ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Grado

Sistema hardware de adquisición de vídeo basado en el sensor de imagen OV7670

AUTOR: Sebastián Cánovas Carrasco

DIRECTOR: Fco. Javier Toledo Moreo

Dar las gracias a mi tutor Fco. Javier Toledo Moreo por ofrecerme la oportunidad de trabajar en este proyecto y ayudarme a culminarlo con éxito, así como a Franz Fidler, mi tutor en mi periodo Erasmus en Austria, por ayudarme en todo lo que estaba en su mano.

Mencionar también a mi compañero Ginés Hidalgo Martínez por compartir su experiencia conmigo y poner su grano de arena para que este proyecto saliera adelante.

Por último, agradecer a todos mis seres queridos su cariño y apoyo incondicional en este periodo tan importante de mi vida.

Autor	Sebastián Cánovas Carrasco
E-mail del autor	sebas.canovas@gmail.com
Director	Fco. Javier Toledo Moreo
E-mail del director	javier.toledo@upct.es
Codirector	-
Título del PFC	Sistema hardware de adquisición de vídeo basado en el sensor de imagen OV7670
Descriptores	

Resumen

El proyecto se basa en el desarrollo de una plataforma hardware de adquisición de vídeo para su procesamiento en sistemas basados en dispositivos FPGA. En particular, la entrada de vídeo al sistema es un módulo con salida digital basado en el sensor OV7670 de *OmniVision*. La plataforma debe ser capaz de adquirir y almacenar vídeo procedente de este módulo, y posibilitar la configuración del mismo. El sistema será controlado por una FPGA, que gestionará la configuración de la cámara, el manejo de los datos recibidos y la visualización de las imágenes

Titulación	Grado en Ingeniería de Sistemas de Telecomunicación
Departamento	Electrónica, tecnología de computadoras y proyectos
Fecha	Octubre - 2014

Índice

Índice de figuras y tablas

Índice de códigos

Capítulo 1: Introducción

1.1 Objetivos

El objetivo principal del proyecto es el desarrollo de una plataforma hardware de adquisición de vídeo para su procesamiento en sistemas basados en dispositivos FPGA. En particular, la entrada de vídeo al sistema es un módulo con salida digital basado en el sensor OV7670 de *OmniVision*. La plataforma debe ser capaz de adquirir y almacenar vídeo procedente de este módulo, y posibilitar la configuración del mismo. El sistema será controlado por una FPGA, que gestionará la configuración de la cámara, el manejo de los datos recibidos y la visualización de las imágenes. La FPGA escogida para abordar el proyecto es la *ZedBoard SoC Zynq XC7X020-1CLG484,* fabricada por la empresa americana *Xilinx.*

El diseño de la plataforma se abordará dividiéndolo en módulos con una función concreta, aprovechando la programación estructural que ofrece VHDL, y, una vez que cada uno de ellos quede diseñado y programado, se interconectarán para que trabajen simultáneamente.

Las fases del proyecto son las siguientes:

1. Familiarización con la arquitectura de los dispositivos *SoC Zynq* y las características de la placa de desarrollo *ZedBoard*.
2. Diseño de módulo hardware para la adquisición de imágenes del sensor OV7670.
3. Diseño de módulo hardware para la visualización de las imágenes en un monitor
4. Diseño de un controlador de bus *SCCB* para la gestión de la configuración del sensor.
5. Desarrollo de un prototipo de la plataforma de procesamiento de video diseñada.

Una vez completada la plataforma, se testeará su funcionamiento en la FPGA, y se realizarán distintas pruebas para conocer las posibilidades que ofrece el sensor de imagen al trabajar con distintas configuraciones.

Durante todo el proyecto se utilizarán varias herramientas que ofrece *Xilinx* para el desarrollo de aplicaciones en sus dispositivos, como el simulador *ISim* para testear el funcionamiento de un módulo antes de incorporarlo a la plataforma final, la herramienta *IP ChipScope™* para monitorizar señales internas sin la necesidad de un osciloscopio o la utilidad *Xilinx CORE Generator™ System* para el diseño de componentes predefinidos por Xilinx configurables por el usuario.

1.2 VHDL

El lenguaje VHDL (*VHSIC – Hardware Description Language*) es un lenguaje de descripción hardware concebido con el fin de diseñar una herramienta estándar e independiente para el modelado, documentación y simulación de los sistemas electrónicos digitales, placas de circuito y componentes, durante todas las fases de su diseño. La primera revisión, desarrollada por *IBM* y *Texas Instruments* vio la luz en 1985. Posteriormente, conscientes de su enorme potencial, la *IEEE Computer Society* decidió adoptarlo y estandarizarlo. Dos versiones se han desarrollado desde entonces por este organismo, bajo las referencias 1076-87 y 1076-93, ésta última incorporando algunas mejoras con respecto a la versión anterior.

VHDL mejoraba los lenguajes de descripción hardware sencillos, tipo *netlist*, que se utilizaban ya en las herramientas de simulación de los entornos CAD para diseño electrónico. Estos lenguajes es que eran capaces de describir un sistema únicamente de forma estructural, y no de forma comportamental, lo que los limitaba a lo hora de diseñar proyectos de cierta envergadura. En cambio, VHDL posee una amplia sintaxis que permite el modelado de circuitos a partir de diferentes niveles de abstracción: descripción estructural o de interconexionado de elementos; descripción arquitectural, a nivel de transferencia de registros; y descripción funcional o comportamental, que describe el comportamiento esperado del circuito.

Basándonos en esta dualidad estructural/comportamental que nos ofrece VHDL desarrollaremos todo el proyecto, combinando las ventajas de cada una para abordar cada parte. De esta forma, concebimos el proyecto como una serie de módulos interconectados, apoyándonos en la descripción estructural, dónde cada uno de ellos se implementa independientemente, aprovechando la descripción comportamental.

1.3 La plataforma *ZedBoard*

En el mercado hay un amplio rango de ofertas de FPGAs, como *Artix*, *Virtex*, *Spartan* o *Zynq*, entre otras. Sin embargo, *ZedBoard* ha sido la elegida para este proyecto, concretamente el modelo que monta un *SoC Zynq XC7X020-1CLG484,* fabricada por la empresa americana *Xilinx*. Las razones para escoger esta *ZedBoard* en lugar de cualquiera de las demás son: los diferentes tipos de memoria que ofrece, el procesador *ARM Cortex-A9* de doble núcleo que incorpora, la interfaz para conectar una amplia variedad de periféricos y su arquitectura *SoC*.

1.3.1 *Tipos de memoria*

El modelo de *ZedBoard* utilizado incluye distintos tipos de memorias, como los 512MB de memoria RAM DDR3, 256MB de memoria flash, 560 KB de Block RAM y un slot para tarjeta SD. Esto da la flexibilidad de almacenar sistemas de reducido tamaño en la memoria flash, con las ventajas que ofrece este tipo de memoria, al mismo que permite almacenar sistemas más grandes en la memoria SD, por ejemplo un sistema operativo como Linux.

1.3.2 *Microprocesador ARM Cortex-A9 de doble núcleo*

Los microprocesadores ARM cuentan con una arquitectura RISC (*Reduced Instruction Set Computer*) de 32 bits desarrollada por la empresa *ARM Holdings*. Su simplicidad los hace ideales para aplicaciones de baja potencia, lo que ha hecho que estén presentes es la mayoría de dispositivos inteligentes de hoy en día, como smartphones, tablets, videoconsolas o smart TVs, entre otros. Por tanto, familiarizarse con este tipo de microprocesadores es muy recomendable por el gran mercado que tienen. Además, el ARM Cortex-A9 que incluye esta *ZedBoard* cuenta con dos núcleos capaces de llegar a una frecuencia de 1 GHz, por lo que no habrá en ningún momento falta de potencia de procesado.

1.3.3 *Interfaces*

Otro aspecto muy importante es el amplio rango de dispositivos que se pueden conectar a la *ZedBoard*, tales como monitores, a través de las salidas VGA y HDMI; dispositivos USB, por el puerto USB OTG 2.0; conexión Ethernet de hasta 1Gbps; o micrófonos y altavoces, por las conexiones mini-Jack de 3.5 mm.

Además, aparte de estas interfaces ampliamente conocidas, cuenta con una serie de conectores PMOD y FMC que hacen de la *ZedBoard* una FPGA muy versátil, pudiendo conectar multitud de dispositivos que no son compatibles con las interfaces típicas. Un ejemplo claro

de esto es el sensor que empleamos en este proyecto, que conectaremos a través de la interfaz FMC con una placa extensora, como veremos más adelante.

1.3.4 *Arquitectura SoC*

La *ZedBoard* es una placa de desarrollo basada en el *Xilinx Zynq-7000 All Programmable SoC (System-on-a-chip)*, lo que hace que puede ser empleada para multitud de aplicaciones. Por tanto, no es sólo una FPGA, es también un *SoC* totalmente programable, ya que combina el sistema de procesado (*PS*), que incluye el microprocesador ARM, con la parte de lógica programable (*PL*), que incluye a todos los periféricos, como vemos en la Figura 1. Esto hace que pueda trabajar a una velocidad más alta y con unas dimensiones más reducidas que una FPGA tradicional.

En concreto, la *ZedBoard* combina el microprocesador *Cortex-A9* con 85.000 celdas de lógica programable de la serie 7 de *Xilinx*.

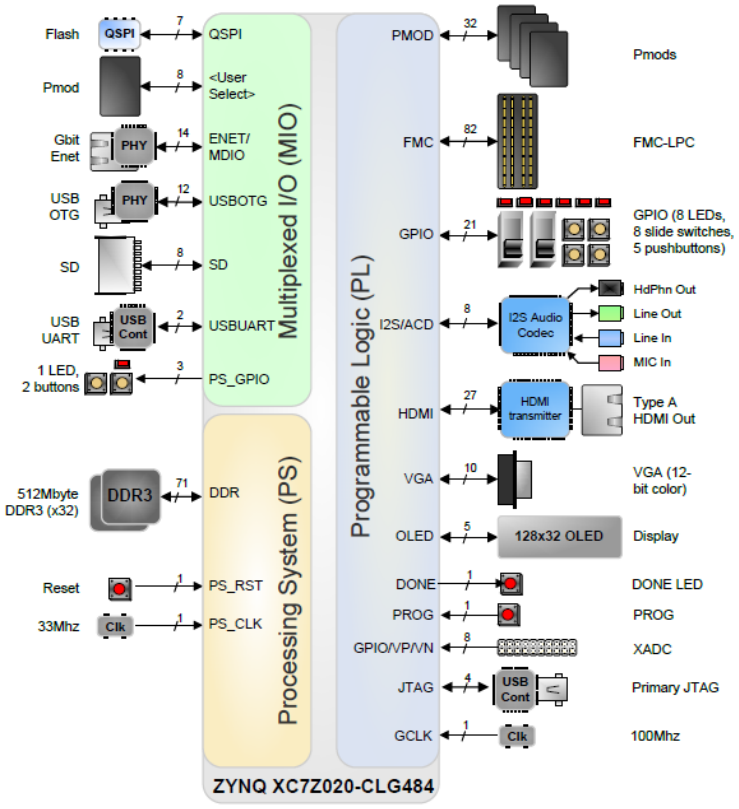


Figura 1. Diagrama de bloques de la *ZedBoard*

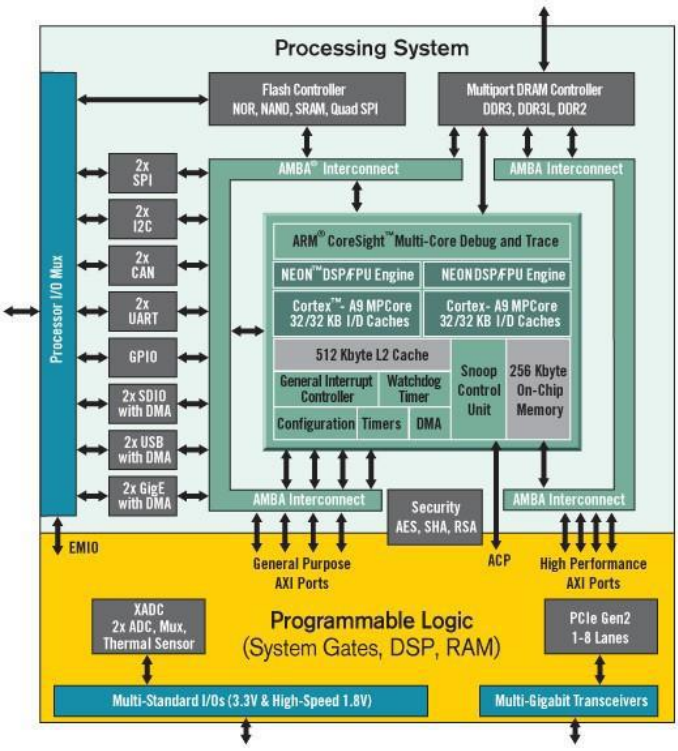


Figura 2. Diagrama completo del *SoC Zynq-7000*

1.4 Software de Xilinx

En este capítulo haremos un repaso de las herramientas de *Xilinx* que utilizaremos para trabajar con la *ZedBoard* y desarrollar el proyecto.

1.4.1 *ISE® Design Suite*

La herramienta *ISE® Design Suite* es el entorno de diseño que ofrece Xilinx al programador para trabajar con cualquier dispositivo de la familia Zynq-7000, permitiendo el acceso a todas las funciones y herramientas adicionales para programar estas FPGAs. Durante todo este proyecto emplearemos la versión 14.4, que permite trabajar con los *SoC* Z-7010, Z-7020, y Z-7030.

1.4.2 *Xilinx CORE Generator™ System (LogiCore)*

Xilinx CORE Generator™ System es una funcionalidad incluida en el pack *ISE® Design Suite* facilita en gran medida el diseño de componentes usados con frecuencia pero que requieren mucho tiempo para su programación. Para ello *Xilinx* ofrece componentes predefinidos, denominados *Intellectual Properties (IP),* para FPGAs configurables por el usuario. Estos *IPs* abarcan un rango muy amplio de aplicaciones, siendo los más comunes bloques de memoria, FIFOs o filtros y adaptadores. Usar estos *IPs* puede ahorrar días o incluso meses de tiempo de diseño, haciendo que un determinado proyecto pueda salir antes al mercado.

1.4.3 *ISE® Simulator (ISim)*

ISim es un completo simulador HDL integrado en el pack *ISE*. La simulación HDL puede ser un paso fundamental en el diseño de cualquier proyecto, pudiendo testear el funcionamiento de cualquier módulo programado antes de aplicarlo sobre la plataforma real.

Por este motivo, *ISim* se convierte en una herramienta básica la trabajar con la programación de FPGAs.

1.4.4 *IP ChipScope™*

El analizador lógico integrado, *Integrated Logic Analyzer (ILA)* en inglés, que emplea la herramienta *IP ChipScope*™ es un analizador configurable que puede ser usado para monitorizar cualquier señal interna del diseño. El núcleo del *ILA* incluye funcionalidades avanzadas de lógica moderna, pudiendo habilitar distintas señales de disparo e incluyendo la opción de añadir expresiones booleanas como disparo. También permite configurar la cantidad de muestras adquiridas en cada captura. Debido a que el núcleo del *ILA* es síncrono con el diseño que está siendo monitorizado, todas las restricciones de reloj que sean aplicadas al diseño son también aplicadas a los componentes dentro del *ILA*.

Capítulo 2: Sensor de imagen OV7670

2.1 Introducción

OmniVision Technologies es una empresa de desarrollo de dispositivos de procesado de imagen conocida a nivel mundial, con presencia en multitud de plataformas, como smartphones, ordenadores portátiles, seguridad, video cámaras, automóviles y sistemas de monitorización médicos.

Debido a la importante presencia de esta marca en el mercado tecnológico, resulta muy interesante familiarizarse con el uso de algunos de sus dispositivos, como es el sensor de imagen OV7670. Este sensor de imagen de bajo voltaje ofrece un amplio abanico de posibilidades debido a su versatilidad y posibilidades de configuración.

El OV7670 es un "*system on a chip* (*SoC*)" basado en tecnología CMOS con una capacidad de procesado de señal que ofrece distintas funciones de control de imagen, tales como control de exposición automática (AEC), control del balance de blancos (AWB), cancelación de ruido, además del control automático o manual de brillo, saturación, gamma y demás parámetros de imagen. Además cuenta con la posibilidad de trabajar a distintas resoluciones (VGA, QVGA, CIF, QCIF) y tasa de imagen, hasta un máximo de 640 x 480 píxeles a 30 imágenes por segundo, y diferentes espacios de color (RGB555, RGB565 y YUV). Todas estas funcionalidades son controladas mediante una serie de registros, a los cuales se accede mediante la interfaz SCCB, estandarizada para dispositivos de *OmniVision*.

A continuación se muestra un esquema de los distintos bloques que componen el sistema de procesado de señal junto con la función de los más importantes:

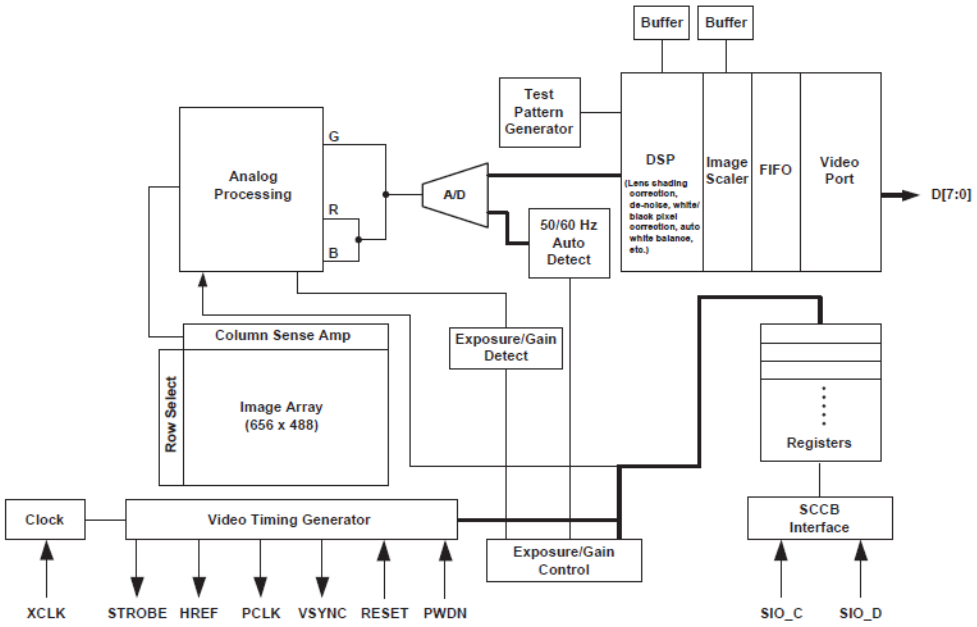


Figura 3. Esquema de bloques funcional del sensor OV7670

- ***Image Sensor Array:*** sensor de 656 x 488 píxeles para un total de 320.128 píxeles, de los cuales 640 x 480 son activos (307.200).

- ***Timing generator:*** en general, controla las siguientes funciones:
  - Control del sensor de imagen y de la generación de fotogramas
  - Generación y distribución de las señales temporales internas

  - Generación de las salidas de sincronía (VSYNC, HREF y PCLK)
  - Control de la tasa de fotogramas por segundo
  - Incluye AEC (*Automatic Exposure Control*)

- ***Analog Signal Processor:*** Este bloque se encarga de todas las funcionalidades analógicas de imagen, incluyendo AGC (*Automatic Gain Control*) y AWB (*Automatic White Balance*).

- ***A/D Converter:*** Después del bloque de procesamiento analógico, la señal analógica original alimenta a un convertidor analógico digital de 10 bits. Este convertidor trabaja hasta un máximo de 12 MHz.

- ***Test Pattern Generator:*** Genera patrones de color y de escala de grises de 8 bits.

- ***Digital Signal Processor (DSP):*** Este bloque controla la interpolación de la señal original a RGB e incorpora un control de calidad de imagen con las siguientes funcionalidades:
  - Realce de bordes
  - Convertidor de espacio de color (RGB o YUV)
  - Control de saturación
  - Corrección de píxeles y de ruido
  - Control automático de gamma+
  - Convertidor de 10 a 8 bits

- ***Image Scaler:*** Controla todos los datos de salida y los adapta al formato requerido antes de enviar la imagen. Puede escalar la imagen de VGA a CIF y casi a cualquier tamaño por debajo de CIF.

- ***SCCB Interface:*** Esta interfaz controla toda la configuración del sensor empleando el bus *SCCB* de *OmniVision*.

2.2 Esquema del sensor

En este proyecto trabajaremos con el sensor OV7670 instalado sobre un circuito impreso con unas dimensiones de 34x34 mm y fabricada en fibra de vidrio (FR4).

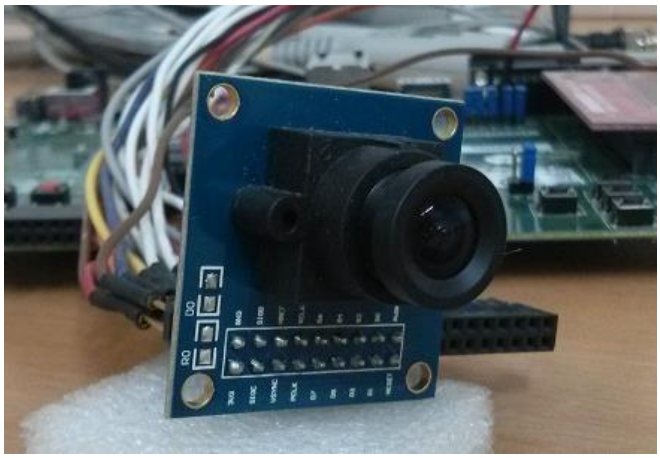


Figura 4. Sensor OV7670

El esquemático de la placa se muestra en la siguiente figura:

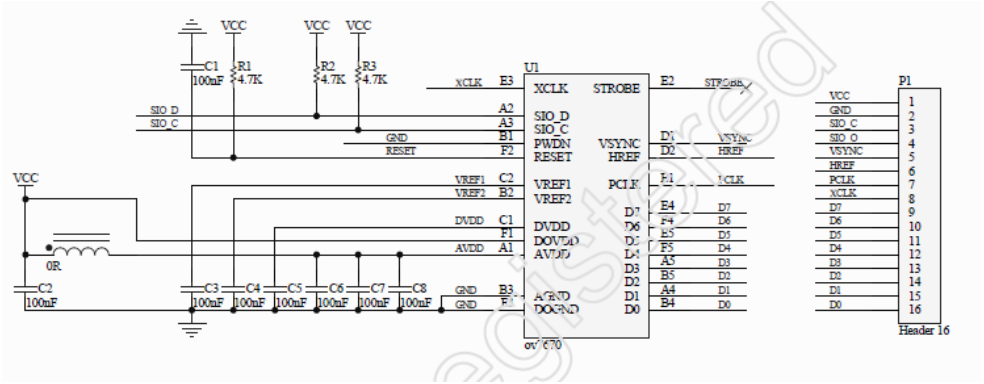


Figura 5. Esquemático sensor OV7670

De acuerdo con el *datasheet* del OV7670, de forma resumida, la función de cada uno de los pines es la siguiente:

Nombre	Tipo	Función
SIOD	Entrada/Salida	Datos de la interfaz SCCB
SIOC	Entrada	Reloj de la interfaz SCCB
RESET	Entrada	Resetea el sensor cuando se pone a nivel bajo
PWDN	Entrada	Apaga el sensor cuando se pone a nivel alto
HREF	Salida	Señal de referencia horizontal
VSYNC	Salida	Señal de referencia vertical
XCLK	Entrada	Reloj del sistema
PCLK	Salida	Reloj para sincronización de píxel
DATA0	Salida	Bit 0 de la salida de vídeo digital
DATA1	Salida	Bit 1 de la salida de vídeo digital
DATA2	Salida	Bit 2 de la salida de vídeo digital
DATA3	Salida	Bit 3 de la salida de vídeo digital
DATA4	Salida	Bit 4 de la salida de vídeo digital
DATA5	Salida	Bit 5 de la salida de vídeo digital
DATA6	Salida	Bit 6 de la salida de vídeo digital
DATA7	Salida	Bit 7 de la salida de vídeo digital
VCC	Alimentación	Alimentación del sensor a 3.3V
GND	Tierra	Tierra

Tabla 1. Pines del sensor OV7670

2.3 Señales de sincronía y formatos de color

Para implementar el módulo de captura de imagen, hay que estudiar cómo transmite el sensor las secuencias de bits que componen la imagen con cada espacio de color y las señales de sincronía que emplea.

