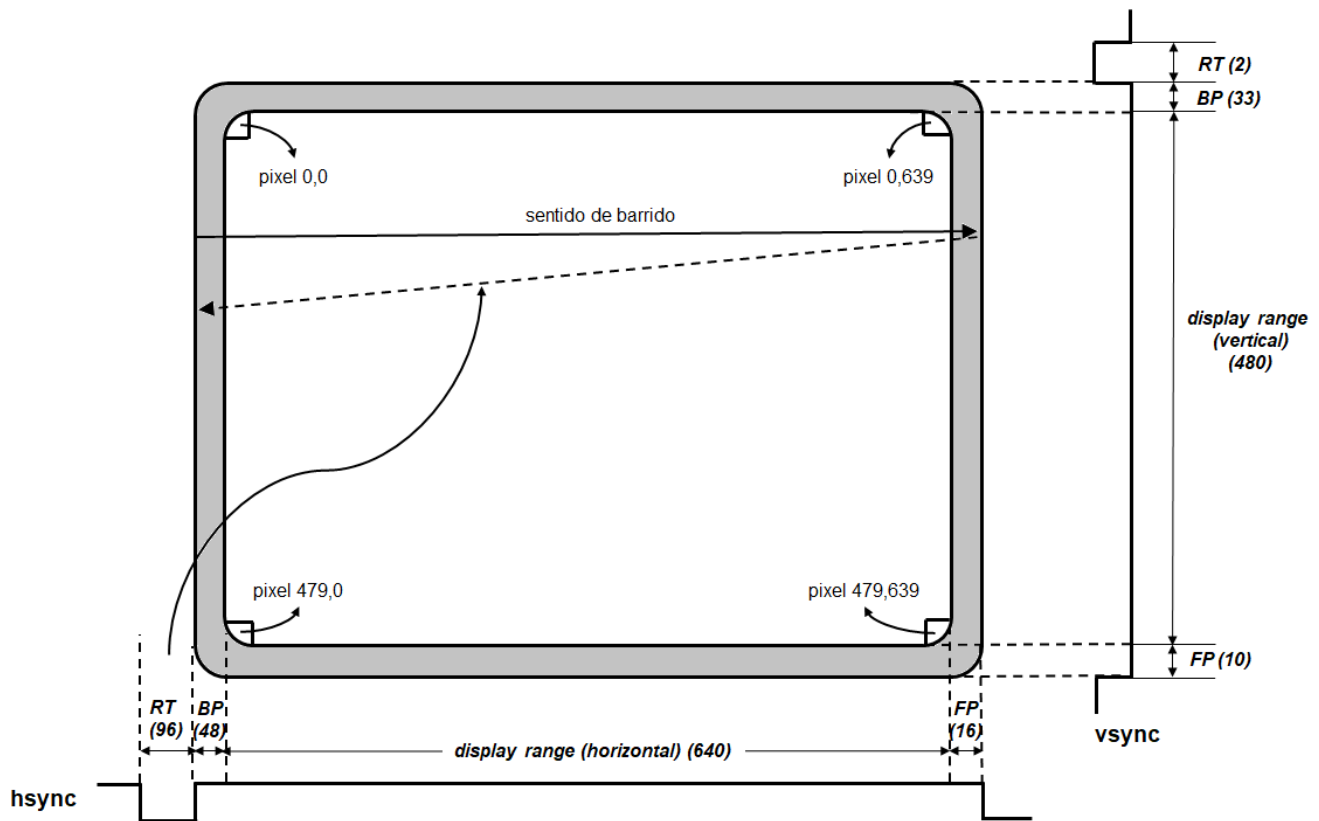


Fig. 7. Representación esquemática del barrido y las regiones del visualizador según VGA y las señales de sincronismo correspondientes para una resolución de 640x480. Tomado y adaptado de [3]

De manera similar a las secciones horizontales, se definen las secciones verticales. En este caso, **RT**, **BP** y **FP** son las siglas de **retrazo**, **back porch** (en este caso el borde superior) y **front porch** (ahora borde inferior). La sección de retrazo en el vertical posee igualmente un significado similar al descrito con anterioridad, pero ahora en el sentido de la vertical, indicando el tiempo que demora el visualizador en comenzar el **back porch**, luego de haber finalizado el último pixel del **front porch**. Al igual que en el caso anterior, ninguna información de video debe visualizarse mientras duren estas señales.

Finalmente, un controlador de video VGA debe asegurar el cumplimiento estricto de los procedimientos descritos anteriormente, así como con los requerimientos para la frecuencia de barrido, lo que, a su vez, garantiza el respeto de los tiempos de cada sección y con ello, los tiempos de pixel.

### 1.1.3. Formato VGA 640x480 a 60Hz

El formato cuyo nombre encabeza esta sección es simplemente eso, otro formato más. La singularidad radica fundamentalmente en su popularidad, sencillez, y, sobre todo, porque es el que emplea el módulo IP desarrollado en este trabajo.

La visualización mediante VGA empleando esta resolución se volvió muy popular en las primeras microcomputadoras IBM PC desarrolladas sobre la década de los 80. Actualmente aún se emplea en muchas aplicaciones, como la videovigilancia en ambientes extremos o agresivos, interfaces gráficas de instrumentos y herramientas industriales, entre otros. Constituye el modo de video básico y el más bajo que soporta Windows, en este caso a 16 bits, aun sin tarjetas ni adaptadores gráficos.

Como se ya se había visto, aunque esta resolución está compuesta por 640x480 pixeles visibles, en total deben manejarse  $800 \times 525 = 420\,000$  pixeles contando todas las zonas que componen la pantalla en las que no se muestra video. Calculando entonces el tiempo de refrescamiento de cada pixel para una frecuencia de refrescamiento de 60Hz, se obtiene  $(1/60 \times 420000) \approx 25$  MHz de tasa de pixel. El inverso de este valor indica el tiempo que debe permanecer activa la información de determinado pixel en la pantalla. Tomando como referencia este valor, se obtienen entonces los parámetros listados en la *Tabla 2* y *Tabla 3* respectivamente. El estándar establece como negativa la polaridad de los pulsos de **hsync** y **vsync**.

Tabla 2. Restricciones temporales asociadas al sincronismo horizontal. Confeccionada a partir de [4].

Scanline part	Pixels	Time [ $\mu$ s]
Visible area	640	25.422045680238
Front porch	16	0.63555114200596
Sync pulse	96	3.8133068520357
Back porch	48	1.9066534260179
Whole line	800	31.777557100298

Tabla 3. Restricciones temporales asociadas al sincronismo vertical. Confeccionada a partir de [4]

Frame part	Lines	Time [ms]
Visible area	480	15.253227408143
Front porch	10	0.31777557100298
Sync pulse	2	0.063555114200596
Back porch	33	1.0486593843098
Whole frame	525	16.683217477656

## 1.2. Módulo IP VGA desarrollado

### 1.2.1. Características Generales

Básicamente, como cualquier otro controlador de VGA, el módulo IP desarrollado tendrá como principal función generar las señales de sincronización horizontales y verticales, necesarias para la visualización, así como las salidas seriales para cada uno de los 3 colores existentes.



El módulo en general cumple una función netamente demostrativa, dado que en esencia solo genera patrones de colores en el visualizador VGA compatible. Estos patrones pueden definirse de forma manual o automática, donde en cada caso tendrá una forma de operación distinta; por lo que su diseño resulta relativamente sencillo, y las funcionalidades que implementa se basan en la interacción directa con el software de aplicación. Todo esto será explicado más adelante en el epígrafe

### **Funcionalidades.**

Sin embargo, en cuanto a la parte física de su implementación, el módulo VGA posee una organización interna específica basada en tres bloques fundamentales. El primero constituye un bloque controlador básico cuya función será determinar en todo momento las señales de sincronización y el orden y coherencia necesarios para el barrido de la pantalla, suministrando en cada momento dos vectores independientes con la información de la coordenada actual del pixel que se está visualizando a los demás bloques que componen el diseño. Es precisamente este bloque quien determina la resolución de 640x480 pixeles y maneja las señales **hsync** y **vsync** como activas a nivel bajo. Las demás especificaciones temporales y de tamaño de las regiones que conforman la pantalla se corresponden con el estándar descrito en el epígrafe **Formato VGA 640x480 a 60Hz**.

El segundo y tercer bloque se encuentran asociados más bien a las funcionalidades que ofrece el dispositivo. De forma general, existe un bloque que posibilita la generación de patrones de colores en pantalla de forma automática atendiendo a una determinada temporización, y otro bloque que se encarga de brindar el soporte necesario para establecer los patrones en pantalla de forma manual mediante la interacción con el usuario. Naturalmente, el empleo de una u otra funcionalidad dependerá de las entradas de selección de un multiplexor colocado a la salida del módulo, que selecciona para las salidas entre las señales de color generadas por uno u otro bloque en particular.

El esquema general del sistema desarrollado se muestra en la *Fig. 8*, donde se pueden apreciar las entradas (a la izquierda) y las salidas (derecha) del diseño implementado. Este esquema representa el controlador VGA visto de manera aislada, aún sin interconectar a ningún procesador específico.



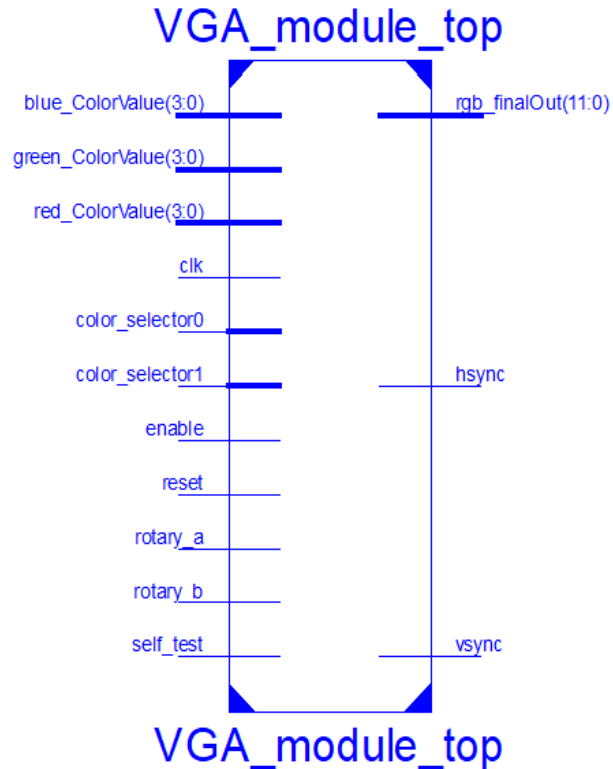


Fig. 8. Vista general de bloque del diseño del controlador VGA implementado. Fuente: Elaboración propia.

A partir de la figura anterior, se pueden enunciar algunas de las características más relevantes del módulo en cuestión. El mismo, emplea una profundidad de color de 12 bits en formato RGB, esto es, como se había mencionado ya con anterioridad, un vector de 12 bits donde se emplean 4bits para definir las componentes del Rojo, el Verde y el Azul respectivamente, siendo este último los 4bits menos significativos.

Por otra parte, puede observarse en la misma figura, la existencia de señales de control que posibilitan varias de las funcionalidades que ofrece el módulo. Para el bloque encargado de generar los patrones de forma automática, se encuentra la señal de **self\_test**, la cual precisamente constituye la entrada de selección del multiplexor a la salida mencionado anteriormente. Para el caso del bloque encargado de generar los patrones de forma manual, se tienen entonces las señales de **blue\_ColorValue**, **green\_ColorValue**, **red\_ColorValue**, **color\_selector**, **rotary\_a** y **rotary\_b**, todas como entradas, cuyas funciones serán explicadas más adelante -véase **Estructura interna y operación**-. De forma global, se encuentran finalmente las señales **enable**, que posibilita la habilitación de todo el controlador de VGA en general, además de **reset** y **clk**.

La Tabla 4 muestra un reporte detallado de la utilización de recursos en el FPGA xc3s700a de la *Spartan 3A Starter Kit* del módulo VGA implementado, en la síntesis final antes de ser importado como módulo de propiedad intelectual en el flujo de diseño de *Xilinx Platform Studio (XPS)*. Esta tabla

da una idea bastante precisa del tamaño del módulo y del espacio que ocupa dentro del FPGA empleado para su verificación.

Tabla 4. Reporte de utilización de recursos del FPGA del módulo VGA implementado de forma aislada. Tomado de *Xilinx ISE Project Navigator*

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slices	197	5888	3%
Number of Slice Flip Flops	123	11776	1%
Number of 4 input LUTs	379	11776	3%
Number of bonded IOBs	34	372	9%
Number of GCLKs	1	24	4%

El código VHDL correspondiente a este bloque del diseño puede consultarse de manera íntegra en el *Anexo 8* al final de este documento.

### 1.2.2. Estructura interna y operación

El diseño del bloque presentado en la *Fig. 8* ha sido realizado en correspondencia con la jerarquía de diseño que se muestra en la *Fig. 9*.

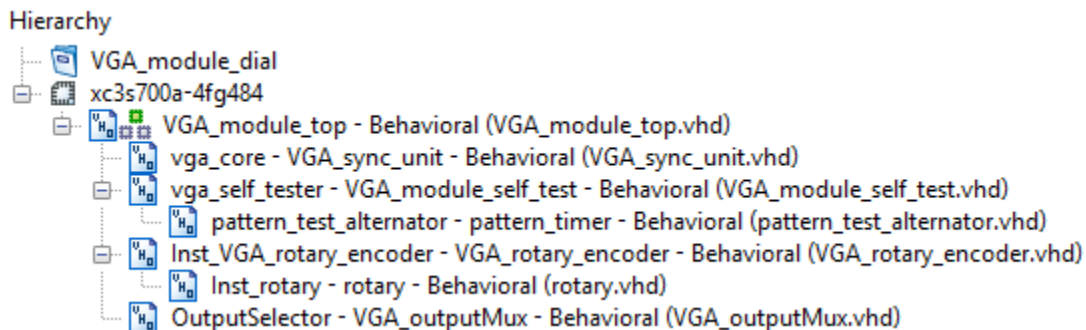


Fig. 9. Jerarquía del diseño propuesto para el controlador VGA.

En la figura anterior pueden observarse los tres bloques ya mencionados anteriormente. **VGA\_sync\_unit** resulta el encargado del control de la resolución y de generar las señales de sincronismo horizontal y vertical. Se puede observar también, los otros dos bloques básicos que componen el diseño, denominados **Bloque VGA\_module\_self\_test** y **Bloque VGA\_rotary\_encoder**. El primero tiene la función de generar un patrón de 48 colores a lo largo de toda la pantalla que irá orientándose de forma horizontal o vertical de forma automática cada un segundo, según un contador diseñado al efecto, denominado en la *Fig. 9* **pattern\_timer**, mientras que el segundo bloque, **VGA\_rotary\_encoder**, será quien permita establecer a través de varias fuentes -todas controladas por el usuario- el color deseado para la franja que se seleccione.

La Fig. 10 muestra de manera esquemática simplificada de la estructura interna del diseño realizado, así como las interconexiones entre los diversos bloques que componen el sistema, representando las señales de mayor importancia para el funcionamiento del conjunto. Este diagrama constituye un resumen del diseño RTL obtenido en Xilinx ISE® para la jerarquía representada en la Fig. 9, razón por la cual no se muestran otras señales comunes a todo el circuito, como el reloj y el reset.

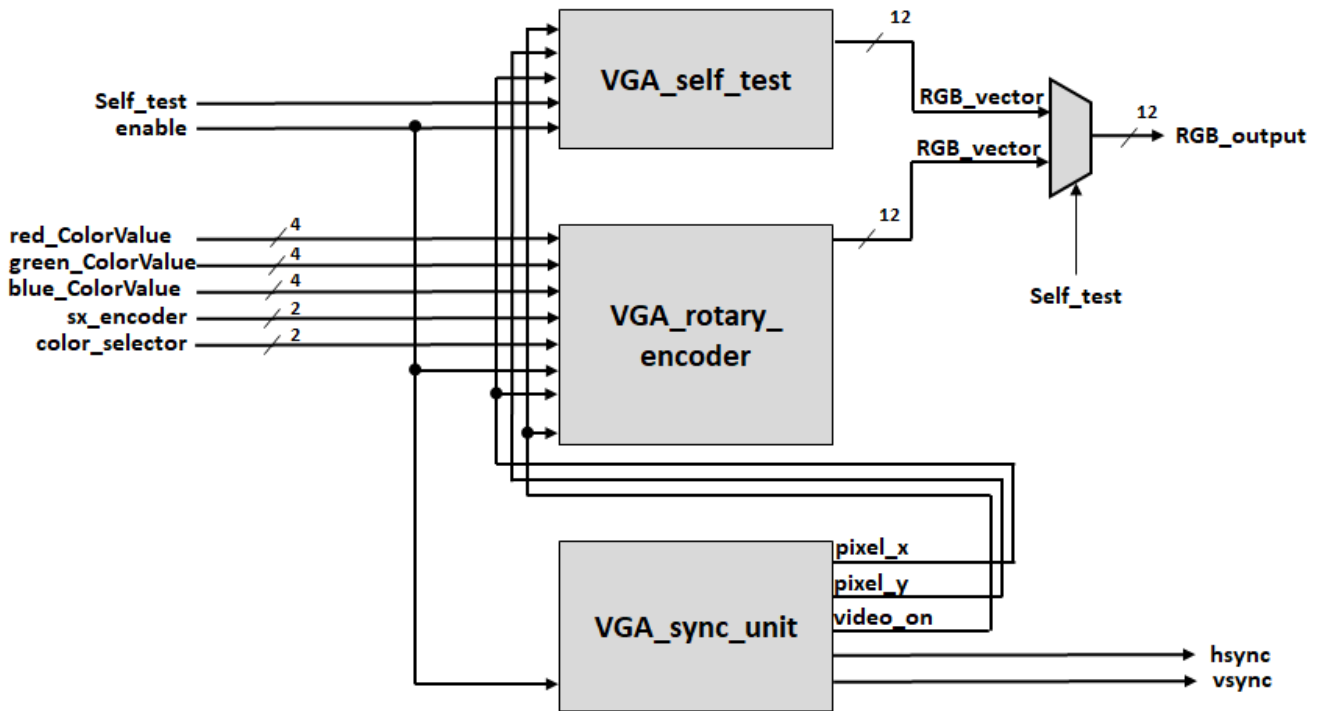


Fig. 10. Estructura interna del controlador VGA implementado.

No obstante, el hecho de que el diagrama anterior obvie la señal de reloj, no significa que carezca de importancia. De hecho, en multitud de diseños consultados, lograr el sincronismo entre todos los elementos que componen un sistema resulta un aspecto esencial y a la vez complicado. La obtención de la frecuencia de pixel guarda estrecha relación con este problema, la cual puede ajustarse de diversas maneras, dependiendo, lógicamente del valor necesario para la resolución a implementar y de la frecuencia de reloj disponible en la placa de desarrollo de FPGA. Típicamente en muchos diseños de referencia que suministran los diversos fabricantes de FPGA, y en muchos otros disponibles en la web, se emplea memoria de video externa para la aplicación VGA, y con ello, el empleo de frecuencias de reloj “incomodas” a la hora de lograr una base de tiempo conveniente para un refrescamiento típico a 60Hz o 70Hz. Esto se pone de manifiesto tanto en [5] como en [6], y en realidad, sucede muy a menudo, dado que el empleo más común de un controlador VGA, aunque no el único, es para la visualización de imágenes o videos; lo cual implica el uso de una memoria de video considerablemente mayor a la disponible localmente dentro del FPGA, que casi siempre termina siendo alguna memoria externa SDRAM DDR (en cualquiera de sus variantes) y el controlador asociado.

Esto último resulta un inconveniente para cuya solución muchos optan por generar directamente desde un módulo IP generador de señal de reloj una señal con una base de tiempo más amigable para realizar el resto del diseño, provocando con esto que el sistema se divida en dos dominios de reloj fundamentales, lo cual complica sustancialmente el proyecto y conlleva precauciones de diseño extras.

Sin embargo, tanto en [3] como en [7] el módulo VGA diseñado no es precisamente para la visualización de imágenes y videos, sino para generar texto, gráficos simples en la pantalla a modo de juego interactivo, o una combinación de ambas cosas. En ambos casos, la aplicación no demanda el empleo de altas resoluciones, por lo que se visualiza a 640x480 a 60Hz sin necesidad de emplear memoria de video externa. Para ello, el sistema opera con la misma señal de reloj de 50MHz disponible en la placa de desarrollo, y simplemente implementan un divisor de frecuencia base 2 para obtener los 25MHz necesarios en la visualización. Esta alternativa resulta sumamente sencilla y es la que se emplea en el presente diseño, dado que no se realiza ninguna visualización de imagen desde memoria de video externa.

#### 1.2.2.1. Bloque *VGA\_sync\_unit*

Este bloque resulta la piedra angular en el diseño del controlador VGA de manera general. Su principal función NO es manejar las salidas de video, sino las señales de temporización necesarias para controlarlo. Este bloque no maneja información de color alguna, solo indica a los otros módulos cuando y donde enviar o no esta información al visualizador. La Fig. 11 muestra una representación del circuito mostrando sus entradas (izquierda) y salidas (derecha).

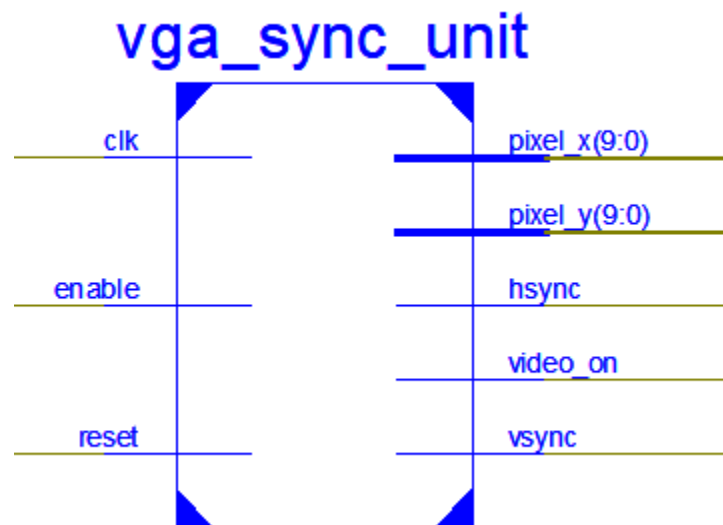


Fig. 11. Bloque *VGA\_sync\_unit*.

Básicamente, su arquitectura consiste en dos contadores binarios ascendentes, encargados de llevar el conteo de las filas y columnas que se van recorriendo en el visualizador. Dado que la resolución

empleada es de 640x480, la pantalla se divide en un total de 525 líneas con 800 píxeles cada una, valores que constituyen las bases de los contadores verticales y horizontales respectivamente y establecen una resolución de 10 bits para cada contador. Estos contadores marcan el tiempo de duración de cada pixel por individual, por lo que son alimentados por una señal de reloj denominada **pixel\_tick**, que no es más que la salida del divisor de frecuencia que genera la señal de reloj de 25MHz necesaria a partir de los 50MHz de entrada.

Para generar las señales de sincronismo horizontal y vertical en correspondencia con las diferentes zonas del visualizador, se emplean un grupo de comparadores cuyas entradas son las salidas de los contadores horizontales y verticales y las distintas constantes que delimitan la sección izquierda, derecha, superior, inferior y el retrazo para ambos casos. Cada salida de cada contador se compara con una constante a la vez en el caso horizontal y vertical, según corresponda. Existen comparadores que indican el fin de la línea horizontal y el fin de pantalla (vertical), y otros que detectan la entrada a las distintas secciones, indicando en cada caso si el pixel a visualizar corresponde se encuentra o no en la zona visible. Cuando los valores de estos contadores se correspondan con una zona no visualizable (**FP**; **BP** o **RT**), se desactivará una señal llamada **video\_on** indicando que las salidas correspondientes a los colores deben desactivarse durante este intervalo. En caso contrario, dicha señal permanecerá siempre activa a nivel alto. Por su significado, esta señal resulta extremadamente útil para los demás módulos, en conjunto con las salidas del contador horizontal y vertical.

Sin embargo, las señales **hsync** y **vsync** como tal solo serán desactivadas (puestas a nivel alto, dado que su polaridad debe ser negativa), en el intervalo de retrazo respectivo -recordar que, aunque se denominan igual y en esencia significan lo mismo, el mismo intervalo en ambos casos no posee igual duración-, a partir de la indicación del comparador correspondiente en cada caso.

La Fig. 12 muestra la simulación obtenida en **ISim®** para este bloque en particular. En esta imagen, aunque se muestran las formas de onda correspondientes a las salidas de ambos contadores, puede apreciarse mucho mejor el comportamiento de la salida del contador vertical, denominado **pixel\_y**, indicando el barrido por cada fila de forma consecutiva. Obsérvese también el comportamiento de las señales **hsync** y **vsync** con respecto a la señal **video\_on**, esta última activa mientras dura el barrido de una línea completa.

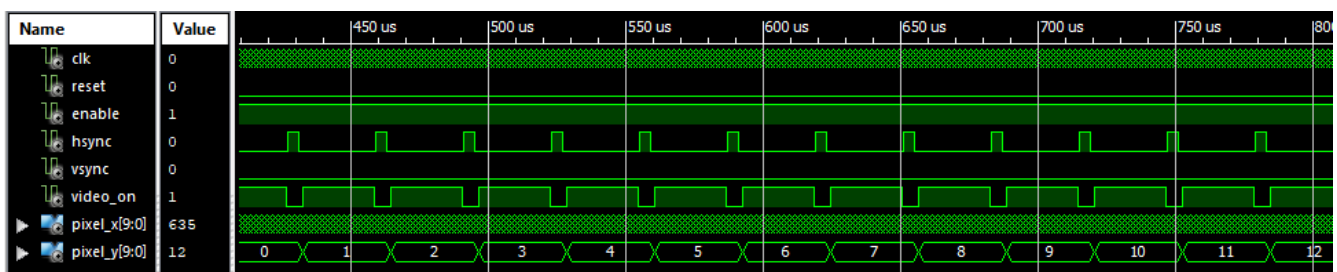


Fig. 12. Diagrama temporal obtenido de la simulación del bloque **VGA\_sync\_unit**

En la Fig. 13 puede verse con mayor claridad el comportamiento de las señales referidas anteriormente al barrer las últimas líneas de la pantalla. Como se esperaba, la señal **video\_on** permanece desactivada a partir de la línea 479 hasta la 524, a partir de la cual el contador vertical vuelve a iniciar su conteo a partir de 0. La señal **vsync** solamente se desactiva en el período de retrazo, a partir de la línea 490 hasta la 492, lo cual puede constatare observando el valor de esta señal en el momento que indica el cursor (línea vertical amarilla en la figura), situado en la parte inferior izquierda de la figura y contando la cantidad de ciclos de la señal **pixel\_y** en el diagrama temporal. De igual forma, si se cuentan la cantidad de ciclos a partir de la última activación de **video\_on** en el diagrama hasta su siguiente activación, podrá constatare el transcurso de **FP+BP+RT** líneas correspondiente al tiempo de video inactivo que indica el estándar (Véase Anexo 1) como se había explicado en el epígrafe **Formato VGA 640x480 a 60Hz**.

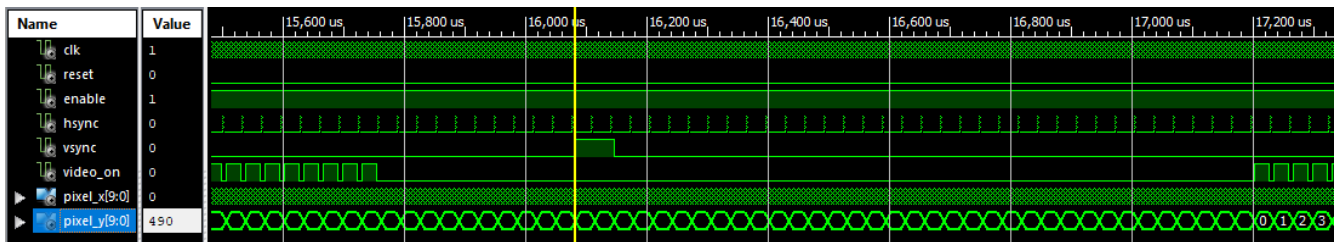
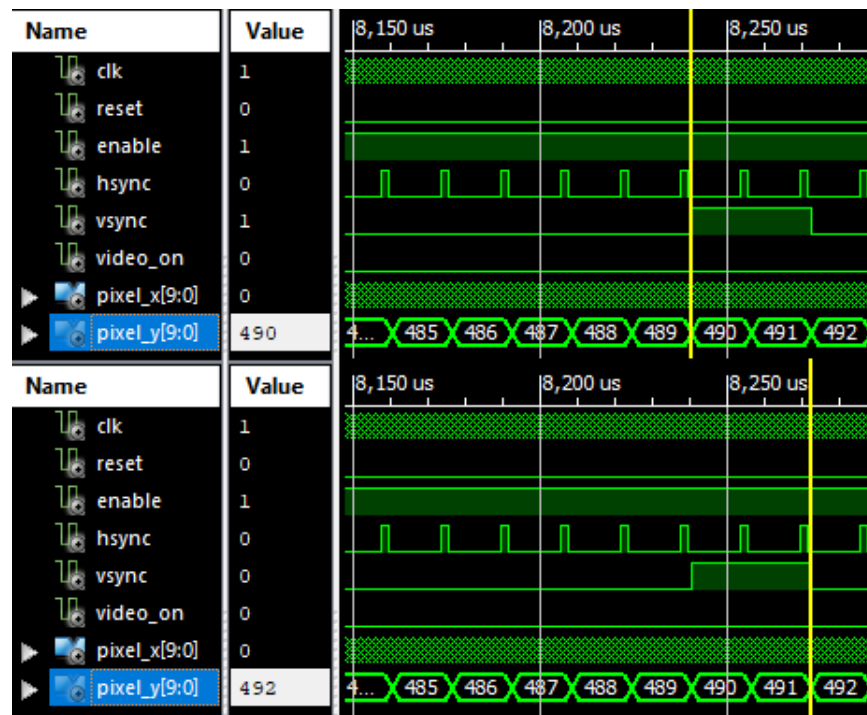


Fig. 13. Comportamiento de las señales **vsync** y **video\_on** al finalizar un barrido completo de la pantalla.

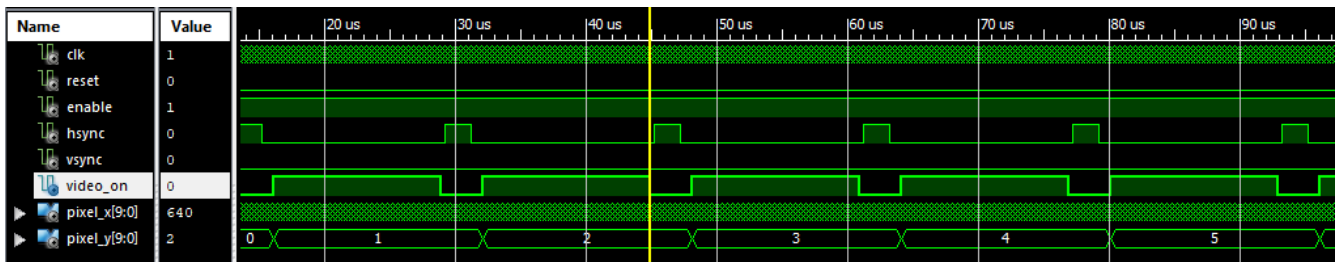
Ampliando la Fig. 13 se obtiene la Fig. 14, donde puede apreciarse con mayor calidad la duración del pulso de la señal **v\_sync** comparando el valor del vector **vsync** que aparece indicado en la forma de onda del vector **pixel\_y**. Obsérvese además en el momento en que se desactiva (líneas de retrazo) y como la señal **video\_on** permanece desactivada durante este intervalo.

Fig. 14. Duración del pulso de la señal **v\_sync**

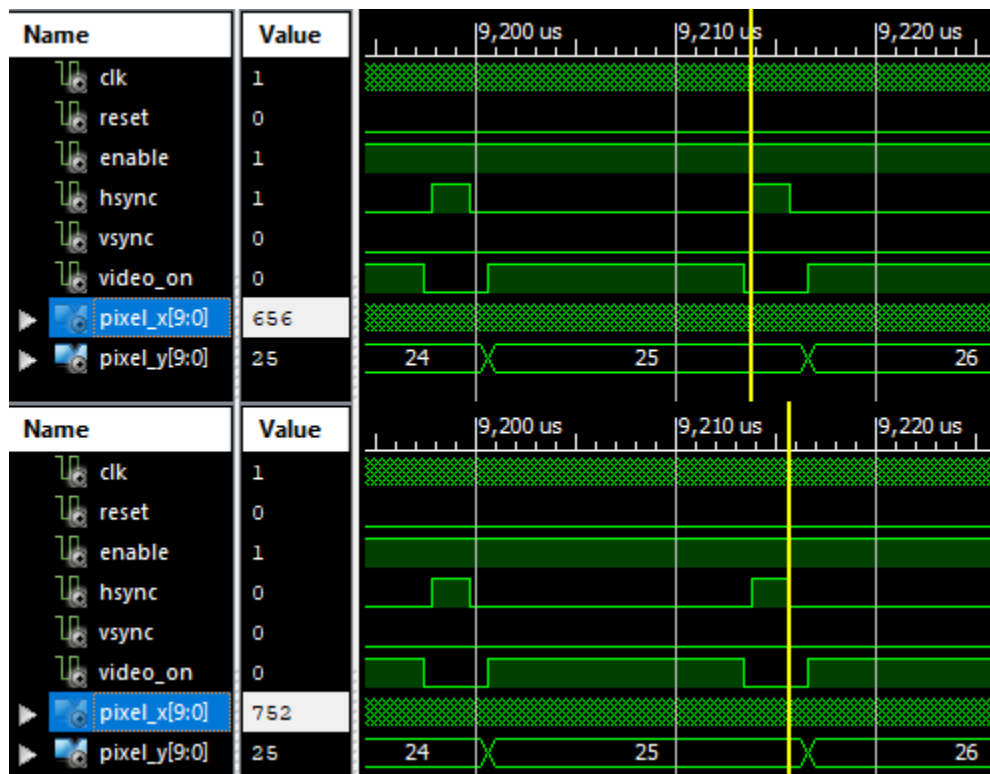
La señal **hsync** se comporta de manera similar a **vsync**, solo que tiene mucha mayor frecuencia que esta última. Esta señal se mantiene igualmente activa a nivel bajo mientras no se entre en la zona de retraso, independientemente de la línea en que se encuentre, lo cual se puede apreciar en la Fig. 14.

La señal **pixel\_x** en este caso, lleva el conteo de las columnas, o del barrido en el sentido horizontal (eje x) de la pantalla, lo que se corresponde con la salida del contador horizontal descrito con anterioridad y como es de suponer, realiza 800 conteos durante cada línea en la vertical; es decir, barre 800 pixeles por cada línea.

La Fig. 15 muestra una vista un poco más ampliada del comportamiento de las salidas de este bloque, donde se aprecia con mejor calidad el comportamiento de la señal **hsync**, con respecto a **pixel\_y**. En ella, puede verse como **hsync** se desactiva casi al final de cada período de conteo de **pixel\_y**, indicando que el barrido por **pixel\_x** ha llegado a la zona de retraso, lo que es de esperar dado que existe un retraso al final de cada horizontal. Nótese que la señal **video\_on** se mantiene activa mientras dura el intervalo de la zona visible hasta el píxel 640, indicado por el marcador amarillo horizontal en la figura.

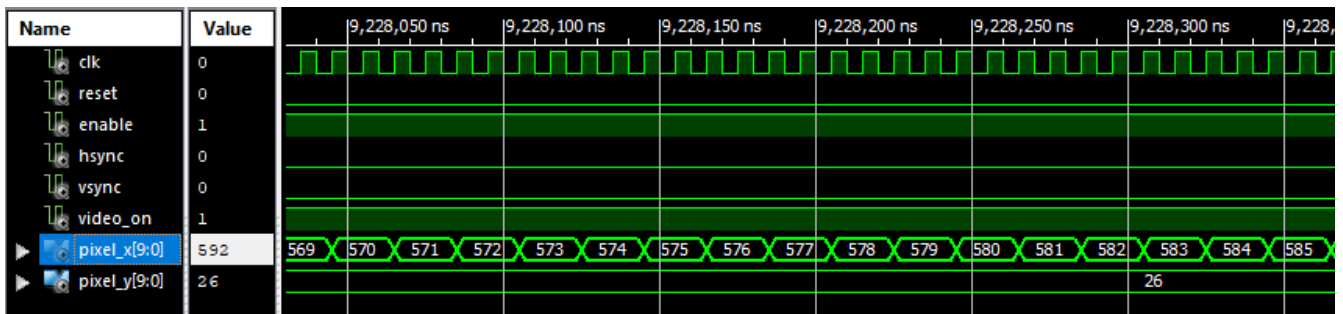
Fig. 15. Vista ampliada del comportamiento de **hsync**, **pixel\_x** y **video\_on**

En la Fig. 16 puede apreciarse de la misma forma que en la Fig. 14 la duración y momento, en pixeles, dada por los valores del vector **pixel\_x**, del pulso de la señal **hsync**. Al igual que en el caso anterior, obsérvese como la señal **video\_on** ya se encuentra desactivada en el momento en que se da dicho pulso en **hsync**.

Fig. 16. Duración del pulso de la señal **hsync**

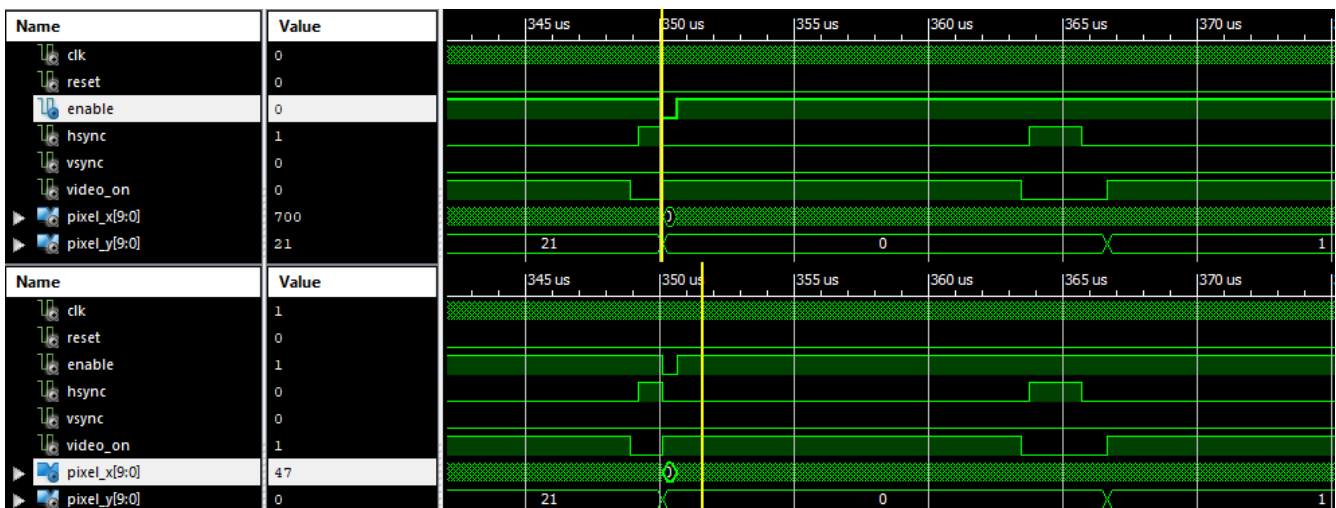
Una vista más detallada del comportamiento del vector de **pixel\_x** puede apreciarse en la Fig. 17, donde puede observarse el cambio incremental ascendente de la forma de onda asociada al mismo, indicando el tiempo activo de cada pixel. Para la captura, se ha escogido la zona activa de la pantalla. Nótese como el valor de **pixel\_y** no cambia durante todo el intervalo, y **video\_on** se mantiene activo.



Fig. 17. Comportamiento del vector `pixel_x`

Otro aspecto importante que se pone de manifiesto en todos los gráficos presentados hasta el momento, es la presencia de la señal **enable**. Esta señal activa a nivel alto condiciona el funcionamiento de todo el módulo, así como del controlador VGA en general. En caso de que esta señal baje a nivel bajo, el bloque completo inhibirá su funcionamiento y restaurará todas sus salidas a los valores iniciales, como si se tratara de un **reset**.

La Fig. 18 muestra el efecto de la desactivación de esta señal sobre el comportamiento del sistema. Para la captura, se ha escogido el momento en el que el barrido se encuentra en la zona de retrazo, donde se activa **hsync**. Como puede observarse la figura, en este momento se encuentra activo el pixel 700 de la línea 21, y se ha bajado el **enable**, por lo que una vez devuelto a nivel alto, el conteo horizontal y vertical comienzan nuevamente desde 0, y en consecuencia se actualizan las demás señales que dependen de esto, como **video\_on**. Lo anterior puede constatarse si se observan los valores de **pixel\_x**, **pixel\_y** y **hsync** antes y después de bajar la señal **enable** en los gráficos superior e inferior respectivamente.

Fig. 18. Efecto de la señal **enable** sobre el comportamiento del bloque **VGA\_sync\_unit**

El código VHDL para la implementación del bloque descrito, así como una descripción detallada de sus pines, puede consultarse en el Anexo 2.

### 1.2.2.2. Bloque *VGA\_module\_self\_test*

Este bloque recibe su nombre por la función que realiza. Su propósito es permitir una auto verificación del bloque de sincronización estableciendo en pantalla un patrón de colores específico, para lo cual se hace necesario el empleo las señales de salida de **VGA\_sync\_unit**. La información de colores ya se encuentra fijada internamente en el módulo, por lo que, para su funcionamiento, básicamente se toman las salidas correspondientes a las coordenadas de pixel, en conjunto con la señal **video\_on** y por medio de comparadores, se envía a la salida a través de un registro interno la información de color correspondiente a cada segmento del patrón.

**VGA\_module\_self\_test**, genera un patrón de colores similar al de la Fig. 5, variando cada vector de color de 4 bits por separado (manteniendo en cero el resto) dentro de las 16 combinaciones posibles, a lo largo de toda la pantalla, obteniéndose así un total de 48 franjas horizontales o verticales correspondientes a cada color que se obtenga. Estas franjas además se intercalan en la pantalla de forma horizontal o vertical de forma automática cada un segundo, a partir de un contador binario ascendente diseñado para este intervalo de tiempo integrado dentro de este mismo módulo, denominado **pattern\_timer**.

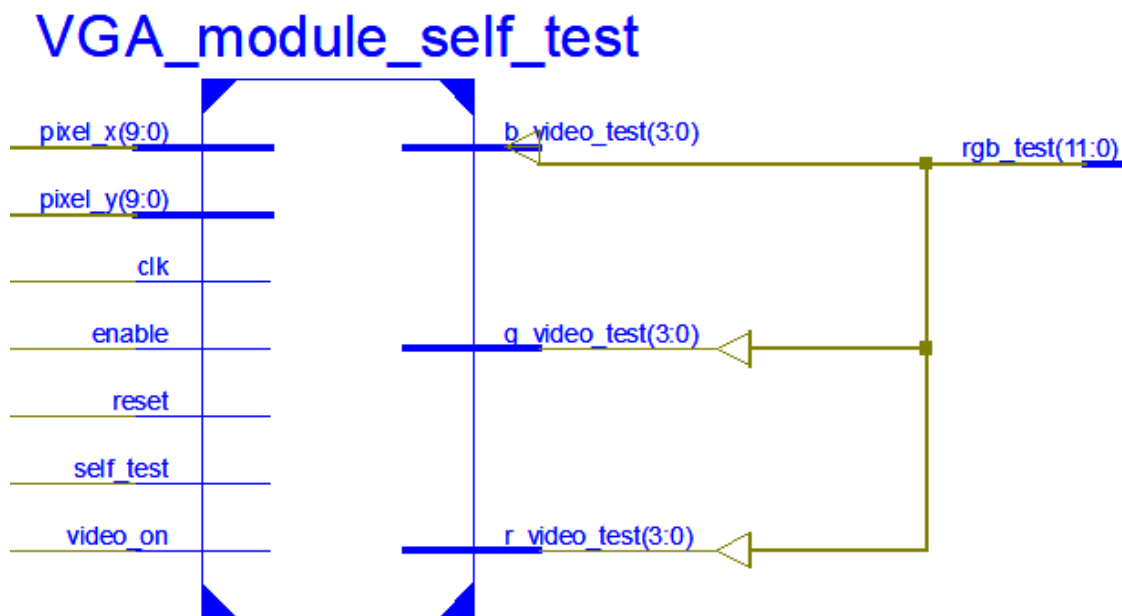


Fig. 19. Bloque **VGA\_module\_self\_test**

Básicamente el funcionamiento interno de este bloque se basa en un conjunto de registros cuya habilitación constituye la señal **enable**. Cuando **enable** se desactiva, simplemente estos registros mantienen sus salidas fijas, pero, sin embargo, no se verá nada en la pantalla puesto que el bloque **VGA\_sync\_unit** tampoco funcionará. Cuando **enable** se encuentra activo, entonces estos registros pondrán a sus salidas los valores de las salidas de color correspondientes, para dar lugar a un vector

de 12 bits de salida, en función de las señales **video\_on**, **pixel\_y** y **pixel\_x**, como puede observarse en la Fig. 19.

Precisamente, estas últimas señales son empleadas por un circuito condicional asincrónico implementado dentro de este bloque, el cual establece en cada caso, las entradas de los registros de salida descritos anteriormente, colocando el valor de color correspondiente según el intervalo especificado. Estos multiplexores son controlados precisamente por comparadores para cada intervalo de pixel especificado en la descripción del hardware, los cuales toman los vectores **pixel\_x** y **pixel\_y**, según la orientación del patrón que se encuentre activa en el momento (horizontal o vertical), y los comparan con una constante prefijada, para entonces determinar el valor del color a visualizar en el rango correspondiente.

La dirección del patrón a visualizar es determinada por un contador interno de 1s, diseñado al efecto. Este contador, denominado **pattern\_timer**, es básicamente un divisor de frecuencia con entrada de habilitación diseñado para dar desbordarse cada un segundo, a partir de una señal de entrada de 50MHz. La salida de este divisor es una señal de salida llamada **sec\_tick** que va también a estos multiplexores y es la que determina cuando se visualiza el patrón de colores de forma horizontal, y cuando de forma vertical. Esta señal se va complementando en cada desbordamiento del contador, permaneciendo un segundo en nivel bajo, y otro después, en nivel alto, y así sucesivamente. La Fig. 20 muestra la forma de onda de la salida de este módulo.

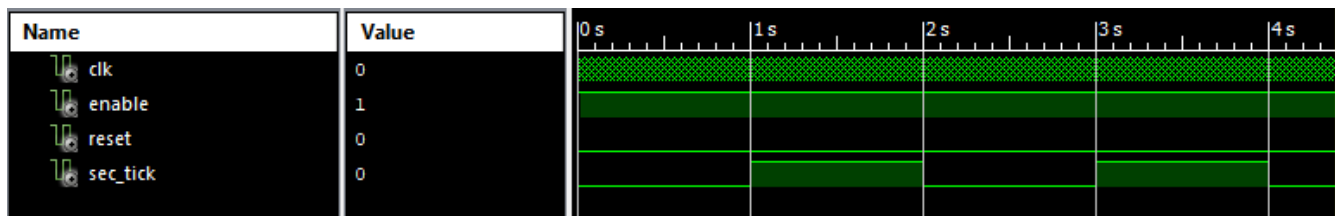


Fig. 20. Forma de onda de **sec\_tick**, salida de **pattern\_timer**, dentro de **VGA\_module\_self\_test**. Cabe destacar que el contador descrito anteriormente es interno al bloque **VGA\_module\_self\_test**, por lo que la señal **sec\_tick** es igualmente interna a este módulo. Su entrada de habilitación en este caso no constituye la señal **enable**, sino más bien una señal de igual nombre, pero en este caso, es el AND lógico entre el **enable** global y la señal **self\_test**. Lo anterior significa que este contador no solo depende de **enable** (global) para su funcionamiento, sino también de la señal **self\_test**, la cual entra al bloque **VGA\_module\_self\_test** justamente para este contador. Este segmento del diseño fue implementado de esta manera con la intención de hacer de **self\_test**, un bit que solo habilite o no el contador de 1s, activando o desactivando con ello el cambio en el patrón, dejando la señal **enable** para la habilitación global de todos los bloques del diseño.

El código completo de este divisor de frecuencia puede consultarse en el *Anexo 4*, mientras que el código para el bloque **VGA\_module\_self\_test** puede ser examinado de manera íntegra en el *Anexo 3*.

### 1.2.2.3. Bloque VGA\_rotary\_encoder

Este bloque recibe su nombre precisamente por el uso que hace del encoder rotatorio que se encuentra disponible en la placa de desarrollo empleada para el trabajo. Su principal función es manejar el conjunto de funcionalidades que brinda el módulo -véase **Funcionalidades**- de cara al usuario. Al igual que en el bloque anterior, **VGA\_rotary\_encoder** genera un patrón de colores, pero esta vez muy distinto al caso anterior. En este bloque, el patrón de colores a generar se basa en tres franjas de colores principales, correspondientes al Rojo, Verde y Azul, los cuales pueden ser modificados a voluntad por el usuario, ya sea empleando el encoder rotatorio mencionado anteriormente, o por medio de comandos, a través de la interfaz serie conectada al sistema con microprocesador. La franja de color que se desee modificar, puede ser seleccionada por medio de los pines habilitados al efecto, denominados **color\_selector(1:0)**. En la *Fig. 21* puede observarse una representación del bloque implementado, donde pueden apreciarse el conjunto de pines de entrada (izquierda) y salidas (derecha) que componen el módulo en cuestión.

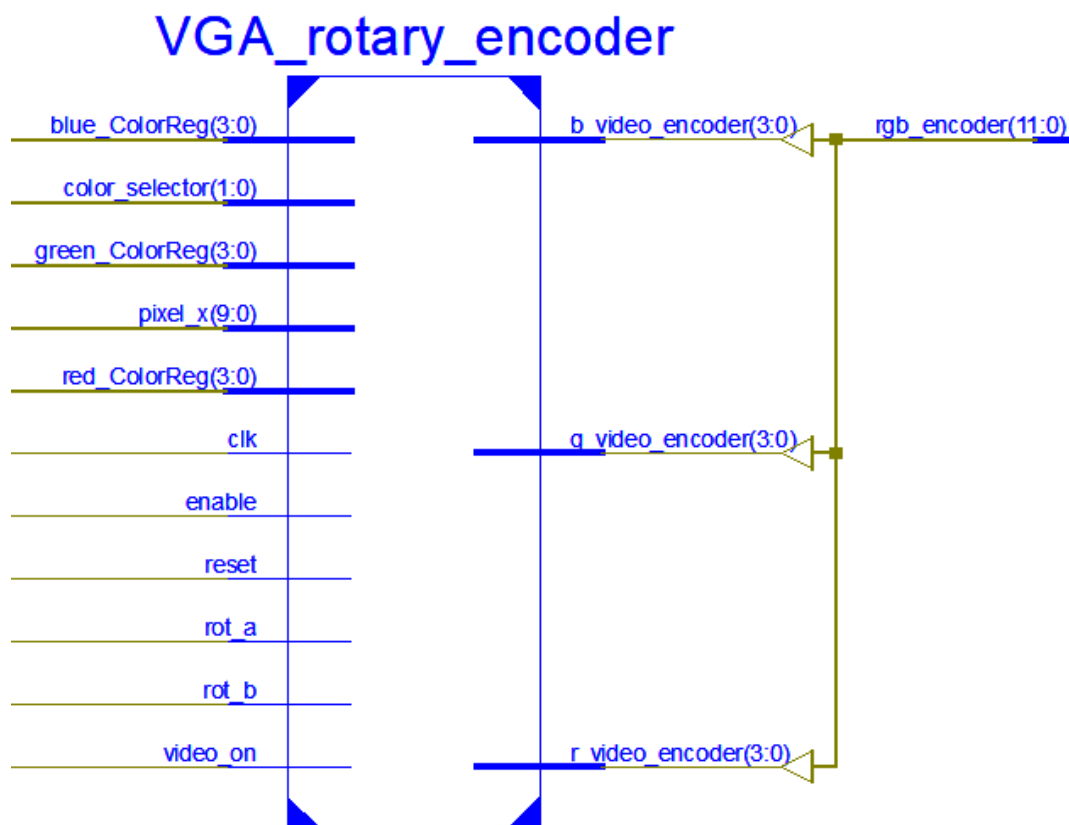


Fig. 21. Bloque **VGA\_rotary\_encoder**

Ya de la figura anterior, al igual que en el resto de los bloques presentados, se evidencian algunas de las características más importantes de este bloque en particular. Como es sabido, las salidas de video componen un vector de 12 bits que se obtiene al concatenar todas las salidas de colores independientes. Este solo vector, será posteriormente multiplexado junto a las salidas del bloque **VGA\_module\_self\_test** para obtener finalmente la señal de color a visualizar en el display -véase **Bloque VGA\_outputMux**.

Por otra parte, la figura muestra también la señal **enable**, que actúa como habilitación global de todo el sistema, y con ello, de este bloque también. Dado que este bloque solo muestra el patrón de color de tres franjas de manera vertical, no necesita de la señal **pixel\_y**, proveniente de **VGA\_sync\_unit**, por lo que solo emplea **pixel\_x**, además de **video\_on**, naturalmente. De esta forma, en el rango de los 640 pixeles visibles, se divide la pantalla en intervalos de 213 pixeles aproximadamente, para obtener el patrón deseado. Las restantes entradas del sistema, intervienen más bien, a la hora de interactuar con el usuario.

Las entradas **rot\_a** y **rot\_b** constituyen la interfaz para la conexión del encoder rotatorio. Estas entradas pertenecen más bien a un circuito interno diseñado para atender el encoder, denominado rotary. Los vectores de entrada **red\_ColorReg**, **green\_ColorReg** y **blue\_ColorReg**, así como **color\_selector** van directamente a este circuito. Las tres primeras establecen un valor de color personalizado que se escribe directamente de forma externa al módulo, y la última determina la activación del registro de color que será modificado por la acción del encoder.

Sin embargo, a este nivel del diseño, el circuito consta básicamente de un registro de 4 bits para cada color, al igual que en el *top level* de **VGA\_module\_self\_test** -véase **Bloque VGA\_module\_self\_test**. De igual forma, la señal **enable** controla el comportamiento del circuito, y posibilita la visualización mientras la misma se encuentre activa. En caso contrario, los registros internos mantienen sus salidas no obstante en el visualizador no se mostrará patrón alguno, ya que el módulo de sincronización **VGA\_sync\_unit** permanecerá igualmente deshabilitado. Adicionalmente, un circuito combinacional asíncrono compuesto en su mayoría por comparadores y multiplexores, será el encargado de velar por el valor del vector de posición **pixel\_x** para enviar al visualizador el valor que contenga el registro de color correspondiente al segmento de pantalla prefijado, según lo indique la señal **video\_on**.

Las demás funcionalidades quedan a cargo del bloque **rotary**, ya mencionado anteriormente, ubicado internamente dentro de **VGA\_rotary\_encoder**. El código VHDL para la implementación de este bloque (su *top level*), así como una descripción más detallada de sus pines, puede consultarse en detalle en el Anexo 5.

### 1.2.2.3.1. Bloque *rotary*

Como ya se había mencionado, este circuito es el encargado de manejar el encoder rotatorio que permite la modificación manual del valor de color de cada franja, según se seleccione. Este subcircuito es precisamente el que brinda el soporte necesario para las funcionalidades manuales que ofrece el módulo.

La Fig. 22 muestra de manera general, el bloque implementado para este circuito. Nótese la similitud entre las entradas y salidas de este subcircuito y las del *top level VGA\_rotary\_encoder* de la Fig. 21.

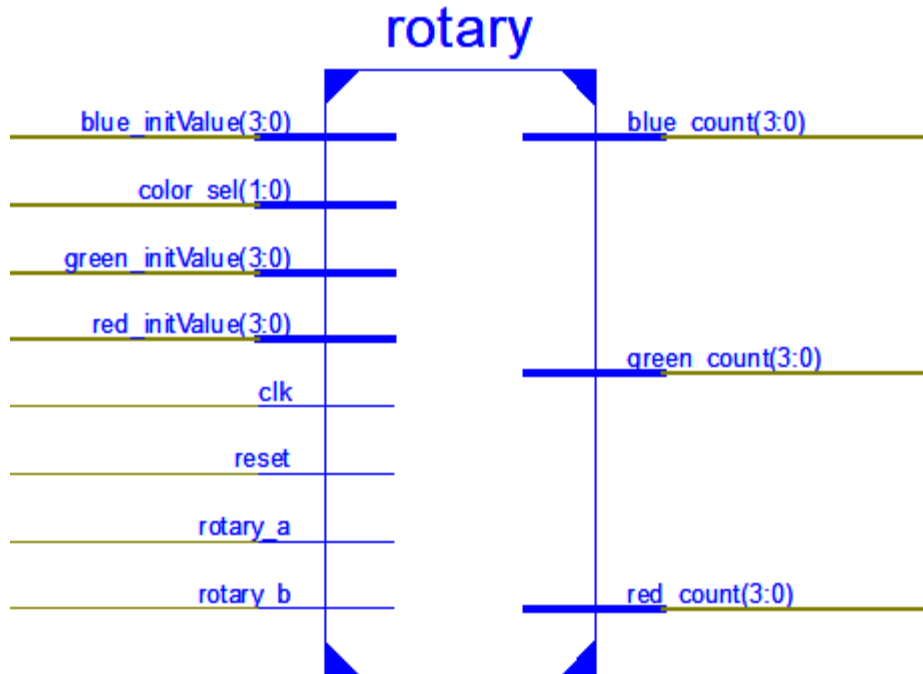


Fig. 22. Bloque *rotary* de *VGA\_rotary\_encoder*

En esencia, el subcircuito posee dos partes, una encargada de atender y decodificar el movimiento del encoder rotatorio, y otra referida a las salidas de colores.

Una representación esquemática del encoder rotatorio disponible en la placa de desarrollo *Spartan 3A Starter Ki* puede verse en la Fig. 23. Básicamente, el encoder rotatorio posee tres *switches* asociados, dos que se activan según el movimiento rotatorio, y otro relacionado con la acción de pulsar el encoder como si fuera un botón. Para la aplicación realizada, este último carece de importancia, empleándose solo los dos primeros. Estos *switches*, denominados **A** y **B** respectivamente, se activan uno primero y otro después según el sentido de movimiento.

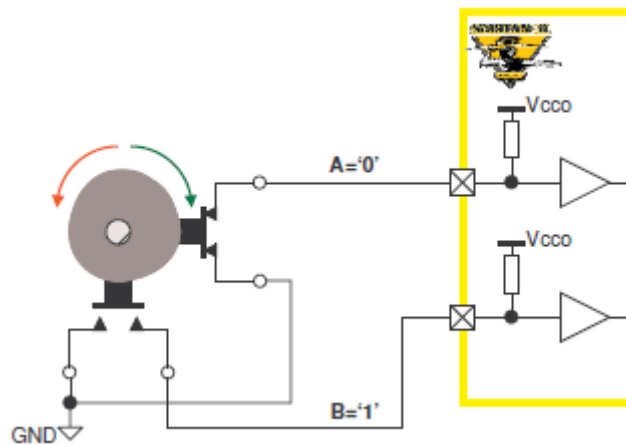


Fig. 23. Diagrama simplificado del encoder rotatorio disponible en la placa de desarrollo *Spartan 3A Starter Kit*. Tomado de [8]

De igual forma, al finalizar la acción de rotar, estos dos switches se desactivan en orden inverso al que fueron activados inicialmente. Este mecanismo permite distinguir entonces el movimiento en cualquiera de los dos sentidos, para lo cual las salidas de **A** y **B** toman la forma que muestra la Fig. 24

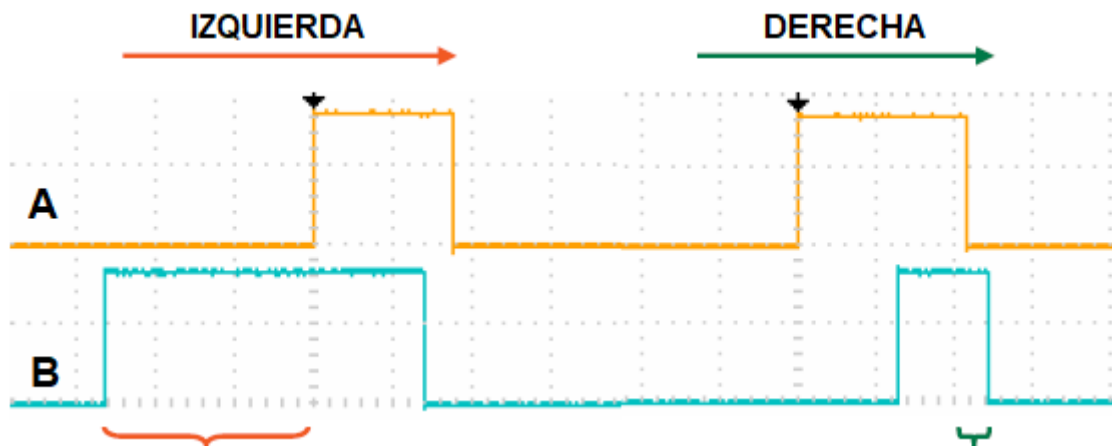


Fig. 24. Formas de onda para entradas del encoder rotatorio según dirección de movimiento. Tomado y adaptado de [8]

Las señales de la Fig. 24 constituyen precisamente las entradas a la parte del diseño dedicada al encoder rotatorio, la cual, para su atención, consta solo de dos circuitos. El primero, denominado **rotary\_filter** en la descripción VHDL del hardware, es básicamente un filtro sincronizador para eliminar el efecto del rebote y la acción mecánica sobre los pines del encoder, representados en la Fig. 22 como **rotary\_a** y **rotary\_b** (estas son entradas que vienen de **VGA\_rotary\_encoder**).