2.3.1 *Señales de sincronía*

Antes de centrarnos en cada espacio de color, conviene hacer un breve repaso de las distintas señales de sincronización de las que dispone el sensor. Éste transmite un flujo de bits controlado por las señales HREF (referencia horizontal), VSYNC (sincronía vertical) y PCLK (reloj de píxel):

- **HREF**: señal de referencia horizontal. Esta señal indica cuando acaba una línea de píxeles poniéndose a nivel bajo, de forma que el sensor está enviando datos válidos siempre que esta señal se encuentre a nivel alto.

- **VSYNC**: señal de sincronía vertical. Indica cuando termina un fotograma y comienza el siguiente. Cambia a nivel alto cuando la imagen ha terminado de transmitirse y vuelva a nivel bajo cuando comienza el siguiente.

- **PCLK**: reloj de píxel. Este reloj indica cuando leer el dato de los pines de salida. El dato (8 bits) está listo para leerse en cada flanco ascendente.

En las siguientes figuras, proporcionadas por el fabricante, vemos unos diagramas que describen el funcionamiento de las tres señales de sincronía descritas:

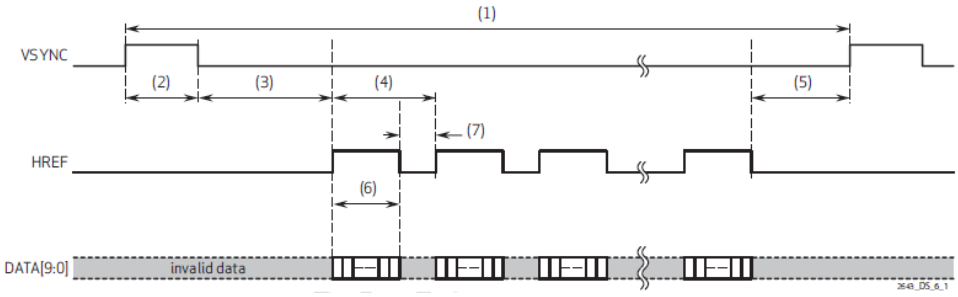


Figura 6. Diagrama PCLK y HREF

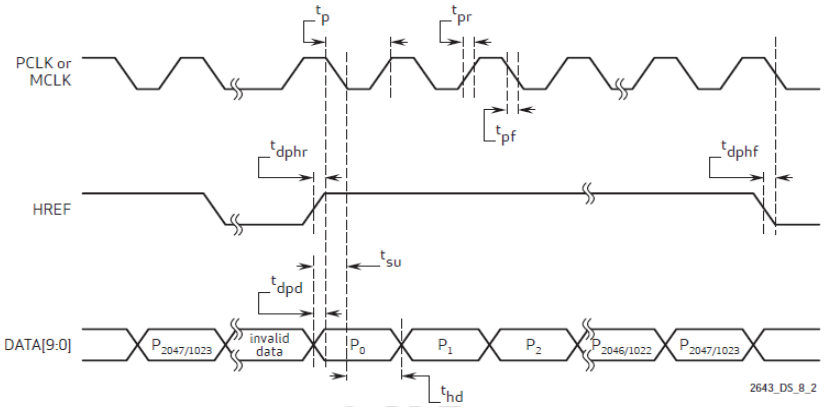


Figura 7. Diagrama PCLK y HREF

2.3.2 *RGB555*

Uno de los espacios de color de los que dispone el sensor es RGB en su versión 555, en el que la información de color de cada píxel queda almacenada en quince bits, cinco para el color rojo, cinco para el verde y cinco para el azul.

De acuerdo con el *datasheet* del sensor, cada píxel se transmite en dos ciclos de reloj siguiendo el siguiente orden:

- Primer ciclo (primer *byte*):

DATA7	DATA6	DATA5	DATA4	DATA3	DATA2	DATA1	DATA0
X	R_4	R_3	R_2	R_1	R_0	G_4	G_3

Tabla 2. Secuencia de transmisión RGB555 (primer ciclo)

- Segundo ciclo (segundo *byte*):

DATA7	DATA6	DATA5	DATA4	DATA3	DATA2	DATA1	DATA0
G_2	G_1	G_0	B_4	B_3	B_2	B_1	B_0

Tabla 3. Secuencia de transmisión RGB555 (segundo ciclo)

Los cinco bits correspondientes al rojo y los dos primeros del verde se transmiten en el primer ciclo de PCLK, por tanto deberemos almacenar estos siete bits hasta el siguiente ciclo, cuando se reciben los cinco del azul y los tres restantes del verde (el bit más significativo del primer ciclo se descarta, es un bit de relleno). Es ahora cuando el píxel se ha recibido completamente y puede ser procesado. La frecuencia a la que vamos procesando los píxeles es la mitad de frecuencia que la del PCLK.

Al procesar el píxel debemos tener en cuenta que la salida VGA trabaja con el formato RGB444, por tanto, nos quedamos únicamente con los cuatro bits más significativos de cada componente, pasando de 15 a 12 bits por píxel. Esto supone bajar la calidad de color que nos proporciona el sensor, pero al eliminar sólo tres bits tampoco se resiente en exceso.

2.3.3 *RGB565*

El espacio de color RGB565 funciona de la misma forma que RGB555, con la única particularidad de que la componente verde tiene más peso sobre la roja y la azul, concretamente un bit más, ocupando dieciséis bits cada píxel. La secuencia de bits que emplea el sensor es la siguiente:

- Primer ciclo (primer *byte*):

DATA7	DATA6	DATA5	DATA4	DATA3	DATA2	DATA1	DATA0
R_4	R_3	R_2	R_1	R_0	G_5	G_4	G_3

Tabla 4. Secuencia de transmisión RGB565 (segundo ciclo)

- Segundo ciclo (segundo *byte*):

DATA7	DATA6	DATA5	DATA4	DATA3	DATA2	DATA1	DATA0
G_2	G_1	G_0	B_4	B_3	B_2	B_1	B_0

Tabla 5. Secuencia de transmisión RGB565 (segundo ciclo)

Esta vez no hay bit de relleno en la secuencia, siendo todos bits de datos. Como vemos, los tres bits más significativos de la componente verde se transmiten en el primer ciclo y los tres restantes en el segundo. Al igual que con RGB555 tendremos que ajustar el formato a RGB444, eliminando esta vez los dos bits menos significativos de la componente verde.

2.3.4 *YUV422*

El tercer espacio de color que emplearemos es YUV422, empleando tres canales diferentes: luminancia (Y), crominancia U y crominancia V. Esta vez se necesitan cuatro ciclos de PCLK para transmitir dos píxeles, manteniendo la misma tasa de píxeles que con RGB pero con distinto formato de transmisión. La secuencia que sigue el sensor para transmitir los distintos canales luminancia y crominancia es configurable (registro TSLB), tenemos las opciones YUYV, YVYU, UYVY y VYUY. Optamos por la opción YUYV, que sigue el siguiente orden:

	Primer ciclo	Segundo ciclo	Tercer ciclo	Cuarto ciclo	Quinto ciclo	Sexto ciclo
DATA[7:0]	Y[7:0]	U[7:0]	Y[7:0]	V[7:0]	Y[7:0]	U[7:0]

Tabla 6. Secuencia de transmisión YUV422

De acuerdo con la tabla 5, en cada ciclo se reciben los ocho bits de correspondientes a la información de un canal, actualizándose el canal Y cada dos ciclos, de forma que cada píxel tiene un valor de luminancia diferente. En cambio, los canales U y V se actualizan cada cuatro ciclos, por tanto, dos píxeles consecutivos comparten valores de crominancia. Este es el principio en que se basa el formato YUV, que el canal de luminancia varía más rápidamente que el de crominancia, pudiendo éste ser transmitido con la mitad de ancho de banda.

Para que la idea quede más clara, en la tabla 8 se muestra una serie de píxeles con los valores de luminancia y crominancia que toma cada uno.

	Luminancia (Y)	Crominancia U	Crominancia V
Píxel 1	Y_1	U_{01}^*	V_{01}^*
Píxel 2	Y_2	U_{23}	V_{01}
Píxel 3	Y_3	U_{23}	V_{23}
Píxel 4	Y_4	U_{45}	V_{23}
Píxel 5	Y_5	U_{45}	V_{45}

* Se supone que antes se han transmitido otros píxeles

Tabla 7. Ejemplo de transmisión de píxeles en YUV422

Vemos que cada píxel tiene un valor de luminancia único, mientras que los valores de crominancia se comparten entre dos píxeles, de ahí los subíndices con dos dígitos. Como el valor de crominancia U es el que primero se actualiza con este formato (*YUYV*), el primer píxel de cada par tendrá el valor de crominancia V del par anterior. Esto no supone una limitación significativa en la calidad de imagen, ya que, como hemos comentado, los valores de crominancia varían más lentamente y pares consecutivos de píxeles pueden compartirlos.

2.4 Conexión del sensor OV7670 a la *ZedBoard*

La conexión del sensor a la *ZedBoard* se hace a la parte de PL, por lo que se puede hacer a los conectores PMOD y al conector FMC. Por comodidad, optamos por utilizar el conector

FMC, ya que las conexiones de los pines son más fáciles de hacer y la que presentan es mayor, evitando desconexiones indeseadas.

Para sacar pines del conector FMC a los que poder conectar el sensor, empleamos una placa extensora FMC, concretamente el modelo *TB-FMCL-PH* de la marca *Inrevium*. Esta placa ofrece tres bancos de pines, donde se incluyen pines de conexión a PL junto con pines de alimentación y tierra.

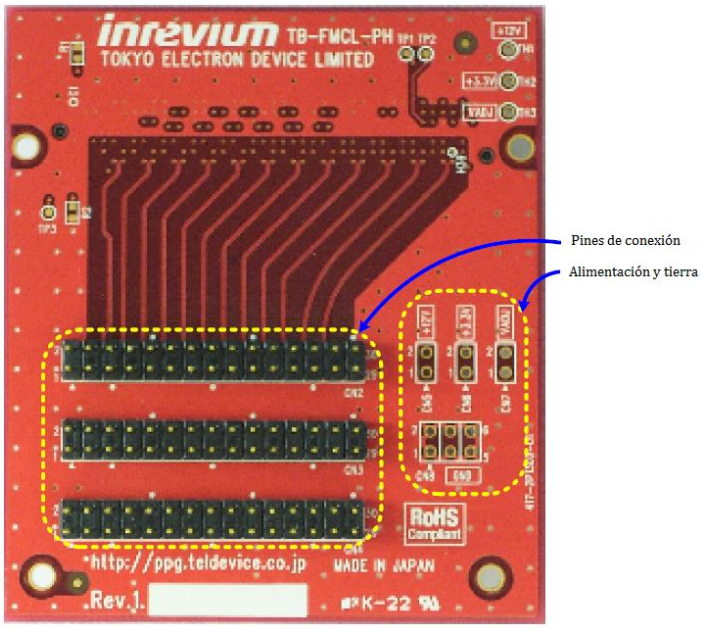


Figura 8. Placa extensora FMC

Para la conexión del sensor es suficiente con un banco, por lo que optamos por el primer banco de pines, denominado CN2. Los dos primeros pines del banco son pines conectados a tierra y los dos siguientes están reservados para señales de reloj, por lo que se conecta el sensor a partir del pin 5, como vemos en la Figura 9.

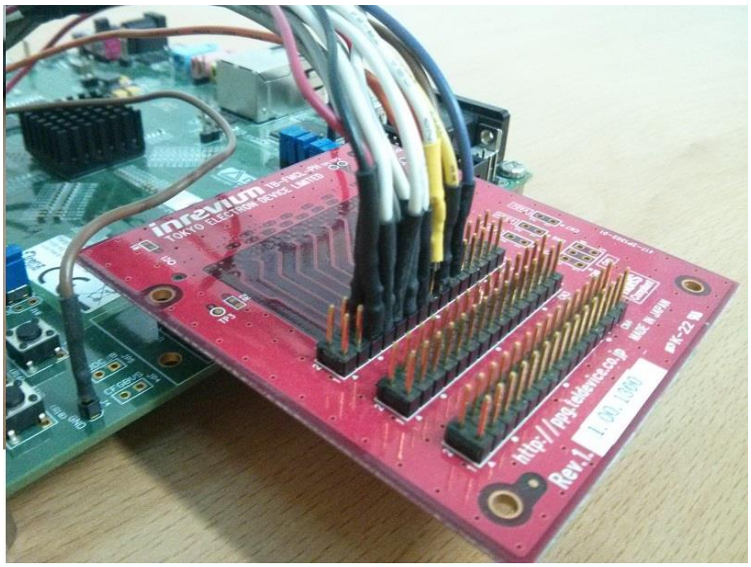


Figura 9. Sensor OV7670

Las conexiones entre los pines de la placa y los pines del sensor se muestran en la siguiente tabla:

Pin sensor	Pin placa	Pin sensor	Pin placa
PWDN	5	**DATA<6>**	13
RESET	6	**DATA<7>**	14
DATA<0>	7	**XCLK**	15
DATA<1>	8	**PCLK**	16
DATA<2>	9	**HREF**	17
DATA<3>	10	**VSYNC**	18
DATA<4>	11	**SIOD**	19
DATA<5>	12	**SIOC**	20

Tabla 8. Conexiones de los pines del sensor a la placa FMC

Capítulo 3: Diseño hardware del módulo de adquisición de imagen

3.1 Introducción

Como hemos visto en el capítulo anterior, el sensor ofrece distintas configuraciones en cuanto a espacios de color se refiere. Para entender bien este concepto conviene hacer una pequeña explicación antes de tratar con él.

Un espacio de color es un modelo matemático con el que se representan los distintos colores como secuencias numéricas, lo que nos permite hacer un tratamiento digital de los colores que componen una imagen.

En función del modelo empleado, los colores se representan de una forma u otra. Por ejemplo, en RGB se tienen tres componentes de color: la roja (R), la verde (G) y la azul (B), que son los tres colores de luz primarios. A partir de estas tres componentes es posible representar un color mediante la mezcla por adición de cada una. Dependiendo del valor que se asigne a cada una de ellas obtendremos distintos colores, donde un valor más alto de una componente hace que ésta tenga mayor intensidad.

Por tanto, como vemos en la Figura 10, podemos abarcar desde el color negro, que se representa con todas las componentes a cero, hasta el blanco, con todas las componentes con el valor máximo.

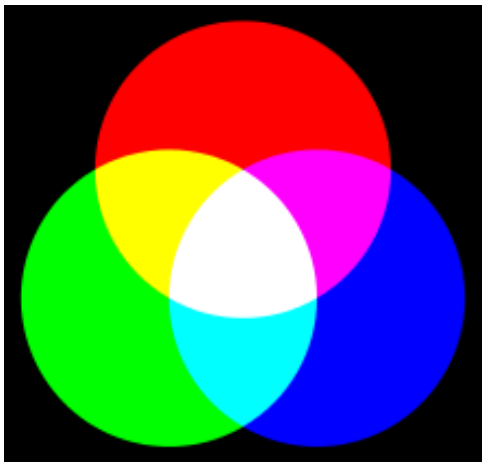


Figura 10. Modelo aditivo de RGB

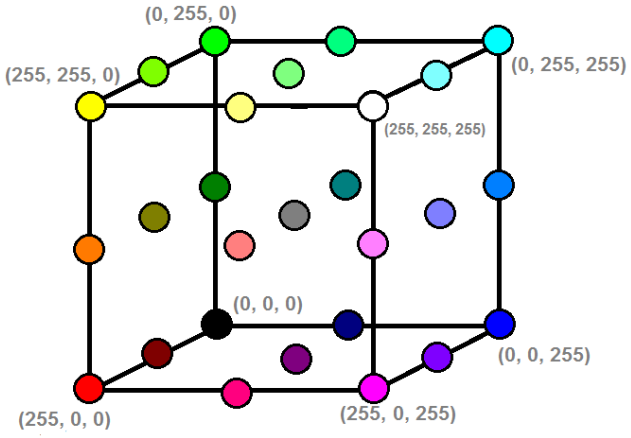


Figura 11. Cubo RGB888

El número de bits que se le asigna a cada componente también puede variar, obteniéndose una mayor paleta de colores cuanto mayor es el número de bits. En nuestro caso, nos centraremos en RGB555 (5 bits para cada componente) y RGB565 (5 bits para el rojo y el azul y 6 bits para el verde), ya que son los dos formatos RGB con los que trabaja el sensor.

El otro espacio de color que nos atañe es YUV, un espacio basado en la forma que una persona percibe una imagen, haciendo que se necesite un ancho de banda menor para las componentes de diferencia de color, denominadas crominancia U y V. Esto hace que los errores de transmisión o las imperfecciones en la captación de la imagen se oculten mejor que en RGB.

Este espacio se compone de tres canales: luminancia (Y), que transporta la señal monocromática (en blanco y negro); crominancia U, que lleva la diferencia de azul respecto a

la luminancia, y crominancia V, con la diferencia de rojo. La componente de luminancia es la que más definición necesita, ya que el ojo humano es más sensible a las imágenes monocromáticas que a las de color, haciendo que las señales de crominancia no necesiten tanto ancho de banda y puedan ser comprimidas sin que la calidad de imagen se resienta en exceso.

El formato empleado por el sensor el YUV422, lo que indica que se tiene el doble de información de luminancia que de cada crominancia. Cuando empleemos este formato habrá que hacer la transformación a RGB para que sea representado correctamente en pantalla. Hay muchas expresiones diferentes pasar a RGB, pero una transformación con buenos resultados es la siguiente:

```
R = 1.164(Y - 16) + 1.596(V - 128)

G = 1.164(Y - 16) - 0.813(V - 128) - 0.391(U - 128)

B = 1.164(Y - 16) + 2.018(U - 128)
```

Expresiones de conversión de YUV a RGB

3.2 Implementación con VHDL

Sabiendo cómo trabaja el sensor con cada formato de color y las distintas señales de sincronía que emplea, pasamos a implementar el módulo de captura, *ov7670_captura_pixel*, para cada uno de ellos.

3.2.1 *RGB555*

La señal PCLK es la que dicta cuando se debe leer cada *byte* de información, por tanto, éste va a ser el reloj que hace trabajar el módulo. Las señales HREF y VSYNC, como hemos visto en el apartado anterior, deben estar a nivel alto al mismo tiempo para que el sensor envíe píxeles válidos. Por tanto, debemos imponer las condiciones de que cada píxel se lea cuando se produzca un flanco ascendente de PCLK y HREF y VSYNC estén a '1'.

Un aspecto importante a tener en cuenta es que la frecuencia a la que la cámara envía los píxeles es diferente a la que emplea VGA para mostrarlos en pantalla. Por tanto, se debe emplear un espacio de memoria (*buffer*) para almacenar los píxeles recibidos y sincronizar el proceso. De este *buffer* hablaremos detalladamente en el siguiente apartado, lo único a tener en cuenta ahora es que cuenta con una entrada que indica la dirección de memoria en la que se escribe el dato y otra que indica cuando se escribe, por lo que implementamos en este módulo una salida, *addr,* que actualice esa posición de memoria con cada píxel recibido y no haya sobreescritura, y otra, *we* (*write enable*)*,* que se active cuando se haya recibido el píxel completo e indique que se puede escribir en memoria. En el *buffer* se almacena una imagen completa, por lo que hay que reiniciar la posición de memoria cada vez que se recibe un fotograma, es decir, cuando VSYNC cambia a nivel alto.

La siguiente parte se encarga de leer correctamente el píxel completo teniendo en cuenta la secuencia para el formato de color RGB555. Para controlar los ciclos de escritura, habilitamos la señal *ciclo,* que cambia de valor en cada ciclo y se reinicia al recibir el píxel completamente. En este caso, como cada píxel se recibe en dos ciclos, sólo será necesario que tenga un bit.

Por otro lado, instanciamos tres señales, *rojo*, *verde* y *azul,* una para cada componente de color, para controlar la información de color que se recibe en cada ciclo y almacenarla en el orden correcto. Como veíamos anteriormente, en RGB555 el sensor envía en el primer ciclo

un bit de relleno, seguido de los cinco bits de rojo y los dos bits más significativos de verde. En el segundo ciclo se reciben los tres bits restantes de verde y los cinco de azul. Una vez recibidos los quince bits del píxel, se activa la salida *we* para escribir el dato en memoria y se aumenta en uno la dirección de memoria donde se escribe. La señal de entrada de datos, *data*, tiene una longitud de ocho bits y está conectada directamente a los pines de entrada DATA del sensor.

Los quince bits recibidos tienen que quedarse en doce para ser compatible con el formato RGB444 que emplea VGA para representar la imagen en el monitor. Por tanto, para no usar más memoria de la necesaria, sólo almacenamos doce bits para cada píxel, desechando el bit menos significativo de cada componente de color. De esta forma, el módulo queda definido como un componente con los siguientes puertos de entrada y salida:

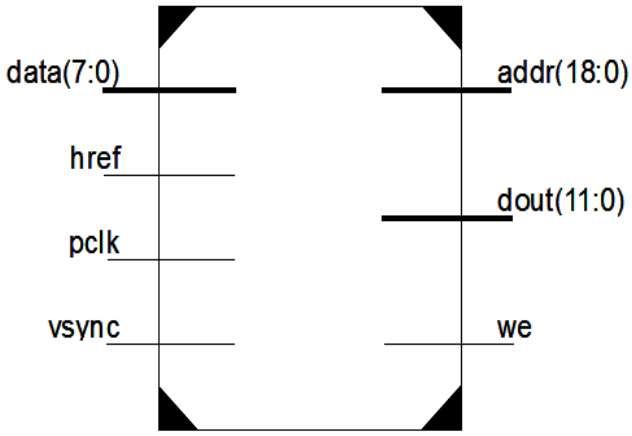


Figura 12. Módulo *ov7670_captura_pixel.* Puertos de entrada (izquierda) y salida (derecha)

La implementación del módulo completo queda así:

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

entity ov7670_captura_pixel is
    Port ( pclk  : in   std_logic;
           vsync : in   std_logic;
           href  : in   std_logic;
           data  : in   std_logic_vector (7 downto 0);
           addr  : out  std_logic_vector (18 downto 0);
           dout  : out  std_logic_vector (11 downto 0);
           we    : out  std_logic);
end ov7670_captura_pixel;

architecture Behavioral of ov7670_captura_pixel is

begin
   addr <= addr_aux;
   dout <= rojo(4 downto 1) & verde(4 downto 1) & azul(4 downto 1);

captura_pixel:process(pclk)
   begin
      if rising_edge(pclk) then
     if vsync = '1' then
            addr_aux <= (others => '0');
            we <= '0';
```

```
            ciclo <= '0';
         else
         if href='1' then
           if (ciclo = '0') then
             rojo <= data(6 downto 2);
             verde <= data(1 downto 0) & "000";
             we <= '0';
             ciclo <= '1';
           else
             verde <= verde(4 downto 3) & data(7 downto 5);
             azul <= data(4 downto 0);
             we <= '1';
             ciclo <= '0';
             addr_aux <= std_logic_vector(unsigned(addr_aux)+1);
           end if;
         else
           ciclo <= '0';
           we <= '0';
         end if;
          end if;
        end if;
   end process;
end Behavioral;
```

* La declaración de señales auxiliares se ha omitido en todo el código mostrado.

Código 1. Módulo completo de captura de píxeles para RGB555

3.2.2 *RGB565*

El formato RGB565 funciona igual que RGB555 con la excepción de que esta vez recibimos en el primer ciclo cinco bits de rojo y los tres más significativos de verde, sin bit de relleno, y en el segundo los tres menos significativos de verde y los cinco de azul. La señal de verde, por tanto, tiene esta vez seis bits de longitud. El truncamiento para guardar sólo doce bits en memoria es igual que antes, desechando esta vez dos bits de la componente verde. Los puertos de entrada y salida son los mismos que con RGB555. La implementación es la siguiente:

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

entity ov7670_captura_pixel is
    Port ( pclk  : in   std_logic;
           vsync : in   std_logic;
           href  : in   std_logic;
           data  : in   std_logic_vector (7 downto 0);
           addr  : out  std_logic_vector (18 downto 0);
           dout  : out  std_logic_vector (11 downto 0);
           we    : out  std_logic);
end ov7670_captura_pixel;

architecture Behavioral of ov7670_captura_pixel is

begin
   addr <= addr_aux;
   dout <= rojo(4 downto 1) & verde(5 downto 2) & azul(4 downto 1);

captura_pixel:process(pclk)
   begin
```

```
        if rising_edge(pclk) then
       if vsync = '1' then
              addr_aux <= (others => '0');
              we <= '0';
         ciclo <= '0';
         else
         if href='1' then
           if (ciclo = '0') then
             rojo <= data(7 downto 3);
             verde <= data(2 downto 0) & "000";
             we <= '0';
             ciclo <= '1';
           else
             verde <= verde(5 downto 3) & data(7 downto 5);
             azul <= data(4 downto 0);
             we <= '1';
             ciclo <= '0';
             addr_aux <= std_logic_vector(unsigned(addr_aux)+1);
           end if;
         else
           ciclo <= '0';
           we <= '0';
         end if;
          end if;
        end if;
    end process;
end Behavioral;
```

Código 2. Módulo completo de captura de píxeles para RGB565

3.2.3 *YUV422*

Para el formato YUV422 el tratamiento de las señales de sincronía sigue siendo el mismo que en los anteriores pero la implementación de la lectura de píxeles sí cambia, de acuerdo a lo visto en el apartado 3.2.4, así como la transformación necesaria para pasar a RGB444.