Su funcionalidad se basa en detectar solo el primer cambio en cualquiera de las dos señales de entrada, e ignorar cualquier actividad subsecuente en la misma señal, hasta que la otra cambie

también. La lógica anterior provoca que las señales obtenidas para cada entrada difieran ligeramente de las entradas originales, obteniéndose las señales **rotary\_q1** y **rotary\_q2**.

La Fig. 25 muestra las formas de onda correspondientes a **rotary\_q1** y **rotary\_q2**, respectivamente, en comparación con las dos entradas del encoder **A** y **B**. Con estas señales, aunque difieran de las originales, es posible determinar aún el sentido de rotación. Observando detenidamente la figura, puede constatarse la regla que surge para determinar la dirección: Cuando **rotary\_q1** pase de nivel bajo a nivel alto, entonces **rotary\_q2** indica la dirección de rotación.

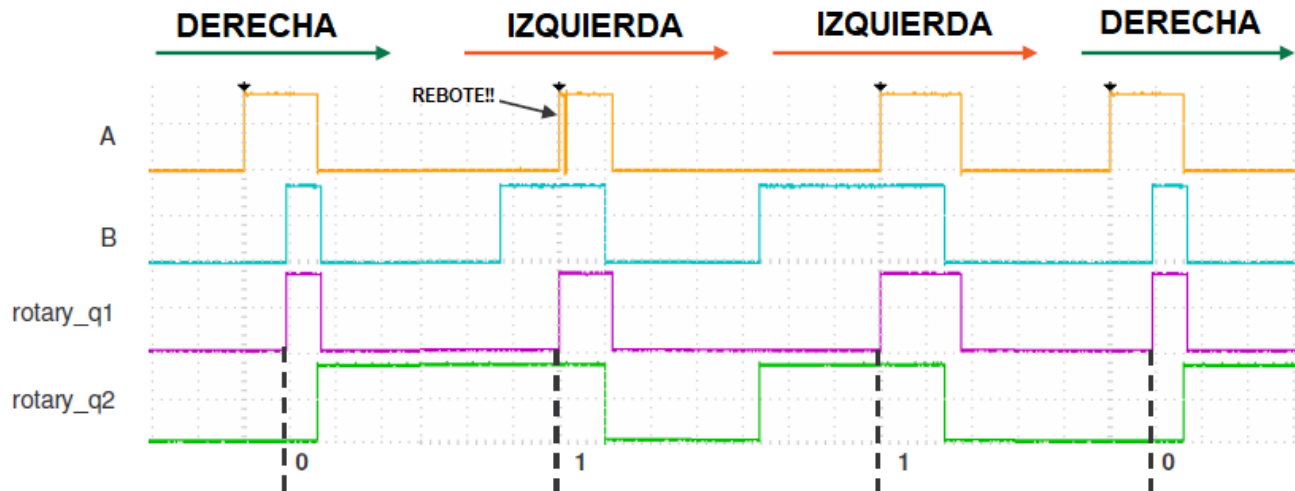


Fig. 25. Señales de entrada del encoder rotatorio, salidas del filtro y decodificación. Tomado de [8]

Precisamente, la señal **rotary\_q1** se emplea entonces para indicar la ocurrencia de un evento de rotación, mientras que **rotary\_q2** indica el sentido de la misma. Siguiendo esta lógica, el otro circuito asociado al encoder, denominado **direction**, se encarga de esto mismo, aprovechando estas dos señales para obtener a su salida una señal que indique la dirección de rotación, y otra que indique el evento. De este modo, **direction** queda, de forma simplificada, como muestra la Fig. 26.

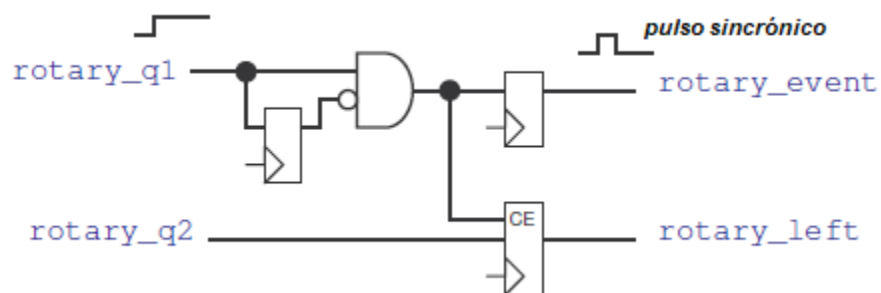


Fig. 26. Circuito implementado para determinar la activación del encoder y el sentido de movimiento.

Tomado de [8]

Una descripción mucho más detallada de los procedimientos y razonamientos realizados para el diseño del circuito que atiende el encoder, puede ser consultada en [2] y [8], materiales utilizados para el desarrollo de este segmento del diseño.



La otra parte del diseño del subcircuito **rotary**, dedicada a los colores, hace un uso exhaustivo de las señales **rotary\_event** y **rotary\_left**, mostradas en la Fig. 26. Esta parte del diseño está basada en tres contadores binarios de 4 bits ascendentes/descendentes, con entradas de reset, habilitación, carga paralela y dirección de conteo sincrónicas. Estos contadores controlan el valor binario del rojo, el verde y el azul respectivamente a enviar a la salida del bloque **VGA\_rotary\_encoder**, y con ello, al visualizador si se encuentra habilitada la selección necesaria. Cada uno de estos contadores, tiene como entrada de dirección de conteo la señal **rotary\_left**, que indicará en cada caso el incremento o decremento del valor de color según la actividad del encoder.

Adicionalmente, cada contador puede variar arbitrariamente su valor de conteo a partir de la entrada de carga paralela. En cada caso, los vectores de entrada **red\_initValue**, **green\_initValue** y **blue\_initValue** se conectan directamente a la carga paralela del contador del color correspondiente, y a partir de la señal de control de carga paralela, establecer sus valores a la salida del bloque en general. Esto permite establecer de forma externa al módulo, a través de estos vectores, un valor de color de forma manual. La señal de control de carga paralela en este caso se decodifica en un circuito independiente para cada contador, encargado de esta funcionalidad.

Este circuito, denominado **color\_change\_detector**, donde “color” se refiere a los colores rojo (*red*), verde (*green*) y azul (*blue*) en cada caso, constituye básicamente un comparador síncrono que evalúa constantemente si el valor del vector en la entrada de carga paralela difiere con su valor previo, indicándolo con la señal **changeColor** (entiéndase el sufijo **color** al igual que el caso anterior). Para ello, el circuito emplea un registro para almacenar el valor de esta entrada, que en cada ciclo de reloj pasará a ser el valor previo, al compararse con el valor actual del mismo. La Fig. 27 muestra un diagrama de bloques simplificado de la lógica concebida para este circuito.

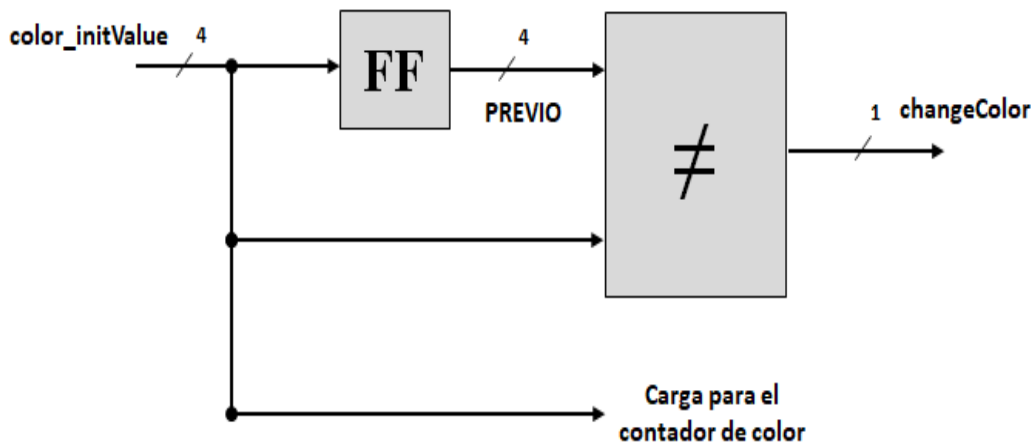


Fig. 27. Diagrama simplificado para el controlador de carga paralela de los contadores de color.

Fuente: Elaboración propia.

En cuanto a la habilitación de estos contadores, encuentra utilidad la señal **rotary\_event**, mencionada ya anteriormente. Esta señal, en conjunto con otra señal, denominada **color\_sel** (una

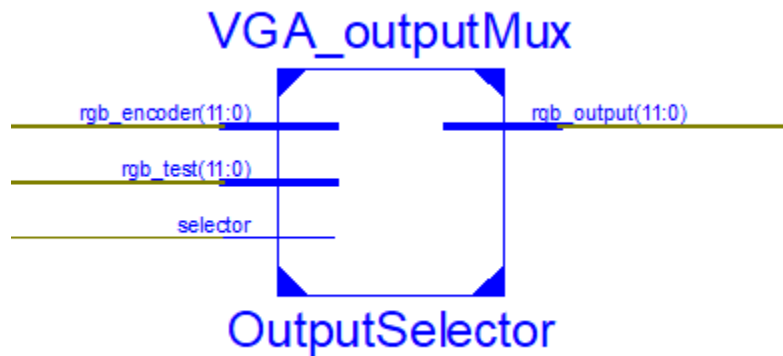
vez más, el sufijo **color** indica la existencia de tres señales, una para cada color) forman un **AND** lógico cuya salida se conecta directamente a la habilitación de estos contadores.

Las señales **color\_sel** provienen de otro circuito independiente, denominado **color\_selector**, que es quien controla la activación de cada señal **color\_sel** por separado. Este circuito de control se encuentra implementado de forma singular, dado que su función es manejar el estado de las señales **color\_sel** en función de las entradas globales del mismo nombre **color\_selector(1:0)**, descritas en epígrafe **Bloque VGA\_rotary\_encoder**. Al igual que los vectores de entrada **red\_initValue**, **green\_initValue** y **blue\_initValue**, el vector de entrada global **color\_selector(1:0)** permite seleccionar externamente de forma manual el contador de color a modificar directamente por la acción del encoder rotatorio. El circuito **color\_selector** emplea únicamente lógica combinatorial para su funcionamiento, el cual en esencia actúa como un multiplexor, poniendo en cada combinación de sus entradas de selección, el nivel alto en una de las tres señales **color\_sel** y el nivel bajo en el resto. Nótese que el vector **color\_selector(1:0)** es de dos bits, por lo que se puede decodificar de cuatro formas distintas, mas solo interesan tres de ellas; un color en cada caso, la cuarta combinación simplemente pone a cero todas las señales **color\_sel**, deshabilitando los tres contadores al mismo tiempo. De forma similar, el efecto de las entradas de selección para la habilitación de los contadores de color no influye sobre la carga paralela de cualquiera de ellos. Ello significa que el usuario puede establecer el valor de conteo en cualquier contador, en cualquier momento, independientemente de que esté habilitado para el conteo o no. La habilitación descrita anteriormente, solo restringe el efecto de la variación del encoder rotatorio sobre los tres contadores, haciendo que solo uno de ellos varíe su valor en cada caso, o ninguno, en caso de que se mantengan los tres deshabilitados.

Al igual que el resto de los bloques del diseño explicados con anterioridad, el código VHDL correspondiente al subcircuito **rotary**, así como una breve descripción de cada uno de sus pines, puede consultarse de manera íntegra en el Anexo 6, al final de este documento.

#### 1.2.2.4. Bloque VGA\_outputMux

Este bloque resulta particularmente simple, pues se trata simplemente de un multiplexor de dos entradas, a saber, las salidas de color de los bloques **VGA\_module\_self\_test** y **VGA\_rotary\_encoder**, respectivamente, y una salida que constituye la salida final de todo el sistema. La entrada de selección del bloque constituye el pin **self\_test**, entrada del módulo general, como indica la Fig. 8 -véase **Características Generales**-.

Fig. 28. Bloque **VGA\_outputMux**

Una representación esquemática de este bloque, que permite comprender más claramente su función en el sistema puede verse en la Fig. 10. El código VHDL correspondiente a este bloque puede consultarse en el Anexo 7, al final de este documento.

### 1.2.3. Funcionalidades

Las funcionalidades que posee el módulo VGA diseñado han sido en gran medida explicadas a lo largo de los epígrafes anteriores, puesto que guardan estrecha relación con la estructura interna y operación de cada una de sus partes.


Xilinx facilita el acoplamiento de cualquier hardware específico diseñado por sus usuarios al bus PLB que interconecta el sistema de procesamiento empujado basado en *MicroBlaze*, por medio de plantillas disponibles para módulos hardware de usuario. Estas plantillas permiten acoplar cualquier diseño propio como módulo IP disponible para su uso como periférico de *MicroBlaze*, independientemente de aquellos que ya se encuentran disponibles en el entorno de desarrollo EDK (*Embedded Development Kit*) de Xilinx. Constan de dos componentes fundamentales, descritos en VHDL, la lógica de usuario, o *user\_logic*, que constituye la interfaz donde irá instanciado el hardware desarrollado por el usuario, en este caso, el controlador de VGA, y el IPIF (*Intellectual-property interface*) para su acoplamiento con el bus PLB. En esta última se incluye el *user\_logic*, donde se encuentra el módulo desarrollado.

Para lograr un acoplamiento exitoso empleando estas plantillas, el diseño del módulo se realiza teniendo en cuenta las entradas que tendrá el mismo de cara al programador, para lo cual mediante el asistente del IPIF se declaran los registros correspondientes según las entradas que se requieran. Estas entradas no tendrán almacenador alguno que retenga el valor necesario, puesto que ya el asistente de creación del módulo de propiedad intelectual se encargará de crearlos, y todo lo que se tiene que hacer es simplemente interconectar en el *user\_logic* las señales correspondientes a los registros creados previamente por el asistente de Xilinx. Para aquellas entradas que quedan fuera del control del software, y son externas al módulo, como es el caso de las entradas del encoder rotatorio

en el diseño propuesto, simplemente se deben declarar a cada nivel como entradas del módulo IP en general, quedando como externas al mismo nivel de aquellas señales asociadas al bus PLB en el *top level* del hardware diseñado.

La *Tabla 5* muestra un reporte detallado del consumo de recursos del módulo VGA implementado ya interconectado al IPIF e instanciado en la lógica de usuario, listo para ser importado como módulo de propiedad intelectual compatible con bus PLB de *MicroBlaze*. Nótese la diferencia entre las estadísticas que presenta la *Tabla 4* correspondientes al mismo módulo IP aún sin interconectar y sintetizado de forma aislada; lo que evidencia un aumento del consumo de recursos debido a la presencia de las plantillas de interconexión para el bus PLB de *MicroBlaze*.

Tabla 5. Reporte de utilización de recursos del FPGA del módulo VGA implementado ya integrado al IPIF. Tomado de *Xilinx ISE Project Navigator 12.4*

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slices	414	5888	7%
Number of Slice Flip Flops	413	11776	3%
Number of 4 input LUTs	586	11776	4%
Number of bonded IOBs	257	372	69%
Number of GCLKs	2	24	8%

El módulo VGA desarrollado posee en esencia solo dos modos de operación, denominados **Modo Manual** y **Modo automático** o **Modo self test**, así como cinco registros en total para el control de la operación de todo el sistema. En cualquiera de los dos modos de operación, se hace un uso exhaustivo del bloque **VGA\_sync\_unit** descrito con anterioridad, más el **Modo Manual** es controlado a través del bloque **VGA\_rotary\_encoder** mientras que el **Modo Automático** se maneja mediante el bloque **VGA\_module\_self\_test**. El paso de un modo a otro es controlado mediante un registro de control que permite modificar mediante software las señales **enable** y **self\_test** respectivamente.

La *Fig. 29* muestra la estructura de este registro, así como el orden y descripción de la funcionalidad de cada uno de sus bits. Este registro puede ser leído y escrito y su organización se encuentra en correspondencia con el formato de datos *big-endian* que soporta el procesador en el que está basado el resto del sistema.

**CTRL\_REG – CONTROL REGISTER (ADDRESS: VGA\_MODULE\_BASEADDR + 0x10)**

		R/W	R/W
<b>RESERVADO</b>		<b>EN</b>	<b>ST</b>
0			31
<b>Bit 31</b>	<b>ST: Self Test bit</b> <u>Si EN = 1:</u> 0 = Modo Manual Activado 1 = Modo Automático Activado <u>Si EN = 0:</u> Este bit se ignora		
<b>Bit 30</b>	<b>EN: Global Enable bit</b> 0 = Módulo VGA desactivado 1 = Módulo VGA activado		
<b>Bit 0-29</b>	<b>Reservado</b>		

Fig. 29. Registro de control **CTRL\_REG**

El sistema en general hace uso del LCD disponible en la placa de desarrollo de FPGA empleada, a través de puertos de E/S de propósito general (GPIO, por sus siglas en inglés) como elemento visualizador de información y estado de la aplicación. Mediante el mismo, el programa de aplicación muestra en todo momento del modo actual del módulo VGA, y para el caso manual, los últimos valores escritos en cada franja de color. De igual modo, el LCD indica cuando el módulo se encuentra deshabilitado globalmente. La *Fig. 30* muestra el aspecto de la placa de desarrollo de FPGA mientras el software de aplicación ha configurado el módulo VGA en **Modo Manual** y se encuentra visualizando precisamente los patrones que muestra la *Fig. 33*.

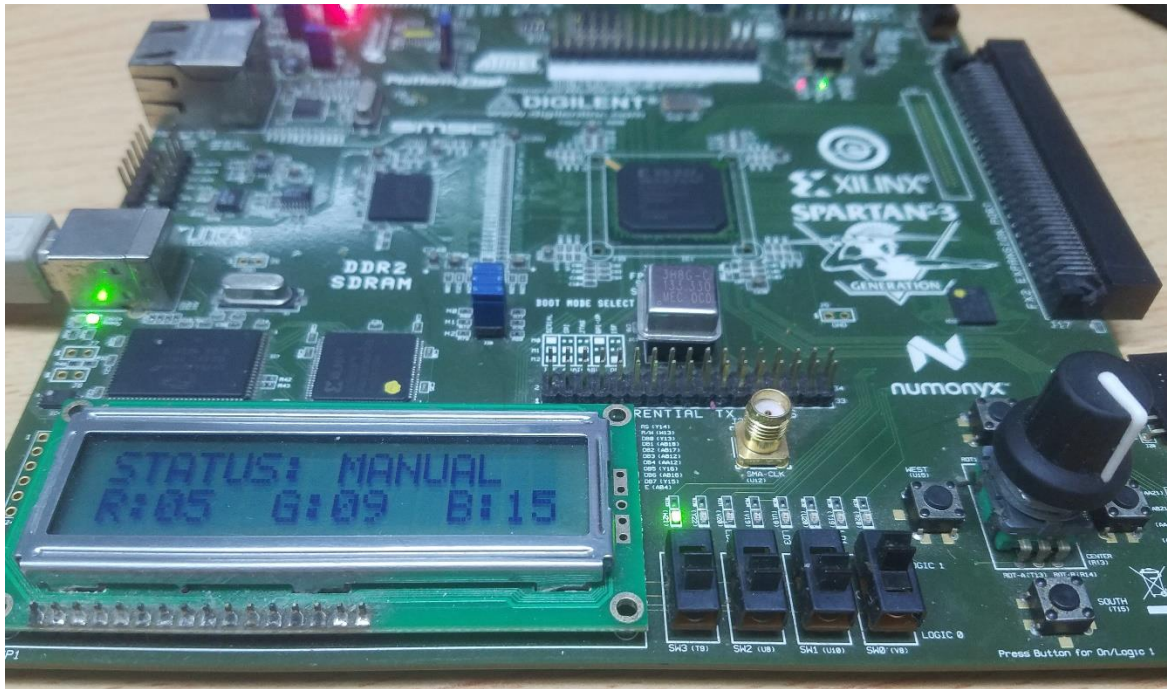


Fig. 30. Información del estado del programa de aplicación en **Modo Manual**. Fuente: Elaboración propia.

Cuando se cambia el modo de operación del módulo VGA a automático, tal información puede verse reflejada igualmente en el display LCD de la placa de desarrollo, aunque se siguen mostrando los valores escritos anteriormente en los registros de colores asociados al **Modo Manual**. La Fig. 31 muestra el aspecto de la placa mientras el módulo VGA permanece en el **Modo Automático**.



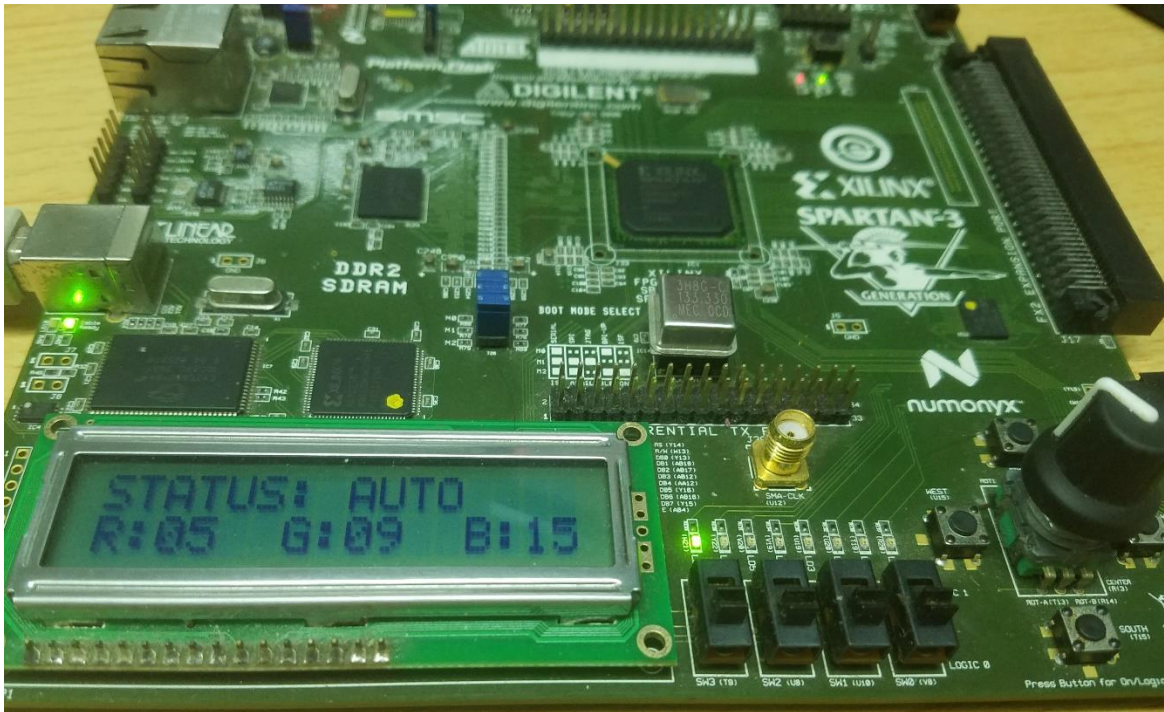


Fig. 31. Información del estado del programa de aplicación en **Modo Automático**. Fuente: Elaboración propia.

Para el caso en el que el programa de aplicación manda a deshabilitar el módulo VGA, el display LCD igualmente muestra esta información, pero sin embargo mantiene aún los últimos valores de color escritos en los registros de color correspondientes. La Fig. 32 muestra el estado del display LCD en la placa de desarrollo para este caso en particular.



Fig. 32. Información del estado del programa de aplicación mientras permanece deshabilitado el módulo VGA. Fuente: Elaboración propia.

Las demás funcionalidades específicas asociadas a cada bloque del diseño serán explicadas en los epígrafes subsecuentes de este informe.

### 1.2.3.1. Modo manual

Tal como se había mencionado, el modo manual, está controlado únicamente por el bloque **VGA\_rotary\_encoder** descrito con anterioridad -véase **Bloque VGA\_rotary\_encoder**-. Este modo genera un patrón de tres franjas de colores verticales estáticas en pantalla, cuyo color podrá ser modificado ya sea mediante software o mediante la acción del encoder rotatorio, con previa activación de la franja a modificar, lo cual se realiza igualmente mediante software. La *Fig. 33* muestra el patrón de colores obtenido al habilitar este modo de operación. En la figura, se puede observar a la izquierda el patrón de franjas por defecto que aparece en pantalla, mientras que, a la derecha se encuentra el mismo patrón, pero con las franjas modificadas.



Fig. 33. Patrón de colores en modo manual. Por defecto (izquierda) y modificado (derecha). Fuente: Elaboración propia

Es en este modo de operación donde radica casi toda la interacción que permite el módulo con el usuario. Una vez activado el bit correspondiente para su habilitación, es posible escribir el valor de color deseado en cada una de las franjas a través de tres registros dedicados al efecto. Estos tres registros hacen uso de las entradas de carga paralela de los contadores binarios implementados en el bloque **rotary**.

La estructura y organización de los registros implementados para la modificación de los colores rojo, verde y azul aparecen representados en la *Fig. 34*, *Fig. 35* y *Fig. 36* respectivamente. De igual forma, en estas mismas figuras, se muestran las direcciones de estos registros tomando como referencia la dirección base donde se mapee el módulo en general dentro del espacio de direcciones del bus del procesador, denominada **VGA\_MODULE\_BASEADDR**; la capacidad de lectura o escritura de cada bit por separado y su condición inicial de reset.



**RED\_REG – RED COLOR REGISTER (ADDRESS: VGA\_MODULE\_BASEADDR + 0)****Bit 29-31 RD3:RD0: Color data bits**Si EN = 0:

Estos bits se ignoran

Si EN = 1:

RD3:RD0: Unsigned 4-bit Red Color data bits

**Bit 0-28 Reservado**Fig. 34. Registro de dato para el color rojo **RED\_REG****GREEN\_REG – GREEN COLOR REGISTER (ADDRESS: VGA\_MODULE\_BASEADDR + 4)****Bit 29-31 GR3:GR0: Color data bits**Si EN = 0:

Estos bits se ignoran

Si EN = 1:

GR3:GR0: Unsigned 4-bit Green Color data bits

**Bit 0-28 Reservado**Fig. 35. Registro de dato para el color verde **GREEN\_REG**

**BLUE\_REG – BLUE COLOR REGISTER (ADDRESS: VGA\_MODULE\_BASEADDR + 8)****Bit 29-31 BL3:BL0: Color data bits**Si EN = 0:

Estos bits se ignoran

Si EN = 1:

BL3:BL0: Unsigned 4-bit Green Color data bits

**Bit 0-28 Reservado**Fig. 36. Registro de dato para el color azul **BLUE\_REG**

Cualquiera de estos tres registros puede ser escrito independientemente de que se encuentre o no habilitado el **Modo Manual**, puesto que son valores independientes del resto del resto del módulo. En este caso, aunque no se encuentre activado, al volver al mismo, en pantalla aparecerán las franjas coloreadas según el valor que haya sido escrito en cada caso.

De forma similar, para la escritura en estos registros no se toma en cuenta los valores de selección en el registro de selección implementado en el módulo. Este quinto registro es quien determina la franja de color que podrá ser modificada manualmente desde el encoder rotatorio, puesto que hace uso de las entradas de selección del módulo denominadas **color\_selector(1:0)**. La Fig. 37 muestra la organización y estructura de este registro, junto con la funcionalidad de cada bit y el estado inicial de reset.

**SEL\_REG – COLOR SELECTOR REGISTER (ADDRESS: VGA\_MODULE\_BASEADDR + 12)****Bit 30-31 SEL1:SEL0: Color Selector Bits**Si EN = 0:

Estos bits se ignoran

Si EN = 1:

00 = El encoder rotatorio modifica la franja roja

01 = El encoder rotatorio modifica la franja verde

10 = El encoder rotatorio modifica la franja azul

11 = El encoder rotatorio no modifica ninguna franja

**Bit 0-29 Reservado**Fig. 37. Registro de selección de franja de color a modificar por el encoder **SEL\_REG**

### 1.2.3.2. Modo automático

Mediante el registro de control puede establecerse el modo de operación automático del módulo VGA. Sobre este modo, no tienen efecto los demás registros descritos anteriormente; pues el módulo toma valores fijos cableados internamente para visualizar la información de color correspondiente. Como ya se había mencionado, el patrón en este caso está formado por 48 franjas de color que se obtienen variando cada vector de color de uno en uno, a lo largo de los 16 valores posibles para cada uno de ellos. En total, se obtienen  $16+16+16$  franjas de colores desplegadas a lo largo de la pantalla los cuales alternarán su dirección de horizontal a vertical y viceversa de modo indefinido hasta que no se cambie al **Modo Manual**. La Fig. 38 muestra el patrón generado en este modo de operación, así como las dos posiciones que adopta en pantalla cada 1s.