Esta vez, la señal ciclo tiene dos píxeles para contar los cuatro ciclos de reloj que son necesarios para la lectura de la secuencia, así sea *YUYV*, *YVYU*, *UYVY* o *VYUY*. En este caso emplearemos *YUYV*, aunque para las demás el único cambio necesario sería variar el canal leído en cada ciclo, las escrituras en memoria serían las mismas.

Para saber en qué ciclo de reloj se encuentra la transferencia de datos y leer el canal correcto, empleamos un *case* controlado por la señal *ciclo,* que se incrementa en cada ciclo, es decir, cada vez que se lee un *byte*. La escritura en memoria la hacemos cada dos ciclos, de la misma forma que para RGB, actualizando los canales de luminancia y crominancia de la forma vista anteriormente.

```
          if href='1' then
            case ciclo is
              when "00" =>
                Y <= data;
                we <= '0';
                ciclo <= ciclo+1;

              when "01" =>
                U <= data;
                we <= '1';
                addr_aux <= addr_aux+1;
```

```
            ciclo <= ciclo+1;

          when "10" =>
            V <= data;
            we <= '0';
            ciclo <= ciclo+1;

          when others =>
            Y <= data;
            addr_aux <= addr_aux+1;
            we <= '1';
            ciclo <= (others => '0');
        end case;
```

Código 3. Adquisición de píxeles en YUV422

Para hacer la transformación a RGB, empleamos otro módulo aparte que trabaja tres con entradas de ocho bits, una para cada canal, y las transforma a una salida de doce bits preparada para ser escrita en el *buffer*. Este módulo tiene implementadas las expresiones de cambio de formato vistas en el apartado 3.1. Aquí el truncamiento de bits se hace en este módulo, ya que la transformación directa para entradas de ocho bits serían veinticuatro (RGB888), con ocho bits para cada componente.

Por tanto, las salidas del módulo *ov7670_captura_pixel* serán, además de las salidas *addr* y *we* que cumplen la misma función que en los anteriores, tres salidas de ocho píxeles que contienen cada uno de los canales YUV sin comprimir. El módulo completo queda así:

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

entity ov7670_captura_pixel is
    Port ( pclk  : in   std_logic;
           vsync : in   std_logic;
           href  : in   std_logic;
           data  : in   std_logic_vector (7 downto 0);
           addr  : out  std_logic_vector (18 downto 0);
           we    : out  std_logic;

         Y   : out std_logic_vector(7 downto 0);
         U   : out std_logic_vector(7 downto 0);
         V   : out std_logic_vector(7 downto 0));

end ov7670_captura_pixel;

architecture Behavioral of ov7670_captura_pixel is

begin
   addr <= addr_aux;

 captura_pixel:process(pclk)
   begin
      if rising_edge(pclk) then

         if vsync = '1' then
            addr_aux <= (others => '0');
       we <= '0';
       ciclo <= (others => '0');
         else
```

```
        -- Patron Y U Y V
        if href='1' then
           case ciclo is
              when "00" =>
                 Y <= data;
                 we <= '0';
                 ciclo <= ciclo+1;

              when "01" =>
                 U <= data;
                 we <= '1';
                 addr_aux <= addr_aux+1;
                 ciclo <= ciclo+1;

              when "10" =>
                 Y <= data;
                 we <= '0';
                 ciclo <= ciclo+1;

              when others =>
                 V <= data;
                 addr_aux <= addr_aux+1;
                 we <= '1';
                 ciclo <= (others => '0');
           end case;
        else
           ciclo <= (others => '0');
           we <= '0';
        end if;

          end if;
       end if;
    end process;
end Behavioral;
```

Código 4. Módulo completo de captura de píxeles para YUV422

3.3 Implementación del *buffer*

Como hemos comentado anteriormente, para sincronizar la frecuencia de adquisición de fotogramas, que es variable con la configuración del PLL interno del sensor, y la de muestra de éstos por la salida VGA es necesario emplear un *buffer* para guardar cada imagen adquirida y así poder trabajar a los 25 MHz necesarios en VGA para mostrar 30 imágenes por segundo.

La resolución con la que trabajamos es de 640x480 píxeles, por lo que para guardar un fotograma completo debemos disponer de 307.200 posiciones de memoria de doce bits cada una, ya que almacenamos los bits en formato RGB444, listo para ser mostrado por pantalla. Esto hace un total de 450 *kilobytes* de memoria necesarios para almacenar una imagen, cosa que no supone un problema para la FPGA con la que trabajamos, que cuenta con 560 KB de Block RAM, que es la memoria que emplearemos para este propósito.

Para implementar este *buffer* empleamos la herramienta *LogiCORE™ IP Block Memory Generator,* proporcionada por Xilinx. A continuación se detalla paso a paso cómo emplear esta herramienta para crear el *buffer* adaptado a nuestros requisitos.

En primer lugar, seleccionamos la opción *New Source* en ISE y, una vez que se abre la nueva ventana con los distintos tipos de archivos a crear, escogemos *IP (CORE Generator &*

Architecture Wizard) y le damos un nombre, en este caso *buffer_imagen,* como vemos en la Figura 13.

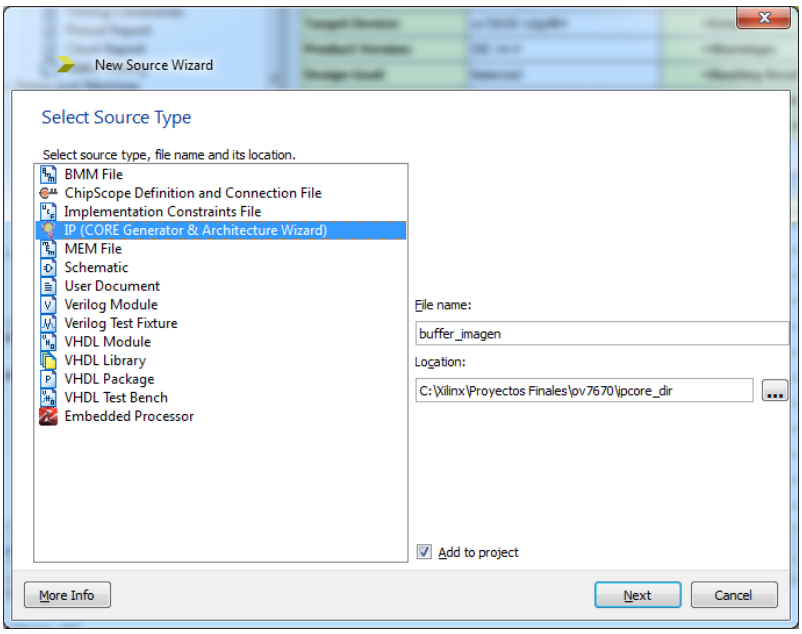


Figura 13. Seleccionar tipo de archivo

Seleccionamos la casilla *Add to project* para que lo añada directamente al proyecto y presionamos *Next*. En la siguiente ventana aparecen multitud de componentes predefinidos para crear, en nuestro caso escogemos la carpeta *Memories & Storage Elements – RAMs & ROMs* y ahí el elemento *Block Memory Generator*. Es recomendable seleccionar la casilla *Only IP compatible with chosen part* para no tener problemas de compatibilidad con la versión del componente, que en este caso es la 7.3.

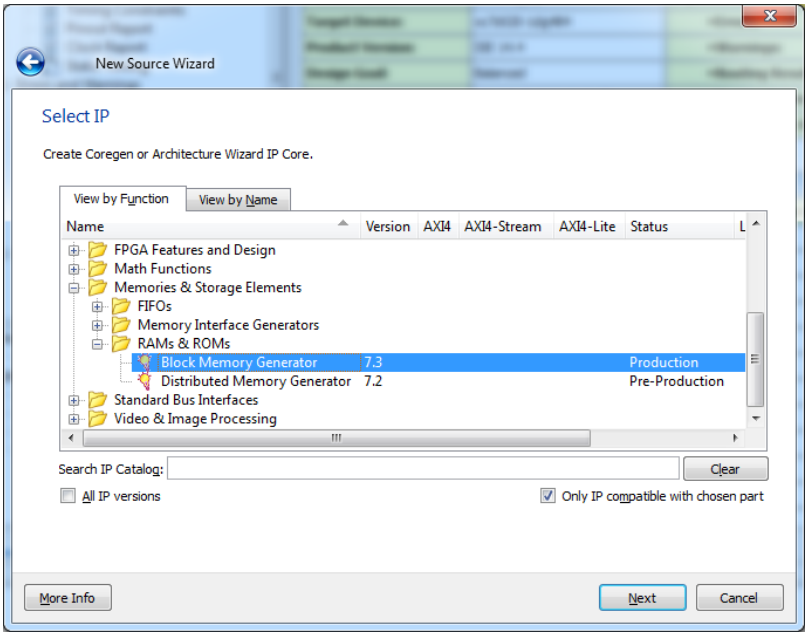


Figura 14. Seleccionar componente

Una vez que presionamos *Next* de nuevo, aparece otro cuadro de diálogo, presionamos *Finish* y arranca la herramienta *LogiCORE.* La primera opción que podemos configurar es el tipo de interfaz, seleccionamos la opción por defecto, *Native*. La opción AXI4 ofrece

funcionalidades que no vamos a utilizar en este proyecto, por lo que una interfaz más sencilla es la más adecuada.

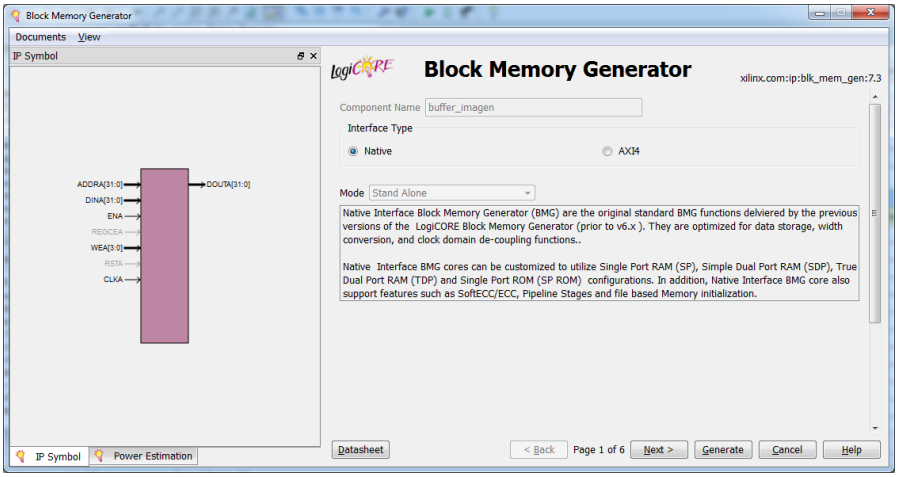


Figura 15. Tipo de interfaz

Presionamos *Next* y pasamos a la siguiente página, donde podemos seleccionar el tipo de memoria. En este caso necesitamos un bloque que admita lectura y escritura de forma independiente, a distintas frecuencias, por lo que escogemos la opción *Simple Dual Port RAM.* Desactivamos la opción *Enable 32-bit Address* porque no necesitamos reservar tantas direcciones de memoria, el *buffer* debe adaptarse a los requisitos, sin usar más memoria de la necesaria.

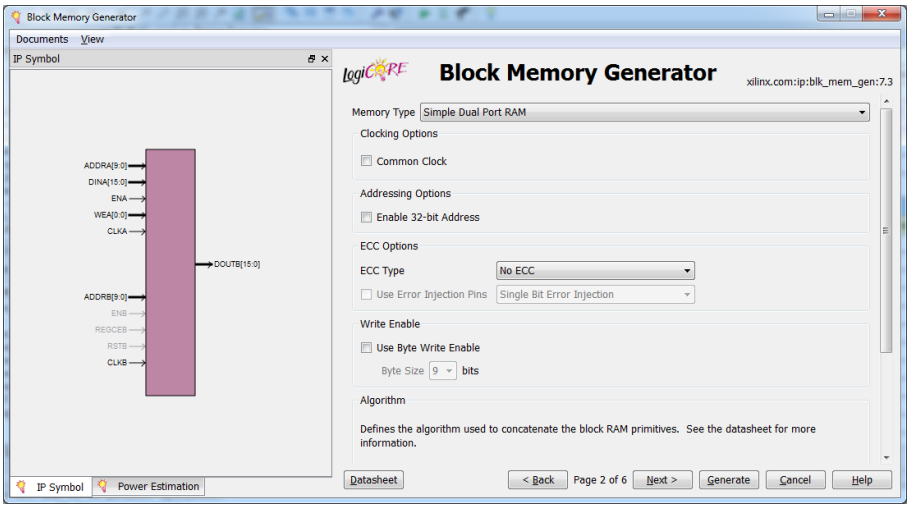


Figura 16. Tipo de memoria

Dejamos las demás opciones por defecto y pasamos a la siguiente ventana. Aquí configuramos el tamaño del bloque, pudiendo configurar la longitud de los datos guardados, *Write Width,* y las direcciones de memoria reservadas, *Write Depth*. Como hemos señalado anteriormente, necesitamos exactamente 307.200 posiciones de memoria con doce bits almacenados en cada una, por lo que introducimos estos dos valores. Las opciones de lectura van a ser idénticas, con doce bits de salida.

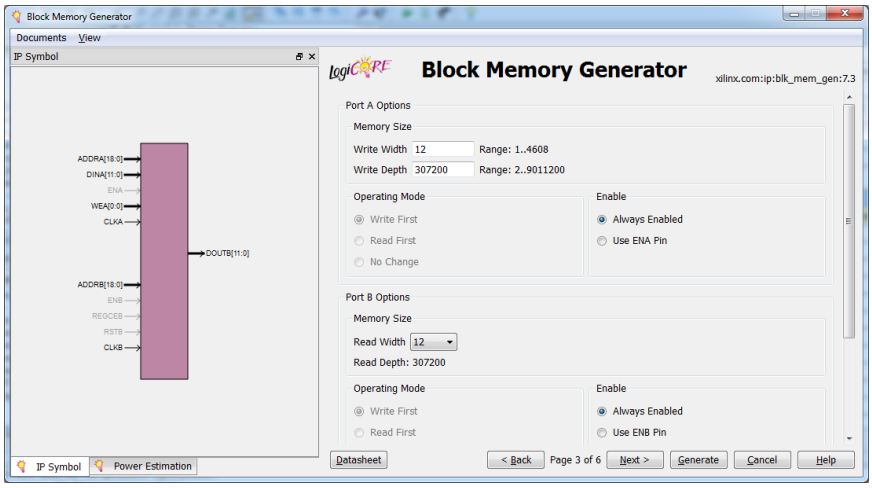


Figura 17. Configuración del tamaño del *buffer*

También nos ofrece la opción de habilitar un pin, *ENA,* para activar o desactivar los demás pines, pero en este no lo vamos a utilizar, así que seleccionamos la opción *Always Enable.* Lo mismo para *ENB*.

Las páginas 4 y 5 ofrecen funcionalidades que no vamos a utilizar, por lo que pasamos directamente a la última página.

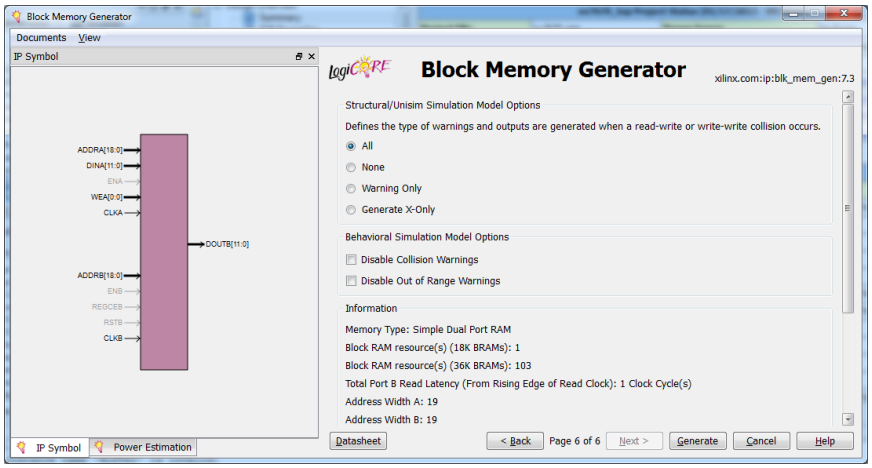


Figura 18. Información del *buffer* generado

Dejamos seleccionada la opción *All* para que lance un aviso cuando se produzca una colisión de lectura-escritura o escritura-escritura. Por último, nos muestra un resumen de los recursos empleados por el *buffer* generado. Se ha hecho uso de 103 bloques de 36Kb, de un máximo de 140, de la Block RAM, por lo que, como preveíamos, la placa cuenta con Block RAM para guardar una imagen completa. La dirección de memoria tiene una longitud de 19 bits para poder abarcar todo el rango necesario, factor que debemos tener en cuenta a la hora de instanciar el *buffer* en el proyecto. Presionamos *Generate* y el *buffer* que creado y listo para instanciar en el proyecto.

Capítulo 4: Diseño del módulo para la visualización de imagen en un monitor VGA

4.1 Introducción

En un monitor VGA podemos distinguir dos tipos de señales: señales de sincronización y señales de color.

Cada uno de los píxeles puede tener un color dentro de una paleta, definida por el número de bits empleados, a más bits mayor número de colores y más extensa es la gama, obteniendo más variedad de colores. El espacio de color que emplea VGA es RGB, por lo que la información de color llega al monitor a través de tres señales distintas, como veíamos anteriormente: una para el nivel de rojo (R), otra para el verde (G) y otra para el azul (B). En nuestro caso, la salida VGA trabaja con cuatro bits por componente, por lo que tendremos una paleta de 4096 colores.

4.2 Señales de sincronía

En cuanto a las señales de sincronía, son necesarias dos señales, una de sincronía horizontal y otra de sincronía vertical. La primera indica cuando se han generado todos los píxeles de una línea y debe pasarse al primer píxel de la línea siguiente, y la segunda para indicar cuando se han generado todas las líneas de un fotograma y debe pasarse al siguiente, mostrando el primer píxel de la primera línea de este nuevo fotograma. Las dos señales se activan a nivel bajo para indicar el cambio de línea y de fotograma, respectivamente.

El estándar VGA trabaja a una resolución de 640 x 480 píxeles. Las Figuras 19 y 20 muestran el diagrama temporal de acuerdo con esta resolución.

La sincronía horizontal marca el final de una línea y asegura que se muestren todos los píxeles entre los márgenes izquierdo y derecho del área visible de la pantalla. Pero hay que tener en cuenta que no se envían píxeles válidos durante todo el tiempo de duración de la línea, hay un periodo donde la información que llega al monitor es color negro (todas las componentes a cero). Con esto conseguimos los márgenes negros a izquierda y derecha de la pantalla. Estas franjas negras las controlamos con la señal *inhibicion_color*, que manda color negro cuando toma el valor '1'. Los periodos de tiempo empleados se muestran en la Figura 19:

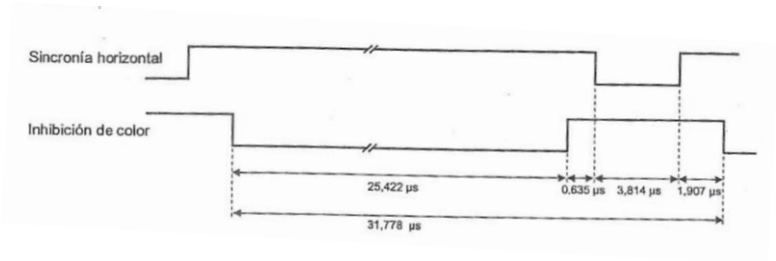


Figura 19. Sincronía horizontal

En cuanto a la sincronía vertical, marca el final de un fotograma y asegura que las líneas que lo componen se muestren entre los límites superior e inferior del área visible de la pantalla. De la misma forma que con la sincronía horizontal, no todo el periodo de duración de un fotograma contiene información de color, se generan dos franjas horizontales en los márgenes superior e inferior de la pantalla. La misma señal *inhibicion_color* controla esto, siguiendo el comportamiento que se muestra en la Figura 20.

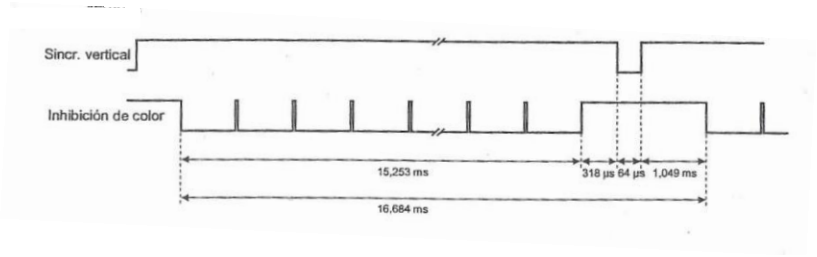


Figura 20. Sincronía vertical

Como se observa, *inhibicion_color* está activa durante la sincronía vertical y durante las primeras y últimas líneas de cada fotograma, y, para las demás líneas, durante la sincronía horizontal y en los primeros y últimos píxeles de cada línea.

Las señales de sincronía y de inhibición de color son generadas por el módulo *OV7670_senales_VGA*. Este módulo únicamente contiene la entrada *clk*, por la que introducimos el reloj de 25 MHz, frecuencia a la que se logra una resolución adecuada, y tres salidas, las señales de sincronía horizontal (*vga_hsync*) y vertical (*vga_vsync*) y la de inhibición de color (*vga_inhibicion_color*).

Para controlar correctamente los periodos de tiempo se cuentan el número de píxeles que forman cada línea y el de líneas que forman cada pantalla. Para ello se usan los contadores *pixel* y *línea*, actualizados en los procesos *pixel_actual* y *línea_actual*. Sabiendo la frecuencia de generación de píxeles, en este caso 25 MHz, y con la información temporal de las Figuras 19 y 20 se puede determinar el número de ellos en cada línea y cuando deben cambiar las señales de sincronía horizontal y de inhibición de color, así como las líneas necesarias para rellenar la pantalla. El valor de las constantes empleadas para controlar estos intervalos de tiempo y conseguir una resolución de 640x480 píxeles son los siguientes:

```
constant PERIODO_H:  integer := 800;
constant SINC_H_ON:  integer := 656;
constant SINC_H_OFF: integer := 752;
constant INHIBICION_COLOR_H: integer := 640;
constant PERIODO_V:  integer := 525;
constant SINC_V_ON:  integer := 490;
constant SINC_V_OFF: integer := 492;
constant INHIBICION_COLOR_V: integer := 480;
```

Código 5. Constantes para el control temporal de las señales VGA

Los procesos *control_sincronia_horiz, control_sincronia_vert* y *control_inhibicion_color* tienen en cuenta los contadores y las constantes para controlar las señales de sincronía horizontal, vertical e inhibición de color, respectivamente. La implementación de cada uno de ellos se muestra a continuación:

```
control_sincronia_horiz:process(clk)
  begin
  if rising_edge(clk25) then
       if(pixel > SINC_H_ON and pixel < SINC_H_OFF) then
         vga_hsync <= '0';
       else
         vga_hsync <= '1';
       end if;
  end if;
end process;
```

Código 6. Proceso de control de sincronía horizontal para VGA

```
     control_sincronia_vert:process(clk)
       begin
       if rising_edge(clk25) then
            if(linea > SINC_V_ON and linea < SINC_V_OFF) then
              vga_vsync <= '0';
            else
              vga_vsync <= '1';
            end if;
       end if;
     end process;
```

Código 7. Proceso de control de sincronía vertical para VGA

```
     control_inhibicion_color:process(clk)
       begin
       if rising_edge(clk25) then
            if(pixel > INHIBICION_COLOR_H or linea > INHIBICION_COLOR_V)
then
              vga_inhibicion_color <= '1';
            else
              vga_inhibicion_color <= '0';
            end if;
       end if;
     end process;
```

Código 8. Proceso de control de inhibición de color para VGA

Siguiendo con el modelo de agrupar en este módulo todas las señales de sincronización, añadimos una señal que controle la dirección de memoria en la que se lee cada píxel. Como se ha introducido en el apartado 4, en el *buffer* se almacena una imagen completa, escribiendo cada píxel de uno en uno desde el primer píxel de la primera línea, hasta el último píxel de la última línea. Por tanto, la lectura debe seguir la misma secuencia, comenzando a dibujar cada imagen desde la primera posición de memoria y acabando con la última, respetando los periodos de tiempo de inhibición de color para que no se hagan lecturas en momentos incorrectos. Una vez dibujada completamente la imagen, se reinicia el contador para volver de nuevo a la primera posición de memoria y estar preparado para mostrar la siguiente imagen.

La señal encargada de controlar la posición de memoria se denomina *pixel_address*, que se actualiza con cada píxel leído. Esta señal emplea las constantes que controlan la inhibición de color, *INHIBICION_COLOR_H* e *INHIBICION_COLOR_V*, para sincronizar correctamente cuando se lee un píxel en cada línea y cuando ha terminado de dibujarse la imagen, reiniciándose la posición de memoria. Esta señal auxiliar se conecta al puerto de salida *pixel_addr* para hacer llegar esta señal al *buffer.*

```
     direccion_memoria:process(clk)
     begin
       if rising_edge(clk25) then
         if(linea > INHIBICION_COLOR_V) then
           pixel_address <= (others => '0');
         else
           if(pixel < INHIBICION_COLOR_H) then
             pixel_address <= pixel_address+1;
           end if;
         end if;
       end if;
     end process;
```

```
pixel_addr <= std_logic_vector(pixel_address);
```

Código 9. Proceso de control de la posición de memoria de lectura

4.3 Señales de color

La información de color que es enviada al monitor es controlada por el módulo *OV7670_controlador_RGB,* que emplea las señales generadas en el módulo de sincronía para controlar el envío de píxeles válidos o, por el contrario, enviar color negro. Como entradas tiene el mismo reloj de 25 MHz, además de los contadores píxel y línea, la señal de inhibición de color y la información de color en RGB444.

Cuando el píxel a enviar se encuentra dentro del área visible de la pantalla se envía directamente la información de color en la entrada a la salida. El formato de color de la salida es RGB con cuatro bits para cada componente, por lo que ya está preparada para que el monitor pueda recibirla. La información de color en la entrada se obtiene del *buffer*, donde la dirección de memoria leída en cada ciclo de reloj viene determinada por la señal *pixel_addr*. Notar que hay que ser cuidadoso con la dirección de memoria a leer, ya que cualquier descuadre entre escritura y lectura supone que la imagen no se muestre correctamente en el monitor.

Si, por el contrario, se ha llegado al final del área visible y hay que dibujar los márgenes negros (la señal de inhibición de color cambia a nivel alto), a la salida de color se envían doce ceros, color negro.

```
control_RGB:process(clk)
begin
     if rising_edge(clk25) then
        if(inhibicion_color = '0') then
             rgb_out <= rgb_in;
        else
           rgb_out <= (others => '0');
        end if;
   end if;
end process;
```

Código 10. Proceso de envío de señal de color

4.4 Módulo VGA

Para hacer el código más legible y hacer un diseño más estructurado, agrupamos los dos módulos descritos anteriormente en un sistema de nivel superior, instanciando cada módulo como un componente. Más adelante hablaremos de cómo trabajar con la programación estructural en VHDL, en este punto nos centraremos en el funcionamiento del módulo VGA.

Los puertos de salida serán uno para cada componente de color, *VGA_ROJO*, *VGA_VERDE* y *VGA_AZUL;* uno para cada sincronía, vertical y horizontal, *VGA_HSYNC* y *VGA_VSYNC;* y otro para la dirección de memoria de lectura*, pixel_addr.* Los puertos asociados a la salida VGA irán directamente conectados a los pines del conector VGA en el diseño final, por lo que deben tener el formato adecuado para que no se produzcan fallos en la visualización de la imagen en pantalla.

Los puertos de entrada serán el reloj de 25 MHz que sincronice todo el módulo, *clk25*; una entrada de doce bits, *pixel_in,* que se conecte a la salida de datos del *buffer* y pase al módulo la información de color de cada píxel; y, como funcionalidad adicional para la

plataforma final, una entrada, *susp,* para poner en modo suspensión al módulo, de forma que la pantalla quede totalmente en negro cuando se active. Añadimos un proceso para enviar color negro al monitor cuando *susp* se activa, en caso contrario, se envía la información de color que sale del módulo *controlador_RGB*, dividiéndola en las tres componentes, con cuatro bits para cada una.

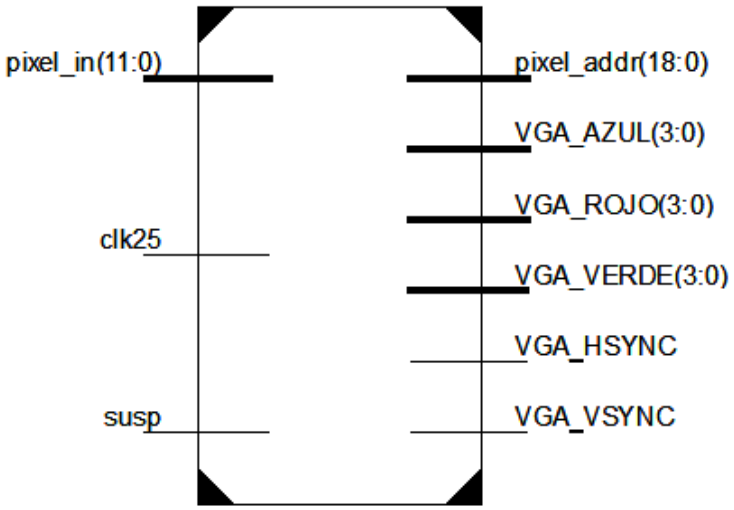


Figura 21. Módulo VGA. Puertos de entrada y salida

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

entity VGA is
  Port (
     clk25     : in    STD_LOGIC;
     pixel_in  : in    STD_LOGIC_VECTOR(11 downto 0);
     susp      : in    STD_LOGIC;
     pixel_addr: out   STD_LOGIC_VECTOR(18 downto 0);
     VGA_ROJO  : out   STD_LOGIC_VECTOR(3 downto 0);
     VGA_VERDE : out   STD_LOGIC_VECTOR(3 downto 0);
     VGA_AZUL  : out   STD_LOGIC_VECTOR(3 downto 0);
     VGA_HSYNC : out   STD_LOGIC;
     VGA_VSYNC : out   STD_LOGIC);
end VGA;

architecture Behavioral of VGA is
   COMPONENT controlador_RGB_v2
   PORT (
     clk : in  std_logic;
     inhibicion_color : in std_logic;
     rgb_in : in std_logic_vector(11 downto 0);
     rgb_out : out  std_logic_vector (11 downto 0));
   END COMPONENT;

   COMPONENT senales_vga
   PORT (
     clk : in std_logic;
     vga_hsync  : out std_logic;
     vga_vsync  : out std_logic;
     vga_inhibicion_color : out std_logic;
     pixel_addr : out std_logic_vector(18 downto 0));
   END COMPONENT;

begin
```

```
 modulo_senales_vga : senales_vga
   Port Map(
      clk => clk25,
      vga_hsync  => VGA_HSYNC,
      vga_vsync  => VGA_VSYNC,
      vga_inhibicion_color => inhibicion_color,
      pixel_addr => pixel_addr);

 modulo_controlador_RGB_imagen_original : controlador_RGB_v2
    Port map(
       clk => clk25,
       inhibicion_color => inhibicion_color,
       rgb_in => pixel_in,
       rgb_out => rgb_out );

envio_color_a_monitor: process(clk25)
begin
  if rising_edge(clk25) then
    if (susp = '0') then
      VGA_ROJO <= rgb_out(11 downto 8);
      VGA_VERDE <= rgb_out(7 downto 4);
      VGA_AZUL <= rgb_out(3 downto 0);
    else
      VGA_ROJO <= "0000";
      VGA_VERDE <= "0000";
      VGA_AZUL <= "0000";
    end if;
  end if;
end process;end Behavioral;
```

Código 11. Módulo VGA

4.5 Prueba de la resolución de color en VGA

La paleta de colores con la que trabaja la salida VGA de la *ZedBoard* es de 4096 colores, por lo que no es muy extensa. Para comprobar que el módulo trabaja correctamente y ver con qué resolución de color trabajaremos, hacemos un programa que muestre por pantalla las distintas tonalidades de una misma componente, es decir, vamos incrementando de uno en uno el valor de una componente dejando a cero las otras dos. De esta forma, podemos ver los saltos de color que se producen entre tonos consecutivos y hacernos una idea de con qué gama de colores trabaja la *ZedBoard* en su salida VGA.

Esta prueba consta de un módulo que va dando valores a la entrada del módulo VGA (*pixel_in*) de forma que se dibujen líneas verticales de una misma tonalidad, apareciendo las distintas tonalidades a lo largo de toda la pantalla. Para que el grosor de las líneas verticales sea suficiente para apreciarlas bien, utilizamos una señal *contador* de ocho bits que se incremente cada dos píxeles pero únicamente usamos los cuatro bits más significativos, de forma que el valor de la componente sólo cambiará cada 16 píxeles, obteniéndose líneas con el suficiente grosor para observar bien los saltos de color. El contador debe reiniciarse al final de cada línea para que las líneas verticales se dibujen correctamente, para ello hacemos uso de la señal de inhibición de color como *reset* del módulo. Además se han añadido dos entradas, *azul* y *verde,* conectadas a dos switch de la placa, para que al activarlos cambie la componente a azul o verde, respectivamente, mientras que si están desactivadas la componente dibujada es la roja.

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

```
use IEEE.numeric_std.ALL;

entity contador is
    Port (  clk : in  STD_LOGIC;
            verde: in STD_LOGIC;
            azul: in STD_LOGIC;
            reset: in STD_LOGIC;
            salida : out  STD_LOGIC_VECTOR (11 downto 0));
end contador;

architecture Behavioral of contador is
begin
salida <= salida_aux;

process(clk)
begin
   if(rising_edge(clk)) then
      if (reset = '1') then
        contador <= (others => '0');
      else
        scaler <= not scaler;
        if (scaler = '1') then
          contador <= contador + 1;
          if(verde = '1') then
            salida_aux <= "0000" & contador(7 downto 4) & "0000";
          elsif(azul = '1') then
            salida_aux <= "00000000" & contador(7 downto 4);
          else
            salida_aux <= contador(7 downto 4) & "00000000";
          end if;
        end if;
      end if;
   end if;
end process;
end Behavioral;
```

Código 12. Módulo para prueba de resolución VGA

Figura 22. Prueba de resolución VGA. Rojo

Figura 23. Prueba de resolución VGA. Verde

Vemos que la resolución de color con la que trabaja la salida VGA no es muy alta, por lo que los colores que conseguiremos en la imagen al trabajar con el sensor no serán de mucha calidad.

Capítulo 5: Diseño del módulo de control y configuración del sensor

Este módulo tiene la función de controlar la configuración del sensor, el cual, como se introdujo en el apartado 3, emplea la interfaz SCCB, utilizada por *OmniVision* en sus dispositivos. Antes de centrarnos en la implementación, conviene hacer una introducción a esta interfaz para familiarizarnos con ella.

5.1 Interfaz SCCB

El SCCB (*Serial Camera Control Bus*) es un bus serie de 3 hilos diseñado por *OmniVision* para el control de la mayoría de funciones de su gama de sensores de imagen. En este caso, el sensor con el que trabajamos presenta una versión simplificada de 2 hilos debido a que está instalado en una placa que permite el acceso a menos pines de los realmente disponibles en el chip.

Este bus se basa en el modelo maestro/esclavo (*master/slave*), muy similar al bus I^2C, en el que un dispositivo que actúa como maestro se conecta al bus SCCB para controlar al menos un dispositivo esclavo. El esquema funcional del bus se muestra en la Figura 24.

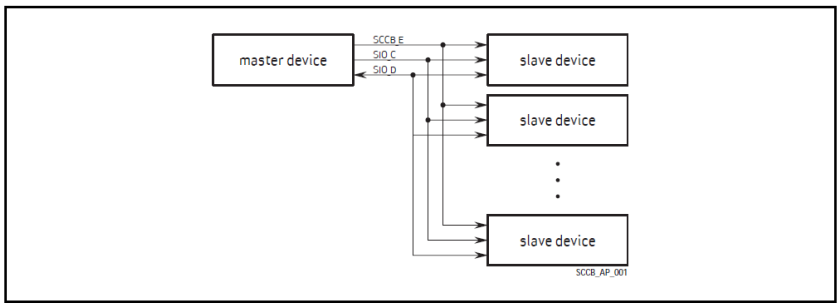


Figura 24. Esquema SCCB

Los tres hilos del bus se denominan SCCB_E, SIOC y SIOD, sus funciones son las siguientes:

- SCCB_E: controlado por el dispositivo maestro. Se pone a nivel alto cuando el bus está libre. Cambia a nivel bajo cuando se está produciendo una transmisión o el sistema está en suspensión.

- SIOC: controlado por el maestro. A nivel alto constante cuando el bus está libre y a nivel bajo constante cuando el sistema está en suspensión. Cuando el bus está activo va cambiando de nivel alto a bajo, actuando como el reloj que sincroniza la transmisión.

- SIOD: pueden acceder a él tanto el dispositivo maestro como el esclavo. Es el hilo que transporta los bits de datos en la transmisión. Cuando el bus está libre queda a nivel alto. A nivel bajo cuando el sistema se suspende.

La placa con la que trabajamos sólo nos permite acceder a los pines SIOC y SIOD, dejando por defecto el SCCB_E a nivel alto. Por lo que dejamos la función de SCCB_E a un lado y nos centramos en SIOC y SIOD.

Al constar de dos hilos, el bus sólo permite un dispositivo maestro que controla únicamente a un dispositivo esclavo, que en este caso es suficiente, ya que sólo trabajamos con un dispositivo maestro (la *ZedBoard*) que controla a un esclavo (el sensor OV7670). El esquema inicial se reduce al mostrado en la Figura 25.

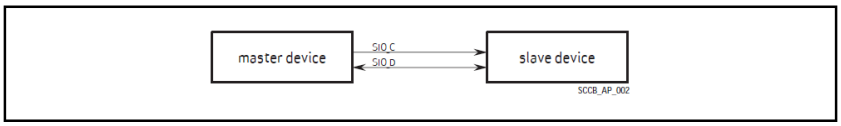


Figura 25. Esquema SCCB de 2 hilos

En la implementación con dos hilos, *OmniVision* advierte de que el dispositivo maestro debe ser capaz de poner la línea de datos, SIOD, en tres estados: alto (‘1’), bajo (‘0’) y alta impedancia (‘Z’). La *ZedBoard* puede soportar los tres estados, por lo que no hay problema a la hora de usar este bus.

5.2 Etapas de transmisión

Una vez vista una visión general del funcionamiento del bus, pasamos a ver cómo trabajar con las señales SIOC y SIOD para realizar la comunicación. La señal SIOC indica cuando se transmite cada bit. Esto se produce en cada flanco ascendente de la señal. El valor del bit transmitido se indica con SIOD, que debe mantener el valor del bit durante un periodo completo de SIOC, que es de 10 µs, lo que supone una frecuencia de 100 kHz. La señal SIOD es bidireccional, ya que puede ser controlada tanto por el dispositivo maestro como por el esclavo, por tanto en el código VHDL la definiremos como un puerto “INOUT”, de entrada y salida.

Distinguimos tres etapas en la transmisión: secuencia de inicio, secuencia final y transmisión de datos.

5.2.1 *Inicio de transmisión*

Partiendo de que SIOC se encuentra a ‘1’ y que SIOD está en el estado de alta impedancia ‘Z’, se indica al sensor que la transmisión ha comenzado cuando SIOD pasa de ‘1’ a ‘0’ mientras que SIOC se mantiene a ‘1’, como se muestra en la Figura 26. Una vez que se ha producido el flanco de bajada de SIOD, SIOC también pasa a ‘0’ y ya comienza la transmisión de datos, transmitiendo el primer bit en el siguiente flanco ascendente de SIOC.

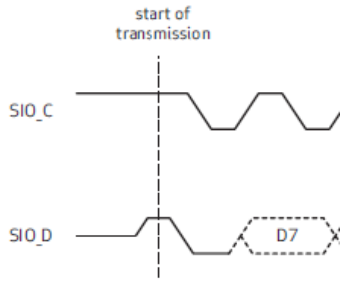


Figura 26. Inicio de transmisión SCCB

5.2.2 *Final de transmisión*

Una vez que se ha transmitido el último bit, se vuelve a poner SIOC a ‘1’ y, con SIOC a nivel alto, se tiene que producir un flanco ascendente de SIOD, pasando de ‘0’ a ‘1’ y manteniéndose a nivel alto.

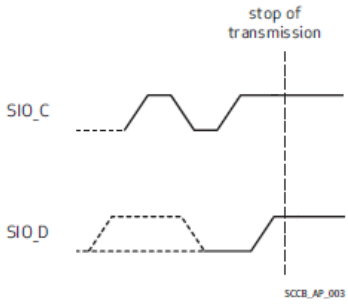


Figura 27. Final de transmisión SCCB

5.2.3 *Transmisión de datos*

La comunicación con el dispositivo esclavo se divide en tres fases de ocho bits, en cada uno de los cuales se transmite: dirección de identificación del dispositivo (*ID Address*), dirección del registro, dato a escribir.

En cada ciclo de transmisión completo podemos escribir un *byte* de datos en un registro del dispositivo esclavo, siguiendo el esquema de la Figura 28. Los bits se transmiten en serie, de uno en uno, siendo el más significativo el primero que se envía en cada fase.

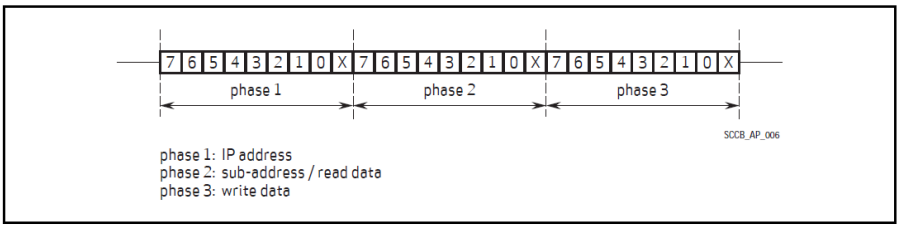


Figura 28. Esquema de transmisión de un *byte*

La dirección de identificación consta de ocho bits e identifica al dispositivo al que el maestro intenta acceder, en este caso, la dirección del sensor OV7670 para escritura es "0x42" (esta dirección es exclusiva para cada dispositivo de *OmniVision*, se debe consultar en su *datasheet*). La dirección de cada registro viene especificada en la documentación del sensor, abarcando de "0x00" a "0xC9", aunque especifica que hay más registros pero quedan reservados. Cada registro cuenta con ocho bits, cuya función se consulta en la documentación para conseguir la configuración deseada.

Para separar cada fase se transmite un noveno bit, llamado "*Don't-care bit*" por el fabricante. Este es un bit de relleno en el que SIOD toma el estado "Z", de ahí que sea necesario que el dispositivo maestro pueda manejar los tres estados.

El esquema de un ciclo de comunicación completo queda así:

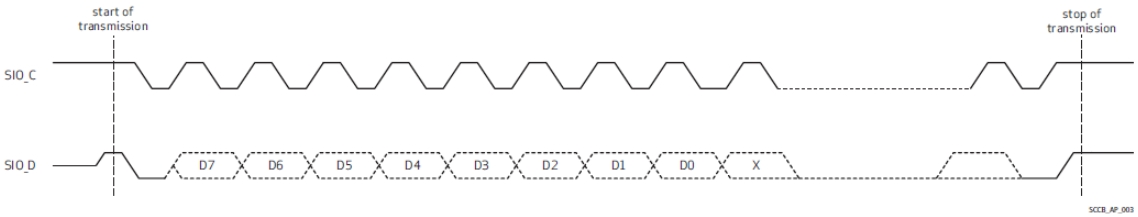


Figura 29. Ciclo de comunicación completo SCCB

5.3 Diseño VHDL

En la implementación del módulo debemos tener en cuenta lo visto anteriormente: las secuencias de inicio y final de cada ciclo de transmisión, el periodo de SIOC (10 µs), la transmisión serie en la que el bit más significativo se transmite primero y el bit de relleno para separar cada etapa.

Con el fin de hacer el diseño más intuitivo, se divide el módulo en dos submódulos, uno para controlar las señales SIOC y SIOD, y otro para controlar el valor de los bits que SIOD toma.

5.3.1 *Submódulo de control de* SIOC *y* SIOD

Este submódulo llamado *SCCB_sender*, controla las señales SIOC y SIOD cumpliendo los estándares de SCCB.

El módulo trabaja con un reloj de 25 MHz, obtenido a partir del divisor de frecuencia, por lo que debemos calcular el número de ciclos de reloj que debemos esperar para cambiar el estado de SIOC y así conseguir la frecuencia de 100 kHz.

$$T_{clk25} = \frac{1}{25\ \text{MHz}} = 40\ \text{ns} \rightarrow \text{Número de ciclos} = \frac{10\ \mu\text{s}}{40\ \text{ns}} = 250$$

Necesitamos esperar 250 ciclos de reloj para conseguir 100 kHz, por tanto, con un contador de 8 bits será suficiente para alcanzar ese valor.

Para distinguir las tres etapas de transmisión empleamos una señal auxiliar, llamada *busy_aux*, que se activa a nivel alto una vez que acaba la secuencia de inicio y comienza la transmisión de datos y se desactiva al finalizar la transmisión.

El módulo cuenta con una entrada que indica cuando debe comenzar un ciclo de transmisión, ya que para configurar el sensor necesitaremos un ciclo por cada registro a configurar. Cuando se activa esta entrada (*send*), se cargan los 24 bits a enviar (entradas *id, reg* y *value*) y comienza todo el proceso de transmisión. Una vez que éste termina se activa la salida *taken* para indicar al módulo de datos que debe cargar en las entradas de datos los nuevos valores a enviar. Una vez que comienza la transmisión el valor de *taken* vuelve a ser '0'. En cada ciclo se cargan los tres *bytes* en la señal *data*, a la que se van añadiendo ceros a la derecha cada vez que se manda un bit, dejando su bit más significativo conectado a la salida SIOD, de forma que se van enviando los 24 bits de uno en uno.