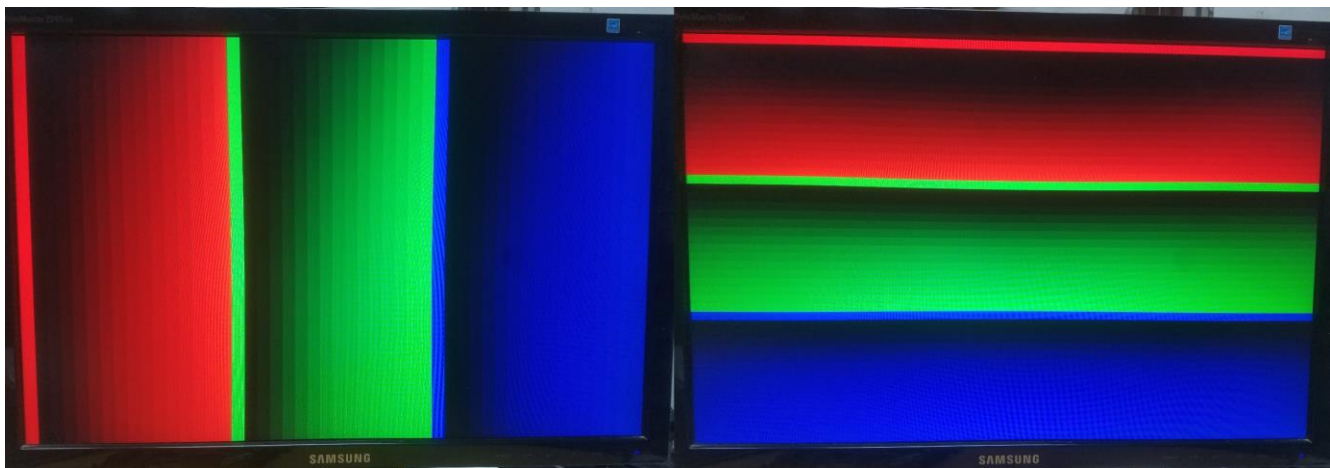


Fig. 38. Patrón generado en el **Modo Automático** alternado cada 1s en pantalla. Fuente: Elaboración Propia.

### 1.2.3.3. Comandos de operación

Para la interacción con el sistema, se utiliza además del encoder rotatorio, un conjunto de comandos determinados desde la aplicación que ejecuta el procesador que son enviados al sistema de procesamiento empujado vía UART, aprovechando las funcionalidades que brinda este módulo IP disponible en el repositorio de módulos IP que brinda Xilinx. Empleando una comunicación serial a 38400 baudios, 8 bits de trama, 1 bit de parada y sin paridad, a partir de una aplicación de alto nivel ejecutada en la PC, es posible el envío de estos comandos y manipular todo el conjunto de registros que posee el módulo IP de VGA diseñado. Estos comandos, añaden otra capa de interacción con el usuario, y evidencian una vez más la estrecha relación que existe entre la plataforma hardware y la plataforma software diseñadas en conjunto.

De forma general, los comandos definidos para la interacción remota con el sistema son enviados en formato ASCII con un tamaño de 2 bytes cada uno, y responden a la sintaxis y funcionalidad que muestra la

Tabla 6. Listado y descripción de comandos habilitados para la interacción remota con el sistema de procesamiento empotrado diseñado en el FPGA.

Comando	Sintaxis (ASCII)	Función
Escribe ROJO	CRXX	Escribe el valor XX en el registro <b>RED_REG</b> . XX representa un número del 00 al 15 especificado en ASCII, donde el primer valor será la decena (incluyendo el 0) y el segundo la unidad. Si la decena es un número distinto de 0 ó 1, se asume que el valor a escribir se encuentra en el rango de 0 a 9, y no es necesario especificar el próximo byte; de lo contrario es necesario que el segundo byte se encuentre en el rango de 0 a 5. Cualquier otro caso provoca la anulación total del comando y no tiene efecto sobre la aplicación.
Escribe VERDE	CGXX	Idem al anterior, pero para <b>GREEN_REG</b> .
Escribe AZUL	CBXX	Idem al anterior, pero para <b>BLUE_REG</b> .
Lee estado	RSXX	Lee el estado XX solicitado directamente desde memoria EEPROM vía IIC, y sobrescribe los valores del módulo VGA con los valores leídos. El estado XX especificado debe encontrarse en un rango válido al igual que CRXX, CGXX y CBXX, con la condición adicional de que se encuentre previamente almacenado en memoria, de lo contrario se envía un mensaje de error. Cualquier otro caso provoca la anulación total del comando y no tiene efecto sobre la aplicación.
Salva estado	SS	Salva los estados de todos los registros del módulo VGA en un buffer de escritura que luego será transferido a la memoria serial EEPROM, vía IIC, hasta que se alcance un total de 10 muestras, visualizando en cada caso la cantidad de muestras almacenadas en un display 7 segmentos vía IIC. Una vez alcanzada esta condición, se indica mediante un led y cualquier otra solicitud de salvar estado será ignorada
Modo Automático	PP	Provoca que el módulo VGA entre en <b>Modo Automático</b> modificando el bit correspondiente en <b>CTRL_REG</b>
Modo Manual	PC	Provoca que el módulo VGA entre en <b>Modo Manual</b> modificando el bit correspondiente en <b>CTRL_REG</b> . No afecta valores de color

Deshabilita VGA	PD	Deshabilita globalmente el módulo VGA modificando el bit correspondiente en <b>CTR_REG</b> . No afecta valores de color.
Habilita VGA	PE	Habilita globalmente el módulo VGA modificando el bit correspondiente en <b>CTRL_REG</b> . No afecta valores de color.

Si se observa detenidamente la tabla anterior, el lector podrá constatar que no existen comandos para establecer la franja de color a modificar por el encoder rotatorio en el **Modo Manual**. Precisamente esta función se realiza a través de la interacción con los switches igualmente presentes en la placa de desarrollo empleada. Estos switches se encuentran conectados a un GPIO de 4 bits, atendido por interrupción.

Todos ellos poseen la misma fuente de interrupción, dado que se encuentran conectados al mismo GPIO, activada en este caso por cualquier cambio de estado que ocurra en cualquiera de los switches asociados. Cada uno de ellos, aunque comparten la misma fuente de interrupción, tiene asociado mediante software una funcionalidad distinta, que, de manera general, se basa en la modificación del registro **SEL\_REG** para la selección de la franja de color a modificar mediante el encoder. La *Tabla 7* muestra las funcionalidades asignadas a cada uno de estos switches y su descripción.

Tabla 7. Funcionalidad de los switches disponibles en la placa de desarrollo.

SWITCH (Nombre en la placa de desarrollo)	FUNCION
SW3	Selecciona la franja roja. Activa led LD7 para indicarlo.
SW2	Selecciona la franja verde. Activa led LD6 para indicarlo.
SW1	Selecciona la franja azul. Activa led LD5 para indicarlo.
SW0	Conmuta entre <b>Modo Automático</b> y <b>Modo Manual</b> según corresponda.

### 1.3. Utilización de periféricos para MicroBlaze

La utilización de periféricos en un sistema de procesamiento basado en el procesador *MicroBlaze* resulta un aspecto esencial en el desempeño del mismo. Para el diseño de un sistema basado en procesador empotrado en FPGA, Xilinx facilita en su paquete de herramientas EDK, además de los entornos de desarrollo para las plataformas software y hardware, un conjunto o repositorio de módulos IP hardware disponibles en versiones libres de costo y de uso, para la interconexión con el procesador que gobierna el sistema. Igualmente, Xilinx pone a disposición del desarrollador una documentación muy completa, necesaria para el trabajo con estos módulos IP, además del soporte

software necesario para la manipulación vía firmware, compuesta por diversos drivers, de alto y bajo nivel y la documentación asociada, ejemplos de aplicación y plantillas de software, así como bibliotecas específicas para la depuración, los flujos de entrada o salida estándar, sistemas operativos, y muchos otros.

Para el trabajo a desarrollar, han sido empleados muchos de estos módulos IP de periféricos que brinda Xilinx, a excepción del controlador de VGA, el cual constituye un diseño propio, que ha sido incorporado al flujo de diseño de XPS para su acoplamiento al bus del procesador *MicroBlaze*.

El sistema ha sido concebido de la forma que se muestra en la *Fig. 39*. En esta figura, se muestran todos los periféricos empleados para el desarrollo interconectados al bus PLB, así como las señales de E/S externas que contiene el sistema. Puede observarse también la presencia de un módulo de depuración denominado **mdm** acoplado al procesador, así como conexión de un controlador de memoria externa **SDRAM DDR2**, disponible en la placa de desarrollo de FPGA empleada, para el almacenamiento de la información de color proveniente del exterior a través de la interfaz de puerto serie **xps\_uartlite**. Esta memoria se encuentra también dispuesta para futuras expansiones del sistema, así como para la gestión de datos y constantes de programa de forma temporal.

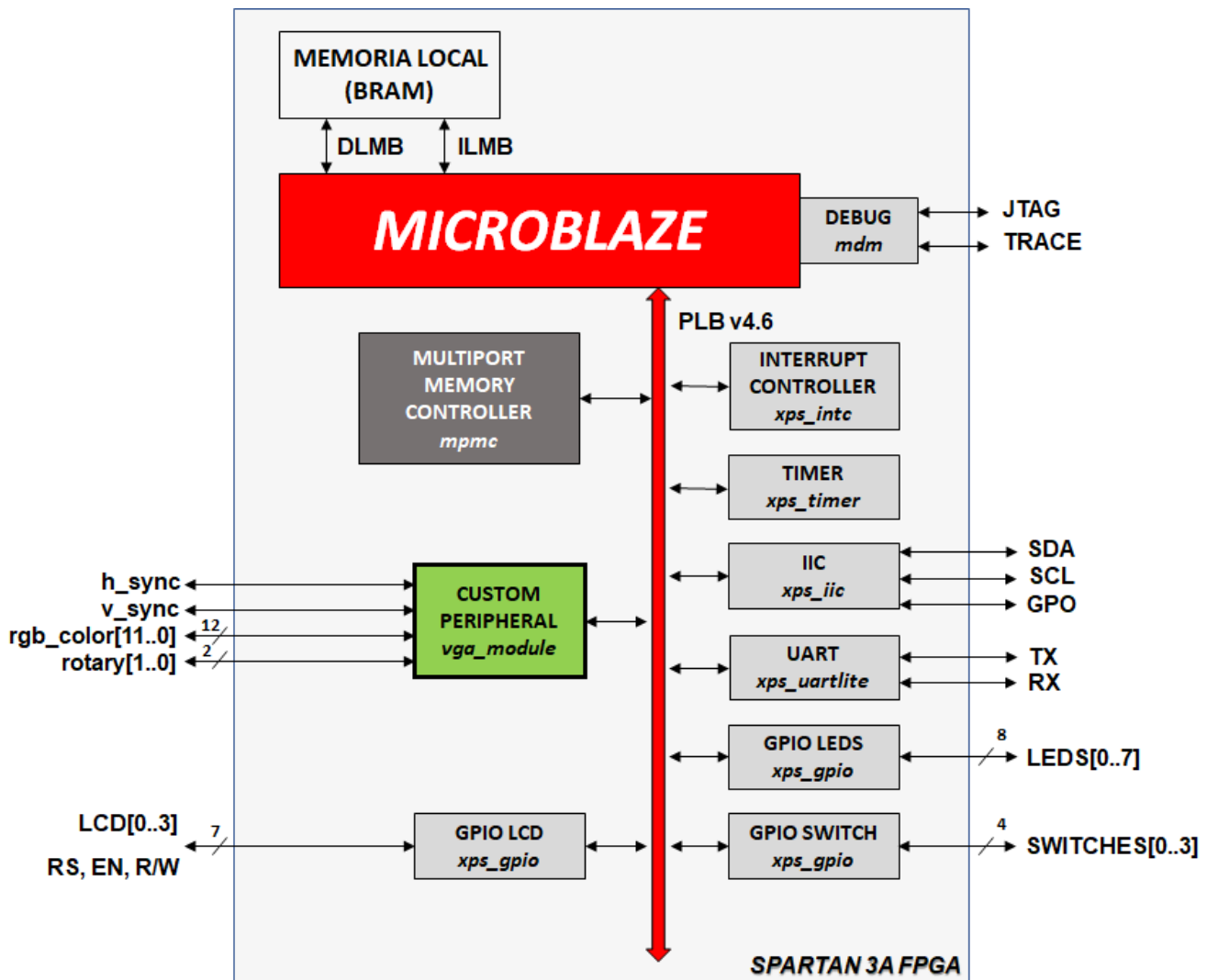


Fig. 39. Representación esquemática simplificada del diseño realizado basado en el procesador *MicroBlaze*. Fuente: Elaboración propia.

Un controlador de interrupciones se encarga de atender todas las fuentes de interrupción asociadas a los diversos periféricos que componen el sistema. De esta forma, se gestionan los eventos de cambio en el estado de los switches, la recepción o transmisión por la interfaz UART, eventos asociados al protocolo y la comunicación mediante IIC, a través del módulo *xps\_iic*, y eventos de temporización asociados a *xps\_timer*.

El entorno de diseño para el diagrama de la figura anterior, se muestra en la Fig. 40, donde se pueden observar, además, desde el aspecto de la ventana de diseño de XPS, las interconexiones de los distintos periféricos empleados a los distintos buses que conforman un sistema de desarrollo empotrado basado en *MicroBlaze*.

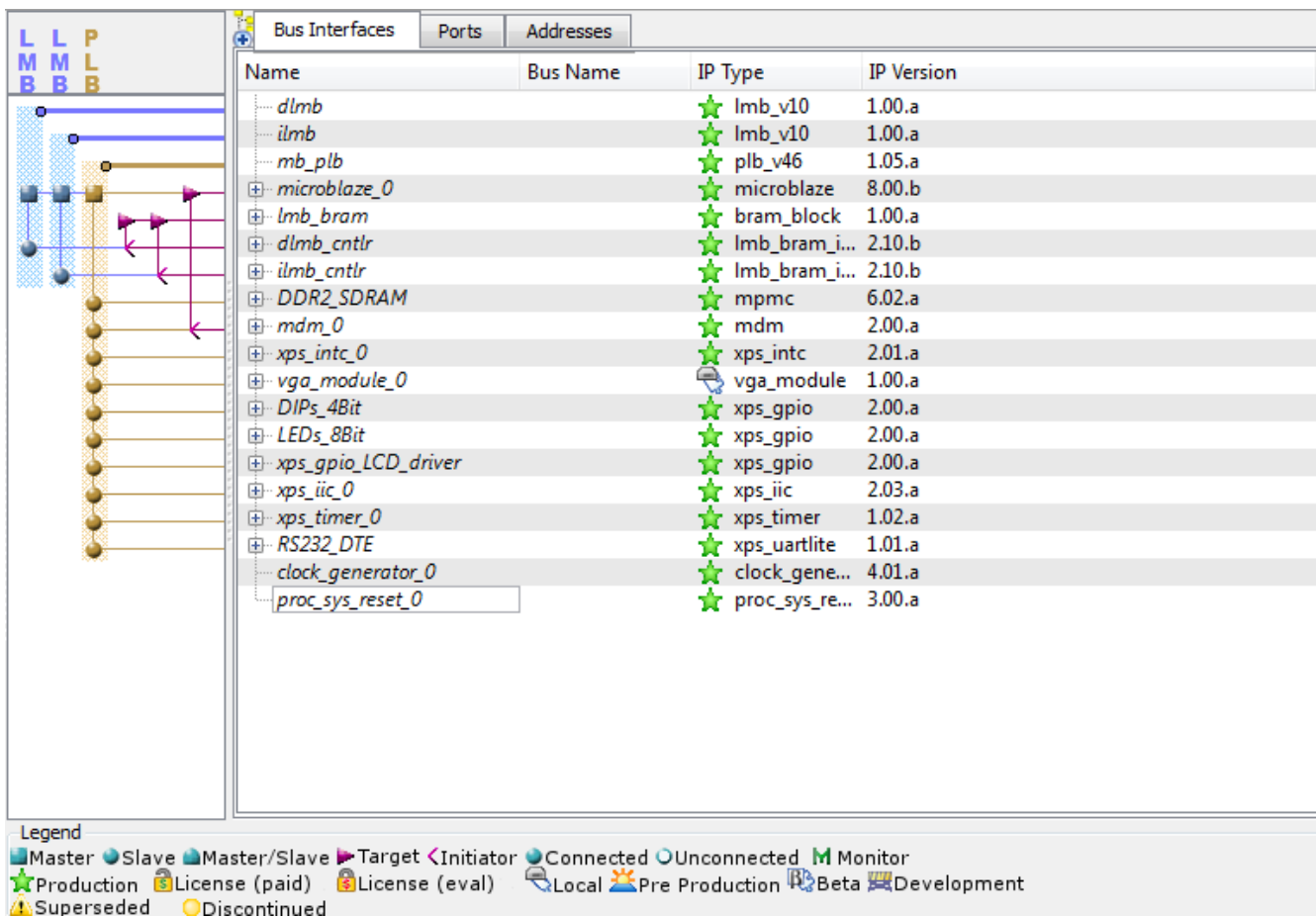


Fig. 40. Listado de periféricos y conexiones para plataforma hardware diseñada mediante la interfaz de XPS. Fuente: Elaboración propia.

En esta figura puede constatar la presencia de otros módulos IP hasta el momento no mencionados, ni ilustrados en la Fig. 39, como el módulo para la generación de la señal de reloj para todo el sistema, así como el módulo para generar la secuencia de RESET de MicroBlaze y el resto de los periféricos asociados.

Estos módulos influyen significativamente en el funcionamiento del sistema resultan indispensables en cualquier sistema de procesamiento, por lo que siempre se encuentran presentes en el flujo de diseño, aunque no son accesibles directamente por software, dado que no poseen drivers ni registros asociados. Su parametrización está dada mediante el fichero de especificaciones de hardware asociado al proyecto en XPS, el cual puede consultarse de manera íntegra en el Anexo 9 al final de este documento.



## 1.4. Programa de Aplicación

### 1.4.1. Características Generales

Para el desarrollo del programa de aplicación se empleó el entorno de *SDK* de *Xilinx EDK 12.4*. La compilación del programa arroja las estadísticas que muestra la *Tabla 8*.

Tabla 8. Estadísticas de compilación para el programa de aplicación concebido.

Segmento	.text	.data	.bss	Total
Tamaño (bytes)	15330	360	2158	17848

El programa principal hace uso intensivo de los *drivers* que componen la capa de aplicación más baja para la interacción con el hardware previsto. Estos *drivers*, disponibles libremente dentro del paquete de herramientas EDK de Xilinx, ofrecen para cada periférico para el que están concebidos, un amplio conjunto de funciones básicas necesarias para la interacción con cada uno de los registros accesibles por el procesador en cada caso, permitiendo de esta forma su configuración y control al más bajo nivel. Naturalmente, esto conlleva una complejidad adicional a la hora de programar funcionalidades complejas y modos de interacción específicos, pues se hace necesario implementar ciertos procedimientos de forma manual, partiendo de estas funciones elementales, que básicamente solo leen y escriben los registros disponibles en cada periférico; pero en cambio, ofrecen ventajas aún mayores en cuanto a rapidez en la ejecución, y menor tamaño del código, lo que permite un ahorro sustancial de los recursos del FPGA, como la memoria local.

En general, la gran mayoría de la lógica que sigue el programa, se encuentra implementada directamente en el programa principal, el cual inicialmente configura e inicializa según los requisitos de operación todos los periféricos necesarios para el posterior funcionamiento del conjunto, así como establece condiciones iniciales en las variables de programa, para luego caer en un ciclo infinito en el cual constantemente se chequean banderas que controlan la ejecución oportuna de determinados segmentos de código que se corresponden con las funcionalidades mencionadas en los epígrafes anteriores.

El resto del código implementado contempla las rutinas de interrupción para la atención de la recepción por el puerto serie, los switches y un temporizador en cada caso, además de otras funciones auxiliares implementadas para su empleo en determinados puntos del programa principal. Las mencionadas banderas que rigen la ejecución de cada tarea desde el programa principal, se controlan precisamente desde las rutinas de atención a la interrupción; de esta forma, se logra una menor densidad de código en estas últimas, permitiendo con ello su rápida ejecución y un aumento de la fiabilidad del sistema, ya que contribuye significativamente a impedir el anidamiento de interrupciones en el sistema, y con ello, una mayor velocidad de respuesta a las mismas.

Por otro lado, el programa hace uso de tablas de memoria, en las cuales se almacenan cadenas de caracteres correspondientes a carteles y tablas para lámparas de siete segmentos, así como de una librería desarrollada específicamente para el manejo del display LCD mediante 4bits de datos en solo lectura a través de GPIO, incluida en el proyecto.

### 1.4.2. Tareas del programa principal

Es precisamente en el programa principal donde se concentran la gran mayoría de las implementaciones realizadas para cada funcionalidad prevista. En esencia, el programa principal tiene tres funciones bien definidas: la configuración e inicialización de variables y periféricos, la actualización de visualizadores y la ejecución de comandos.

La configuración e inicialización de variables y periféricos se trata de establecer condiciones iniciales tanto en el programa como en el hardware antes de comenzar a realizar las tareas previstas. Para el caso del programa, se trata de la inicialización de banderas, contadores, entre otros, previo a su utilización en la lógica del software, y en el caso del hardware, de modificar los registros de control para establecer el modo de operación requerido en cada periférico, y de establecer las condiciones iniciales necesarias en sus salidas. Esto incluye configurar el GPIO empleado para el manejo de LCD y los LEDs como salidas y el GPIO de los switches como entradas, el módulo de IIC, el de puerto serie (UART), el temporizador, el módulo para VGA diseñado y el controlador de interrupciones que se encargará de manejar las tres fuentes de interrupción dispuestas en el sistema. De igual forma, se establece también la tabla de vectores de interrupción asociada a cada ISR, así como la llamada a funciones del SO *standalone* de *MicroBlaze* para la habilitación de la solicitud de interrupciones en el procesador. Estas configuraciones iniciales previas llevan consigo naturalmente la inicialización de todos los periféricos mencionados, lo cual no precisamente representa su configuración. En este caso, se trata del establecimiento de condiciones iniciales hardware teniendo en cuenta el estado de las salidas de interés después del evento de RESET. Esto significa, por ejemplo, encender inicialmente el led correspondiente a la franja de color activa en la pantalla, y apagar los demás, lo cual implica a su vez una escritura inicial de los registros de control de VGA para que el modo por defecto después de RESET sea el manual, así como de algún valor de color en sus registros para que inicialmente se pueda visualizar algo en pantalla; así como de un texto inicial en el LCD y la escritura por IIC al GPIO de expansión de IIC conectado al bus para establecer el número 0 en la pantalla siete segmentos.

Precisamente, el establecimiento de condiciones iniciales en el LCD, así como en la lámpara 7 segmentos, hace uso de la segunda funcionalidad del programa principal, que es la actualización de visualizadores. Ya una vez concluida la primera etapa, se entra en un ciclo infinito en el cual se pregunta continuamente por tres banderas fundamentales, una para actualizar leds, otra para

actualizar carteles en el LCD y otra que indica si se recibió algo vía UART, esta última constituye la tercera funcionalidad del programa principal mencionada anteriormente. La Fig. 41 muestra el diagrama de flujo correspondiente al programa principal, el cual hace referencia al nombre específico de las banderas empleadas en el código de aplicación desarrollado.

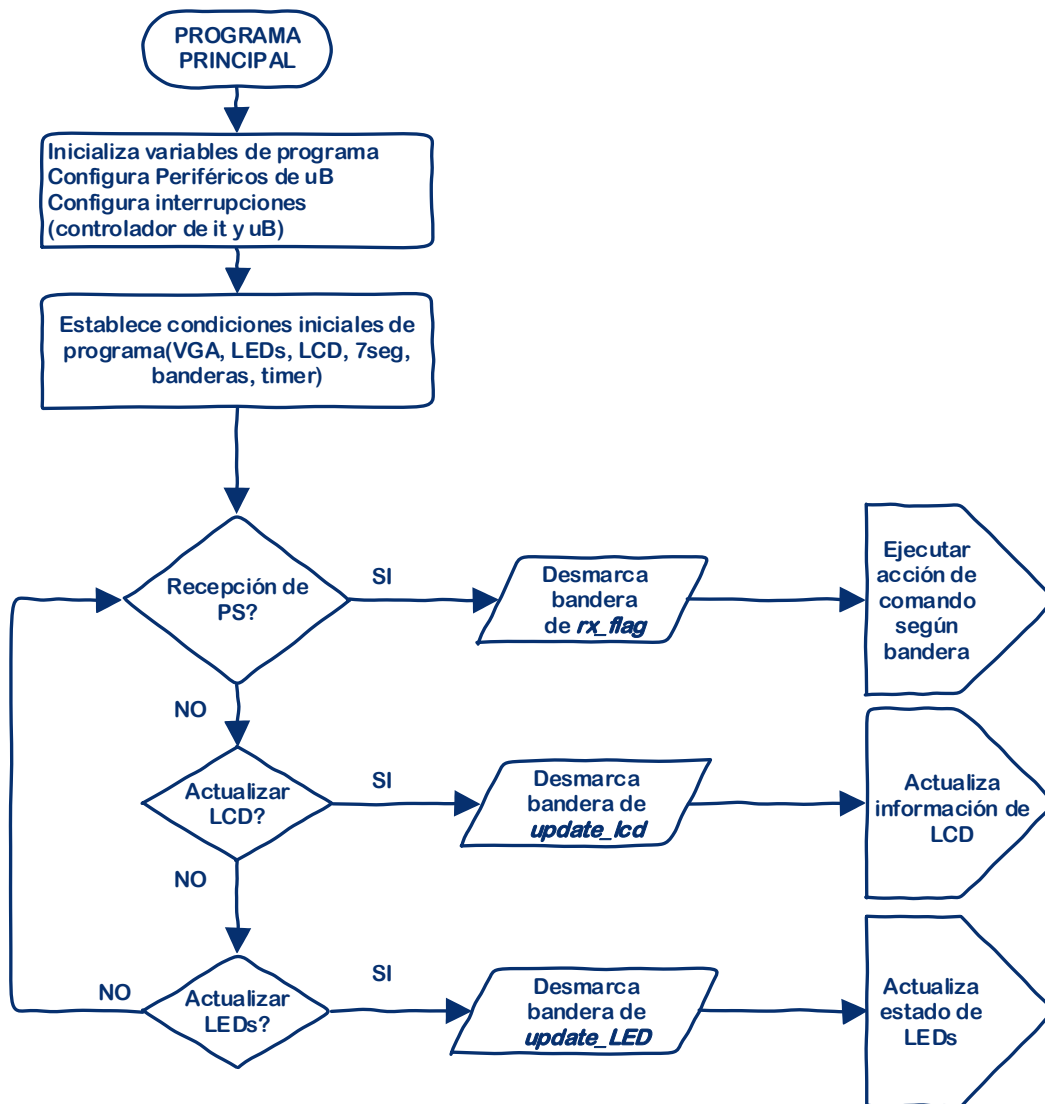


Fig. 41. Diagrama de flujo de programa principal.

Como se aprecia en el diagrama anterior, la ejecución de cada tarea se controla mediante una bandera, la cual se marca o desmarca convenientemente desde cualquier lugar del programa, dando lugar a la ejecución del segmento de código correspondiente. Activando cualquiera de estas banderas, o incluso varias a la vez, garantiza que el procesador ejecute el código necesario, aun habiendo varias activadas, de forma secuencial y ordenada, dado que la inmensa mayoría del tiempo,

este se encuentra preguntando por cada bandera cíclicamente, de forma ordenada según fue escrito el código.

Para el proyecto, el empleo del LCD tiene como objetivo indicar en cada momento el estado del módulo VGA, ya sea **deshabilitado**, **modo manual** o **modo automático**, mostrando un cartel en cada caso y el valor de cada registro de color, para lo cual se hace necesario la lectura, en primer lugar, del registro de control para conocer el estado, y en segundo lugar, de los registros de color para el rojo, el verde y el azul, para luego escribir los carteles pertinentes en las zonas correspondientes del LCD. Este proceso de actualización del LCD se realiza de forma periódica cada 1 segundo temporizado por un *timer*, cuya interrupción invoca una ISR cuyo único objetivo es precisamente marcar esta bandera. En otros casos mucho más específicos, como la indicación de un error en algún otro proceso, se marca directamente esta bandera para restablecer la información en el LCD luego de haber borrado su contenido para mostrar un cartel de error o alguna otra información casual. La Fig. 42 muestra el diagrama de flujo correspondiente a este proceso.

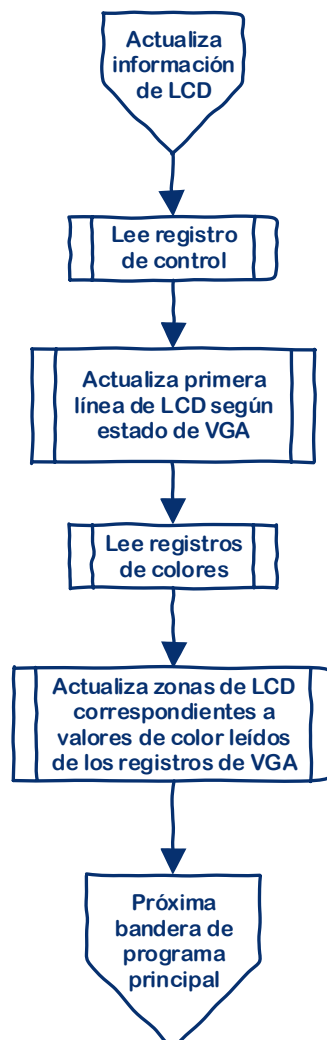


Fig. 42. Diagrama de flujo para la actualización del LCD.

En el caso de los LEDs su actualización ocurre de forma similar. En este caso, se hace uso de 5 leds, cada uno de los cuales tiene un significado específico. La funcionalidad de los tres primeros se corresponde con la operación de los switches según la *Tabla 7*, el cuarto indica condición de EEPROM llena y el quinto un error en la recepción de una imagen vía UART, ya sea porque fue cancelado su envío o porque se recibió el comando de fin de transmisión antes de recibir todos los bytes de imagen esperados -véase **Comandos de imagen**-. Particularmente, la operación de actualización de leds no ocurre de forma periódica, sino que se mandada a ejecutar manualmente desde los segmentos de código correspondientes, dado que su estado solo puede ser modificado por la ocurrencia de las condiciones de programa descritos anteriormente para cada led y no mediante ningún comando. En cualquier caso, cada vez que se ejecuta este segmento de código, se lee el registro de selección del VGA para determinar el valor a escribir en cada led mediante escrituras no destructivas al GPIO que los controla. La *Fig. 43* muestra el diagrama de flujo correspondiente a la lógica implementada.