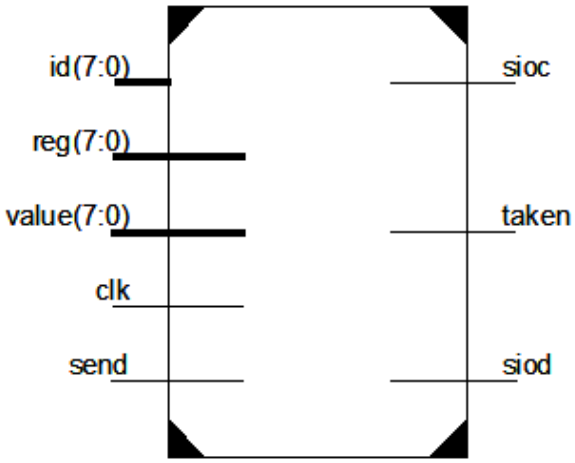


Figura 30. Submódulo *SCCB_sender.* Puertos de entrada y salida

El código que compone el módulo se adjunta a continuación:

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

entity SCCB_sender is
    Port ( clk   : in  std_logic;
           SIOD  : inout  std_logic;
           SIOC  : out  std_logic;
           taken : out  std_logic;
           send  : in  std_logic;
           id    : in  std_logic_vector (7 downto 0);
           reg   : in  std_logic_vector (7 downto 0);
           value : in  std_logic_vector (7 downto 0));
end SCCB_sender;

architecture Behavioral of SCCB_sender is
signal pausa   : unsigned (7 downto 0) := "00000001"; -- contador
para hacer la pausa de 250 ciclos y conseguir el reloj de 100 kHz

begin

process(clk)
begin
  if rising_edge(clk) then

    -- INICIO DE TRANSMISION -- Una transmision completa para cada
registro

    if send = '1' and busy_aux = '0' then
      if pausa = "00000001" then
        SIOC <= '1';
        SIOD <= '1';
        taken <= '0';
        pausa <= pausa + 1;
        data <= id & reg & value;

      else
        if pausa = "00111110" then
          SIOD <= '0';
          pausa <= pausa + 1;
        end if;

        pausa <= pausa + 1;

        if pausa = "01111110" then -- pausa de 125 ciclos (t = 5us)
          SIOC <= '0'; -- SIOC = 0
          busy_aux <= '1';
          SIOD <= data(23);  -- carga 1º bit
          data <= data(22 downto 0) & '0';
          contador <= contador + 1;
          pausa <= "00000001";
        end if;
      end if;
    end if;

    ----------------------------

    -- TRANSMISION DE DATOS --
```

```
      if busy_aux = '1' then
        if contador2 /= "11" then
          if pausa = "11111010" then
            SIOC <= '0';
            pausa <= "00000001";

            if contador = "1000" then
              contador <= "0000"; -- Reinicio contador
              contador2 <= contador2 + 1;

              SIOD <= 'Z'; -- Don't care bit
            else
              SIOD <= data(23);
              data <= data(22 downto 0) & '0';
              contador <= contador + 1;
            end if;

            -- 125 ciclos (Mitad de periodo)
          elsif pausa = "01111101" then
            SIOC <= '1'; -- SIOC = 1
            pausa <= pausa + 1;

          else
            pausa <= pausa + 1;
          end if;
        else

        -- FIN TRANSMISION --

          if pausa = "11111010" then
            if click = '0' then
              SIOC <= '0';
              SIOD <= '0';
              click <= '1';
            else
              busy_aux <= '0';
              SIOD <= '1';
              SIOC <= '1';
              contador2 <= "00";
              taken <= '1';
              click <= '0';

            end if;
            pausa <= "00000001";

          elsif pausa = "01111110" then
            SIOC <= '1';
            pausa <= pausa + 1;

          else
            pausa <= pausa + 1;
          end if;
        end if;
      end if;
    end if;
  end process;
  end Behavioral;
```

Código 13. Implementación submódulo *SCCB_sender*

5.3.2 *Submódulo de carga de datos*

Este submódulo, llamado *SCCB_registers* guarda los valores de cada registro a configurar y se encarga de pasarlos al módulo *SCCB_sender* para que los envíe al sensor.

Para que la sincronización sea correcta, debe trabajar con el mismo reloj que *SCCB_sender*. Este módulo es más simple que el anterior, ya que su función es únicamente cambiar el valor de los datos a enviar cuando el ciclo completo de transmisión ha finalizado. Por tanto, además de la entrada del reloj, presenta dos entradas más: una conectada a la salida *taken* del módulo anterior, llamada *advance*, que indica cuando se debe pasar al siguiente registro, y otra añadida por si queremos volver a configurar el sensor, denominada *resend*, que conectaremos a un botón de la placa para habilitar la función de reconfigurar el sensor mientras se ejecuta el código.

Para controlar el orden en que se envía cada registro, empleamos la señal *address* junto con un bloque *case*. El valor de esta señal, que actúa como contador, se aumenta en uno cada vez que se activa la señal *advance*, de forma que en la salida de datos, de 16 bits, (*command*) se carga el nuevo valor de registro con su correspondiente *byte* de datos.

Por último, para detener el envío de datos, se ha implementado una salida de un bit (*finished*) que se activa cuando la configuración de todos los registros ha sido enviada y el sensor ha quedado completamente configurado. A modo de indicador en tiempo real, esta señal quedará conectada a un LED de la placa para que éste quede encendido cuando la configuración haya terminado.

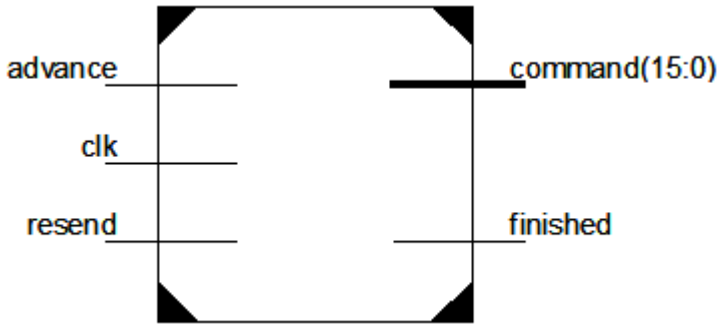


Figura 31. Submódulo *SCCB_registers.* Puertos de entrada y salida

A continuación se muestra una versión simplificada del código con únicamente 4 registros de prueba y se tenga una visión del funcionamiento del módulo. En la versión final cargada sobre el sensor se configurarán muchos más registros, pero esto sólo supone más selecciones en el *case,* aspecto que en este punto no es relevante.

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

entity SCCB_registers is
    Port ( clk      : in  std_logic;
           resend   : in  std_logic;
           advance  : in  std_logic;
           command  : out  std_logic_vector(15 downto 0);
           finished : out  std_logic);
end SCCB_registers;

architecture Behavioral of SCCB_registers is

begin
```

```
command <= sreg;

process(clk)
begin
  if rising_edge(clk) then
    if resend = '1' then
      address <= (others => '0');
    elsif advance = '1' then
      address <= std_logic_vector(unsigned(address)+1);
    end if;

    case address is
      when x"00" => sreg <= x"1280"; -- Reset de software
      when x"01" => sreg <= x"1204";
      when x"02" => sreg <= x"B084";
      when x"03" => sreg <= x"6BCA";

---- Espacio reservado para más registros ----

      when others => sreg <= x"FFFF";
    end case;

    if sreg = x"FFFF" then
      finished <= '1';
    else
      finished <= '0';
    end if;

  end if;
end process;
end Behavioral;
```

Código 14. Implementación submódulo *SCCB_registers*

5.3.3 *Módulo de control*

Para conectar estos dos submódulos, se implementa el módulo *ov7670_controlador*, con un nivel jerárquico por encima de ellos, que será el que se incluya en el diseño final del proyecto. Empleamos la sentencia COMPONENT para declarar cada submódulo, más adelante, en el diseño completo de la plataforma, explicaremos cómo funciona. Este módulo maneja todas las señales de control del sensor, éstas son: SIOD, SIOC, XCLK, PWDN y RESET.

El funcionamiento de las señales SIOD y SIOC ya ha sido estudiado en los submódulos anteriores, por lo que explicamos brevemente la función de las tres señales restantes.

XCLK es el reloj del sistema, el que emplea el sensor para sincronizar la captura de imagen. Según la documentación del sensor, debe tener una frecuencia en 10 y 48 MHz, en este caso introduciremos una señal de reloj de 25 MHz.

PWDN es una señal, activa a nivel alto, empleada para poner al sensor en modo suspensión. Queda conectada a la entrada *susp* para añadir la funcionalidad de poner en suspensión el sensor con un interruptor. Cuando éste se activa, el módulo se suspende, cuando vuelva a desactivarse, el sensor se activará de nuevo.

RESET es la señal empleada para resetear los registros del sistema y devolverlos a sus valores por defecto cuando se conecta a tierra. Optamos por hacer un reseteo de software (bit 7 del registro COM7 activo) al inicio la configuración, por tanto se deja a nivel alto.

Como el reloj de entrada al módulo es de 25 MHz, la salida XCLK la conectamos directamente a este reloj. La salida *config_finished* se añade para conectarla a un LED y que se encienda cuando la configuración ha terminado. Por último comentar que la salida *finished* del módulo *SCCB_registers* se conecta negada a la entrada *send* de *SCCB_sender* para que el módulo deje de enviar datos cuando la configuración finalice y *finished* se active.

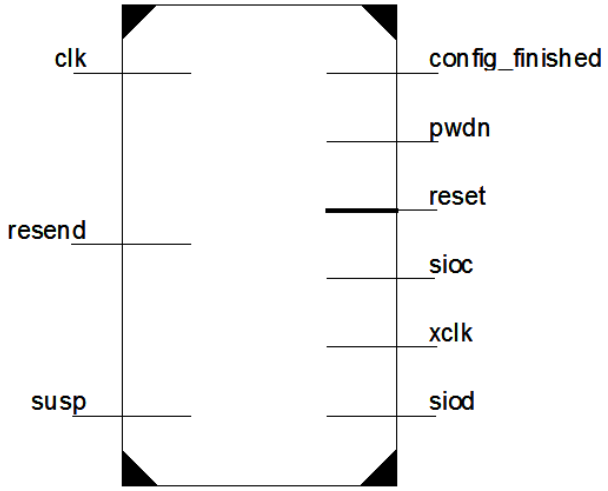


Figura 32. Módulo *ov7670_controlador.* Puertos de entrada y salida

El código VHDL del módulo es el siguiente:

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

entity ov7670_controlador is
    Port (
         clk    : in    std_logic;
         resend : in    std_logic;
         susp   : in STD_LOGIC;
         config_finished : out std_logic;
         SIOC   : out   std_logic;
         SIOD   : inout std_logic;
         reset  : out   std_logic;
         pwdn   : out   std_logic;
         xclk   : out   std_logic
    );
end ov7670_controlador;

architecture Behavioral of ov7670_controlador is

------------ Instanciación de componentes ----------
COMPONENT SCCB_registers
PORT(
  clk      : IN std_logic;
  advance  : IN std_logic;
  resend   : in std_logic;
  command  : OUT std_logic_vector(15 downto 0);
  finished : OUT std_logic
  );
END COMPONENT;

COMPONENT SCCB_sender
PORT(
```

```
    clk   : IN std_logic;
    send  : IN std_logic;
    taken : out std_logic;
    id    : IN std_logic_vector(7 downto 0);
    reg   : IN std_logic_vector(7 downto 0);
    value : IN std_logic_vector(7 downto 0);
    SIOD  : INOUT std_logic;
    SIOC  : OUT std_logic
    );
END COMPONENT;
----------------------------------------------------

-- Dirección de identificación del sensor OV7670
constant camera_address : std_logic_vector(7 downto 0) := x"42";

begin
   config_finished <= finished;

send <= not finished;
mod_SCCB_sender: SCCB_sender PORT MAP(
   clk   => clk,
   taken => taken,
   SIOD  => SIOD,
   SIOC  => SIOC,
   send  => send,
   id    => camera_address,
   reg   => command(15 downto 8),
   value => command(7 downto 0)
);

reset <= '1';
pwdn  <= susp;    -- Si SW = '1' entra en modo suspensión
xclk  <= clk;

mod_SCCB_registers: SCCB_registers PORT MAP(
   clk      => clk,
   advance  => taken,
   command  => command,
   finished => finished,
   resend   => resend
);

end Behavioral;
```

Código 15. Implementación módulo *ov7670_controlador*

5.4 Simulación

Una vez implementados hacemos la simulación de cada uno de ellos para comprobar que funcionan correctamente. Para ello empleamos la herramienta de *test bench* que ofrece Xilinx, *ISim*.

Para hacer la simulación del módulo *SCCB_sender*, damos unos valores cualquiera a los tres *bytes* a enviar. A la dirección del dispositivo le damos el valor "0x42", ya que es la que utilizaremos para el sensor, y a la dirección del registro y el valor que toma éste le damos el valor "10000001", aunque podemos darle cualquier otro. Configuramos el periodo de reloj a 40 ns y lanzamos la simulación, mostrada en la Figura 33:

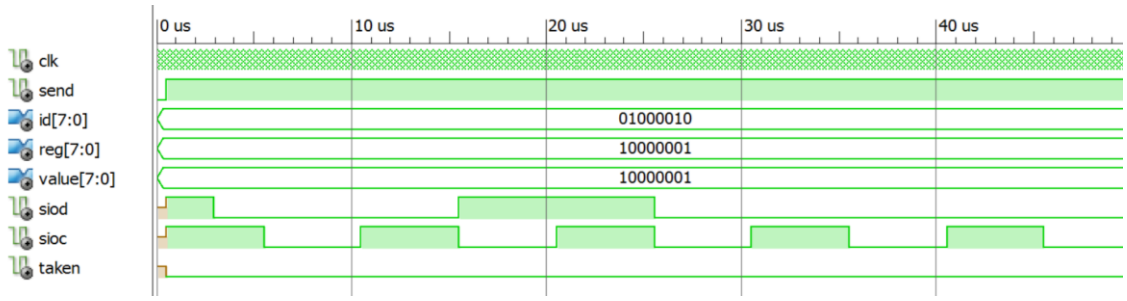


Figura 33. Simulación módulo SCCB_sender. Fase inicial

Como se puede observar, SIOC tiene el periodo que se necesita, 10 µs, y SIOD y SIOC realizan correctamente la secuencia de inicio de transmisión cuando la entrada *send* se activa. Ahora hay ver si SIOD transmite correctamente los tres *bytes* de datos, el primero de ellos se muestra en la Figura 34:

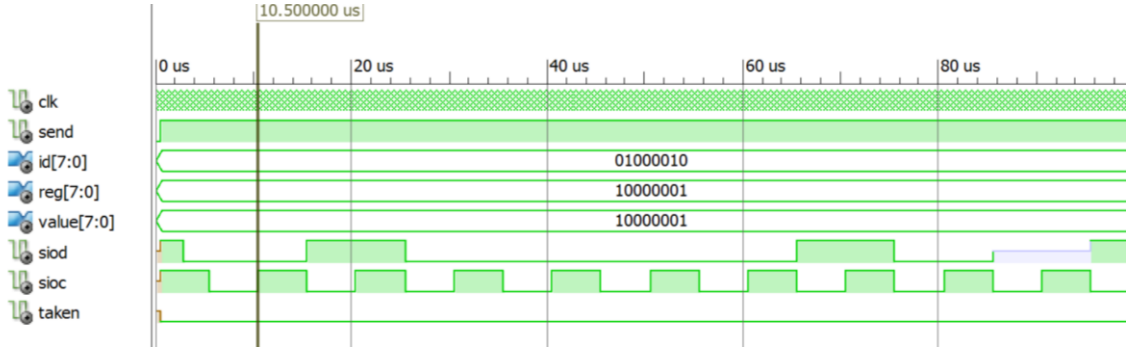


Figura 34. Simulación módulo SCCB_sender. Primer *byte*

La secuencia “01000010”, 42 en hexadecimal, comienza a transmitirse inmediatamente después de producirse la secuencia de inicio de transmisión, concretamente en el flanco de subida de SIOC marcado con el cursor. Después, se observa que en cada flanco ascendente, SIOD toma el valor adecuado para cada bit. En el diseño se ha optado por cambiar el valor del bit en el flanco de bajada de SIOC, de forma que el bus tiene medio ciclo de reloj para estabilizarse para cuando llega el flanco de subida y se escribe el bit en el sensor. Una vez finalizada la transmisión de los ocho bits, se pone SIOD a alta impedancia para que el sensor lea el bit de relleno que indica que la primera fase ha terminado. Cuando se ha leído este bit, comienza la transmisión del segundo bit, como vemos en la Figura 35:

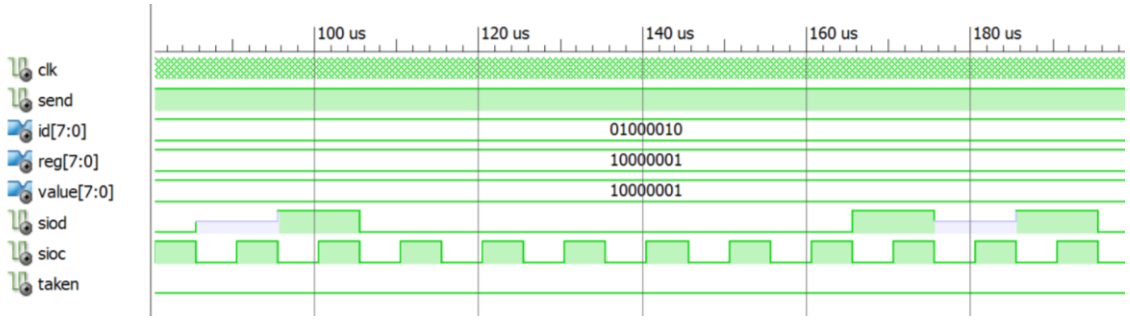


Figura 35. Simulación módulo SCCB_sender. Segundo *byte*

La secuencia “10000001”, correspondiente a la dirección de registro, también se envía correctamente. Al acabar, de la misma forma que con el primer *byte*, se envía el bit de relleno, dando comienzo a la transmisión del tercer y último *byte*, mostrado en la Figura 36:

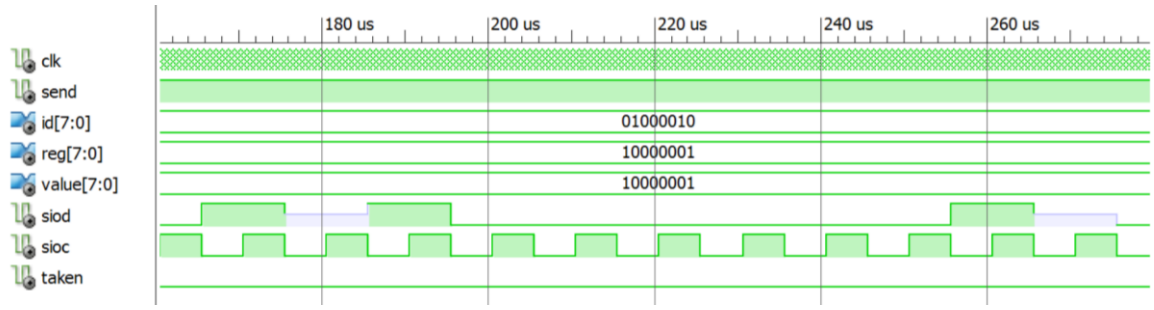


Figura 36. Simulación módulo SCCB_sender. Tercer *byte*

La transmisión del *byte* correspondiente al valor del registro también se transmite correctamente. En esta simulación se ha dejado activa la entrada *send*, de forma que al terminar el ciclo de transmisión automáticamente comienza el siguiente, que en este caso será igual que el anterior, ya que las entradas de datos no cambian. En la Figura 37 se puede ver que, tras un breve periodo de espera, se produce la secuencia de final de transmisión (SIOC a '1' y SIOD pasa de '0' a '1') y comienza de nuevo la secuencia de inicio de transmisión, comenzando de nuevo el ciclo. La salida *taken* se activa durante un breve periodo de tiempo (un ciclo de reloj) para indicar al módulo de datos que se deben cargar los nuevos datos a enviar.

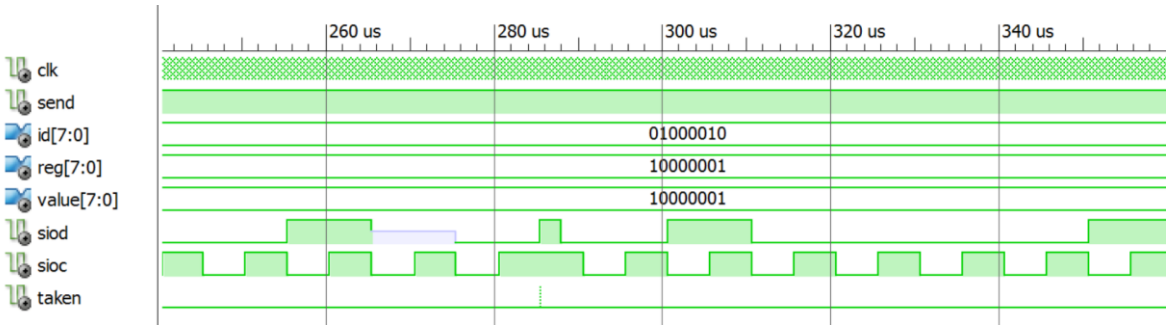


Figura 37. Simulación módulo SCCB_sender. Fase final e inicio de la siguiente

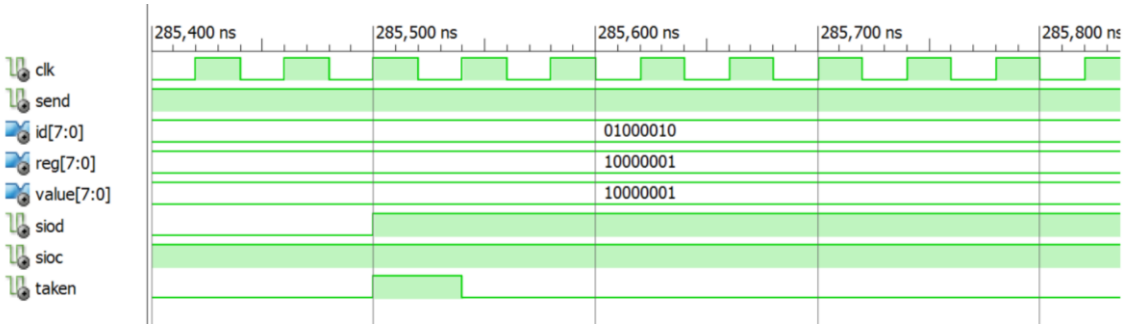


Figura 38. Simulación módulo SCCB_sender. Señal *taken*

Para finalizar la simulación completa del módulo, se desactiva la entrada *send* para simular que el módulo ha quedado configurado y se debe terminar la comunicación. Como se muestra en la Figura 39, las señales SIOD y SIOC quedan activadas tras la secuencia de final de transmisión, siguiendo correctamente el estándar de SCCB.

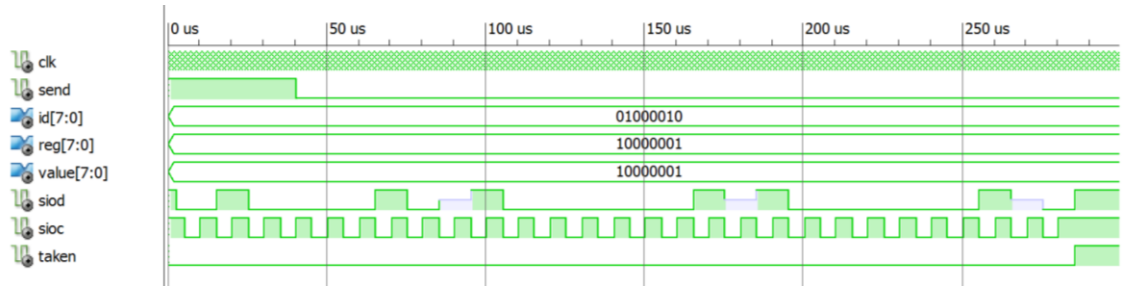


Figura 39. Simulación módulo SCCB. Ciclo completo

El módulo *SCCB_registers* únicamente guarda la dirección y el valor de los registros y los pasa a *SCCB_sender,* por tanto, para comprobar que funciona correctamente, en lugar de hacer una simulación exclusiva para él, es más adecuado implementar el controlador completo, incluyendo los dos submódulos, y realizar la simulación de éste. Seleccionamos un periodo de reloj de 40 ns y lanzamos iSim, empleando los registros de prueba anteriores:

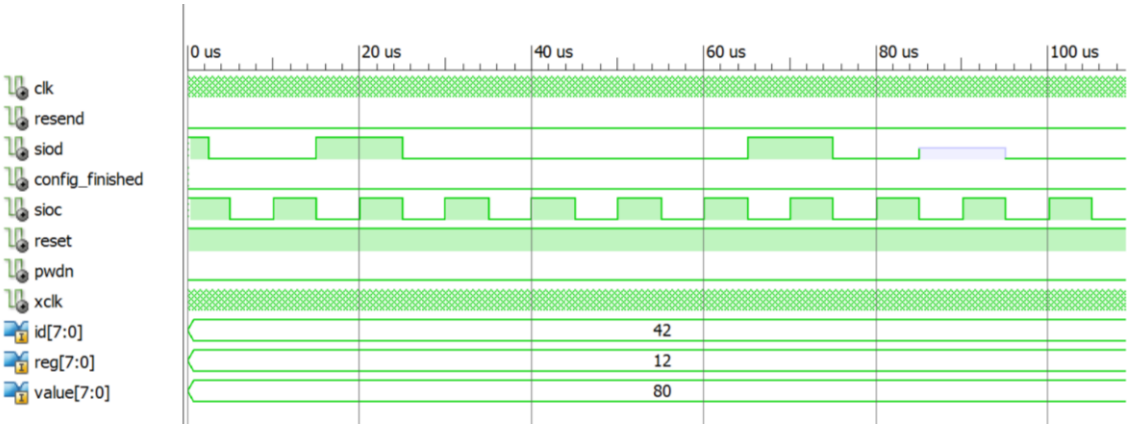


Figura 40. Simulación módulo OV7670_controlador. Inicio de transmisión

El inicio de la transmisión, como se muestra en la Figura 40, se realiza correctamente y la carga de los valores es correcta. Ahora queda ver si una vez terminada la transmisión del primer registro se pasa correctamente al siguiente. Continúa la simulación en la Figura 41:

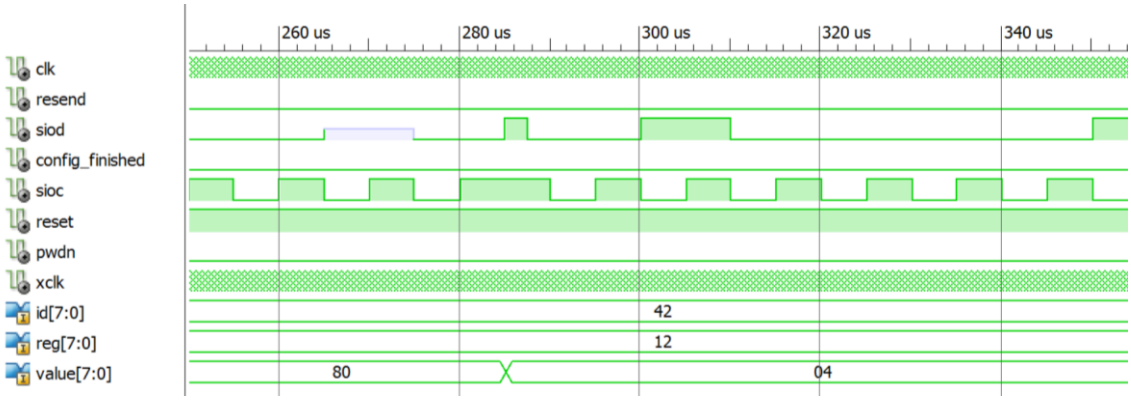


Figura 41. Simulación módulo OV7670_controlador. Cambio de valores

Cuando termina el primer ciclo se cambia de valor (de “0x1280” a “0x1204”) y la transmisión continúa, lo que supone que el módulo *SCCB_registers* funciona correctamente. Por último nos cercioramos de que una vez transmitidos los 4 registros de prueba se finaliza la transmisión y se activa la salida *config_finished*:

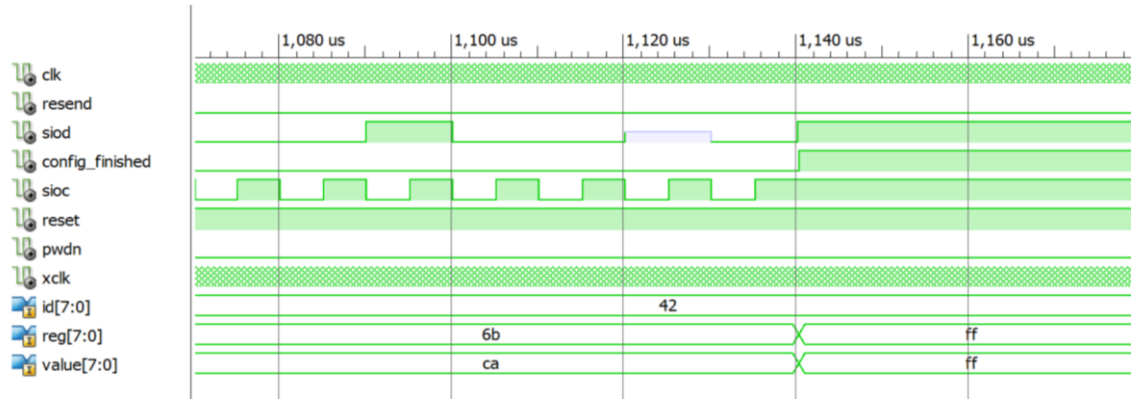


Figura 42. Simulación módulo OV7670_controlador. Fin de transmisión

En la Figura 42 vemos que tras el último registro (6B) se activa la salida *config_finished* y las salidas SIOD y SIOC quedan a nivel alto, finalizando correctamente la transmisión. La Figura 43 muestra una visión global de toda la simulación, mostrando que se cargan los valores correctos en cada momento.

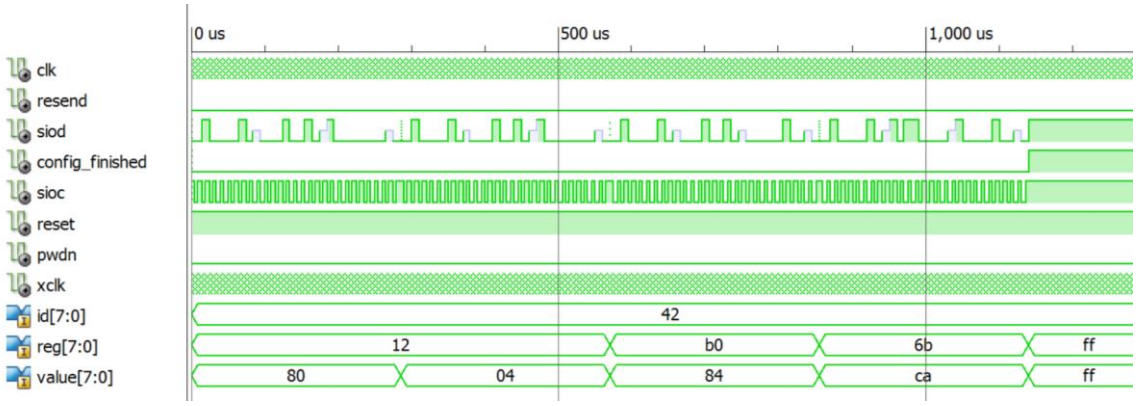


Figura 43. Simulación completa módulo OV7670_controlador

Tras esta simulación, el módulo *OV7670_controlador* está listo para ser incluido en el diseño final del proyecto.

Capítulo 6: Desarrollo de una plataforma de procesamiento de vídeo en tiempo real

Una vez diseñados los módulos necesarios para la adquisición de imagen del sensor, visualización de ésta en un monitor y control del sensor, queda hacer el diseño completo de la plataforma de procesamiento de vídeo en tiempo real, interconectando todos los módulos correctamente y haciendo distintas pruebas con la configuración del sensor para ver las posibilidades que ofrece.

6.1 Diseño general

Para hacer la interconexión de los módulos aprovechamos la programación estructural que ofrece VHDL. Creamos un módulo de nivel jerárquico superior a los demás donde instanciar todos los módulos diseñados en capítulos anteriores y crear la plataforma completa.

6.1.1 *Puertos de entrada y salida*

Los puertos de entrada y salida son comunes a los distintos diseños que implementaremos. Deben abarcar todos los pines del sensor y los de la salida VGA, además de contar con un reloj de entrada para sincronizar todos los procesos. Adicionalmente, añadimos un LED de salida para indicar que la configuración del sensor de ha completado, un botón para reconfigurar el sensor y un interruptor para poner el sistema en modo suspensión. Los puertos de la plataforma son los siguientes:

Nombre	Entrada/Salida	Tipo	Función
OV7670_SIOC	Salida	STD_LOGIC	Asociado al pin SIOC del sensor
OV7670_SIOD	Entrada/Salida	STD_LOGIC	Asociado al pin SIOD del sensor
OV7670_RESET	Salida	STD_LOGIC	Asociado al pin RESET del sensor
OV7670_PWDN	Salida	STD_LOGIC	Asociado al pin PWDN del sensor
OV7670_HREF	Entrada	STD_LOGIC	Asociado al pin HREF del sensor
OV7670_VSYNC	Entrada	STD_LOGIC	Asociado al pin VSYNC del sensor
OV7670_PCLK	Entrada	STD_LOGIC	Asociado al pin PCLK del sensor
OV7670_XCLK	Salida	STD_LOGIC	Asociado al pin XCLK del sensor
OV7670_DATA	Entrada	STD_LOGIC_VECTOR (8 bits)	Asociados a las entradas de datos del sensor
VGA_ROJO	Salida	STD_LOGIC_VECTOR (4 bits)	Salida de la componente roja de VGA
VGA_VERDE	Salida	STD_LOGIC_VECTOR (4 bits)	Salida de la componente verde de VGA
VGA_AZUL	Salida	STD_LOGIC_VECTOR (4 bits)	Salida de la componente azul de VGA

VGA_HSYNC	Salida	STD_LOGIC	Referencia horizontal de VGA
VGA_VSYNC	Salida	STD_LOGIC	Referencia vertical de VGA
CLK100	Entrada	STD_LOGIC	Reloj de entrada del sistema (100 MHz)
LED	Salida	STD_LOGIC	LED
BTN	Entrada	STD_LOGIC	Botón de reconfiguración
SW	Entrada	STD_LOGIC	Switch de suspensión

Tabla 9. Puertos de la plataforma

El esquema de la plataforma, visto como una "caja negra" queda así, con los pines de entrada en la parte izquierda y los de salida a la derecha, con la excepción de SIOD, que queda en la parte derecha pese a ser de entrada y salida:

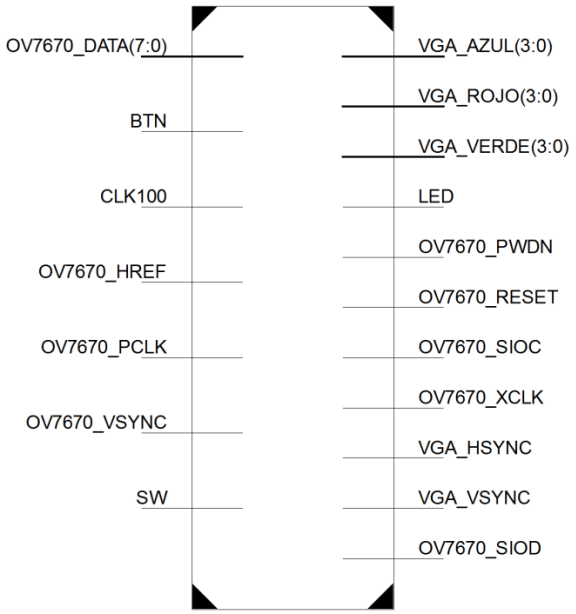


Figura 44. Esquema básico de la plataforma

6.1.2 *Mapeado de los pines*

Para asignar cada uno de los puertos de la plataforma a un pin concreto, debemos crear un archivo en formato *.ucf* que realice el mapeado adecuadamente. En primer lugar mapeamos los pines asignados al sensor, que son los que tienen el código OV7670.

Para conectar el sensor a la *ZedBoard*, como hemos introducido anteriormente, hemos empleado una placa extensora conectada al conector FMC. Por tanto, debemos separar el proceso de mapeado en dos partes: ver, con la documentación de la placa extensora, cuál es el pin del conector FMC asociado a cada pin de la placa donde va conectado el sensor y, una vez hecha esta asignación, asociar cada pin FMC al código con el que lo reconoce la *ZedBoard*, que se encuentra en el *Master UCF*, archivo descargable en *ZedBoard.org*. Este archivo también sirve para saber la alimentación asignada a cada pin, referenciada con el código *IOSTANDARD.*

Los pines de la placa a los que queda conectado el sensor han sido indicados en el apartado 3.3, por lo que hacemos la asociación de esos pines con los pines FMC.

FMC Pin	Signal Name	PinHeader(CN2)		Signal Name	FMC Pin
-	GND	1	2	GND	-
H5	CLK0_M2C_N	3	4	CLK0_M2C_P	H4
H8	LA02_N	5	6	LA02_P	H7
H11	LA04_N	7	8	LA04_P	H10
H14	LA07_N	9	10	LA07_P	H13
H17	LA11_N	11	12	LA11_P	H16
H20	LA15_N	13	14	LA15_P	H19
H23	LA19_N	15	16	LA19_P	H22
H26	LA21_N	17	18	LA21_P	H25
H29	LA24_N	19	20	LA24_P	H28
H32	LA28_N	21	22	LA28_P	H31
H35	LA30_N	23	24	LA30_P	H34
H38	LA32_N	25	26	LA32_P	H37
-	GND	27	28	GND	-
-	NC	29	30	NC	-

Figura 45. Asignación de pines de la placa a pines FMC

De la Figura 45, extraída directamente de la documentación de la placa, podemos hacer la asignación de los pines del 5 al 20, que son los empleados por el sensor.

El siguiente paso es, atendiendo al *Master UCF*, hacer la segunda asignación, que es la que nos da el código que hay que utilizar en el archivo *.ucf* del proyecto.

```
# Bank 34
NET FMC_LA02_N    LOC = P18  | IOSTANDARD=LVCMOS18;  # "FMC-LA02_N"
NET FMC_LA02_P    LOC = P17  | IOSTANDARD=LVCMOS18;  # "FMC-LA02_P"

NET FMC_LA04_N    LOC = M22  | IOSTANDARD=LVCMOS18;  # "FMC-LA04_N"
NET FMC_LA04_P    LOC = M21  | IOSTANDARD=LVCMOS18;  # "FMC-LA04_P"

NET FMC_LA07_N    LOC = T17  | IOSTANDARD=LVCMOS18;  # "FMC-LA07_N"
NET FMC_LA07_P    LOC = T16  | IOSTANDARD=LVCMOS18;  # "FMC-LA07_P"

NET FMC_LA11_N    LOC = N18  | IOSTANDARD=LVCMOS18;  # "FMC-LA11_N"
NET FMC_LA11_P    LOC = N17  | IOSTANDARD=LVCMOS18;  # "FMC-LA11_P"

NET FMC_LA15_N    LOC = J17  | IOSTANDARD=LVCMOS18;  # "FMC-LA15_N"
NET FMC_LA15_P    LOC = J16  | IOSTANDARD=LVCMOS18;  # "FMC-LA15_P"

# Bank 35
NET FMC_LA19_N    LOC = G16  | IOSTANDARD=LVCMOS18;  # "FMC-LA19_N"
NET FMC_LA19_P    LOC = G15  | IOSTANDARD=LVCMOS18;  # "FMC-LA19_P"

NET FMC_LA21_N    LOC = E20  | IOSTANDARD=LVCMOS18;  # "FMC-LA21_N"
NET FMC_LA21_P    LOC = E19  | IOSTANDARD=LVCMOS18;  # "FMC-LA21_P"

NET FMC_LA24_N    LOC = A19  | IOSTANDARD=LVCMOS18;  # "FMC-LA24_N"
NET FMC_LA24_P    LOC = A18  | IOSTANDARD=LVCMOS18;  # "FMC-LA24_P"
```

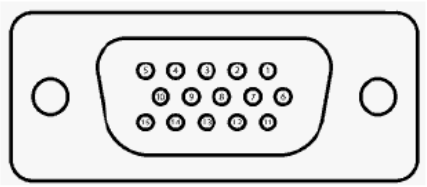
Figura 46. Asignación de pines extraídas del Master UCF de la *ZedBoard*

En la Figura 46 se han extraído del Master UCF los pines empleados en este caso, de forma que ya tenemos asociado cada pin del sensor a su código correspondiente. Recogemos estos códigos en la tabla siguiente:

Puerto	Pin placa	Pin FMC	Código UCF
OV7670_PWDN	5	FMC-LA02_N	**P18**
OV7670_RESET	6	FMC-LA02_P	**P17**
OV7670_DATA<0>	7	FMC-LA04_N	**M22**
OV7670_DATA<1>	8	FMC-LA04_P	**M21**
OV7670_DATA<2>	9	FMC-LA07_N	**T17**
OV7670_DATA<3>	10	FMC-LA07_P	**T16**
OV7670_DATA<4>	11	FMC-LA11_N	**N18**
OV7670_DATA<5>	12	FMC-LA11_P	**N17**
OV7670_DATA<6>	13	FMC-LA15_N	**J17**
OV7670_DATA<7>	14	FMC-LA15_P	**J16**
OV7670_XCLK	15	FMC-LA19_N	**G16**
OV7670_PCLK	16	FMC-LA19_P	**G15**
OV7670_HREF	17	FMC-LA21_N	**E20**
OV7670_VSYNC	18	FMC-LA21_P	**E19**
OV7670_SIOD	19	FMC-LA24_N	**A19**
OV7670_SIOC	20	FMC-LA24_P	**A18**

Tabla 10. Códigos UCF de los pines asociados al sensor

Continuamos el mapeado con las señales de la salida VGA, cuyos códigos UCF se encuentran en la Figura 47, extraída del *datasheet* de la *ZedBoard*.



VGA Pin	Signal	Description	Zynq Pin
1	RED	Red video	V20, U20, V19, V18
2	GREEN	Green video	AB22, AA22, AB21, AA21
3	BLUE	Blue video	Y21, Y20, AB20, AB19
4	ID2/RES	formerly Monitor ID bit 2	NC
5	GND	Ground (HSync)	NC
6	RED_RTN	Red return	NC
7	GREEN_RTN	Green return	NC
8	BLUE_RTN	Blue return	NC
9	KEY/PWR	formerly key	NC
10	GND	Ground (VSync)	NC
11	ID0/RES	formerly Monitor ID bit 0	NC
12	ID1/SDA	formerly Monitor ID bit 1	NC
13	HSync	Horizontal sync	AA19
14	VSync	Vertical sync	Y19
15	ID3/SCL	formerly Monitor ID bit 3	NC

Figura 47. Pines salida VGA

Como vemos, las cuatro componentes del rojo deben ir asociadas a los pines V20, U20, V19 y V18, yendo del menos al más significativo. Las del verde a los pines AB22, AA22, AB21 y AA21 y las del azul a los pines Y21, Y20, AB20 y AB19. Las señales de sincronía horizontal y vertical van a los pines AA19 e Y19, respectivamente. Todo queda recogido en la tabla siguiente.

Puerto	Código UCF	Puerto	Código UCF
VGA_ROJO<0>	V20	VGA_VERDE<3>	AA21
VGA_ROJO<1>	U20	VGA_AZUL<0>	Y21
VGA_ROJO<2>	V19	VGA_AZUL<1>	Y20
VGA_ROJO<3>	V18	VGA_AZUL<2>	AB20
VGA_VERDE<0>	AB22	VGA_AZUL<3>	AB19
VGA_VERDE<1>	AA21	VGA_HSYNC	E20
VGA_VERDE<2>	AB21	VGA_VSYNC	E19

Tabla 11. Códigos UCF de los pines de la salida VGA

El *IOSTANDARD* para estos pines debe ser *LVCMOS33,* de acuerdo con el master UCF.

En cuanto al reloj de entrada del sistema, la *ZedBoard* ofrece un oscilador de 100 MHz que puede ser empleada por la parte de lógica programable (PL). El pin asociado al oscilador, de acuerdo con la documentación de la *ZedBoard*, es el **Y9**.

Por último queda mapear los pines del LED, el botón y el interruptor. Como la *ZedBoard* incluye ocho LEDs, ocho interruptores y cinco botones para la parte de PL, únicamente

tenemos que asociar uno de ellos a las señales de salida de la plataforma, que en este caso han sido el LED *LD0*, el interruptor *SW0* y el botón *BTNU*. Mirando nuevamente el *datasheet* de la *ZedBoard*, los pines asociados son los siguientes:

Puerto	Código UCF
LED	T22
BTN	T18
SW	F22

Tabla 12. Códigos UCF de los pines de los periféricos

Después de conocer el código de cada pin y consultando la alimentación de cada uno en el Master UCF, el mapeado queda así:

```
NET "OV7670_PWDN"       LOC = P18  | IOSTANDARD=LVCMOS18;
NET "OV7670_RESET"      LOC = P17  | IOSTANDARD=LVCMOS18;
NET "OV7670_DATA<0>"    LOC = M22  | IOSTANDARD=LVCMOS18;
NET "OV7670_DATA<1>"    LOC = M21  | IOSTANDARD=LVCMOS18;
NET "OV7670_DATA<2>"    LOC = T17  | IOSTANDARD=LVCMOS18;
NET "OV7670_DATA<3>"    LOC = T16  | IOSTANDARD=LVCMOS18;
NET "OV7670_DATA<4>"    LOC = N18  | IOSTANDARD=LVCMOS18;
NET "OV7670_DATA<5>"    LOC = N17  | IOSTANDARD=LVCMOS18;
NET "OV7670_DATA<6>"    LOC = J17  | IOSTANDARD=LVCMOS18;
NET "OV7670_DATA<7>"    LOC = J16  | IOSTANDARD=LVCMOS18;
NET "OV7670_XCLK"       LOC = G16  | IOSTANDARD=LVCMOS18;
NET "OV7670_PCLK"       LOC = G15  | IOSTANDARD=LVCMOS18;
NET "OV7670_PCLK"       CLOCK_DEDICATED_ROUTE = FALSE;
NET "OV7670_HREF"       LOC = E20  | IOSTANDARD=LVCMOS18;
NET "OV7670_VSYNC"      LOC = E19  | IOSTANDARD=LVCMOS18;
NET "OV7670_SIOD"       LOC = A19  | IOSTANDARD=LVCMOS18;
NET "OV7670_SIOC"       LOC = A18  | IOSTANDARD=LVCMOS18;

NET "CLK100"            LOC = Y9   |IOSTANDARD=LVTTL;

NET VGA_AZUL<0>         LOC = Y21  | IOSTANDARD=LVCMOS33;
NET VGA_AZUL<1>         LOC = Y20  | IOSTANDARD=LVCMOS33;
NET VGA_AZUL<2>         LOC = AB20 | IOSTANDARD=LVCMOS33;
NET VGA_AZUL<3>         LOC = AB19 | IOSTANDARD=LVCMOS33;
NET VGA_VERDE<0>        LOC = AB22 | IOSTANDARD=LVCMOS33;
NET VGA_VERDE<1>        LOC = AA22 | IOSTANDARD=LVCMOS33;
NET VGA_VERDE<2>        LOC = AB21 | IOSTANDARD=LVCMOS33;
NET VGA_VERDE<3>        LOC = AA21 | IOSTANDARD=LVCMOS33;
NET VGA_ROJO<0>         LOC = V20  | IOSTANDARD=LVCMOS33;
NET VGA_ROJO<1>         LOC = U20  | IOSTANDARD=LVCMOS33;
NET VGA_ROJO<2>         LOC = V19  | IOSTANDARD=LVCMOS33;
NET VGA_ROJO<3>         LOC = V18  | IOSTANDARD=LVCMOS33;
NET VGA_HSYNC           LOC = AA19 | IOSTANDARD=LVCMOS33;
NET VGA_VSYNC           LOC = Y19  | IOSTANDARD=LVCMOS33;

NET "LED"               LOC = "T22" | IOSTANDARD=LVTTL;
NET "BTN"               LOC = "T18" | IOSTANDARD=LVCMOS18;
NET "SW"                LOC = "F22" | IOSTANDARD=LVCMOS18;
```

Código 16. UCF de la plataforma

Aclarar que empleamos la línea "`NET "OV7670_PCLK" CLOCK_DEDICATED_ROUTE = FALSE`" para que la *ZedBoard* admita PCLK como señal de reloj.

6.1.3 *Implementación VHDL de la plataforma*

Como hemos comentado anteriormente, para interconectar todos los módulos diseñados anteriormente nos aprovechamos de la programación estructural que ofrece VHDL, instanciando cada módulo como un componente y creando las señales necesarias para las conexiones.

Antes de comenzar a explicar las conexiones, comentar que se añaden a la plataforma dos módulos más, uno para realizar la división de frecuencia del reloj, recordemos que VGA emplea un reloj de 25 MHz y el reloj de entrada del sistema es de 100 MHz; y otro, de menor importancia, para evitar que al presionar el botón de reconfiguración se produzcan rebotes y la *ZedBoard* no interprete bien la señal.

El funcionamiento del divisor de frecuencia es muy sencillo, se basa en una señal que varíe únicamente con los flancos ascendentes de la señal de reloj original, lo que genera un reloj con la mitad de frecuencia que la original. Para obtener una señal de reloj de un cuarto de frecuencia de la señal original, repetimos el proceso anterior pero esta vez la señal varía con los flancos ascendentes de la señal generada con la mitad de frecuencia. Señalar que es necesario emplear dos señales auxiliares, una para cada reloj de salida, ya que el lenguaje no permite dar a una variable el valor de un puerto de salida, por lo que la sentencia "`clk50 <= not clk50`" lanzaría un error.

De esta forma podemos obtener un reloj de 50 y 25 MHz a partir del de 100 MHz. En este caso, aunque no es necesario el reloj de 50 MHz, dejaremos como salidas las dos señales de reloj, por si fuese útil en futuras aplicaciones.

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

entity Clock is
    Port ( clk100 : in  STD_LOGIC;
           clk50 : out  STD_LOGIC;
           clk25 : out  STD_LOGIC);
end Clock;
architecture Behavioral of Clock is

begin
clk50 <= clk50a;
clk25 <= clk25a;
  process(clk100)
  begin
     if rising_edge(clk100) then
       clk50a <= not clk50a;
       if rising_edge(clk50a) then
         clk25a <= not clk25a;
       end if;
     end if;
  end process;
end Behavioral;
```

Código 17. Módulo divisor de frecuencia

El módulo *antirrebote* consiste en hacer una espera cuando se detecta un nivel alto en la entrada y, pasada la espera, activar la salida para enviar la señal al módulo correspondiente. La entrada irá conectada al botón y la salida a la entrada de reconfiguración habilitada en el módulo de control. La pausa generada es de 0.671 segundos, suficiente para evitar rebotes indeseados.