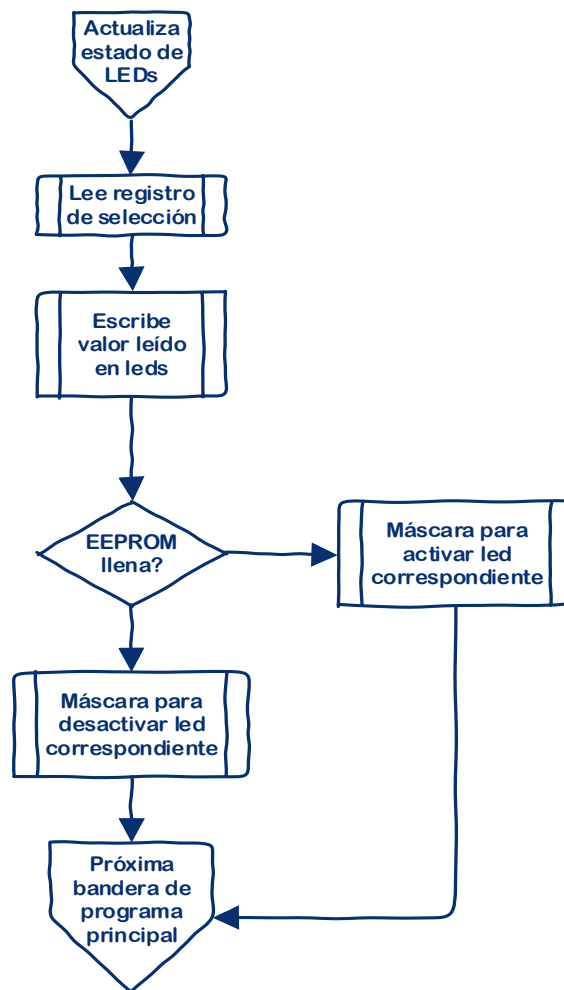


Fig. 43. Diagrama de flujo para la actualización de los LEDs

### 1.4.3. Atención a comandos

Como ya se había mencionado, la mayor parte de la lógica implementada en el programa principal se encuentra dedicada a la atención de todos los comandos validados en el sistema. Según el diagrama de la *Fig. 41*, cada bandera contenida en el ciclo infinito al final del programa principal controla la ejecución de una tarea distinta, según el caso. Una de estas banderas es precisamente la bandera de recepción por UART, la cual, una vez activa, permite que el procesador chequee otro conjunto de tareas controladas de igual manera por banderas, mucho más específicas, según el comando recibido.

En general, la activación de la bandera de recepción por puerto serie ocurre cada vez que el procesador entra a ejecutar la ISR correspondiente al módulo de UART cuando se recibe un byte cualquiera -véase **ISR de UART**-. La activación de esta bandera permite al procesador entrar a revisar las condiciones de banderas que se encuentran anidadas dentro de la condición de bandera de recepción activada, las cuales corresponden a comandos recibidos o a banderas controladas por otras banderas sujetas a determinados comandos. Estas últimas se encuentran organizadas en 3 tipos fundamentales, según su papel en el código: banderas de comandos simples, banderas intermedias y banderas de recepción de constantes.

Dentro de estos tres grupos, adquieren mayor significancia las banderas simples y las intermedias. Mientras que las banderas de recepción de constantes son controladas por las banderas intermedias, estas últimas y las simples son controladas desde la ISR de UART, y son activadas una vez que se ha decodificado algún comando válido con 2 bytes recibidos consecutivamente; formando así una cadena de activación de la forma que sigue:

1. Se detecta un comando válido recibido en dos bytes consecutivos
2. Se decodifica el comando y se activan las banderas correspondientes (según el comando, pueden ser simples o intermedias)
3. Se activa la bandera de recepción de byte (al salir de la ISR)
4. Con la condición de recepción de byte activa, se chequean TODAS LAS BANDERAS contenidas dentro de esta condición.
5. Se detecta la bandera que ha sido activada (simple o intermedia)
  - 5a. Si la bandera activada fue alguna bandera simple se ejecuta una acción y se regresa a encuestar cíclicamente las banderas restantes para luego caer en una encuesta cíclica de las banderas principales según la *Fig. 41*.
  - 5b. Si la bandera activada fue alguna bandera intermedia, se activa otra bandera para en la próxima recepción de byte ejecutar la acción correspondiente. Luego se regresa a

encuestar cíclicamente las banderas restantes para luego caer en una encuesta cíclica de las banderas principales según la Fig. 41.

Las condiciones que se verifican dentro de la condición de recepción de byte se muestran simbólicamente en el diagrama de la Fig. 44. En este diagrama, pueden observarse el conjunto completo de los posibles caminos que puede tomar el programa en función de

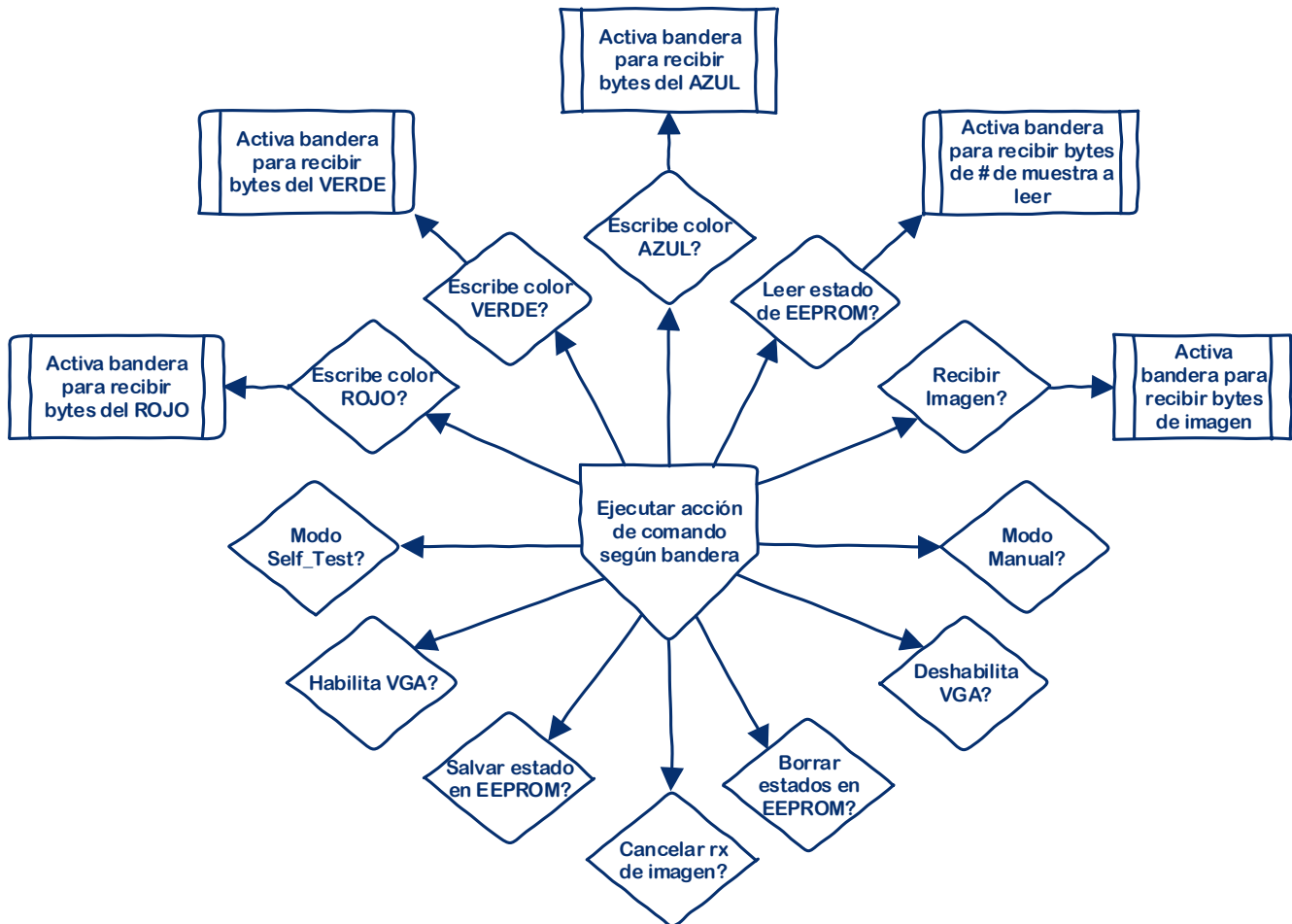


Fig. 44. Diagrama de flujo para la condición de byte recibido desde UART

La razón de ser de las banderas intermedias se debe a la funcionalidad que controlan y a la forma de implementar el procesamiento de los mismos comandos. Mientras que en el caso de las banderas simples una vez activadas solo resta detectar la condición correspondiente en el programa principal y ejecutar la respectiva acción, las banderas intermedias surgen de la necesidad de aislar la sintaxis de determinados comandos del resto de la información que necesitan para operar. Este es el caso de los comandos que conllevan 2 bytes adicionales además de los 2 bytes propios de la sintaxis, como **setColorRed**, **setColorBlue** y **setColorGreen**, entre otros; que necesitan 2 bytes correspondientes al valor de color a escribir en los registros de VGA.

Precisamente, las banderas intermedias permiten realizar tal diferenciación, dado que una vez recibido el código del comando, el programa verificará la bandera de recepción de byte -la cual se activa desde la ISR cada vez que se recibe cualquier byte-, pero como la recepción exitosa del comando no implica su inmediata ejecución, en tanto no se tenga la constante necesaria para escribir, como en el caso de los colores, el valor de color a escribir, si este tipo de bandera no estuviese presente y se activara directamente la escritura de color una vez recibido el comando, el programa interpretaría el último byte del comando, que en estas circunstancias sería el último byte recibido, como el primer byte del valor a escribir, lo cual es incorrecto.

A continuación, se describen los tres tipos de comandos empleados en el programa principal, y su relación con el diagrama de la *Fig. 44* y los tres tipos de banderas mencionados.

#### 1.4.3.1. Comandos simples

Como ya se venía mencionando, los comandos simples son aquellos que activan las banderas simples una vez recibidos, las que a su vez desencadenan determinada acción en concreto; sin activar ninguna otra bandera. Su sintaxis es la más simple de todas, puesto que solo vienen especificados por 2 bytes, y nada más. Naturalmente, todos ellos se ejecutan bajo la condición de bandera de recepción activada, pero la acción correspondiente solo se limita a escribir o leer determinados registros. Estos comandos son:

##### ✓ Comando **saveStatus**

Este comando, como su nombre lo indica, permite salvar el estado de todos los registros del módulo VGA en un buffer de escritura de 5 elementos que luego será escrito en la EEPROM serial empleando el módulo IIC. Esta escritura en EEPROM verifica primero si la memoria está llena, en cuyo caso no realiza salva alguna y marca la bandera de **update\_LED** para indicarlo, de lo contrario escribe el buffer mencionado en la EEPROM y actualiza las variables de dirección y conteo de salvas para una próxima ejecución. Cada vez que se realiza una salva de estado, el contador de salvas almacenadas se muestra en una lámpara de siete segmentos conectada igualmente al bus IIC mediante un expansor de puertos compatible. La implementación de este comando responde al diagrama de flujo de la *Fig. 45*.



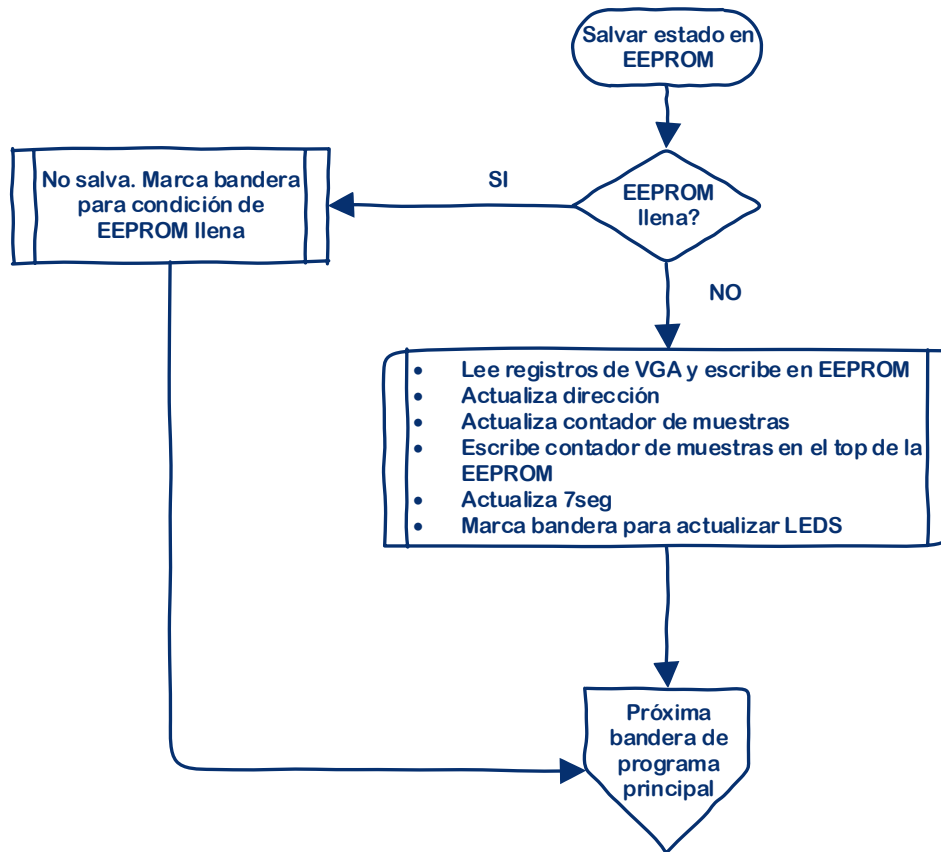


Fig. 45. Diagrama de flujo correspondiente a la operación de salvar estado en memoria EEPROM serial.

✓ Comando **clearStatus**

Este comando tiene un efecto opuesto a **saveStatus**, puesto que su función es la de limpiar completamente todos los datos de estados salvados por la aplicación en la EEPROM. La función que realiza es simplemente limpiar todas las variables asociadas a la escritura de la memoria vía IIC, y actualizar tanto LEDs como la lámpara 7 segmentos conectada igualmente mediante un expasor de puertos vía IIC. El diagrama de flujo de esta operación se muestra simbólicamente en la

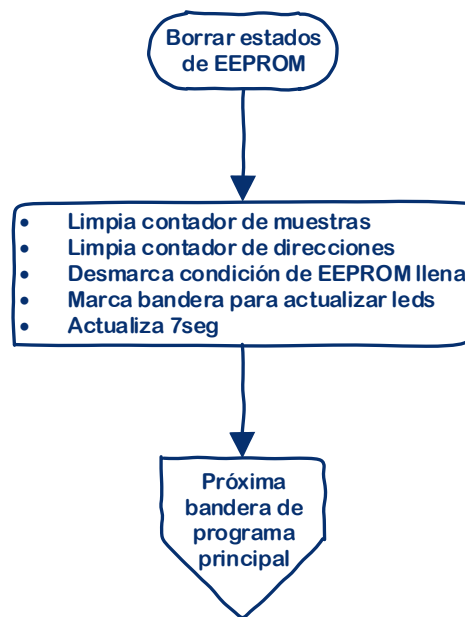


Fig. 46. Diagrama de flujo correspondiente a la operación de limpiar todos los estados salvados en memoria EEPROM serial.

✓ Comandos ***readySelfTest*** y ***readyManualColor***

La funcionalidad de estos comandos resulta muy básica, siendo un comando complemento del otro. En ambos casos, solo se realizan escrituras no destructivas directamente sobre el registro de control del módulo de VGA para modificar exclusivamente el bit de ***self\_test***. Mientras ***readySelfTest*** establece el modo de operación automático sobre el módulo VGA a través de este bit; ***readyManualColor*** establece el modo manual mediante este mismo bit.

✓ Comandos ***readyEnableVGA*** y ***readyDisableVGA***

Al igual que el caso anterior, ambos comandos se complementan el uno al otro; siguiendo exactamente el mismo comportamiento. La única diferencia radica en que el bit modificado ahora es el bit ***enable*** del mismo registro de control de VGA.

#### 1.4.3.2. Comandos compuestos

Se consideran comandos compuestos a aquellos comandos que además de los dos bytes de sintaxis, necesitan otros dos bytes adicionales para especificar determinada constante. Este tipo de comandos, una vez recibidos y decodificados, activan banderas intermedias las que a su vez activan las banderas de recepción de constantes. La operación de este tipo de comandos ya se había explicado parcialmente en el apartado ***Atención a comandos***, por lo que en esta sección solo se

explica la operación de cada comando en particular, atendiendo a sus similitudes y funciones. Los comandos agrupados en esta categoría son:

✓ Comandos de tipo **getColor XX**.

Existen tres comandos que tienen esta forma, de acuerdo a la forma en la que están escritos en el programa de aplicación; estos son los comandos **Escribe Rojo**, **Escribe Verde** y **Escribe Azul**, los cuales operan de acuerdo a la *Tabla 6* en el epígrafe **Comandos de operación**. Estos comandos tienen cada uno una bandera intermedia correspondiente denominada **readyColorRed**, **readyColorGreen** y **readyColorBlue** respectivamente. Cada una de estas banderas se activa desde la propia ISR del módulo de puerto serie UART, y a su vez, activan tres banderas denominadas **getColorRed**, **getColorGreen** y **getColorBlue** respectivamente, una en cada caso, que habilitan el procesamiento de los dos bytes recibidos posteriormente a la recepción de los dos bytes propios del comando en sí. Estos dos bytes son los que aparecen representados en como **XX** en la nomenclatura de esta categoría de comandos.

Cada uno de estos tres comandos, responde la misma lógica, mostrada en el diagrama de flujo de la *Fig. 47*. Esta lógica se corresponde con la *Tabla 6* mencionada anteriormente.

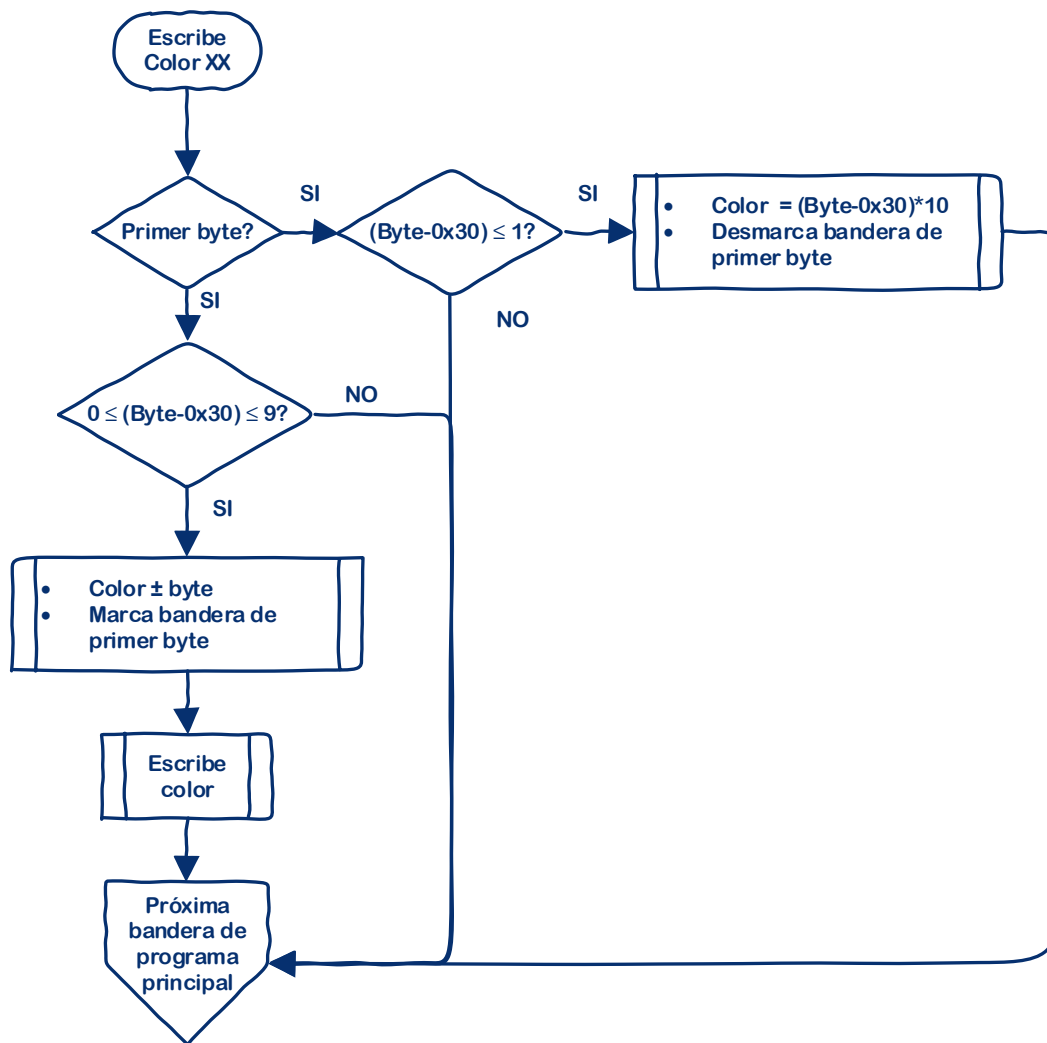


Fig. 47. Diagrama de flujo correspondiente a la operación escribir los colores rojo, verde y azul.

✓ Comando **getStatusNumber XX**.

Similar al caso anterior, este comando espera dos bytes **XX** para su operación. Su función es establecer el estado **XX** de operación en el sistema empotrado diseñado, leyéndolo directamente desde la memoria EEPROM serial vía IIC. Para su ejecución, la lógica verifica que el estado **XX** solicitado se encuentre en efecto, almacenado en la memoria, consultando para ello el contador de muestras, actualizado y almacenado en cada caso en una posición específica de la memoria, además de que el número **XX** especificado se encuentre en un rango válido, al igual que en el caso de los comandos **getColorXX**. En caso de solicitar una muestra que no se encuentra almacenada en memoria, se envía mediante UART un mensaje de error indicando el evento. La lógica de este comando responde al diagrama de flujo que muestra la Fig. 48.

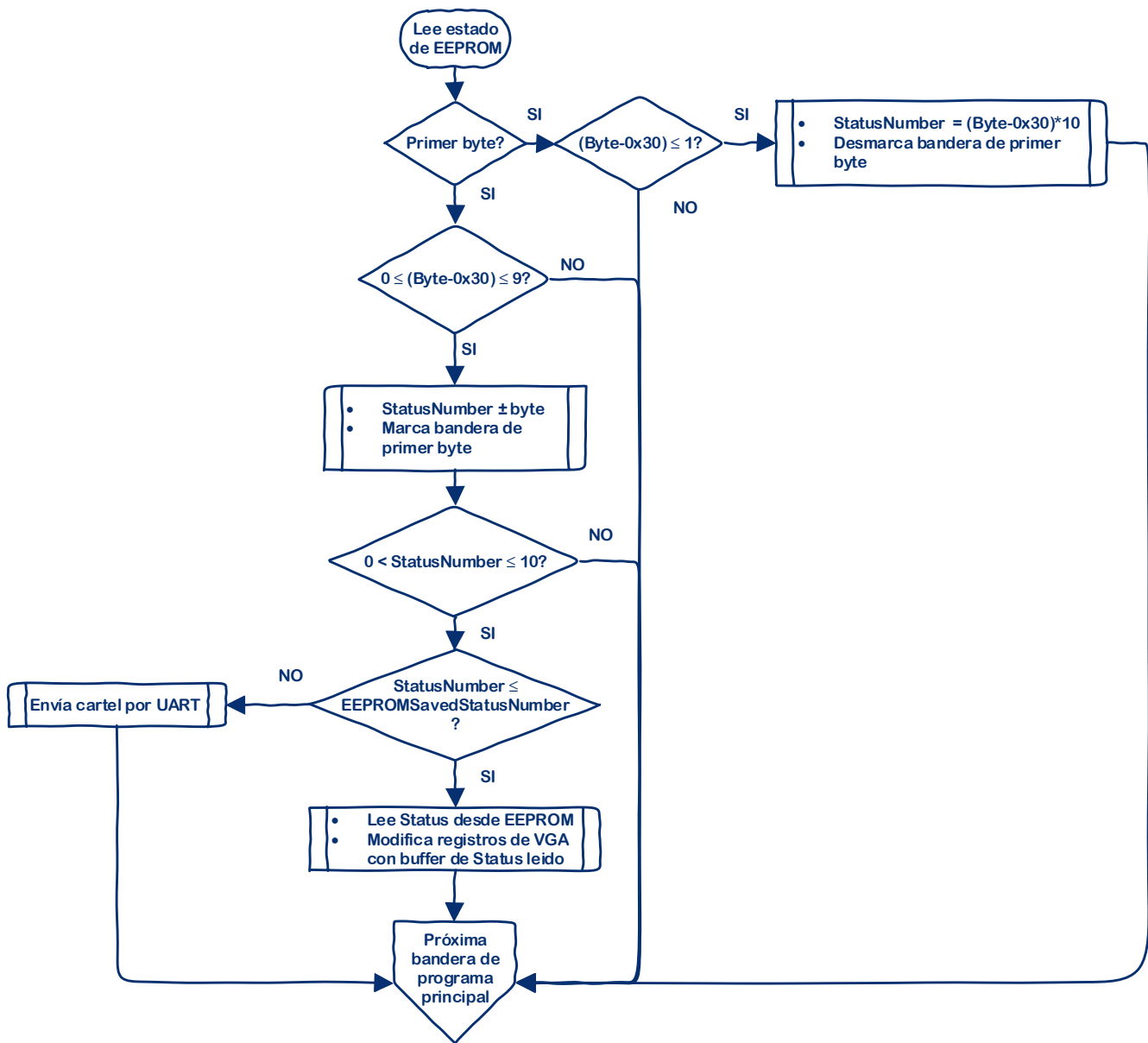


Fig. 48. Diagrama de flujo correspondiente a la operación de leer estado de memoria EEPROM

#### 1.4.3.3. Comandos de imagen

Adicionalmente a los comandos descritos, la aplicación brinda soporte para la recepción de una imagen vía UART. Para el envío de la imagen se ha desarrollado una aplicación de alto nivel para Windows denominada **ImageSender**, la cual emplea dos comandos fundamentales para indicar el comienzo del envío de la imagen y su fin; ya sea después de haber finalizado su transmisión o porque la misma ha sido cancelada manualmente por el usuario.

El envío de la imagen implica la transmisión desde PC de 614400 bytes de información en código binario puro, pertenecientes a los pixeles que componen la imagen. Este envío se realiza bajo condiciones específicas, fijando la resolución de la imagen a 640x480 en formato RGB 444, lo cual representa un total de 307200 pixeles cada uno con 12 bits de información de color. Cada pixel es

enviado desde la aplicación en la PC de una forma igualmente predeterminada, siendo cada pixel 2 bytes donde el primer byte menos significativo contiene en su nibble mas alto los bits del color verde, y en su nibble más bajo los bits del color azul; mientras que el nibble menos significativo del segundo byte a enviar por cada pixel serán los bits del color rojo y el nibble más significativo estará en cero. Este formato se garantiza mediante el empleo de la aplicación, lo cual permite recibir y guardar la imagen en el sistema de procesamiento de forma organizada; lo que determina el algoritmo empleado para la manipulación de toda esta trama.

Los dos comandos asociados al envío y recepción de la imagen son enviados automáticamente por la aplicación en cada proceso de envío de imagen. El comando de inicio de recepción de imagen puede verse como un comando compuesto, ya que al igual que los demás comandos de este tipo, posee igualmente una bandera intermedia que se activará en el momento que se recibe el comando, para luego comenzar a almacenar los bytes subsecuentes como información de imagen de la manera prevista. Este comando desencadena una secuencia que termina ejecutando una acción cuya lógica responde al diagrama de flujo de la *Fig. 49*.

En este caso, el código simplemente realiza la operación inversa a la realizada por la aplicación para enviar la imagen con el formato explicado anteriormente. Dado que la memoria externa SDRAM DDR2 ha sido configurada la interfaz PIM para bus plb; y su ancho en este caso es de 32 bits, es necesario conformar una palabra de 32 bits completa para ir almacenando de forma organizada la información de pixel que va llegando vía UART. Para ello, se cuenta con un buffer de imagen de 16 bits en el cual se irá almacenando mediante operaciones de corrimiento de bits y máscaras la información de color de un pixel de la forma **ORGB**, para 4 bits de color con 3 colores. Una vez obtenido el pixel, se copia en los 16 bits menos significativos de la palabra de 32 bits a salvar en la memoria externa, quedando libre los otros 16 bits más significativos para almacenar el próximo pixel, el cual se obtiene de la misma forma que el anterior. Todo el proceso se rige mediante un contador denominado **bytesCount** el cual indicará como procesar cada byte que va llegando; hasta tener los 4 bytes completos que conforman la palabra que contiene la información de dos pixeles para enviar al controlador de memoria para almacenar. Una vez conformada esta palabra, se escribe en la memoria, se limpian todas las variables empleadas y se actualizan las direcciones de memoria para escribir la próxima palabra; y así sucesivamente hasta recibir todos los bytes de la imagen.

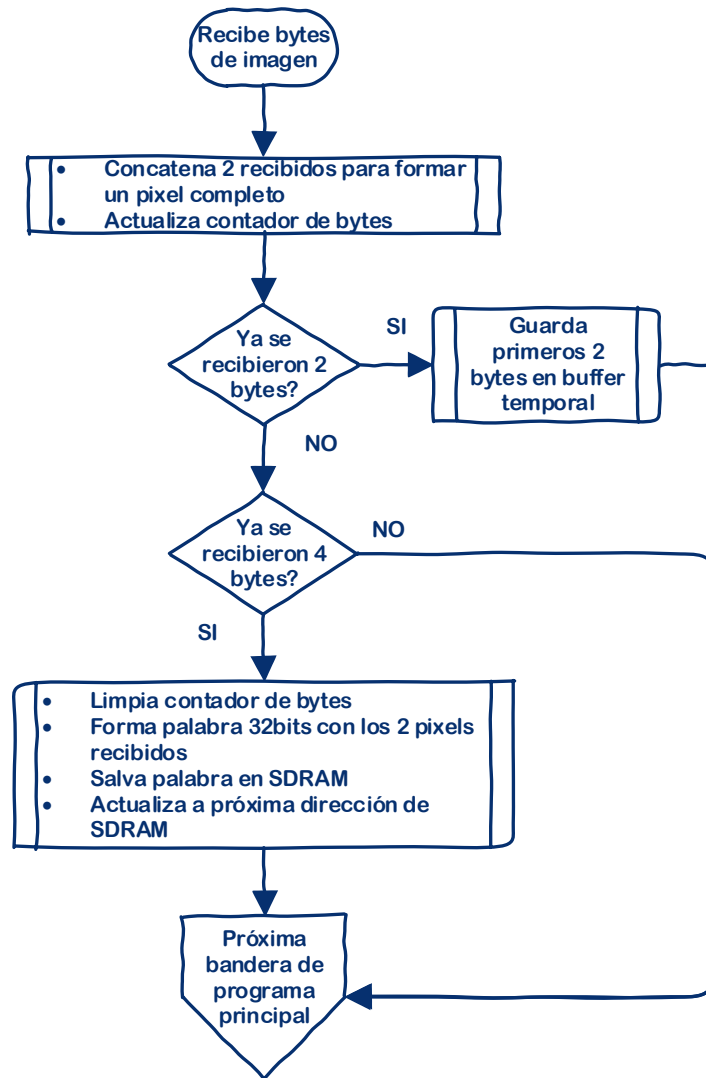


Fig. 49. Diagrama de flujo correspondiente a la operación de recibir una imagen vía UART.

Sin embargo, el flujo de programa de la figura anterior, puede verse interrumpido a petición del usuario, o simplemente porque ha sido enviada completamente la imagen. En ese caso, el programa pregunta por la cantidad de palabras de 32 bits que han sido almacenadas en memoria externa, comparando esta misma con la cantidad de palabras esperadas, o simplemente si se ha recibido el comando de ***image\_cancel*** que indica que se ha recibido un comando de fin de transmisión de imagen producto a la acción de cancelar emitida por la aplicación a petición del usuario. Este comando puede ser catalogado como un comando simple, dado que no activa ninguna otra bandera intermedia, salvo la de actualizar LCD, y puede indicar dos cosas, o que se ha interrumpido la transmisión (en este caso se recibiría el comando de ***image\_cancel*** y a continuación el número de palabras almacenadas sería menor que el esperado), o que la imagen ha sido recibida en su totalidad (se recibe el mismo comando, solo que ahora las palabras almacenadas si se corresponden con la cantidad de palabras esperadas) que en cualquier caso tienen la misma respuesta: restablecer todas

las variables relacionadas con la recepción de la imagen, incluyendo claro está, aquella que contiene la dirección de SDRAM a escribir, así como la indicación del evento, una vez determinada la fuente, a través del display LCD. Esta lógica responde al diagrama de la *Fig. 50* y puede consultarse en su totalidad en el código suministrado para la aplicación en conjunto con los anexos de este trabajo.

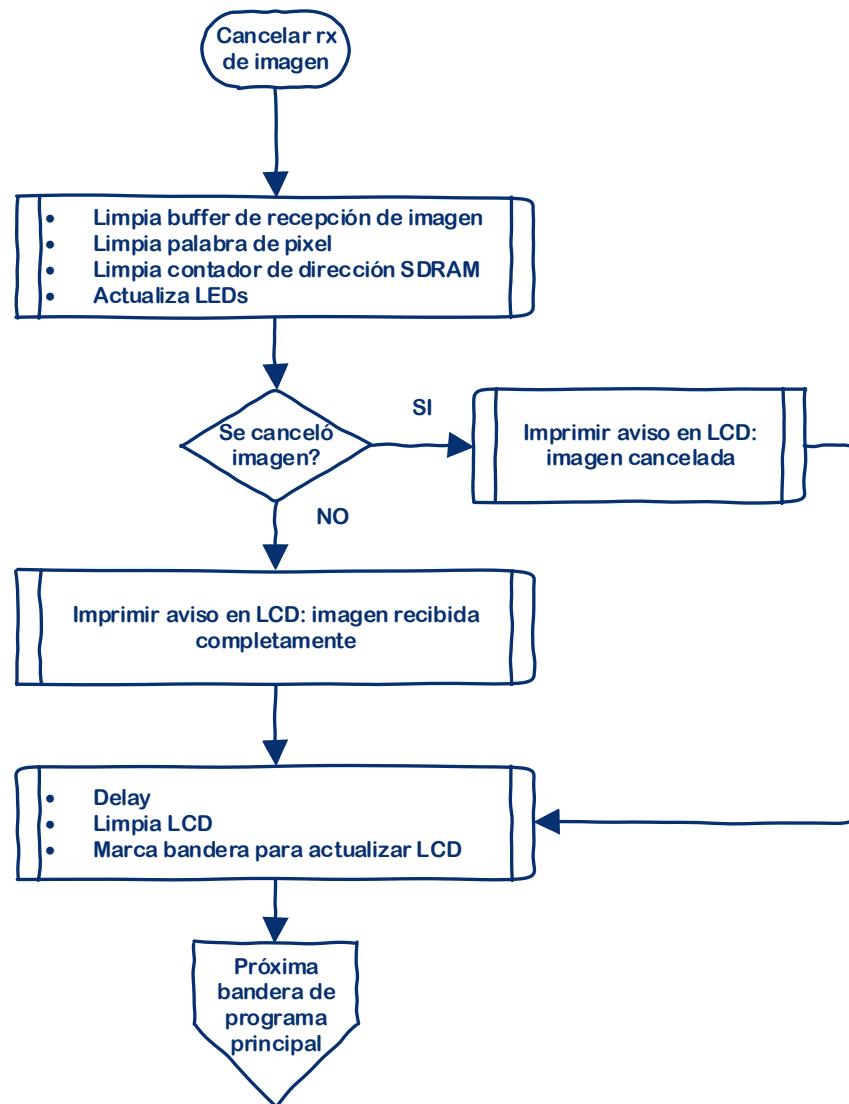


Fig. 50. Diagrama de flujo correspondiente al evento de fin de recepción de imagen vía UART

## 1.5. Rutinas de Interrupción

### 1.5.1. ISR switches

La interrupción de los switches no es más que la interrupción del GPIO que los controla. Este GPIO solicita interrupción ante un cambio de estado en cualquiera de sus bits. Dado que cada switch conectado externamente a este GPIO tiene una función específica, la rutina de interrupción correspondiente, en esencia, tendrá que detectar el bit que ha cambiado -causa de la interrupción-, y



luego, ejecutar una acción en correspondencia. De esta forma, aunque para cualquier switch la fuente de interrupción es la misma, en función de cada uno, se realiza una acción diferente. La *Tabla 7* indica la funcionalidad de cada uno de ellos.

Para lograr esto, se hace necesario tener en cada solicitud de interrupción, el estado de cada switch antes del cambio, para con esto poder comparar contra el estado actual de cada uno y de esta forma determinar que bit fue el que cambió. Tanto el estado previo como el actual de los switches se leen y almacenan en un byte cada uno, denominados ***switchPrevious*** y ***swstatus*** respectivamente, uno al principio del programa principal y al final de la ISR, y el otro a la entrada de la ISR. La comparación entonces se realiza efectuando una operación lógica XOR entre ambos bytes, buscando que el bit que ha sido alterado se ponga a 1, ya que en el byte del estado previo tendrá un valor, y en el actual tendrá otro, mientras que los que no, que en ambos casos permanecerán iguales, se limpien. Posteriormente, se comienza a rotar los bits del resultado de la operación lógica realizada, buscando el 1 lógico resultante, incrementando un contador en cada rotación. Este contador servirá después para indicar el número del switch que ha cambiado, y con ello, realizar la selección de la franja correspondiente escribiendo el registro de selección del módulo de VGA.

Al finalizar la ISR, se limpia el contador empleado para la decodificación del switch y se marca la bandera de actualización de los LEDs. Luego actualiza el estado previo con el actual y se desmarca la solicitud de interrupción en el propio GPIO. La *Fig. 51* muestra el diagrama de flujo para la lógica de atención a la ISR concebida.

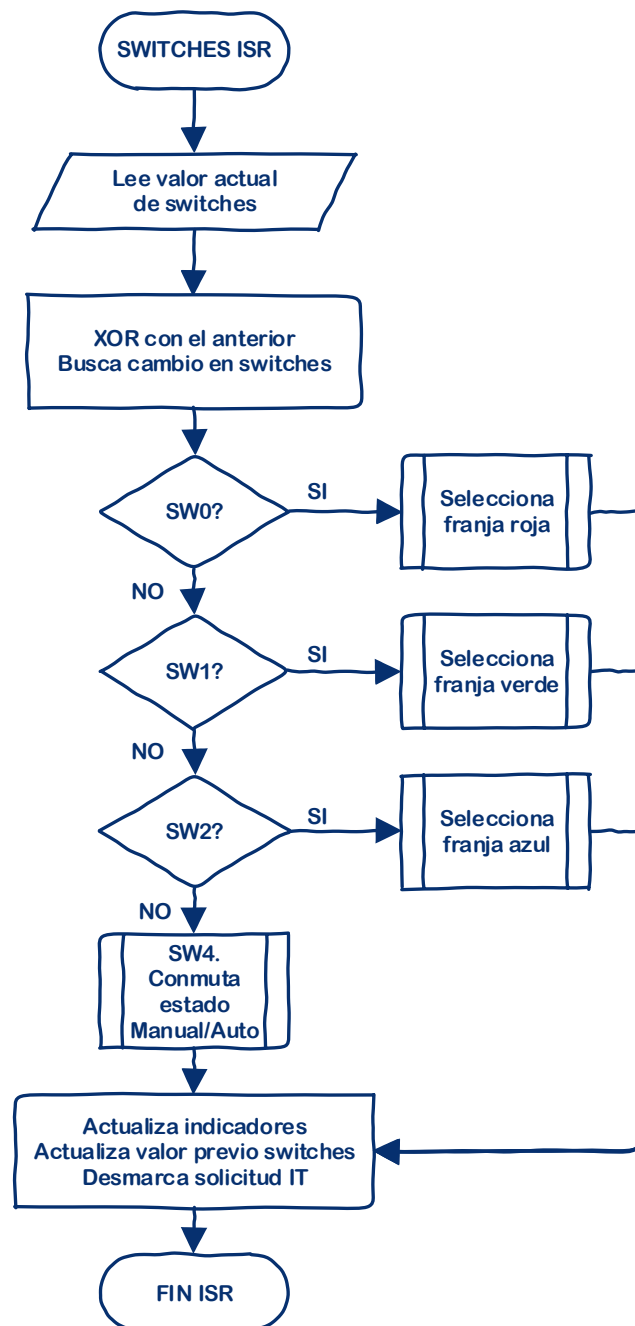


Fig. 51. Diagrama de flujo correspondiente a la ISR de los switches

Es necesario aclarar que, para la atención de esta interrupción en particular, se asume que solo se activa un switch a la vez, dado que no es humanamente posible activar dos o más de ellos al mismo tiempo, estrictamente hablando. En la decodificación de los switches, siempre habrá uno activado, dado que por eso es que se invoca a la interrupción.

En el diagrama de flujo, la última condición no debe interpretarse como “Si es **sw2** se selecciona la franja azul y si no, entonces se conmuta de estado...”, lo cual, aunque en esencia es lo que se hace en el código, resulta una interpretación incorrecta, ya que esto no constituye el caso por defecto de

una estructura **switch-case**, sino el cuarto caso, que por lógica se ejecutará si no se corresponde con ninguno de los casos anteriores. Esto marca una diferencia en el código, dado que, si por error la rotación fue más allá del cuarto bit del resultado de la operación XOR, no se ejecutarán ninguna de las cuatro acciones previstas.

### 1.5.2. ISR Timer

Esta rutina de interrupción se corresponde con el evento de desbordamiento del temporizador presente en el diseño. Este temporizador ha sido configurado pertinentemente para solicitar una interrupción periódicamente cada 1 segundo, en cuya ISR únicamente solo se marca la bandera para actualizar el LCD. El diagrama de flujo resulta extremadamente simple, resultando como muestra la Fig. 52.

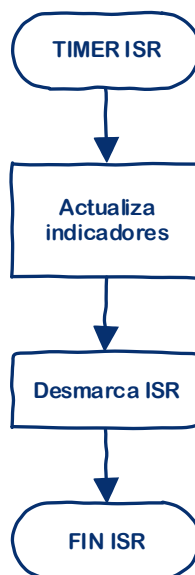


Fig. 52. Diagrama de flujo correspondiente a la ISR del temporizador

### 1.5.3. ISR de UART

Sin lugar a dudas, esta ISR constituye la más importante de las tres rutinas de ISR que componen el sistema. Es aquí donde precisamente adquiere forma la más destacada de todas las funcionalidades: la interacción con el sistema vía UART.

Aunque es invocada tanto en el evento de fin de transmisión como de recepción por UART, debido al diseño del propio hardware al que da soporte, su funcionalidad se basa solo en atender la recepción de datos vía serie. Su implementación guarda estrecha relación con la concepción realizada para la comunicación del sistema con el exterior vía UART, en particular con la estructura de los comandos habilitados. Teniendo en cuenta que estos comandos se componen solo de 2 bytes, para su recepción y posterior decodificación, se hace necesario el empleo de un buffer de 2 bytes capaz de

contener la sintaxis completa de cualquiera de ellos, además de la lógica necesaria para su llenado con cada operación de recepción.

En primer lugar, para lograr discriminar entre una recepción y una transmisión, la rutina procede directamente a preguntar por un bit específico en el registro de estado del módulo que indica si la FIFO de recepción interna del propio módulo está vacía o no. En función de esto, se procede a leer el primer byte de la FIFO y con ello desmarcar la solicitud de interrupción, para luego escribirlo en el byte más alto del buffer de recepción, denominado ***rx\_buffer***. A continuación, se procede a la decodificación pertinente, en la cual se pregunta estrictamente por cada comando válido, en función de lo cual se activa(n) la(s) bandera(s) correspondiente(s). La recepción de cualquier dato vía UART directamente desde teclado de la PC, tendrá lugar en el formato ASCII, razón por la cual la rutina de interrupción deberá preguntar por cualquier comando válido partiendo de su sintaxis en ASCII, lo cual constituye el primer filtro para discriminar entre algún comando válido recibido y un dato binario cualquiera. En cualquier caso, sea cual fuere el byte recibido, esta primera decodificación del buffer de recepción no resultará en ningún comando válido, dado que solo se habrá recibido el primer byte del mismo. Ya en un segundo evento de recepción, el byte recibido será escrito en el byte más bajo del buffer de recepción, conservando el byte más alto escrito previamente, con lo cual ya se tendrán los 16 bits necesarios para la decodificación.

A partir de aquí, solo resta ejecutar las acciones correspondientes para el comando recibido, en caso de haber sido algún comando válido. Una vez hecho esto, la rutina termina marcando la bandera de recepción para el programa principal, con lo cual se entra a buscar la bandera que ha sido activada desde esta ISR y se ejecuta así la acción pertinente.

La *Fig. 53* muestra el diagrama de flujo correspondiente a la lógica implementada para la funcionalidad anterior. En este diagrama puede verse un caso particular en la recepción, referente a la recepción de una imagen. Cuando esto ocurre, es porque previamente se ha recibido el comando que indica el inicio de la transmisión de una imagen desde la PC<sup>5</sup> y lo que se está recibiendo es la información binaria de la misma enviada byte a byte de forma estándar desde la PC con una cantidad de bytes y formatos fijos y conocidos. En este caso, la rutina ignora cada byte recibido entre tanto no se reciba una secuencia de bytes que indique algún comando válido -lo cual solo ocurrirá si se cancela o finaliza el envío de la imagen-, pero siempre marca la bandera de recepción. Esto provoca que en el programa principal, como la bandera de recepción de imagen estará marcada, el segmento de código correspondiente procese el byte leído de la FIFO de recepción del módulo de UART como parte de una imagen cuya recepción está en curso. Por esta razón, en el diagrama de la *Fig. 53* se pregunte simbólicamente luego de decodificar el comando en ASCII por el inicio de recepción de

<sup>5</sup> Tanto el envío de un comando de inicio de transmisión de imagen, como otro de fin de transmisión/imagen cancelada se garantizan a partir del empleo de una aplicación de alto nivel desarrollada al efecto para tal propósito, denominada ***Image\_Sender***, suministrada junto con los materiales de este proyecto.

imagen, aunque en la práctica no se encuentre implementado estrictamente de esta forma en el código de la ISR.

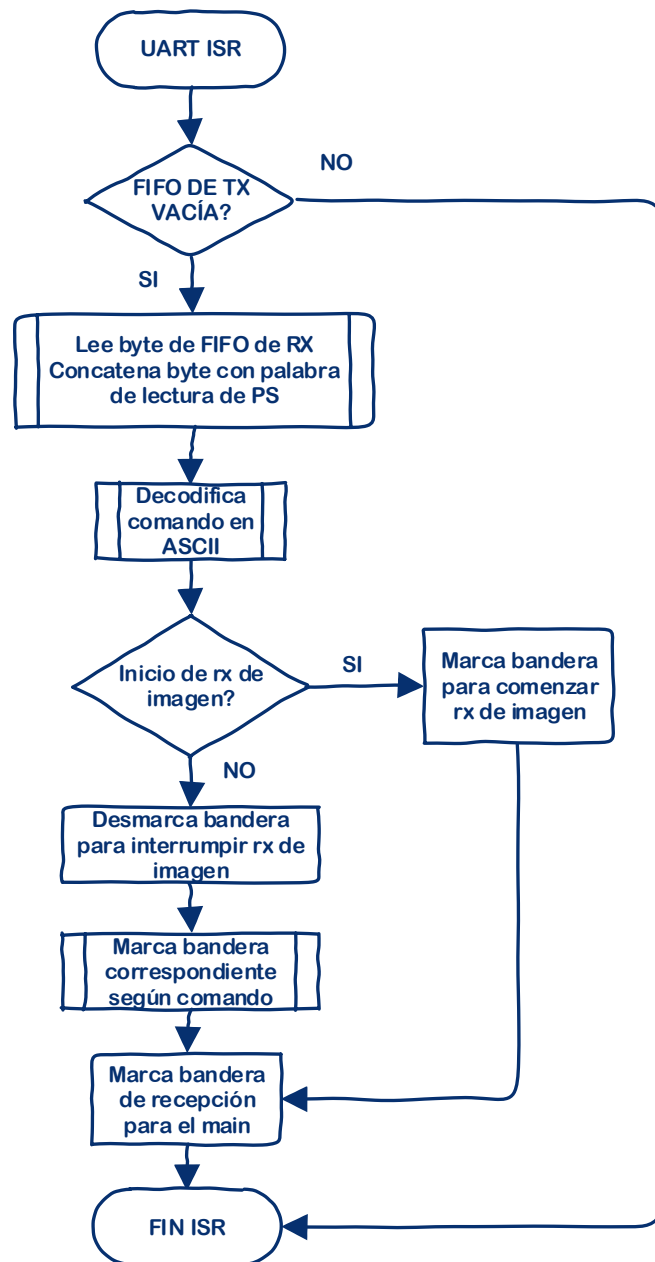


Fig. 53. Diagrama de flujo correspondiente a la ISR del módulo de puerto serie UART

Algo similar ocurre con aquellos comandos que implican la recepción de dos bytes adicionales correspondientes a algún valor específico, como el caso de los comandos para escribir los colores, o para leer algún estado de EEPROM. En este caso, una vez recibidos los dos bytes correspondientes al comando en sí, los siguientes 2 bytes recibidos se interpretarán como los dos bytes correspondientes a la constante, independientemente de que estos constituyan un comando válido o no. Precisamente, una vez activada la bandera intermedia correspondiente (acción provocada por el comando recibido previamente), el próximo byte que se reciba irá a parar al segmento de código

correspondiente a la validación de la constante recibida, donde finalmente, sea lo que sea, será interpretado de una forma u otra como valor de constante. Este comportamiento resulta totalmente lógico ya que el byte recibido primero pasará por el filtro de la decodificación de esta ISR como el byte más alto del buffer de recepción y no será ningún comando válido -dado que será un solo byte que además fue recibido luego de recibir dos bytes de comando válido-, pero como la ISR activa la bandera de recepción cada vez que recibe un byte cualquiera, cuando se salga de esta última, este byte irá a parar al segmento de código correspondiente a la validación de la constante recibida, donde finalmente, sea lo que sea, será interpretado de una forma u otra como valor de constante.

.

## CONCLUSIONES

Después de realizar las evaluaciones y utilizar los métodos ya descritos con anterioridad, se puede concluir que es posible materializar el diseño, más que de un controlador de VGA, de cualquier otro módulo hardware de diseño propio con propósito específico e incorporarlo al flujo de diseño de EDK, en conjunto con el resto de los componentes que proporciona Xilinx para integrarlo a un sistema de procesamiento empotrado basado en hardware reconfigurable, lo que posibilita su reusabilidad en diseños posteriores, así como su portabilidad de un entorno de trabajo a otro.

Justamente, la versatilidad de las herramientas que brinda Xilinx dentro del paquete de EDK encuentra aquí su verdadero propósito, permitiendo implementaciones híbridas hardware/software en consonancia con las necesidades del usuario, así como su posterior depuración mediante diversos métodos, para ambas plataformas. El desarrollo de este trabajo es un ejemplo de ello, dado que el diseño se realiza partiendo de la disponibilidad de los diversos componentes hardware a partir de su descripción software, mediante *softcores*, *firmcores* o *hardcores*, y pone de manifiesto su fácil incorporación a un diseño dado, lo que hace pensar que cualquier componente o sistema que se necesite incorporar al diseño requerido, se encuentra simplemente al alcance de la mano -o de un pago-, con relativamente bajos costos, en poco tiempo y con poco esfuerzo.

A criterio de los autores, la interacción con todas las herramientas que componen este entorno aporta al diseñador una capacidad de abstracción y conocimientos increíblemente vasta, dado la enorme diferencia que existe entre esta filosofía de diseño y los métodos convencionales para el desarrollo de sistemas empotrados basados en otras plataformas fuera del hardware reconfigurable. A este nivel, dado que todas las interconexiones e implementaciones del sistema específico yacen dentro de un solo chip, se hace necesario el empleo exhaustivo de herramientas de alto nivel desarrolladas al efecto, y una comprensión cabal de todo el proceso que la misma realiza a más bajo nivel en ambas plataformas, para con ello lograr los parámetros de diseño y la funcionalidad deseadas, tomar decisiones y detectar errores que permitan optimizar, más de lo que la herramienta ya lo hace, el diseño final dentro del FPGA. Todo esto, lógicamente, se traduce en un ahorro considerable de recursos, mejoras en el desempeño y robustez en el producto final. Precisamente, mientras algunas consideraciones e implementaciones quedan de la mano del asistente de trabajo, otras más sutiles y específicas, pero no menos importantes, quedan sujetas a una inevitable supervisión y manejo por parte del diseñador, lo que aporta una capa adicional de complejidad al trabajo, pero que al mismo tiempo lo embellece y magnifica, de modo que encuentra siempre, su utilidad práctica.



## REFERENCIAS BIBLIOGRAFICAS

1. (US) DC. VGA Controller (VHDL) 2018. Atlassian Confluence 6.6.13:[Available from: <https://www.digikey.com/eewiki/pages/viewpage.action?pagelId=15925278#space-menu-link-content>.
2. Xilinx. Spartan-3A FPGA Starter Kit Board User Guide. UG330 (v13): Xilinx; 2007. p. 60-3.
3. Chu PP. FPGA Prototyping by VHDL Examples: Xilinx Spartan™-3 Version. Sons CJW, editor: Wiley-Interscience; 2007.
4. VESA. VESA and Industry Standards and Guidelines for Computer Display Monitor Timing (DMT). Monitor Timing Standard: VESA; 2007. p. 17-20.
5. Tran V-H, Tran X-T. An Efficient Architecture Design for VGA Monitor Controller. International Conference on Consumer Electronics, Communications and Networks (CECNet) 2011.
6. Kadlec J. UTIA EdkDSP Demonstrator in Xilinx 3S700AN FPGA with Embedded FLASH and NV RAM. Rev.3. In: v.v.i. UAC, editor.: UTIA AV CR v.v.i.; 2015.
7. Mishra A, Kumar A, Parihar R. VGA Application in Text Display Using FPGA. Advances in Intelligent Systems and Computing: Springer, Singapore; 2018.
8. Chapman K. Rotary Encoder Interface for Spartan-3E Starter Kit. In: Xilinx, editor.: Xilinx; 2006.
9. Hamid M. Writing Efficient Testbenches. In: Xilinx, editor. XAPP199. v1.1 ed: Xilinx; 2010.
10. Maaref M. Creating an OPB IPIF-based IP and Using it in EDK. In: Xilinx, editor. XAPP967 v1.1 ed: Xilinx; 2007.
11. Xilinx. MicroBlaze™ Software Reference Guide. Xilinx; 2002.
12. Xilinx. XST User Guide. UG627. v11.3 ed: Xilinx; 2009.
13. Xilinx. EDK Concepts, Tools, and Techniques. A Hands-On Guide to Effective Embedded System Design. UG683. v14.6 ed: Xilinx; 2012.
14. Xilinx. ISim In-Depth Tutorial. UG682 v14.3 ed: Xilinx; 2012.
15. Xilinx. ISim User Guide. UG660. v14.3 ed: Xilinx; 2012.
16. Xilinx. Synthesis and Simulation Design Guide. UG626. v14.4 ed: Xilinx; 2012.
17. Xilinx. Embedded System Tools Reference Manual. UG1043. v2014.1 ed: Xilinx; 2014.

## ANEXOS

Anexo 1. Especificaciones de tiempo para diversos estándares VGA. Tomado de [1]

Resolución (pixels)	Frecuencia Refrescamiento	Pixel Clock (MHz)	Horizontal (pixel clock)				Vertical (filas)			
			D <sup>1</sup>	FP <sup>2</sup>	SP <sup>3</sup>	BP <sup>4</sup>	D <sup>1</sup>	FP <sup>2</sup>	SP <sup>3</sup>	BP <sup>4</sup>
640x400	85	31.5	640	32	64	96	400	1	3	41
640x480	60	25.175	640	16	96	48	480	10	2	33
640x480	75	31.5	640	16	64	120	480	1	3	16
720x400	85	35.5	720	36	72	108	400	1	3	42
768x576	60	34.96	768	24	80	104	576	1	3	17
800x600	60	40	800	40	128	88	600	1	4	23
800x600	75	49.5	800	16	80	160	600	1	3	21
1024x768	60	65	1024	24	136	160	768	3	6	29
1024x768	70	75	1024	24	136	144	768	3	6	29
1024x768	85	94.5	1024	48	96	208	768	1	3	36
1280x1024	60	108	1280	48	112	248	1024	1	3	38
1280x1024	75	135	1280	16	144	248	1024	1	3	38
1400x1050	100	214.39	1400	112	152	264	1050	1	3	58
1600x1200	70	189	1600	64	192	304	1200	1	3	46
1600x1200	60	162	1600	64	192	304	1200	1	3	46
1920x1440	60	234	1920	128	208	344	1440	1	3	56
1920x1440	75	297	1920	144	224	352	1440	1	3	56

<sup>1</sup>: Región de visualización (**display range**).

<sup>2</sup>: Región de retrazo (**sync pulse/retrace**).

<sup>3</sup>: Región de visualización (**display range**).

<sup>4</sup>: Borde izquierdo (**back porch**).

Anexo 2. Código VHDL correspondiente a la implementación del bloque **VGA\_sync\_unit**.

```

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

entity VGA_sync_unit is
    port(
        clk          : in std_logic;
        reset        : in std_logic;
        enable       : in std_logic;
        hsync        : out std_logic;
        vsync        : out std_logic;
        video_on     : out std_logic;
        p_tick       : out std_logic;
        pixel_x      : out std_logic_vector (9 downto 0);
        pixel_y      : out std_logic_vector (9 downto 0));
end VGA_sync_unit;
-----
--Descripción de pines
--hsync    =>> Señal de sincronización para el barrido horizontal. Especifica
--              el tiempo necesario para un barrido de línea completo, con
--              bordes y todo.
--vsync    =>> Señal de sincronización para el barrido vertical.
--              Especifica el tiempo necesario para un barrido de la pantalla
--              completa.
--video_on =>> Señal para inhabilitar display en los bordes y en el retrase.
--p_tick   =>> Clock para incrementar los pixeles a 25MHz, correspondiente a
--              una resolución de 640 x 480.
--              p*1*s = 800*525*60 = 25M pixels/sec aprox.
--pixel_x  =>> Vector para coordenada horizontal de pixel.
--pixel_y  =>> Vector para coordenada vertical de pixel.
-----

```

```

architecture Behavioral of VGA_sync_unit is
--definiendo como constantes el tamaño de cada región de la pantalla
--para el barrido horizontal las unidades son pixels
--para el barrido vertical las unidades están en líneas
--OJO PARA UNA PANTALLA DE 640X480

--constantes para el barrido horizontal
constant HD: integer:= 640;      --espacio horizontal visualizable
constant HF: integer:= 16;      --espacio del borde derecho
constant HB: integer:= 48;      --espacio del borde izquierdo
constant HR: integer:= 96;      --retrace. durante este espacio hsync va a cero
                                --para volver a empezar

--constantes para el barrido vertical
constant VD: integer:= 480;      --espacio vertical visualizable (cantidad de
                                --líneas).
constant VF: integer:= 10;      --espacio del borde inferior
constant VB: integer:= 33;      --espacio del borde superior
constant VR: integer:= 2;       --retrace vertical. Demora del barrido entre el
                                --momento en que llega al final de la pantalla
                                --y vuelve a empezar

--señales para generar el pixel_tick
signal mod2_reg    : std_logic;
signal mod2_next   : std_logic;

--señales para los contadores del barrido vertical
signal v_count_reg    : unsigned(9 downto 0);
signal v_count_next   : unsigned(9 downto 0);

--señales para los contadores del barrido horizontal
signal h_count_reg    : unsigned(9 downto 0);
signal h_count_next   : unsigned(9 downto 0);

--buffer de salida
signal v_sync_reg, h_sync_reg    : std_logic;
signal v_sync_next, h_sync_next  : std_logic;

--señales de estado
signal h_end: std_logic;          --señal para indicar el fin del barrido horizontal
                                --para una línea.
signal v_end    : std_logic;      --idem al h_end pero para el barrido vertical
signal pixel_tick : std_logic;    --señal de reloj para los 25M pixel/sec (salida del
divisor de frecuencia)

```

**begin**

```
--registros globales del sistema
```

```
process (clk, reset)
```

```
begin
```

```
    if reset = '1' then
```

```
        mod2_reg    <= '0';
```

```
        v_count_reg <= (others => '0');
```

```
        h_count_reg <= (others => '0');
```

```
        v_sync_reg  <= '0';
```

```
        h_sync_reg  <= '0';
```

```
    elsif (clk' event and clk = '1') then
```

```
        if enable = '1' then
```

```
            mod2_reg    <= mod2_next;
```

```
            v_count_reg <= v_count_next;
```

```
            h_count_reg <= h_count_next;
```

```
            v_sync_reg  <= v_sync_next;
```

```
            h_sync_reg  <= h_sync_next;
```

```
        else
```

```
            mod2_reg    <= '0';
```

```
            v_count_reg <= (others => '0');
```

```
            h_count_reg <= (others => '0');
```

```
            v_sync_reg  <= '0';
```

```
            h_sync_reg  <= '0';
```

```
        end if;
```

```
    end if;
```

```
end process;
```

```
--señal para generar el pixel_tick. Dado que el clk es de 50MHz, esto básicamente  
es un divisor de reloj por 2, obteniendo 25MHz
```

```
mod2_next <= not mod2_reg;
```

```
--25MHz pixel tick
```

```
pixel_tick <= '1' when mod2_reg = '1' else '0';
```

```
--status
```

```
h_end <= '1' when h_count_reg = (HD+HF+HB+HR-1) else '0';
```

```
v_end <= '1' when v_count_reg = (VD+VF+VB+VR-1) else '0';
```

--ahora process para el contador para los 800 pix. Barrido horizontal

```
process (h_count_reg, h_end, pixel_tick)
begin
    if pixel_tick = '1' then
        if h_end = '1' then
            h_count_next <= (others => '0');
        else
            h_count_next <= h_count_reg + 1;
        end if;
    else
        h_count_next <= h_count_reg;
    end if;
end process;
```

--ahora process para el contador para los 525 pix. Barrido vertical

```
process (v_count_reg, h_end, v_end, pixel_tick)
begin
    if pixel_tick = '1' and h_end = '1' then
        if (v_end = '1') then
            v_count_next <= (others => '0');
        else
            v_count_next <= v_count_reg + 1;
        end if;
    else
        v_count_next <= v_count_reg;
    end if;
end process;
```

--metiendole buffers para evitar glitches

```
h_sync_next <= '1' when (h_count_reg >= (HD+HF)) and (h_count_reg <= (HD+HF+HR-1))
               else '0';
v_sync_next <= '1' when (v_count_reg >= (VD+VF)) and (v_count_reg <= (VD+VF+VR-1))
               else '0';
```

--lógica para señal de video on/off

```
video_on <= '1' when (h_count_reg < HD) and (v_count_reg < VD) else '0';
```

--output signals

```
hsync      <= h_sync_reg;
vsync      <= v_sync_reg;
pixel_x    <= std_logic_vector(h_count_reg);
pixel_y    <= std_logic_vector(v_count_reg);
p_tick     <= pixel_tick;
```

```
end Behavioral;
```

Anexo 3. Código VHDL correspondiente a la implementación del bloque **VGA\_self\_test**.

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

```
entity VGA_module_self_test is
    port(
```

```
        clk                : in  std_logic;
        reset               : in  std_logic;
        enable              : in  std_logic;
        self_test           : in  std_logic;
        video_on            : in  std_logic;
        pixel_x             : in  std_logic_vector (9 downto 0);
        pixel_y             : in  std_logic_vector (9 downto 0);
        r_video_test        : out  std_logic_vector (3 downto 0);
        g_video_test        : out  std_logic_vector (3 downto 0);
        b_video_test        : out  std_logic_vector (3 downto 0)
```

```
    );
```

```
end VGA_module_self_test;
```

```
-----
--Descripción de pines
```

```
--  self_test  =>> Aunque esté habilitado, el bloque no funcionará hasta que
--                  no se active esta señal, para cuando se seleccione el otro
--                  bloque, este no se quede funcionando por gusto, aunque no
--                  salga.
--  video_on    =>> Entrada desde VGA_sync_unit. Indica zona inactiva de video.
--  pixel_x     =>> Entrada desde VGA_sync_unit. Coordenada horizontal de pixel.
--  pixel_y     =>> Entrada desde VGA_sync_unit. Coordenada vertical de pixel.
--  r_video_test=>> Vector de salida para color rojo.
--  g_video_test=>> Vector de salida para color verde.
--  b_video_test=>> Vector de salida para color azul.
```

```
-----
architecture Behavioral of VGA_module_self_test is
```

```
--registros para almacenar el color del pixel que está saliendo.
```

```
signal r_video_reg      : std_logic_vector (3 downto 0);
signal g_video_reg      : std_logic_vector (3 downto 0);
signal b_video_reg      : std_logic_vector (3 downto 0);
```

```
--señales internas para la conexión de pattern_timer (cables)
```

```
signal en_patter_timer  : std_logic;      --enable del contador
signal sec_clock        : std_logic;      --salida de 1s del contador
```

```
--declarando valores para coordenadas de pixel
```

```
signal pixel_x_int      : integer range 1586 downto 0;
signal pixel_y_int      : integer range 524 downto 0;
```



```
--instanciando contador 1s
```

```
COMPONENT pattern_timer
  PORT(
    clk      : IN std_logic;
    enable   : IN std_logic;
    reset    : IN std_logic;
    sec_tick  : OUT std_logic
  );
END COMPONENT;
```

```
begin
```

```
--conectando contador 1s
```

```
pattern_test_alternator: pattern_timer
  PORT MAP(
    clk      => clk,
    enable   => en_patter_timer,
    reset    => reset,
    sec_tick  => sec_clock);
```

```
--convirtiendo valores de vectores de entrada directamente en enteros para la comparación
```

```
pixel_x_int <= CONV_INTEGER (pixel_x);
pixel_y_int <= CONV_INTEGER (pixel_y);
```

```
--declarando registros globales para salidas de color
```

```
process (clk,reset,r_video_reg, g_video_reg, b_video_reg)
begin
  if reset = '1' then
    r_video_test <= "0000";
    g_video_test <= "0000";
    b_video_test <= "0000";
  elsif clk='1' and clk'event then
    if enable = '1' then
      r_video_test <= r_video_reg;
      g_video_test <= g_video_reg;
      b_video_test <= b_video_reg;
    end if;
  end if;
end process;
```

```
--CLC para generar colores. Se compara la coordenada del pixel para saber cuál color mostrar en cada momento
```

```
--para patron vertical |||||
```

```
r_video_reg <= "1111" when ((pixel_x_int >= 0) and (pixel_x_int < 13) and
  (video_on = '1') and (sec_clock = '1')) else
  "0000" when ((pixel_x_int >= 13) and (pixel_x_int < 26) and
  (video_on = '1') and (sec_clock = '1')) else
  "0001" when ((pixel_x_int >= 26) and (pixel_x_int < 39) and
  (video_on = '1') and (sec_clock = '1')) else
  "0010" when ((pixel_x_int >= 39) and (pixel_x_int < 52) and
  (video_on = '1') and (sec_clock = '1')) else
```

```

"0011" when ((pixel_x_int >= 52) and (pixel_x_int < 65) and
             (video_on = '1') and (sec_clock = '1')) else
"0100" when ((pixel_x_int >= 65) and (pixel_x_int < 78) and
             (video_on = '1') and (sec_clock = '1')) else
"0101" when ((pixel_x_int >= 78) and (pixel_x_int < 91) and
             (video_on = '1') and (sec_clock = '1')) else
"0110" when ((pixel_x_int >= 91) and (pixel_x_int < 104) and
             (video_on = '1') and (sec_clock = '1')) else
"0111" when ((pixel_x_int >= 104) and (pixel_x_int < 117) and
             (video_on = '1') and (sec_clock = '1')) else
"1000" when ((pixel_x_int >= 117) and (pixel_x_int < 130) and
             (video_on = '1') and (sec_clock = '1')) else
"1001" when ((pixel_x_int >= 130) and (pixel_x_int < 143) and
             (video_on = '1') and (sec_clock = '1')) else
"1010" when ((pixel_x_int >= 143) and (pixel_x_int < 156) and
             (video_on = '1') and (sec_clock = '1')) else
"1011" when ((pixel_x_int >= 156) and (pixel_x_int < 169) and
             (video_on = '1') and (sec_clock = '1')) else
"1100" when ((pixel_x_int >= 169) and (pixel_x_int < 182) and
             (video_on = '1') and (sec_clock = '1')) else
"1101" when ((pixel_x_int >= 182) and (pixel_x_int < 195) and
             (video_on = '1') and (sec_clock = '1')) else
"1110" when ((pixel_x_int >= 195) and (pixel_x_int < 208) and
             (video_on = '1') and (sec_clock = '1')) else

```

--para patron horizontal =

```

"1111" when ((pixel_y_int >= 0) and (pixel_y_int < 10) and
             (video_on = '1') and (sec_clock = '0')) else
"0000" when ((pixel_y_int >= 10) and (pixel_y_int < 20) and
             (video_on = '1') and (sec_clock = '0')) else
"0001" when ((pixel_y_int >= 20) and (pixel_y_int < 30) and
             (video_on = '1') and (sec_clock = '0')) else
"0010" when ((pixel_y_int >= 30) and (pixel_y_int < 40) and
             (video_on = '1') and (sec_clock = '0')) else
"0011" when ((pixel_y_int >= 40) and (pixel_y_int < 50) and
             (video_on = '1') and (sec_clock = '0')) else
"0100" when ((pixel_y_int >= 50) and (pixel_y_int < 60) and
             (video_on = '1') and (sec_clock = '0')) else
"0101" when ((pixel_y_int >= 60) and (pixel_y_int < 70) and
             (video_on = '1') and (sec_clock = '0')) else
"0110" when ((pixel_y_int >= 70) and (pixel_y_int < 80) and
             (video_on = '1') and (sec_clock = '0')) else
"0111" when ((pixel_y_int >= 80) and (pixel_y_int < 90) and
             (video_on = '1') and (sec_clock = '0')) else
"1000" when ((pixel_y_int >= 90) and (pixel_y_int < 100) and
             (video_on = '1') and (sec_clock = '0')) else
"1001" when ((pixel_y_int >= 100) and (pixel_y_int < 110) and
             (video_on = '1') and (sec_clock = '0')) else
"1010" when ((pixel_y_int >= 110) and (pixel_y_int < 120) and
             (video_on = '1') and (sec_clock = '0')) else
"1011" when ((pixel_y_int >= 120) and (pixel_y_int < 130) and
             (video_on = '1') and (sec_clock = '0')) else

```

```

"1100" when ((pixel_y_int >= 130) and (pixel_y_int < 140) and
(video_on = '1') and (sec_clock = '0')) else
"1101" when ((pixel_y_int >= 140) and (pixel_y_int < 150) and
(video_on = '1') and (sec_clock = '0')) else
"1110" when ((pixel_y_int >= 150) and (pixel_y_int < 160) and
(video_on = '1') and (sec_clock = '0')) else
"0000";--este vector 0000 cuando llegue al final de la pantalla

--para patron vertical |||||
g_video_reg <= "1111" when ((pixel_x_int >= 208) and (pixel_x_int < 221) and
(video_on = '1') and (sec_clock = '1')) else
"0000" when ((pixel_x_int >= 221) and (pixel_x_int < 234) and
(video_on = '1') and (sec_clock = '1')) else
"0001" when ((pixel_x_int >= 234) and (pixel_x_int < 247) and
(video_on = '1') and (sec_clock = '1')) else
"0010" when ((pixel_x_int >= 247) and (pixel_x_int < 260) and
(video_on = '1') and (sec_clock = '1')) else
"0011" when ((pixel_x_int >= 260) and (pixel_x_int < 273) and
(video_on = '1') and (sec_clock = '1')) else
"0100" when ((pixel_x_int >= 273) and (pixel_x_int < 286) and
(video_on = '1') and (sec_clock = '1')) else
"0101" when ((pixel_x_int >= 286) and (pixel_x_int < 299) and
(video_on = '1') and (sec_clock = '1')) else
"0110" when ((pixel_x_int >= 299) and (pixel_x_int < 312) and
(video_on = '1') and (sec_clock = '1')) else
"0111" when ((pixel_x_int >= 312) and (pixel_x_int < 325) and
(video_on = '1') and (sec_clock = '1')) else
"1000" when ((pixel_x_int >= 325) and (pixel_x_int < 338) and
(video_on = '1') and (sec_clock = '1')) else
"1001" when ((pixel_x_int >= 338) and (pixel_x_int < 351) and
(video_on = '1') and (sec_clock = '1')) else
"1010" when ((pixel_x_int >= 351) and (pixel_x_int < 364) and
(video_on = '1') and (sec_clock = '1')) else
"1011" when ((pixel_x_int >= 364) and (pixel_x_int < 377) and
(video_on = '1') and (sec_clock = '1')) else
"1100" when ((pixel_x_int >= 377) and (pixel_x_int < 390) and
(video_on = '1') and (sec_clock = '1')) else
"1101" when ((pixel_x_int >= 390) and (pixel_x_int < 403) and
(video_on = '1') and (sec_clock = '1')) else
"1110" when ((pixel_x_int >= 403) and (pixel_x_int < 416) and
(video_on = '1') and (sec_clock = '1')) else

--para patron horizontal =
"1111" when ((pixel_y_int >= 160) and (pixel_y_int < 170) and
(video_on = '1') and (sec_clock = '0')) else
"0000" when ((pixel_y_int >= 170) and (pixel_y_int < 180) and
(video_on = '1') and (sec_clock = '0')) else
"0001" when ((pixel_y_int >= 180) and (pixel_y_int < 190) and
(video_on = '1') and (sec_clock = '0')) else
"0010" when ((pixel_y_int >= 190) and (pixel_y_int < 200) and
(video_on = '1') and (sec_clock = '0')) else
"0011" when ((pixel_y_int >= 200) and (pixel_y_int < 210) and

```

```

        (video_on = '1') and (sec_clock = '0')) else
"0100" when ((pixel_y_int >= 210) and (pixel_y_int < 220) and
        (video_on = '1') and (sec_clock = '0')) else
"0101" when ((pixel_y_int >= 220) and (pixel_y_int < 230) and
        (video_on = '1') and (sec_clock = '0')) else
"0110" when ((pixel_y_int >= 230) and (pixel_y_int < 240) and
        (video_on = '1') and (sec_clock = '0')) else
"0111" when ((pixel_y_int >= 240) and (pixel_y_int < 250) and
        (video_on = '1') and (sec_clock = '0')) else
"1000" when ((pixel_y_int >= 250) and (pixel_y_int < 260) and
        (video_on = '1') and (sec_clock = '0')) else
"1001" when ((pixel_y_int >= 260) and (pixel_y_int < 270) and
        (video_on = '1') and (sec_clock = '0')) else
"1010" when ((pixel_y_int >= 270) and (pixel_y_int < 280) and
        (video_on = '1') and (sec_clock = '0')) else
"1011" when ((pixel_y_int >= 280) and (pixel_y_int < 290) and
        (video_on = '1') and (sec_clock = '0')) else
"1100" when ((pixel_y_int >= 290) and (pixel_y_int < 300) and
        (video_on = '1') and (sec_clock = '0')) else
"1101" when ((pixel_y_int >= 300) and (pixel_y_int < 310) and
        (video_on = '1') and (sec_clock = '0')) else
"1110" when ((pixel_y_int >= 310) and (pixel_y_int < 320) and
        (video_on = '1') and (sec_clock = '0')) else
"0000";--este vector 0000 cuando llegue al final de la pantalla

```

--para patron vertical |||||

```

b_video_reg <= "1111" when ((pixel_x_int >= 416) and (pixel_x_int < 429) and
        (video_on = '1') and (sec_clock = '1')) else
"0000" when ((pixel_x_int >= 429) and (pixel_x_int < 442) and
        (video_on = '1') and (sec_clock = '1')) else
"0001" when ((pixel_x_int >= 442) and (pixel_x_int < 455) and
        (video_on = '1') and (sec_clock = '1')) else
"0010" when ((pixel_x_int >= 455) and (pixel_x_int < 468) and
        (video_on = '1') and (sec_clock = '1')) else
"0011" when ((pixel_x_int >= 468) and (pixel_x_int < 481) and
        (video_on = '1') and (sec_clock = '1')) else
"0100" when ((pixel_x_int >= 481) and (pixel_x_int < 494) and
        (video_on = '1') and (sec_clock = '1')) else
"0101" when ((pixel_x_int >= 494) and (pixel_x_int < 507) and
        (video_on = '1') and (sec_clock = '1')) else
"0110" when ((pixel_x_int >= 507) and (pixel_x_int < 520) and
        (video_on = '1') and (sec_clock = '1')) else
"0111" when ((pixel_x_int >= 520) and (pixel_x_int < 533) and
        (video_on = '1') and (sec_clock = '1')) else
"1000" when ((pixel_x_int >= 533) and (pixel_x_int < 546) and
        (video_on = '1') and (sec_clock = '1')) else
"1001" when ((pixel_x_int >= 546) and (pixel_x_int < 559) and
        (video_on = '1') and (sec_clock = '1')) else
"1010" when ((pixel_x_int >= 559) and (pixel_x_int < 572) and
        (video_on = '1') and (sec_clock = '1')) else
"1011" when ((pixel_x_int >= 572) and (pixel_x_int < 585) and
        (video_on = '1') and (sec_clock = '1')) else

```

```

"1100" when ((pixel_x_int >= 585) and (pixel_x_int < 598) and
(video_on = '1') and (sec_clock = '1')) else
"1101" when ((pixel_x_int >= 598) and (pixel_x_int < 611) and
(video_on = '1') and (sec_clock = '1')) else
"1110" when ((pixel_x_int >= 611) and (pixel_x_int < 640) and
(video_on = '1') and (sec_clock = '1')) else

--para patron horizontal =
"1111" when ((pixel_y_int >= 320) and (pixel_y_int < 330) and
(video_on = '1') and (sec_clock = '0')) else
"0000" when ((pixel_y_int >= 330) and (pixel_y_int < 340) and
(video_on = '1') and (sec_clock = '0')) else
"0001" when ((pixel_y_int >= 340) and (pixel_y_int < 350) and
(video_on = '1') and (sec_clock = '0')) else
"0010" when ((pixel_y_int >= 350) and (pixel_y_int < 360) and
(video_on = '1') and (sec_clock = '0')) else
"0011" when ((pixel_y_int >= 360) and (pixel_y_int < 370) and
(video_on = '1') and (sec_clock = '0')) else
"0100" when ((pixel_y_int >= 370) and (pixel_y_int < 380) and
(video_on = '1') and (sec_clock = '0')) else
"0101" when ((pixel_y_int >= 380) and (pixel_y_int < 390) and
(video_on = '1') and (sec_clock = '0')) else
"0110" when ((pixel_y_int >= 390) and (pixel_y_int < 400) and
(video_on = '1') and (sec_clock = '0')) else
"0111" when ((pixel_y_int >= 400) and (pixel_y_int < 410) and
(video_on = '1') and (sec_clock = '0')) else
"1000" when ((pixel_y_int >= 410) and (pixel_y_int < 420) and
(video_on = '1') and (sec_clock = '0')) else
"1001" when ((pixel_y_int >= 420) and (pixel_y_int < 430) and
(video_on = '1') and (sec_clock = '0')) else
"1010" when ((pixel_y_int >= 430) and (pixel_y_int < 440) and
(video_on = '1') and (sec_clock = '0')) else
"1011" when ((pixel_y_int >= 440) and (pixel_y_int < 450) and
(video_on = '1') and (sec_clock = '0')) else
"1100" when ((pixel_y_int >= 450) and (pixel_y_int < 460) and
(video_on = '1') and (sec_clock = '0')) else
"1101" when ((pixel_y_int >= 460) and (pixel_y_int < 470) and
(video_on = '1') and (sec_clock = '0')) else
"1110" when ((pixel_y_int >= 470) and (pixel_y_int < 480) and
(video_on = '1') and (sec_clock = '0')) else
"0000";--este vector 0000 cuando llegue al final de la pantalla
en_patter_timer <= self_test and enable; --habilitación del contador 1s
end Behavioral;

```

#### Anexo 4. Código VHDL correspondiente a la implementación del contador para 1s en el bloque **VGA\_self\_test**

```

entity pattern_timer is
    port(
        clk            : in std_logic;
        enable         : in std_logic;
        reset          : in std_logic;
        sec_tick       : out std_logic
    );
end pattern_timer;

-----
--Descripción de pines
--  enable    =>>  Habilitación global.
--  sec_tick   =>>  Salida que complementa su estado cada 1s.
-----

architecture Behavioral of pattern_timer is
    signal count      : std_logic_vector (25 downto 0);
    signal tick       : std_logic;
begin
    process (clk)
        begin
            if clk = '1' and clk 'event then
                if reset='1' then
                    count <= (others => '0');
                    tick  <= '0';
                elsif enable = '1' then
                    count <= count + 1;
                    if(count = 49999999) then
                        --reseteando el contador y cambia salida
                        tick <= not tick;
                        count <= (others => '0');
                    end if;
                else
                    tick <= '0';
                    count <= (others => '0');
                end if;
            end if;
        end process;
        --cableando tick a la salida
        sec_tick <= tick;
    end Behavioral;

```

Anexo 5. Código VHDL correspondiente a la implementación del bloque **VGA\_rotary\_encoder**

```

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

entity VGA_rotary_encoder is

    Port (
        clk                : in STD_LOGIC;
        reset               : in std_logic;
        enable              : in std_logic;
        video_on            : in std_logic;
        rot_a               : in std_logic;
        rot_b               : in std_logic;
        color_selector      : in std_logic_vector (1 downto 0);
        red_ColorReg        : in std_logic_vector (3 downto 0);
        green_ColorReg      : in std_logic_vector (3 downto 0);
        blue_ColorReg       : in std_logic_vector (3 downto 0);
        pixel_x             : in std_logic_vector (9 downto 0);
        r_video_encoder     : out std_logic_vector (3 downto 0);
        g_video_encoder     : out std_logic_vector (3 downto 0);
        b_video_encoder     : out std_logic_vector (3 downto 0);
    );

end VGA_rotary_encoder;

```

---

--Descripción de pines

-- <b>video_on</b>	=>>	Entrada desde VGA_sync_unit
-- <b>pixel_x</b>	=>>	Entrada desde VGA_sync_unit
-- <b>color_selector</b>	=>>	Vector de entrada externa al módulo para seleccionar el registro de color a escribir
--		Entrada para rotary.
-- <b>rot_a</b>	=>>	Entrada para rotary. Pin para conectar el encoder de la placa. Va directo al filtro de entrada de rotary para decodificar movimiento del encoder.
--		
-- <b>rot_b</b>	=>>	Entrada para rotary. Pin para conectar el encoder de la placa. Va directo al filtro de entrada de rotary para decodificar movimiento del encoder.
--		
-- <b>red_ColorReg</b>	=>>	Vector de entrada externa al módulo con valor para establecer color rojo.
-- <b>green_ColorReg</b>	=>>	Vector de entrada externa al módulo con valor para establecer color verde.
-- <b>blue_ColorReg</b>	=>>	Vector de entrada externa al módulo con valor para establecer color azul.
-- <b>r_video_encoder</b>	=>>	Vector de salida para color rojo.
-- <b>g_video_encoder</b>	=>>	Vector de salida para color verde.
-- <b>b_video_encoder</b>	=>>	Vector de salida para color azul.

---



architecture Behavioral of VGA\_rotary\_encoder is

```

signal pixel_x_int          : integer range 1586 downto 0;

--registros finales de salida
signal r_video_reg          : std_logic_vector (3 downto 0);
signal g_video_reg          : std_logic_vector (3 downto 0);
signal b_video_reg          : std_logic_vector (3 downto 0);

--registros para almacenar el valor que viene del encoder
signal red_register         : std_logic_vector (3 downto 0);
signal green_register       : std_logic_vector (3 downto 0);
signal blue_register        : std_logic_vector (3 downto 0);

--componente para atender al encoder rotatorio
COMPONENT rotary
  PORT (
    clk                : IN std_logic;
    reset              : IN std_logic;
    rotary_a           : IN std_logic;
    rotary_b           : IN std_logic;
    color_sel          : IN STD_logic_vector (1 downto 0);
    red_initValue      : IN std_logic_vector (3 downto 0);
    green_initValue    : IN std_logic_vector (3 downto 0);
    blue_initValue     : IN std_logic_vector (3 downto 0);
    red_count          : OUT std_logic_vector (3 downto 0);
    green_count        : OUT std_logic_vector (3 downto 0);
    blue_count         : OUT std_logic_vector (3 downto 0)
  );
END COMPONENT;
```

**begin**

--convirtiendo valores de vectores de entrada directamente en enteros para la comparación

```
pixel_x_int <= CONV_INTEGER (pixel_x);
```

```
Inst_rotary: rotary PORT MAP(
```

```

    red_initValue      => red_ColorReg,
    green_initValue    => green_ColorReg,
    blue_initValue     => blue_ColorReg,
    red_count          => red_register,
    green_count        => green_register,
    blue_count         => blue_register,
    color_sel          => color_selector,
    reset              => reset,
    rotary_a           => rot_a,
    rotary_b           => rot_b,
    clk                => clk

```

```
);
```



--registros para la salida de colores

```
video_output: process (clk, reset, r_video_reg, g_video_reg, b_video_reg)
begin
    if reset = '1' then

        r_video_encoder <= "0000";
        g_video_encoder <= "0000";
        b_video_encoder <= "0000";

    elsif clk = '1' and clk'event then

        if enable = '1' then

            r_video_encoder <= r_video_reg;
            g_video_encoder <= g_video_reg;
            b_video_encoder <= b_video_reg;

        end if;

    end if;

end process;
```

--CLC para generar colores. Idem a VGA\_self\_test

```
r_video_reg <= red_register when (pixel_x_int >= 0 and pixel_x_int < 213 and
video_on = '1') else "0000";

g_video_reg <= green_register when
(pixel_x_int >= 213 and pixel_x_int < 426 and video_on = '1')
else "0000";

b_video_reg <= blue_register when (pixel_x_int >= 426 and pixel_x_int < 640 and
video_on = '1') else "0000";

end Behavioral;
```

Anexo 6. Código VHDL correspondiente a la implementación al circuito **rotary** perteneciente al bloque **VGA\_rotary\_encoder**.

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

entity rotary is
  Port (
    clk           : in STD_LOGIC;
    reset         : in STD_LOGIC;
    rotary_a      : in STD_LOGIC;
    rotary_b      : in STD_LOGIC;
    color_sel     : in STD_LOGIC_VECTOR (1 downto 0);
    red_initValue : in STD_LOGIC_VECTOR (3 downto 0);
    green_initValue : in STD_LOGIC_VECTOR (3 downto 0);
    blue_initValue : in STD_LOGIC_VECTOR (3 downto 0);
    red_count     : out STD_LOGIC_VECTOR (3 downto 0);
    green_count   : out STD_LOGIC_VECTOR (3 downto 0);
    blue_count    : out STD_LOGIC_VECTOR (3 downto 0));
end rotary;
```

-----  
 --Descripción de pines

```
--  rotary_a      ==>  Entrada desde VGA_rotary_encoder. Pin de
                        interfaz para el encoder rotatorio de la placa.
                        Esta señal primero se sincroniza y luego pasa
                        al filtro de entrada para obtener señales de
                        movimiento.

--  rotary_b      ==>  Idem al anterior.
--  color_sel     ==>  Entrada desde VGA_rotary_encoder. Seleccionan
                        el registro de color donde será escrito el
                        valor de la entrada color_initValue

--  red_initValue ==>  Entrada desde VGA_rotary_encoder. Valor de
                        color a cargar en el registro del contador del
                        color rojo

--  green_initValue ==>  Misma funcionalidad que el anterior, pero para
                        el color verde.

--  blue_initValue ==>  Misma funcionalidad que el anterior, pero para
                        el color azul.

--  red_count     ==>  Registro del contador que establece el valor
                        del color rojo a visualizar.

--  green_count   ==>  Registro del contador que establece el valor
                        del color verde a visualizar.

--  blue_count    ==>  Registro del contador que establece el valor
                        del color azul a visualizar.
-----
```

**architecture Behavioral of rotary is**

```

--FF para sincronizador de señales de entrada del encoder
signal rotary_a_in      : std_logic;
signal rotary_b_in      : std_logic;

--signal para concatenar las salidas del sincronizador (cable)
signal rotary_in        : std_logic_vector (1 downto 0);

--registros de salida del filtro para el encoder.
signal rotary_q1        : std_logic;
signal rotary_q2        : std_logic;

--latch para almacenar el estado anterior de rotary_q1 y comparar con el actual en
--rotary_q1
signal delay_rotary_q1  : std_logic;

--signal para indicar la ocurrencia de algún cambio en el encoder
signal rotary_event     : std_logic;

--signal para indicar sentido de rotación. Rotary_left = 1 increase
signal rotary_left      : std_logic;

-----
--Señales y registros asociados a los contadores de color
--  color_number          =>>  Registro para contador de color
--  color_numberPrevious  =>>  Registro para almacenar valor anterior del
--                               contador de color
--  changeColor           =>>  Señal para indicar un cambio de valor
-----

--para el rojo
signal red_number          : std_logic_vector (3 downto 0);
signal red_numberPrevious  : std_logic_vector (3 downto 0);
signal changeRed           : std_logic;

--para el verde
signal green_number        : std_logic_vector (3 downto 0);
signal green_numberPrevious : std_logic_vector (3 downto 0);
signal changeGreen         : std_logic;

--para el azul
signal blue_number         : std_logic_vector (3 downto 0);
signal blue_numberPrevious  : std_logic_vector (3 downto 0);
signal changeBlue          : std_logic;

--decoder
signal red_sel             : std_logic;
signal green_sel           : std_logic;
signal blue_sel            : std_logic;

```

## begin

```

--process para sincronizar entradas del encoder y decodifica orden de activación
--de los switches del encoder
rotary_filter: process(clk)
begin
    if clk'event and clk='1' then
        --Sincronizando entradas con reloj del sistema.
        rotary_a_in <= rotary_a;
        rotary_b_in <= rotary_b;
        rotary_in <= rotary_b_in & rotary_a_in;
        --determinando estado de switches del encoder según entradas
        case rotary_in is
            when "00" => rotary_q1 <= '0';
                           rotary_q2 <= rotary_q2;

            when "01" => rotary_q1 <= rotary_q1;
                           rotary_q2 <= '0';

            when "10" => rotary_q1 <= rotary_q1;
                           rotary_q2 <= '1';

            when "11" => rotary_q1 <= '1';
                           rotary_q2 <= rotary_q2;

            when others => rotary_q1 <= rotary_q1;
                           rotary_q2 <= rotary_q2;

        end case;
    end if;

end process rotary_filter;

--tomando valor de entradas y decodificando dirección de movimiento (izquierda o
--derecha)
direction: process(clk)
begin
    if clk'event and clk='1' then
        delay_rotary_q1 <= rotary_q1;
        if rotary_q1='1' and delay_rotary_q1='0' then
            rotary_event <= '1';
            rotary_left <= rotary_q2;

        else
            rotary_event <= '0';
            rotary_left <= rotary_left;

        end if;

    end if;

end process direction;

```

```

--Multiplexor para la selección de la señal de habilitación a activar para
--escribir registros de contador de colores según la entrada de selección
--color_sel
color_selector: process (color_sel)
    begin
        case color_sel is

            when "00" => red_sel    <= '1';
                        green_sel <= '0';
                        blue_sel  <= '0';

            when "01" => green_sel <= '1';
                        blue_sel  <= '0';
                        red_sel    <= '0';

            when "10" => blue_sel  <= '1';
                        red_sel    <= '0';
                        green_sel  <= '0';

            when others => blue_sel <= '0';
                        red_sel    <= '0';
                        green_sel  <= '0';

        end case;
    end process;

```

```

-----
--RED-
-----
--detector de cambio para contador rojo. Compara el valor del conteo actual con el
--previo y si cambia activa señal para indicarlo
red_change_detector: process (clk)
begin
    if clk'event and clk='1' then
        red_numberPrevious <= red_number;
        if (red_numberPrevious /= red_number) then
            changeRed <= '1';
        else
            changeRed <= '0';
        end if;
    end if;
end process;

--contador rojo
--contador binario up/down con entrada de carga paralela.
--Entradas de reset, habilitación y dirección sincrónicas.

-----
--DESCRIPCION DE PINES
--  reset      =>>  RESET global del sistema
--  clk         =>>  CLK global del sistema
--  changeRed   =>>  control de carga paralela
--  red_initValue =>>  entrada carga paralela
--  clock_enable =>>  rotary event AND red sel. Entrada de habilitación del
--                   contador.
--  rotary_left  =>>  Control de dirección de conteo
-----
red_counter: process(clk)
begin
    if clk'event and clk='1' then
        if reset = '1' then
            red_count <= (others => '0');
        else
            if changeRed = '1' then
                red_number <= red_initValue;
            end if;
            if rotary_event = '1' and red_sel = '1' then
                if rotary_left = '1' then
                    red_number <= red_number+'1'; --INCREASE
                else
                    red_number <= red_number-'1'; --DECREASE
                end if;
            end if;

            red_count <= red_number;
        end if;
    end if;
end process red_counter;

```

```

-----
--GREEN-
-----

--detector de cambio para contador verde. Idem al rojo
green_change_detector: process (clk)
begin
    if clk'event and clk = '1' then
        green_numberPrevious <= green_number;
        if ( green_numberPrevious /= green_number ) then
            changeGreen <= '1';
        else
            changeGreen <= '0';
        end if;
    end if;
end process;

--contador verde
--contador binario up/down con entrada de carga paralela.
--Entradas de reset, habilitación y dirección sincrónicas.

-----
--DESCRIPCION DE PINES
--  reset          =>>  RESET global del sistema
--  clk             =>>  CLK global del sistema
--  changeGreen     =>>  control de carga paralela
--  green_initValue =>>  entrada carga paralela
--  clock_enable    =>>  rotary event AND green sel. Entrada de habilitación
--                  del contador.
--  rotary_left     =>>  Control de dirección de conteo
-----

green_counter: process(clk)
begin
    if clk'event and clk='1' then
        if reset = '1' then
            green_count <= (others => '0');
        else
            if changeGreen = '1' then
                green_number <= green_initValue;
            end if;
            if rotary_event = '1' and green_sel = '1' then

                if rotary_left = '1' then
                    green_number <= green_number+'1'; --INCREASE
                else
                    green_number <= green_number-'1'; --DECREASE
                end if;
            end if;
            green_count <= green_number;
        end if;
    end if;
end process green_counter;

```

```

-----
-----BLUE-----
-----
--detector de cambio para contador azul. Idem al rojo
blue_change_detector: process (clk)
    begin
        if clk'event and clk = '1' then
            blue_numberPrevious <= blue_number;
            if (blue_numberPrevious /= blue_number) then
                changeBlue <= '1';
            else
                changeBlue <= '0';
            end if;
        end if;
    end if;
end process;

--contador azul
--contador binario up/down con entrada de carga paralela.
--Entradas de reset, habilitación y dirección sincrónicas.

-----
--DESCRIPCION DE PINES
--  reset          =>>  RESET global del sistema
--  clk             =>>  CLK global del sistema
--  changeBlue      =>>  control de carga paralela
--  blue_initValue  =>>  entrada carga paralela
--  clock_enable    =>>  rotary event AND blue sel. Entrada de habilitación
--                  del contador.
--  rotary_left     =>>  Control de dirección de conteo
-----
blue_counter: process(clk)
    begin
        if clk'event and clk='1' then
            if reset = '1' then
                blue_count <= (others => '0');
            else
                if changeBlue = '1' then
                    blue_number <= blue_initValue;
                end if;
                if rotary_event = '1' and blue_sel = '1' then
                    if rotary_left = '1' then
                        blue_number <= blue_number+'1'; --INCREASE
                    else
                        blue_number <= blue_number-'1'; --DECREASE
                    end if;
                end if;
                blue_count <= blue_number;
            end if;
        end if;
    end if;
end process blue_counter;
end Behavioral;

```



Anexo 7. Código VHDL correspondiente al multiplexor de salida **VGA\_outputmux**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity VGA_outputMux is
Port (
    selector          : in  STD_LOGIC;
    rgb_encoder        : in  STD_LOGIC_VECTOR (11 downto 0);
    rgb_test           : in  STD_LOGIC_VECTOR (11 downto 0);
    rgb_output         : out STD_LOGIC_VECTOR (11 downto 0));
end VGA_outputMux;

architecture Behavioral of VGA_outputMux is

begin
    rgb_output <= rgb_test WHEN selector ='1' ELSE rgb_encoder;

end Behavioral;

```

Anexo 8. Código VHDL correspondiente al *top level* de todo el diseño, **VGA\_module\_top**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity VGA_module_top is

    port (
        clk                : in std_logic;
        reset               : in std_logic;
        enable              : in std_logic;
        self_test           : in std_logic;
        rotary_a            : in std_logic;
        rotary_b            : in std_logic;
        color_selector0     : in std_logic;
        color_selector1     : in std_logic;
        red_ColorValue      : in std_logic_vector (3 downto 0);
        green_ColorValue    : in std_logic_vector (3 downto 0);
        blue_ColorValue     : in std_logic_vector (3 downto 0);
        hsync               : out std_logic;
        vsync               : out std_logic;
        rgb_finalOut        : out std_logic_vector(11 downto 0));

end VGA_module_top;

```

-----  
 --EL PINOUT DE ESTE BLOQUE COINCIDE CON LA DISTRIBUCIÓN DE PINES PRESENTADA EN LA  
 --Fig. 8 DEL EPIGRAFE Características Generales  
 -----

```

architecture Behavioral of VGA_module_top is

    signal video_on_wire   : std_logic;
    signal pixel_x_wire    : std_logic_vector (9 downto 0);
    signal pixel_y_wire    : std_logic_vector (9 downto 0);

    signal r_encoder       : std_logic_vector (3 downto 0);
    signal g_encoder       : std_logic_vector (3 downto 0);
    signal b_encoder       : std_logic_vector (3 downto 0);

    signal r_test          : std_logic_vector (3 downto 0);
    signal g_test          : std_logic_vector (3 downto 0);
    signal b_test          : std_logic_vector (3 downto 0);

    signal rgb_muxTest     : std_logic_vector (11 downto 0);
    signal rgb_muxencoder  : std_logic_vector (11 downto 0);

end Behavioral;

```

```

COMPONENT VGA_rotary_encoder
  PORT (
    clk                : IN std_logic;
    reset              : IN std_logic;
    enable              : IN std_logic;
    video_on            : IN std_logic;
    color_selector      : IN std_logic_vector (1 downto 0);
    red_ColorReg        : in std_logic_vector (3 downto 0);
    green_ColorReg      : in std_logic_vector (3 downto 0);
    blue_ColorReg       : in std_logic_vector (3 downto 0);
    rot_a               : IN std_logic;
    rot_b               : IN std_logic;
    pixel_x             : IN std_logic_vector (9 downto 0);
    r_video_encoder     : OUT std_logic_vector (3 downto 0);
    g_video_encoder     : OUT std_logic_vector (3 downto 0);
    b_video_encoder     : OUT std_logic_vector (3 downto 0);
  );
END COMPONENT;

component vga_sync_unit
  PORT (
    clk                : in std_logic;
    reset              : in std_logic;
    enable              : in std_logic;
    hsync              : out std_logic;
    vsync              : out std_logic;
    video_on            : out std_logic;
    pixel_x             : out std_logic_vector (9 downto 0);
    pixel_y             : out std_logic_vector (9 downto 0));
end component;

COMPONENT VGA_module_self_test
  PORT (
    clk                : IN std_logic;
    reset              : IN std_logic;
    enable              : IN std_logic;
    self_test          : IN std_logic;
    video_on            : IN std_logic;
    pixel_x             : IN std_logic_vector (9 downto 0);
    pixel_y             : IN std_logic_vector (9 downto 0);
    r_video_test       : OUT std_logic_vector (3 downto 0);
    g_video_test       : OUT std_logic_vector (3 downto 0);
    b_video_test       : OUT std_logic_vector (3 downto 0);
  );
END COMPONENT;

COMPONENT VGA_outputMux
  PORT (
    selector            : IN std_logic;
    rgb_encoder         : IN STD_LOGIC_VECTOR (11 downto 0);
    rgb_test            : IN std_logic_vector (11 downto 0);
    rgb_output          : OUT std_logic_vector (11 downto 0));
END COMPONENT;

begin

```

```

vga_core      : vga_sync_unit
  PORT MAP (
    clk          => clk,
    reset        => reset,
    enable       => enable,
    hsync        => hsync,
    vsync        => vsync,
    video_on     => video_on_wire,
    pixel_x      => pixel_x_wire,
    pixel_y      => pixel_y_wire);
vga_self_tester: VGA_module_self_test
--izquierda: E/S del módulo VGA_module_self_test
  PORT MAP (
    clk          => clk,
    reset        => reset,
    enable       => enable,
    self_test    => self_test,
    video_on     => video_on_wire,
    pixel_x      => pixel_x_wire,
    pixel_y      => pixel_y_wire,
    r_video_test => r_test,
    g_video_test => g_test,
    b_video_test => b_test);
Inst_VGA_rotary_encoder: VGA_rotary_encoder
  PORT MAP (
    clk          => clk,
    reset        => reset,
    enable       => enable,
    video_on     => video_on_wire,
    color_selector (0) => color_selector0,
    color_selector (1) => color_selector1,
    red_ColorReg  => red_ColorValue,
    green_ColorReg => green_ColorValue,
    blue_ColorReg => blue_ColorValue,
    rot_a         => rotary_a,
    rot_b         => rotary_b,
    pixel_x       => pixel_x_wire,
    r_video_encoder => r_encoder,
    g_video_encoder => g_encoder,
    b_video_encoder => b_encoder);
--concatenando vectores
rgb_muxTest      <= r_test&g_test&b_test;
rgb_muxEncoder    <= r_encoder&g_encoder&b_encoder;
OutputSelector: VGA_outputMux
  PORT MAP (
    selector      => self_test,
    rgb_encoder    => rgb_muxEncoder,
    rgb_test       => rgb_muxTest,
    rgb_output     => rgb_finalOut);
end Behavioral;

```

## Anexo 9. Fichero de especificaciones de hardware (MHS) del sistema empotrado implementado.

```

#####
# Created by Base System Builder Wizard for Xilinx EDK 12.4 Build EDK_MS4.81d
# Tue Jun 11 02:44:25 2019
# Target Board:  Xilinx Spartan-3A Starter Kit Rev D
# Family:      spartan3a
# Device:      xc3s700a
# Package:     fg484
# Speed Grade:  -4
# Processor number: 1
# Processor 1: microblaze_0
# System clock frequency: 62.5
# Debug Interface: On-Chip HW Debug Module
#####
PARAMETER VERSION = 2.1.0

PORT fpga_0_RS232_DTE_RX_pin = fpga_0_RS232_DTE_RX_pin, DIR = I
PORT fpga_0_RS232_DTE_TX_pin = fpga_0_RS232_DTE_TX_pin, DIR = 0
PORT fpga_0_LEDs_8Bit_GPIO_IO_0_pin = fpga_0_LEDs_8Bit_GPIO_IO_0_pin, DIR = 0,
VEC = [0:7]
PORT fpga_0_DIPs_4Bit_GPIO_IO_I_pin = fpga_0_DIPs_4Bit_GPIO_IO_I_pin, DIR = I,
VEC = [0:3]
PORT fpga_0_DDR2_SDRAM_DDR2_Clk_pin = fpga_0_DDR2_SDRAM_DDR2_Clk_pin, DIR = 0
PORT fpga_0_DDR2_SDRAM_DDR2_Clk_n_pin = fpga_0_DDR2_SDRAM_DDR2_Clk_n_pin, DIR = 0
PORT fpga_0_DDR2_SDRAM_DDR2_CE_pin = fpga_0_DDR2_SDRAM_DDR2_CE_pin, DIR = 0
PORT fpga_0_DDR2_SDRAM_DDR2_CS_n_pin = fpga_0_DDR2_SDRAM_DDR2_CS_n_pin, DIR = 0
PORT fpga_0_DDR2_SDRAM_DDR2_ODT_pin = fpga_0_DDR2_SDRAM_DDR2_ODT_pin, DIR = 0
PORT fpga_0_DDR2_SDRAM_DDR2_RAS_n_pin = fpga_0_DDR2_SDRAM_DDR2_RAS_n_pin, DIR = 0
PORT fpga_0_DDR2_SDRAM_DDR2_CAS_n_pin = fpga_0_DDR2_SDRAM_DDR2_CAS_n_pin, DIR = 0
PORT fpga_0_DDR2_SDRAM_DDR2_WE_n_pin = fpga_0_DDR2_SDRAM_DDR2_WE_n_pin, DIR = 0
PORT fpga_0_DDR2_SDRAM_DDR2_BankAddr_pin = fpga_0_DDR2_SDRAM_DDR2_BankAddr_pin,
DIR = 0, VEC = [1:0]
PORT fpga_0_DDR2_SDRAM_DDR2_Addr_pin = fpga_0_DDR2_SDRAM_DDR2_Addr_pin, DIR = 0,
VEC = [12:0]
PORT fpga_0_DDR2_SDRAM_DDR2_DQ_pin = fpga_0_DDR2_SDRAM_DDR2_DQ_pin, DIR = IO, VEC
= [15:0]
PORT fpga_0_DDR2_SDRAM_DDR2_DM_pin = fpga_0_DDR2_SDRAM_DDR2_DM_pin, DIR = 0, VEC
= [1:0]
PORT fpga_0_DDR2_SDRAM_DDR2_DQS_pin = fpga_0_DDR2_SDRAM_DDR2_DQS_pin, DIR = IO,
VEC = [1:0]
PORT fpga_0_DDR2_SDRAM_DDR2_DQS_n_pin = fpga_0_DDR2_SDRAM_DDR2_DQS_n_pin, DIR =
IO, VEC = [1:0]
PORT fpga_0_DDR2_SDRAM_DDR2_DQS_Div_0_pin = fpga_0_DDR2_SDRAM_DDR2_DQS_Div_0_pin,
DIR = 0
PORT fpga_0_DDR2_SDRAM_DDR2_DQS_Div_I_pin = fpga_0_DDR2_SDRAM_DDR2_DQS_Div_I_pin,
DIR = I
PORT fpga_0_clk_1_sys_clk_pin = dcm_clk_s, DIR = I, SIGIS = CLK, CLK_FREQ =
50000000
PORT fpga_0_rst_1_sys_rst_pin = sys_rst_s, DIR = I, SIGIS = RST, RST_POLARITY = 1

```

```

PORT xps_gpio_LCD_driver_GPIO_IO_pin = xps_gpio_LCD_driver_GPIO_IO, DIR = IO, VEC
= [0:6]
PORT vga_module_0_rotary_a_in_pin = vga_module_0_rotary_a_in, DIR = I
PORT vga_module_0_rotary_b_in_pin = vga_module_0_rotary_b_in, DIR = I
PORT vga_module_0_hsync_out_pin = vga_module_0_hsync_out, DIR = 0
PORT vga_module_0_vsync_out_pin = vga_module_0_vsync_out, DIR = 0
PORT vga_module_0_rgb_finalOut_pin = vga_module_0_rgb_finalOut, DIR = 0, VEC =
[11:0]
PORT xps_iic_0_Gpo_pin = xps_iic_0_Gpo, DIR = 0, VEC = [31:31]
PORT xps_iic_0_Sda = xps_iic_0_Sda, DIR = IO
PORT xps_iic_0_Scl = xps_iic_0_Scl, DIR = IO

```

```

BEGIN microblaze
PARAMETER INSTANCE = microblaze_0
PARAMETER C_AREA_OPTIMIZED = 1
PARAMETER C_USE_BARREL = 1
PARAMETER C_DEBUG_ENABLED = 1
PARAMETER HW_VER = 8.00.b
PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 1
PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 1
BUS_INTERFACE DPLB = mb_plb
BUS_INTERFACE IPLB = mb_plb
BUS_INTERFACE DEBUG = microblaze_0_mdm_bus
BUS_INTERFACE DLMB = dlmb
BUS_INTERFACE ILMB = ilmb
PORT MB_RESET = mb_reset
PORT INTERRUPT = microblaze_0_Interrupt
END

```

```

BEGIN plb_v46
PARAMETER INSTANCE = mb_plb
PARAMETER HW_VER = 1.05.a
PORT PLB_Clk = clk_62_5000MHz
PORT SYS_Rst = sys_bus_reset
END

```

```

BEGIN lmb_v10
PARAMETER INSTANCE = ilmb
PARAMETER HW_VER = 1.00.a
PORT LMB_Clk = clk_62_5000MHz
PORT SYS_Rst = sys_bus_reset
END

```

```

BEGIN lmb_v10
PARAMETER INSTANCE = dlmb
PARAMETER HW_VER = 1.00.a
PORT LMB_Clk = clk_62_5000MHz
PORT SYS_Rst = sys_bus_reset
END

```

```

BEGIN lmb_bram_if_cntlr
  PARAMETER INSTANCE = dlmb_cntlr
  PARAMETER HW_VER = 2.10.b
  PARAMETER C_BASEADDR = 0x00000000
  PARAMETER C_HIGHADDR = 0x00003fff
  BUS_INTERFACE SLMB = dlmb
  BUS_INTERFACE BRAM_PORT = dlmb_port

```

```

END

```

```

BEGIN lmb_bram_if_cntlr
  PARAMETER INSTANCE = ilmb_cntlr
  PARAMETER HW_VER = 2.10.b
  PARAMETER C_BASEADDR = 0x00000000
  PARAMETER C_HIGHADDR = 0x00003fff
  BUS_INTERFACE SLMB = ilmb
  BUS_INTERFACE BRAM_PORT = ilmb_port

```

```

END

```

```

BEGIN bram_block
  PARAMETER INSTANCE = lmb_bram
  PARAMETER HW_VER = 1.00.a
  BUS_INTERFACE PORTA = ilmb_port
  BUS_INTERFACE PORTB = dlmb_port

```

```

END

```

```

BEGIN xps_uartlite
  PARAMETER INSTANCE = RS232_DTE
  PARAMETER C_BAUDRATE = 38400
  PARAMETER C_DATA_BITS = 8
  PARAMETER C_USE_PARITY = 0
  PARAMETER C_ODD_PARITY = 0
  PARAMETER HW_VER = 1.01.a
  PARAMETER C_BASEADDR = 0x84000000
  PARAMETER C_HIGHADDR = 0x8400ffff
  BUS_INTERFACE SPLB = mb_plb
  PORT RX = fpga_0_RS232_DTE_RX_pin
  PORT TX = fpga_0_RS232_DTE_TX_pin
  PORT Interrupt = RS232_DTE_Interrupt

```

```

END

```

```

BEGIN xps_gpio
  PARAMETER INSTANCE = LEDs_8Bit
  PARAMETER C_ALL_INPUTS = 0
  PARAMETER C_GPIO_WIDTH = 8
  PARAMETER C_INTERRUPT_PRESENT = 0
  PARAMETER C_IS_DUAL = 0
  PARAMETER HW_VER = 2.00.a
  PARAMETER C_BASEADDR = 0x81420000
  PARAMETER C_HIGHADDR = 0x8142ffff
  BUS_INTERFACE SPLB = mb_plb
  PORT GPIO_IO_0 = fpga_0_LEDs_8Bit_GPIO_IO_0_pin

```

```

END

```

```

BEGIN xps_gpio
PARAMETER INSTANCE = DIPs_4Bit
PARAMETER C_ALL_INPUTS = 1
PARAMETER C_GPIO_WIDTH = 4
PARAMETER C_INTERRUPT_PRESENT = 1
PARAMETER C_IS_DUAL = 0
PARAMETER HW_VER = 2.00.a
PARAMETER C_BASEADDR = 0x81440000
PARAMETER C_HIGHADDR = 0x8144ffff
BUS_INTERFACE SPLB = mb_plb
PORT IP2INTC_Irpt = DIPs_4Bit_IP2INTC_Irpt
PORT GPIO_IO_I = fpga_0_DIPs_4Bit_GPIO_IO_I_pin
END

```

```

BEGIN mpmc
PARAMETER INSTANCE = DDR2_SDRAM
PARAMETER C_NUM_PORTS = 1
PARAMETER C_SPECIAL_BOARD = S3A_STKIT
PARAMETER C_MEM_TYPE = DDR2
PARAMETER C_MEM_PARTNO = MT47H32M16-3E
PARAMETER C_MEM_DATA_WIDTH = 16
PARAMETER C_DDR2_DQSN_ENABLE = 1
PARAMETER C_PIM0_BASETYPE = 2
PARAMETER HW_VER = 6.02.a
PARAMETER C_PIM1_BASETYPE = 0
PARAMETER C_MPMC_BASEADDR = 0x8c000000
PARAMETER C_MPMC_HIGHADDR = 0xffffffff
BUS_INTERFACE SPLB0 = mb_plb
PORT MPMC_Clk0 = clk_125_0000MHzDCM0
PORT MPMC_Clk90 = clk_125_0000MHz90DCM0
PORT MPMC_Rst = sys_periph_reset
PORT DDR2_Clk = fpga_0_DDR2_SDRAM_DDR2_Clk_pin
PORT DDR2_Clk_n = fpga_0_DDR2_SDRAM_DDR2_Clk_n_pin
PORT DDR2_CE = fpga_0_DDR2_SDRAM_DDR2_CE_pin
PORT DDR2_CS_n = fpga_0_DDR2_SDRAM_DDR2_CS_n_pin
PORT DDR2_ODT = fpga_0_DDR2_SDRAM_DDR2_ODT_pin
PORT DDR2_RAS_n = fpga_0_DDR2_SDRAM_DDR2_RAS_n_pin
PORT DDR2_CAS_n = fpga_0_DDR2_SDRAM_DDR2_CAS_n_pin
PORT DDR2_WE_n = fpga_0_DDR2_SDRAM_DDR2_WE_n_pin
PORT DDR2_BankAddr = fpga_0_DDR2_SDRAM_DDR2_BankAddr_pin
PORT DDR2_Addr = fpga_0_DDR2_SDRAM_DDR2_Addr_pin
PORT DDR2_DQ = fpga_0_DDR2_SDRAM_DDR2_DQ_pin
PORT DDR2_DM = fpga_0_DDR2_SDRAM_DDR2_DM_pin
PORT DDR2_DQS = fpga_0_DDR2_SDRAM_DDR2_DQS_pin
PORT DDR2_DQS_n = fpga_0_DDR2_SDRAM_DDR2_DQS_n_pin
PORT DDR2_DQS_Div_0 = fpga_0_DDR2_SDRAM_DDR2_DQS_Div_0_pin
PORT DDR2_DQS_Div_I = fpga_0_DDR2_SDRAM_DDR2_DQS_Div_I_pin
END

```

```

BEGIN clock_generator
PARAMETER INSTANCE = clock_generator_0
PARAMETER C_CLKIN_FREQ = 50000000

```



```

PARAMETER C_CLKOUT0_FREQ = 125000000
PARAMETER C_CLKOUT0_PHASE = 90
PARAMETER C_CLKOUT0_GROUP = DCM0
PARAMETER C_CLKOUT0_BUF = TRUE
PARAMETER C_CLKOUT1_FREQ = 125000000
PARAMETER C_CLKOUT1_PHASE = 0
PARAMETER C_CLKOUT1_GROUP = DCM0
PARAMETER C_CLKOUT1_BUF = TRUE
PARAMETER C_CLKOUT2_FREQ = 62500000
PARAMETER C_CLKOUT2_PHASE = 0
PARAMETER C_CLKOUT2_GROUP = NONE
PARAMETER C_CLKOUT2_BUF = TRUE
PARAMETER C_EXT_RESET_HIGH = 1
PARAMETER HW_VER = 4.01.a
PARAMETER C_DEVICE = 3s700a
PARAMETER C_PACKAGE = fg484
PARAMETER C_SPEEDGRADE = -4
PARAMETER C_CLKOUT3_FREQ = 50000000
PORT CLKIN = dcm_clk_s
PORT CLKOUT0 = clk_125_0000MHz90DCM0
PORT CLKOUT1 = clk_125_0000MHzDCM0
PORT CLKOUT2 = clk_62_5000MHz
PORT RST = sys_rst_s
PORT LOCKED = Dcm_all_locked
PORT CLKOUT3 = clock_generator_0_CLKOUT3
END

BEGIN mdm
PARAMETER INSTANCE = mdm_0
PARAMETER C_MB_DBG_PORTS = 1
PARAMETER C_USE_UART = 1
PARAMETER HW_VER = 2.00.a
PARAMETER C_BASEADDR = 0x84400000
PARAMETER C_HIGHADDR = 0x8440ffff
BUS_INTERFACE SPLB = mb_plb
BUS_INTERFACE MBDEBUG_0 = microblaze_0_mdm_bus
PORT Debug_SYS_Rst = Debug_SYS_Rst
END

BEGIN proc_sys_reset
PARAMETER INSTANCE = proc_sys_reset_0
PARAMETER C_EXT_RESET_HIGH = 1
PARAMETER HW_VER = 3.00.a
PORT Slowest_sync_clk = clk_62_5000MHz
PORT Ext_Reset_In = sys_rst_s
PORT MB_Debug_Sys_Rst = Debug_SYS_Rst
PORT Dcm_locked = Dcm_all_locked
PORT MB_Reset = mb_reset
PORT Bus_Struct_Reset = sys_bus_reset
PORT Peripheral_Reset = sys_periph_reset
END

```

```

BEGIN xps_intc
  PARAMETER INSTANCE = xps_intc_0
  PARAMETER HW_VER = 2.01.a
  PARAMETER C_BASEADDR = 0x81800000
  PARAMETER C_HIGHADDR = 0x8180ffff
  BUS_INTERFACE SPLB = mb_plb
  PORT Intr = xps_timer_0_Interrupt & DIPs_4Bit_IP2INTC_Irpt &
xps_iic_0_IIC2INTC_Irpt & RS232_DTE_Interrupt
  PORT Irq = microblaze_0_Interrupt
END

```

```

BEGIN xps_gpio
  PARAMETER INSTANCE = xps_gpio_LCD_driver
  PARAMETER HW_VER = 2.00.a
  PARAMETER C_GPIO_WIDTH = 7
  PARAMETER C_TRI_DEFAULT = 0x00000000
  PARAMETER C_BASEADDR = 0x81400000
  PARAMETER C_HIGHADDR = 0x8140ffff
  BUS_INTERFACE SPLB = mb_plb
  PORT GPIO_IO = xps_gpio_LCD_driver_GPIO_IO
END

```

```

BEGIN xps_timer
  PARAMETER INSTANCE = xps_timer_0
  PARAMETER HW_VER = 1.02.a
  PARAMETER C_ONE_TIMER_ONLY = 1
  PARAMETER C_BASEADDR = 0x83c00000
  PARAMETER C_HIGHADDR = 0x83c0ffff
  BUS_INTERFACE SPLB = mb_plb
  PORT Interrupt = xps_timer_0_Interrupt
END

```

```

BEGIN vga_module
  PARAMETER INSTANCE = vga_module_0
  PARAMETER HW_VER = 1.00.a
  PARAMETER C_BASEADDR = 0xc4400000
  PARAMETER C_HIGHADDR = 0xc440ffff
  BUS_INTERFACE SPLB = mb_plb
  PORT rotary_a_in = vga_module_0_rotary_a_in
  PORT rotary_b_in = vga_module_0_rotary_b_in
  PORT hsync_out = vga_module_0_hsync_out
  PORT vsync_out = vga_module_0_vsync_out
  PORT rgb_finalOut = vga_module_0_rgb_finalOut
  PORT clk_in = clock_generator_0_CLKOUT3
END

```

```
BEGIN xps_iic
PARAMETER INSTANCE = xps_iic_0
PARAMETER HW_VER = 2.03.a
PARAMETER C_BASEADDR = 0x81600000
PARAMETER C_HIGHADDR = 0x8160ffff
PARAMETER C_SCL_INERTIAL_DELAY = 10
BUS_INTERFACE SPLB = mb_plb
PORT IIC2INTC_Irpt = xps_iic_0_IIC2INTC_Irpt
PORT Gpo = xps_iic_0_Gpo
PORT Sda = xps_iic_0_Sda
PORT Scl = xps_iic_0_Scl
END
```

Anexo X+1: Código programa principal completo (textual)

Anexo X+2: Código isr (Cada isr un anexo completo)