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

entity antirrebote is
    Port ( clk : in  STD_LOGIC;
           i : in  STD_LOGIC;
           o : out  STD_LOGIC);
end antirrebote;

architecture Behavioral of antirrebote is

begin
process(clk)
  begin
     if rising_edge(clk) then
        if i = '1' then
           if contador = x"FFFFFF" then
            o <= '1';
          else
            o <= '0';
          end if;
          contador <= contador+1;
       else
          contador <= (others => '0');
          o <= '0';
       end if;
     end if;
  end process;
end Behavioral;
```

Código 18. Módulo antirrebote

Una vez definidos estos dos módulos podemos comenzar a explicar las conexiones necesarias para que todo funcione correctamente. En primer lugar debemos declarar cada uno de los bloques implementados con el comando COMPONENT. Esta sentencia permite definir un subsistema que luego puede reutilizarse instanciando las copias del mismo que sean necesarias, sólo especificando su conexión con el resto de componentes, puesto que su funcionamiento interno ya ha sido definido. A continuación vemos un ejemplo con el bloque de captura de píxeles:

```
COMPONENT ov7670_captura_pixel
PORT(
  pclk  : IN std_logic;
  vsync : IN std_logic;
  href  : IN std_logic;
  data  : IN std_logic_vector(7 downto 0);
  addr  : OUT std_logic_vector(18 downto 0);
  dout  : OUT std_logic_vector(11 downto 0);
  we    : OUT std_logic);
END COMPONENT;
```

Código 19. Declaración del módulo ov7670_captura_pixel

Declaramos cada módulo, indicando su interfaz, y ya podemos instanciarlos las veces que sea necesario, aunque para esta plataforma sólo necesitaremos un bloque de cada tipo.

Siguiendo el ejemplo anterior, comenzamos con la instanciación del módulo *ov7670_captura_pixel*, cuyas entradas van conectadas directamente a los pines del sensor, PCLK, VSYNC, HREF y DATA, mientras que sus salidas van conectadas a las entradas del *buffer* para cada píxel quede almacenado. Para instanciar un componente previamente declarado e indicar su interfase, se emplea la sentencia PORTMAP. De esta forma asignamos cada puerto del componente a una señal interna o a un puerto de la plataforma. La instanciación de este módulo sería la siguiente:

```
mod_captura: ov7670_captura_pixel
PORT MAP(
   pclk  => OV7670_PCLK,
   vsync => OV7670_VSYNC,
   href  => OV7670_HREF,
   data  => OV7670_DATA,
   addr  => buf_addr,
   dout  => buf_data,
   we    => buf_we(0)
);
```

Código 20. Instanciación del módulo ov7670_captura_pixel

Las señales *buf_addr*, *buf_data* y *buf_we* son las que conectan las salidas con las entradas del *buffer.* Señalar que hay que declarar la señal *buf_we* como "*std_logic_vector(0 downto 0)"* porque así es como la define automáticamente la herramienta *LogiCORE* al crear el *buffer,* por lo que declararla como "*std_logic",* que parece lo evidente, lanza un error de incompatibilidad de tipo de datos.

Siguiendo con el *buffer,* lo instanciamos teniendo en cuenta que el reloj de escritura tiene que ser el mismo que el módulo de captura de píxeles, PCLK, para que estén sincronizados y se escriba en memoria en el tiempo correcto. El reloj de lectura, en cambio, tiene que ser el mismo que el del módulo VGA, 25 MHz, y la salida de datos de memoria va al pin de entrada del módulo VGA, que leerá el píxel para mostrarlo en pantalla. La dirección de memoria de lectura queda controlada por el módulo *VGA*, como se comenta en el apartado 5.

```
buf_imag : buffer_imagen
PORT MAP (
  clka  => OV7670_PCLK,
  wea   => buf_we,
  addra => buf_addr,
  dina  => buf_data,

  clkb  => clk25,
  addrb => lectura_addr,
  doutb => lectura_pixel
);
```

Código 21. Instanciación del *buffer*

Para visualizar la imagen en pantalla instanciamos el módulo VGA, haciendo las conexiones directas con los puertos de salida VGA de la plataforma y conectando al *buffer* el puerto de entrada de información de color, a través de la señal *lectura_pixel,* así como el que indica la dirección de memoria de lectura, señal *lectura_addr*. El reloj de entrada es el de 25 MHz y la señal de modo suspensión queda conectada directamente al *switch*.

```
modulo_VGA : VGA
PORT MAP(
  clk25      => clk25,
  pixel_in   => lectura_pixel,
  pixel_addr => lectura_addr,

  VGA_ROJO   => VGA_ROJO,
  VGA_VERDE  => VGA_VERDE,
  VGA_AZUL   => VGA_AZUL,
  VGA_HSYNC  => VGA_HSYNC,
  VGA_VSYNC  => VGA_VSYNC,

  susp       =>  SW
);
```

Código 22. Instanciación del módulo VGA

El módulo de configuración del sensor es el siguiente que nos ocupa, aunque su instanciación en el diseño final es bastante sencilla. Los puertos SIOC, SIOD, PWDN, RESET y XCLK van conectados directamente al sensor, mientras que el puerto *resend* se conecta al botón de reconfiguración a través del módulo *antirrebote*; el puerto que controla la suspensión del sensor se conecta al interruptor y la señal que indica que la configuración ha finalizado, al LED. El reloj de entrada al módulo, de acuerdo con los cálculos realizados en el apartado 6.3, es el de 25 MHz, que está asociado a la señal *clk25.*

```
controlador: ov7670_controlador
PORT MAP(
  clk   => clk25,
  sioc  => OV7670_SIOC,
  resend => resend,
  susp  => SW,
  config_finished => LED,
  siod  => OV7670_SIOD,
  pwdn  => OV7670_PWDN,
  reset => OV7670_RESET,
  xclk  => OV7670_XCLK
);
```

Código 23. Instanciación del módulo ov7670_controlador

Por último, señalamos la instanciación de los módulos divisor de frecuencia y *antirrebote*:

```
mod_divisor : divisor
PORT MAP(
  clk100 => clk100,
  clk25 => clk25
);
```

Código 24. Instanciación del divisor de frecuencia

```
mod_antirrebote: antirrebote
PORT MAP(
  clk => clk25,
  i   => BTN,
  o   => resend
);
```

Código 25. Instanciación del módulo antirrebote

Para tener una visión general de la plataforma, en la página siguiente se muestra un esquema con todos los bloques presentes en ella interconectados:

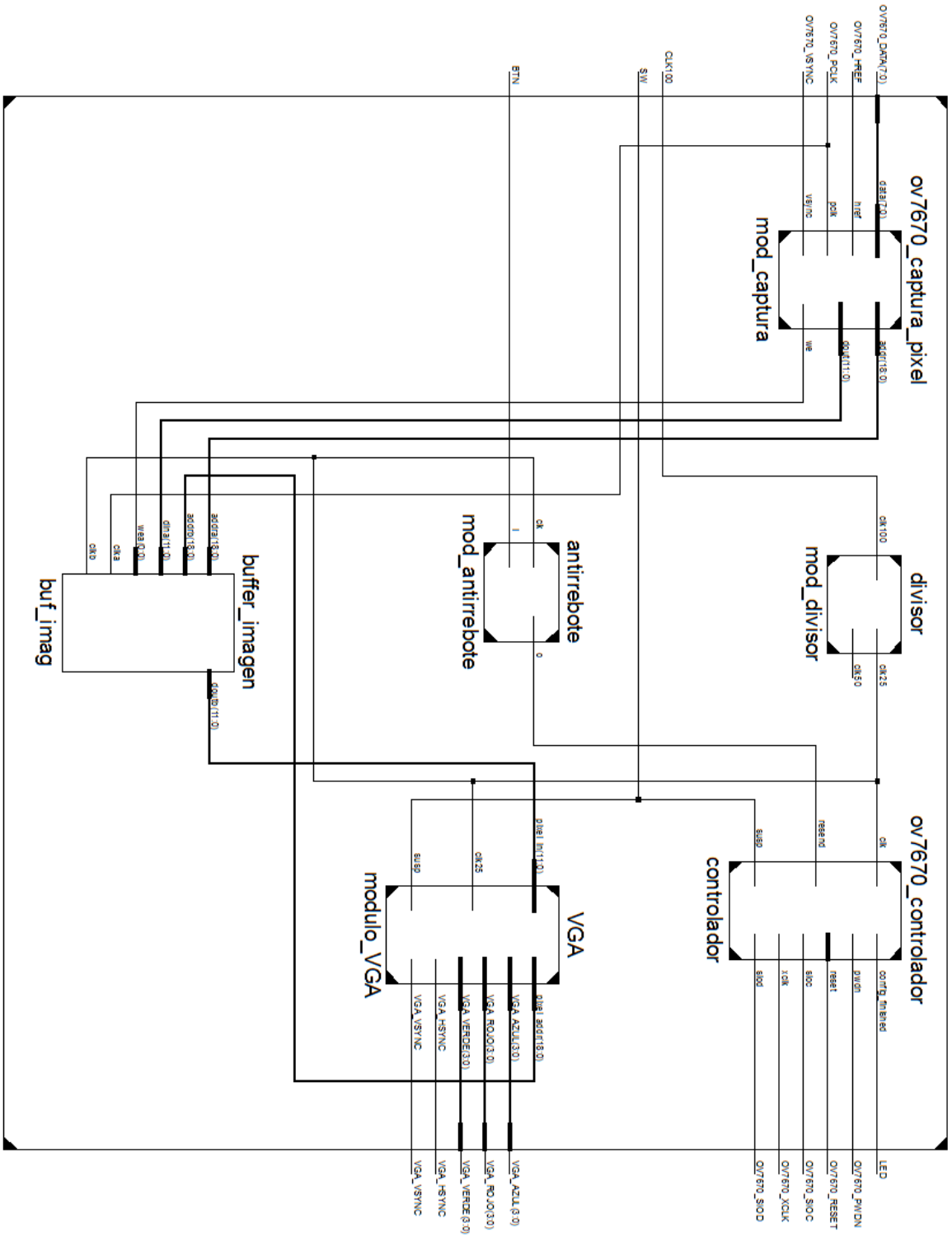


Figura 48. Esquema general de la plataforma

6.2 Pruebas y resultados

Una vez realizado el diseño de la plataforma procedemos a ponerlo en funcionamiento y comprobar que todo funciona correctamente. En este momento hay que poner especial atención en configurar correctamente el sensor, ya que cualquier valor erróneo en un registro puede provocar que la imagen no se visualice correctamente. En el desarrollo del proyecto se han realizado multitud de pruebas con distintos registros, recordemos que este sensor cuenta con aproximadamente 200 registros configurables, pero en este apartado sólo recogeremos las más relevantes y que mejores resultados han arrojado.

En primer lugar, conviene señalar los registros más relevantes, explicando brevemente su función y los valores que deben tomar de acuerdo con las funcionalidades que queramos activar.

Dirección (Hex)	Nombre	Función
00	GAIN	Controla el nivel de ganancia manual del algoritmo AGC
0C	COM3	Activa el escalado de imagen
11	CLKRC	Escala PCLK
12	COM7	Cambia el formato de color, la resolución y permite hacer un *reset* a nivel software
13	COM8	Activa o desactiva los algoritmos AGC, AEC y AWB
14	COM9	Controla el nivel de ganancia automático de AGC, permite congelar los algoritmos AGC y AEC
15	COM10	Cambia el formato de HREF, PCLK y VSYNC
1E	MVFP	Activa el modo espejo y el modo *VFlip*
3A	TSLB	Cambia la secuencia del formato YUV, junto con COM13
3B	COM11	Activa el modo nocturno (baja la tasa de bits enviados)
3D	COM13	Activa el control de gamma, la saturación automática y cambia la secuencia del formato YUV, junto con TSLB
3E	COM14	Activa el escalado manual de imagen y añade un escalado adicional a PCLK
40	COM15	Cambia el formato RGB (RGB555 o RGB565)
55	BRIGHT	Controla el brillo de la imagen
56	CONTRAS	Controla el contraste de la imagen
58	MTXS	Activa el modo de contraste automático
6B	DBLV	Multiplica la frecuencia de PCLK por un factor configurable múltiplo de 2

Tabla 13. Registros más relevantes

Para las primeras pruebas únicamente nos centramos en los registros que controlan el formato de color, el PCLK y las demás señales de sincronía.

Comenzamos con las pruebas en formato RGB565, por lo que empleamos el módulo de captura de píxeles adecuado para él. Para escoger el formato de color, debemos configurar los registros COM7 y COM15. Para dar al registro el valor correcto nos referimos al *datasheet* del sensor, donde aparece la función de cada uno de los ocho bits.

Dirección (Hex)	Nombre	Valor por defecto	Descripción
11	CLKRC	80	**Bit[7]:** Digital PLL option 0: Disable double clock option, meaning the maximum PCLK can be as high as half input clock 1: Enable double clock option, meaning the maximum PCLK can be as high as input clock **Bit[6]:** Use external clock directly (no clock pre-scale available) **Bit[5:0]:** Internal clock pre-scalar F(internal clock) = F(input clock)/(Bit[5:0]+1) • Range: [0 0000] to [1 1111]
40	COM15	C0	**Bit[7:6]:** Data format - output full range enable 0x: Output range: [10] to [F0] 10: Output range: [01] to [FE] 11: Output range: [00] to [FF] **Bit[5:4]:** RGB 555/565 option (must set COM7[2] = 1 and COM7[0] = 0) x0: Normal RGB output 01: RGB 565 11: RGB 555 **Bit[3:0]:** Reserved
6B	DBLV	3A	**Bit[7:6]:** PLL control 00: Bypass PLL 01: Input clock x4 10: Input clock x8 11: Input clock x16 **Bit[5]:** Reserved **Bit[4]:** Regulator control 0: Enable internal regulator 1: Bypass internal regulator **Bit[3:0]:** Clock divider control for DSP scale control (valid only when COM14[3] = 1)
12	COM7		**Bit[7]:** SCCB Register Reset 0: No change 1: Resets all registers to default values **Bit[6]:** Reserved **Bit[5]:** Output format - CIF selection **Bit[4]:** Output format - QVGA selection **Bit[3]:** Output format - QCIF selection **Bit[2]:** Output format - RGB selection (see below) **Bit[1]:** Color bar

0: Disable
1: Enable
Bit[0]: Output format - Raw RGB (see below)

	COM7[2]	COM7[0]
YUV	0	0
RGB	1	0
Bayer RAW	0	1
Processed Bayer RAW	1	1

Tabla 14. Configuración de los registros más importantes (extraída del *datasheet*)

Los valores que tenemos que dar a los registros COM7 y COM15 son '00000100' y '11010000' respectivamente. La resolución escogida es la del estándar VGA, de ahí que desactivemos las demás opciones en COM7. El rango de datos configurable en el registro COM15 lo establecemos de 00 a FF, aunque cambiar este rango, después de hacer distintas pruebas, no es relevante en la calidad de imagen. Para todas las pruebas con el formato RGB565 dejaremos estos registros con estos valores.

Con los registros CLKRC y DBLV se puede controlar la frecuencia de PCLK, lo que supone variar la tasa de bits por segundo que envía el sensor. Variando estos registros podemos cambiar la tasa de imágenes por segundo que se muestran en pantalla. La salida VGA va a mantener las 30 imágenes por segundo, tasa para la que está preparado el monitor, gracias al *buffer* implementado, por lo que podemos variar estos valores sin temer que al monitor le llegue más tasa de refresco de la que soporta. Al realizar las divisiones y multiplicaciones de frecuencia que nos permite el PLL del sensor, partimos de la señal de entrada al sistema, pin XCLK, que es de 25 MHz.

Para ver la frecuencia a la que trabaja PCLK y comprobar que funciona correctamente, sin desestabilizaciones, debemos utilizar la herramienta *ChipScope*, muy útil para ver las señales internas del diseño sin necesidad de emplear equipamiento extra. En el apéndice se explica su funcionamiento.

Partimos de una configuración de PCLK sin variaciones de frecuencia, con un divisor igual a cero (CLKRC = '01000000') y habilitando la opción de aplicar directamente el reloj externo (bit 6).

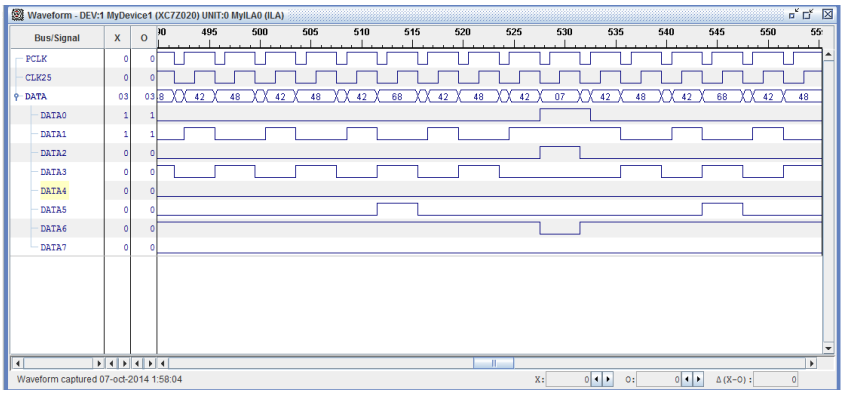


Figura 49. Captura PCLK con PLL desactivado

En la Figura 49, observamos que el PCLK es una señal de reloj estable a 25 MHz, igual que el reloj de entrada, aunque el ciclo de trabajo de la señal es del 25%. Esto no es relevante para el funcionamiento del sistema, ya que la lectura se hace únicamente en los flancos de subida. Las entradas de datos en esta captura se encuentras desactivadas.

Para comprobar que el divisor configurable en el registro CLKRC funciona de acuerdo a lo especificado en la documentación, le damos un valor dejando el PLL de nuevo desactivado. Configuramos CLKRC para que el divisor divida el reloj por un factor 8 (bits de 5 a 0 con un valor igual a 7). Para desactivar el PLL dejamos a los bits 7 y 6 del registro DBLV a '0'.

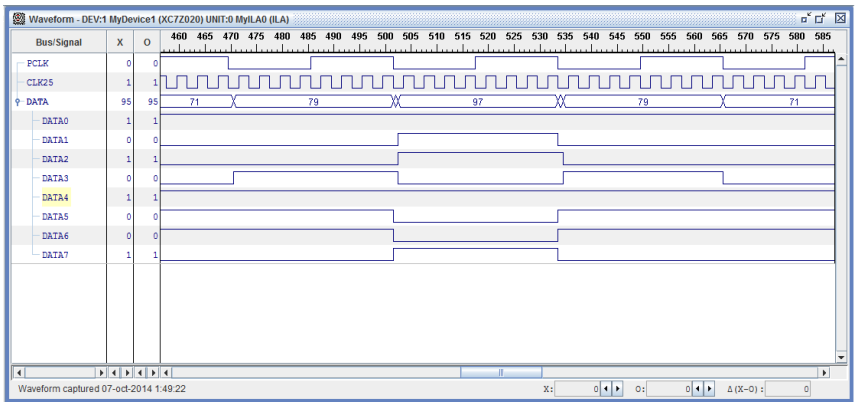


Figura 50. Captura PCLK con divisor por 8

Vemos en la Figura 50 que PCLK tiene una frecuencia ocho veces menor que clk25, por lo que el divisor sigue el comportamiento esperado, siendo PCLK completamente estable. Esta baja tasa de refresco hace que la grabación de video tenga muy poco peso, siendo útil en aplicaciones donde se desee almacenar video durante mucho tiempo, por ejemplo en tareas de vigilancia.

Comprobado que PCLK es estable a baja frecuencia, incrementamos la frecuencia activando el PLL interno (bits 7 y 6 de DBLV). Cuanto mayor sea la tasa de refresco que da el sensor, más fluida será la imagen mostrada en pantalla, ya que los cambios de color en cada píxel de una imagen a la siguiente serán menores.

Los cambios de frecuencia son configurables con los valores del divisor y el PLL, dando multitud de combinaciones posibles. Tras varias pruebas variando uno y otro, se llega a un límite en el que PCLK se desestabiliza y la imagen se distorsiona. Para no hacer demasiado tediosa la presentación de resultados para el lector, no se van a analizar una a una todas las pruebas realizadas, simplemente decir que la frecuencia máxima obtenida sin desestabilizar el sensor es 33.3 MHz. (bits de 5 a 0 en CLKRC con valor 2 y bits 7 y 6 de DBLV a '1'). Un ejemplo de cómo se desestabiliza el sensor cuando el factor es demasiado alto se muestra en la Figura 51, donde se aprecia como PCLK deja de ser periódico, distorsionando totalmente la imagen mostrada en pantalla.

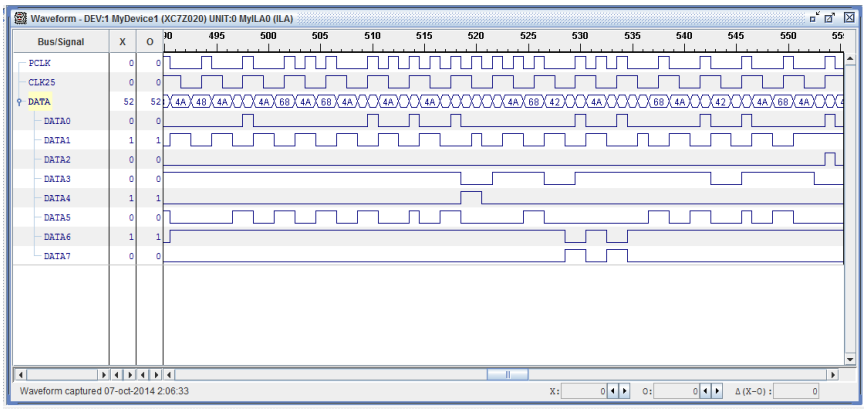


Figura 51. Captura PCLK inestable

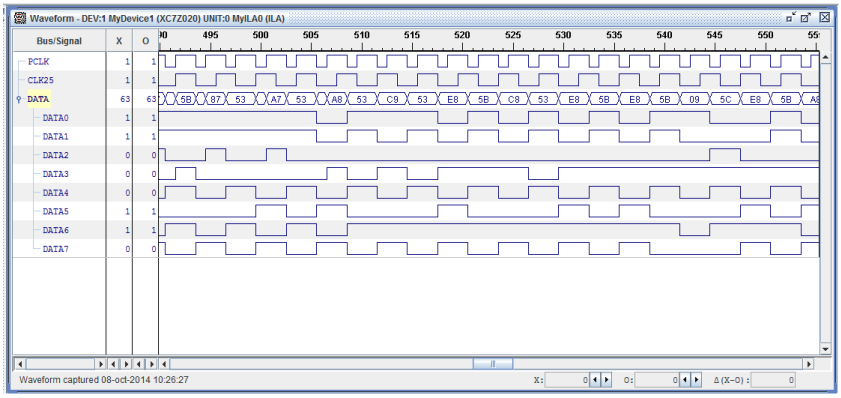


Figura 52. PCLK estable

Con los registros CLKRC y DBLV configurados para que la imagen no se desestabilice, COM7 y COM15 para que el sensor trabaje con el formato de color RGB565 y el registro COM14 con el valor x"00" para desactivar el escalado de PCLK, ya tenemos una configuración básica para que se muestre imagen correctamente en el monitor en tiempo real. La configuración de registros en el módulo *SCCB_registers* para esta prueba de imagen es la siguiente:

```
when x"00" => sreg <= x"1280"; -- Reset de software
when x"01" => sreg <= x"1204"; -- COM7
when x"02" => sreg <= x"40D0"; -- COM15
when x"03" => sreg <= x"6BCA"; -- DBLV
when x"04" => sreg <= x"1102"; -- CLKRC
when x"05" => sreg <= x"3E00"; -- COM14
when x"06" => sreg <= x"B084";
```

Código 26. Configuración de registros en *SCCB_registers*

El primer dato que se escribe en el sensor es el que activa el bit 7 del registro COM7 para hacer un *reset*, ya que el fabricante recomienda hacer un resetear el sensor, ya sea a nivel de software o hardware, antes de cada configuración. Después del *reset*, se envían al sensor los datos de la configuración deseada.

El registro B0 aparece como registro reservado en el *datasheet*, pero es necesaria su configuración por lo que parece un fallo de fabricación en el sensor. Si este registro se deja con su valor por defecto, el sensor no trabaja adecuadamente los colores, obteniéndose una calidad de imagen bastante pobre y con colores diferentes a los reales, como aparece en la Figura 53.

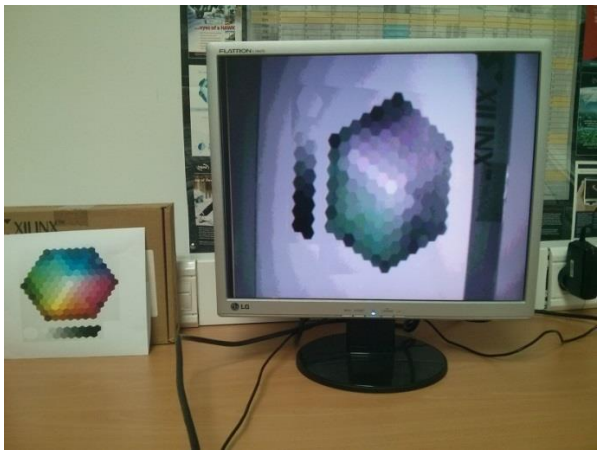


Figura 53. Prueba con registro B0 sin configurar

Investigando la causa de este fallo y tras intentar contactar con el fabricante, que no dio respuesta, un usuario de un foro de desarrollo de aplicaciones con FPGAs avisó de la existencia de este registro y de su valor adecuado, solucionando el problema.

Cargamos la plataforma diseñada en la *ZedBoard* y el sensor comienza a trabajar, mostrando por pantalla imagen en tiempo real con formato RGB565. Se ha optado por enseñar por pantalla una paleta de colores impresa en papel, ya que contiene todos los colores y así podemos ver cómo trabaja el sensor con toda la gama cromática. El sensor y, por tanto, la imagen, están volteados 90 grados porque era más cómodo a la hora de dejar el sensor fijo en una posición.

Un ejemplo de la imagen que se visualiza en el monitor se muestra en la Figura 54:

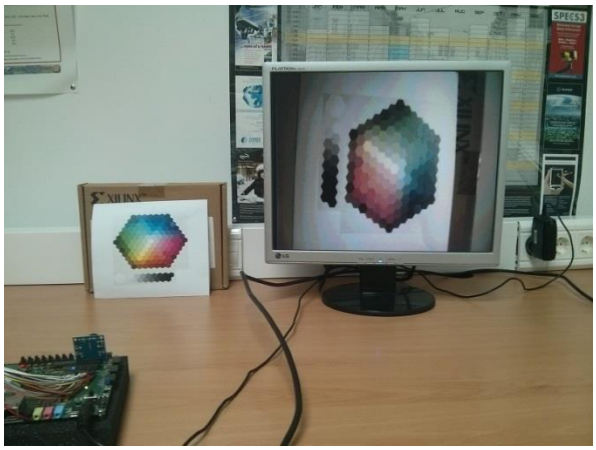


Figura 54. Prueba RGB565

Vemos que la calidad de imagen es bastante aceptable teniendo en cuenta que la paleta de colores con la que se trabaja es bastante limitada, como veíamos en el apartado 5.5. Al activar el interruptor de suspensión la imagen queda de color negro, volviendo al modo normal cuando lo desactivamos.

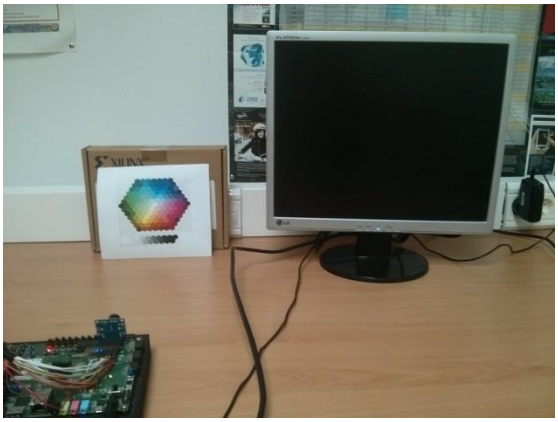


Figura 55. Modo suspensión

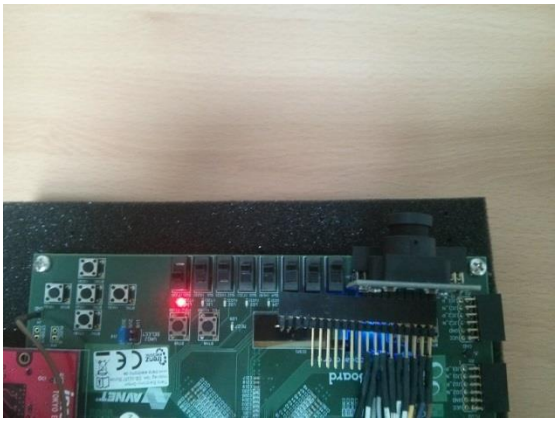


Figura 56. Interruptor activado para modo suspensión

La siguiente prueba es trabajando con formato RGB555, para lo que tenemos que cambiar el módulo de adquisición de píxeles y la configuración del registro COM15, que ahora pasa a valer '11110000' (bits 5 y 4 a '1' para activar el formato RGB555). Los demás registros se mantienen igual.

Figura 57. Prueba RGB555

Pese a que la componente verde en este caso tiene menos peso y cada píxel ha pasado a tener de 16 a 15 bits de información, vemos que la calidad de imagen es muy parecida a la obtenida con RGB565. Esto es consecuencia del truncamiento de bits, con la pérdida de información que conlleva, al pasar a RGB444.

Visto el formato RGB, pasamos a YUV422. Se cambia de nuevo el módulo de adquisición de píxeles por el descrito en el apartado 4.3.3, además de añadir un módulo de conversión de color YUV a RGB. El valor del registro COM7 pasa a ser '00000000' para activar el formato YUV. Dos registros a tener en cuenta son COM13 y TSLB, que determinan la secuencia con la que el sensor manda cada canal. En este caso, trabajamos con la secuencia YUYV, por lo que se deja desactivado el bit 1 de COM13 y el 3 de TSLB. Las demás funcionalidades para YUV configurables en estos registros se dejan en sus valores por defecto. Tras hacer estos cambios, la imagen obtenida por pantalla es la que aparece en la Figura 58.

Figura 58. Prueba YUV422

Vemos que el tratamiento de los canales y la conversión a RGB se realizan correctamente, obteniéndose buena calidad de imagen, muy similar a la obtenida con RGB565 y RGB555, aunque con los colores un poco más vivos. De nuevo, el factor limitante a la hora de obtener unos colores más reales es la resolución de color que ofrece la salida VGA.

Una vez comprobado que el sensor funciona correctamente utilizando una configuración básica con los tres formatos de color, podemos jugar un poco con los demás parámetros para ver cómo varía la imagen con cada uno:

- Cambio de contraste manual, registro CONTRAS. En función del valor dado sube o baja el contraste de la imagen. Para habilitar este registro debemos desactivar la opción de auto-contraste (bit 7 del registro MTXS).

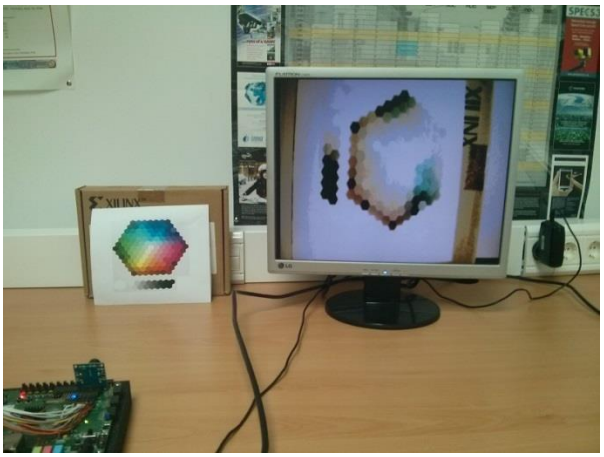


Figura 59. Prueba con máximo contraste

- Cambio de brillo, registro BRIGHT. Funciona igual que el contraste, damos el valor máximo y mínimo para ver el brillo máximo y mínimo de la imagen.

Figura 60. Prueba con máximo brillo

- Algoritmos AGC (Auto Gain Control) y AEC (Auto Exposure Control): controlan automáticamente el tiempo de exposición y la ganancia aplicada a la imagen, lo que supone cambios del brillo. Están activados por defecto, haciendo que el sensor se reconfigure automáticamente en tiempo real en función de la luz captada. Los desactivamos para comprobar cómo varía la calidad de imagen y si el sensor responde más rápido a los cambios de luz y de color.

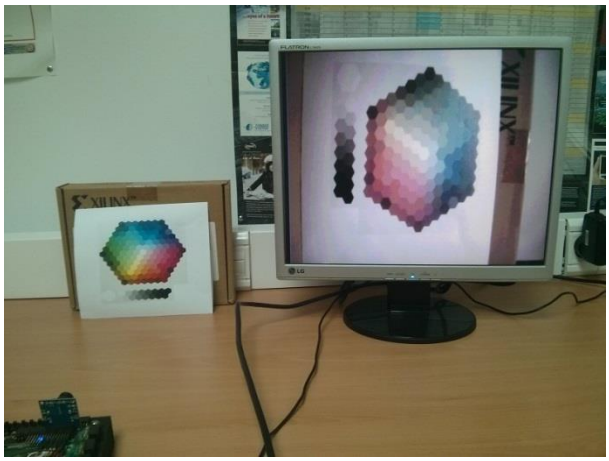


Figura 61. Prueba con AGC y AEC desactivados

La diferencia básica que se observa con esta configuración es que el sensor no cambia automáticamente el brillo de la imagen cuando hay un cambio de luz significativo, siempre se mantiene el mismo nivel de brillo. Como trabajamos en un entorno con bastante luz, el sensor capta mucha y los colores se aprecian más pálidos.

- Modos espejo y *VFlip*: El modo espejo presenta la imagen invertida sobre el eje vertical y el modo Vflip la invierte sobre el eje horizontal. Se pueden activar en el registro MVFP (bits 5 y 4, respectivamente).

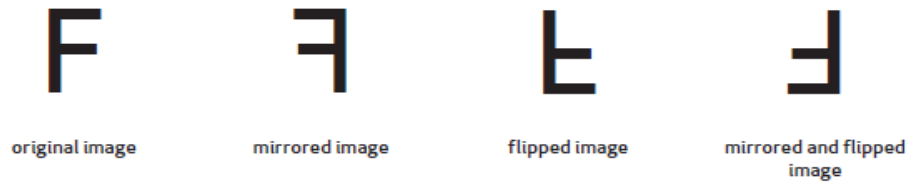


Figura 62. Efecto de los modos espejo y *VFlip*

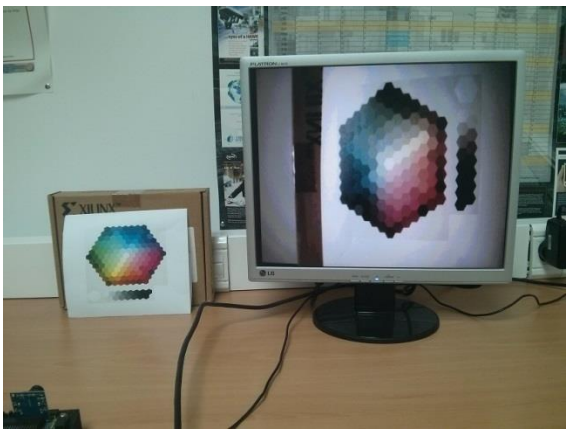


Figura 63. Prueba con modo espejo

Figura 64. Prueba con *VFlip*

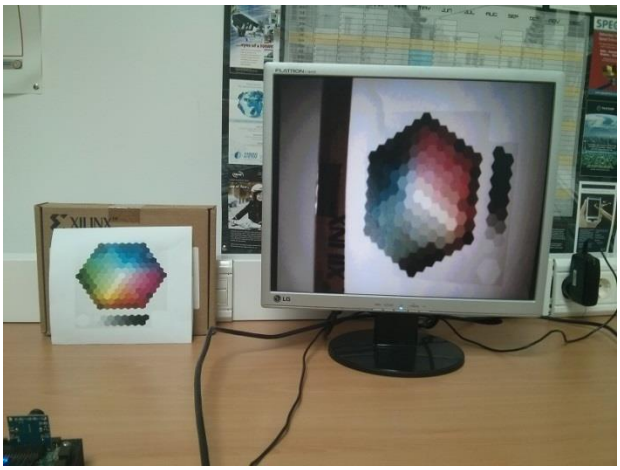


Figura 65. Prueba con modo espejo y *VFlip*

Además de estos parámetros, se han realizado pruebas con otros registros, como el nivel de saturación automático (bit 6 del registro COM13); configurar los niveles de gamma manualmente (por defecto se controlan automáticamente), variando las curvas de gamma (registros de 0x7A a 0x89); o variar manualmente las matrices de color (registros de 0x4F a 0x54), pero ninguna ha dado resultados favorables, empeorando en todas ellas la calidad de imagen mostrada, por lo que no se hará mayor hincapié en ellas.

6.3 Conclusiones

A lo largo del proyecto, se ha desarrollado una plataforma de procesado de vídeo en tiempo real en una FPGA empleando el sensor de imagen OV7670 de *OmniVision*, aunque uno de los puntos fuertes del diseño estructural de la plataforma es que la mayoría de bloques son reutilizables para otros dispositivos del mismo fabricante.

Este proyecto también ha servido para familiarizarse con el lenguaje VHDL y conocer las ventajas que ofrece, así como con el software de *Xilinx*, uno de los fabricantes de FPGAs más importantes del mundo.

Uno de los factores limitantes de la calidad de imagen obtenida ha sido la resolución de color que ofrece la salida VGA, por lo que una propuesta para trabajos futuros que mejoraría la calidad de vídeo sería emplear la salida HDMI que ofrece la ZedBoard, de forma que se puedan aprovechar todos los bits de información de cada píxel. En este caso, para poder almacenar una imagen completa tendríamos que hacer uso de la RAM DDR3, ya que la Block RAM no sería lo suficientemente amplia.

Otro aspecto en el que trabajar para mejorar la calidad, sería emplear otro de los sensores de imagen que ofrece *OmniVision,* como por ejemplo el OV2643, que cuenta con 2 MP y una resolución de 1280x720 píxeles a 30 fps, lo que combinado con el uso de la salida HDMI puede ofrecer una calidad de imagen excelente.

Capítulo 7: Bibliografía

Usado durante todo el proyecto

(1) Toledo Moreo, F. Javier; Garrigós Guerrero, F. Javier; Martínez Álvarez, J. Javier. *Síntesis de Sistemas Digitales con VHDL*. Universidad Politécnica de Cartagena, Septiembre 2003.

Capítulo 1

(2) http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

(3) http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

(4) http://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf

(5) http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

(6) http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm

(7) http://www.xilinx.com/products/intellectual-property/chipscope_ila.htm

(8) http://elecrom.wordpress.com/2010/03/24/xilinx-chipscope-tutorial/

(9) http://es.edaboard.com/topic-5111932.0.html

(10) http://www.xilinx.com/tools/coregen.htm

(11) http://www.xilinx.com/tools/isim.htm

Capítulo 2

(12) http://www.techtoys.com.hk/Components/OmniVision%20Cameras/Schematics/OV_CMOS_PCB25022013.pdf

(13) http://www.voti.nl/docs/OV7670.pdf

(14) http://files.virt2real.ru/docs/OV2643.pdf

(15) http://www.wvshare.com/product/OV7670-Camera-Board.htm

(16) http://developer.mbed.org/media/uploads/edodm85/ov7670_fifo_sch_v2.pdf

(17) http://solutions.inrevium.com/products/pdf/pdf_TB-FMCL-PH_HWUserManual_02_01e.pdf

(18) http://solutions.inrevium.com/products/fmc/peripherals_debug/

Capítulo 3

(19) http://es.wikibooks.org

(20) http://es.wikipedia.org/wiki/YCbCr
(21) http://es.wikipedia.org/wiki/Mosaico_de_Bayer
(22) https://thinksmallthings.wordpress.com
(23) http://www.xilinx.com/support/documentation/application_notes/xapp931.pdf
(24) http://calibracionhd.wordpress.com/2013/05/18/como-elegir-un-espacio-de-color-es-mejor-rgb-o-ycbcr-bt-709-vs-bt-601/

Capítulo 4

(25) http://www.ovt.com/download_document.php?type=document&DID=63

Capítulo 5

(26) http://es.wikipedia.org/wiki/Video_Graphics_Array

Capítulo 6

(27) http://forums.xilinx.com/t5/Virtex-Family-FPGAs/LVTTL-LVCMOS33-and-3-3V/td-p/35010
(28) http://www.xilinx.com/support/documentation/university/ISE-Teaching/HDL-Design/14x/Nexys3/Supporting%20Materials/Nexys3_Master.ucf
(29) http://forums.xilinx.com/t5/Connectivity/Transceiver-core-with-undefined-IOSTANDARD-inputs-and/td-p/248822
(30) http://forums.xilinx.com/t5/Virtex-Family-FPGAs/quot-CLOCK-DEDICATED-ROUTE-FALSE-quot-constraint-seems-to-be-not/td-p/379933
(31) http://www.xilinx.com/support/answers/35453.html
(32) http://forums.xilinx.com/t5/Design-Entry/Clocking-Wizard-v3-6-Inverted-clocks/m-p/344849
(33) http://developer.mbed.org
(34) http://embeddedprogrammer.blogspot.com.es
(35) http://forum.arduino.cc

Apéndice: Uso de la herramienta IP ChipScope™

Para utilizar *ChipScope*, hay que crear un nuevo archivo específico en formato .cdc y añadirlo al proyecto. Esto podemos hacerlo directamente desde ISE, seleccionando la opción *New Source* al clicar sobre el nivel jerárquico más alto de nuestro proyecto. En la ventana que aparece seleccionamos la opción *ChipScope Definition and Connection File*, como vemos en la Figura 66, y le damos un nombre, por ejemplo *DataMonitor*.

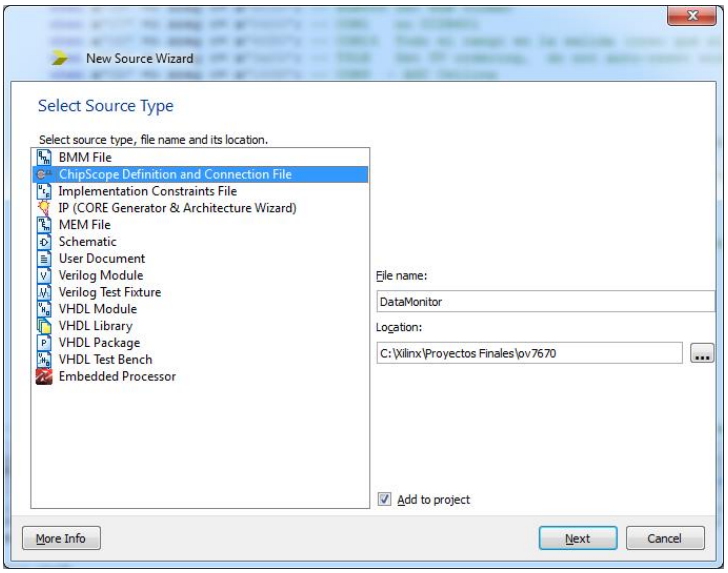


Figura 66. Seleccionar tipo de archivo

Clicamos en *Next* y en la siguiente ventana, donde hace un repaso del módulo que se va a crear, en *Finish.* A continuación se crea el archivo y seleccionamos el *ILA* (Integrated Logic Analyzer) que hemos creado, apareciendo la siguiente ventana:

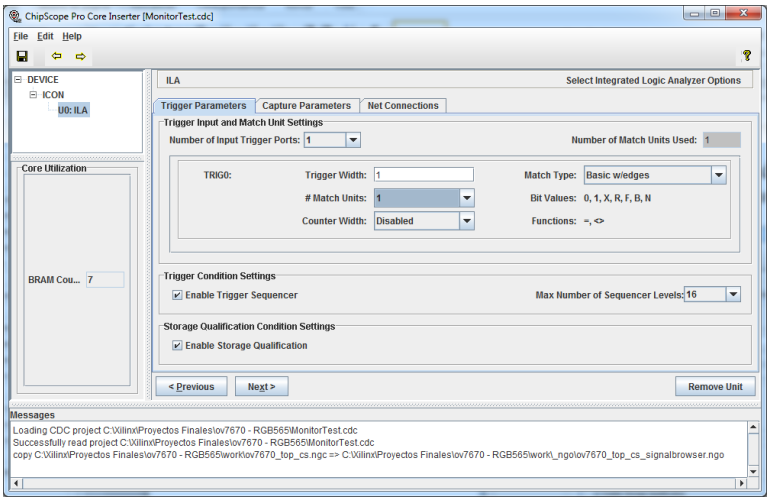


Figura 67. Parámetros de la señal de disparo

Aquí podemos seleccionar los parámetros de la señal de disparo (*trigger*), es decir, el evento que inicia la lectura de datos. Se pueden escoger opciones como el número de señales de disparo, la longitud de cada una o el evento que tiene que darse para que se produzca el disparo. La siguiente pestaña que nos aparece es *Capture Parameters*, donde podemos escoger el número de señales capturadas y las muestras que se tomarán de cada una cuando se produzca el evento de disparo.

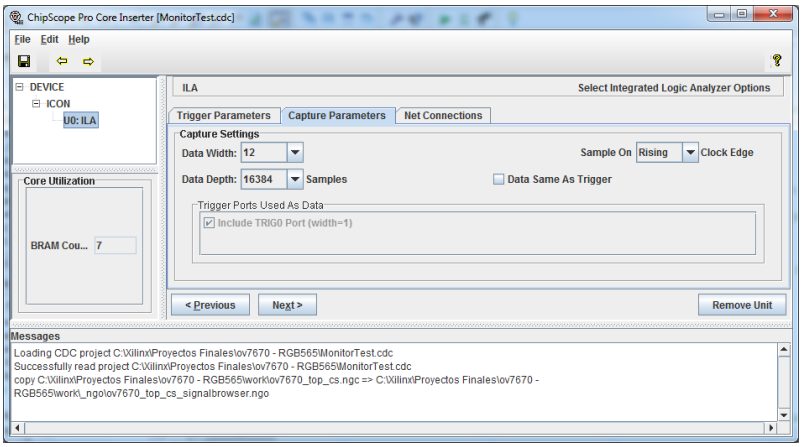


Figura 68. Parámetros de las señales capturadas

En la tercera pestaña, *Net Connections*, seleccionamos las señales del proyecto que queremos capturar, *Data Port,* el reloj que marca la frecuencia de muestreo, *Clock Port,* y la señal de disparo, *Trigger Port.*

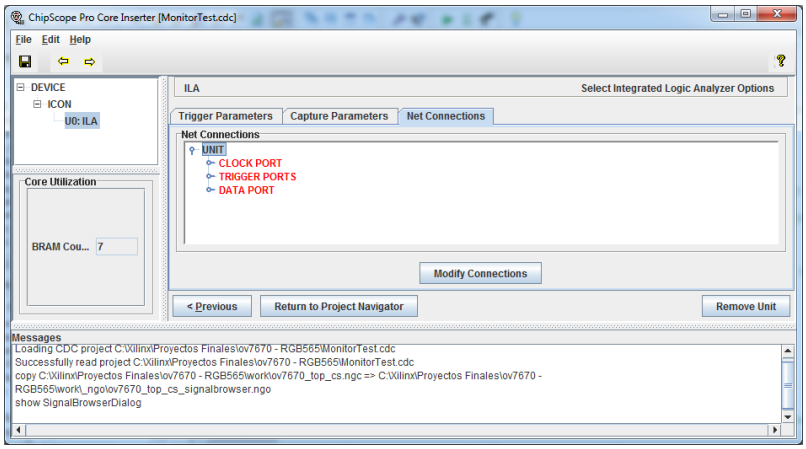


Figura 69. Puertos de reloj, disparo y datos vacíos

Vemos en la Figura 69 que los tres puertos aparecen en rojo, esto indica que no se les han asignado todavía señales válidas. Como reloj de muestreo escogemos el que trabaja a más frecuencia en el sistema, de 100 MHz, para obtener la frecuencia de muestreo necesario para capturar correctamente todas las señales. La señal de disparo en nuestro caso no es muy importante, ya que nuestro objetivo es observar de forma general el comportamiento del sistema. Como hay que seleccionar una, optamos por PCLK, de forma que empiece a capturar cuando detecte un flanco ascendente de ésta. Esto supone que el sistema está en funcionamiento, por lo que la captura será representativa.

Por último, queda escoger las señales capturadas, que en nuestro caso, para este ejemplo, serán PCLK, CLK25, para llevar una referencia de reloj y comprobar que la *ZedBoard* está alimentando al sensor y los pines de datos de salida del sensor. En total capturamos diez señales, número que debemos actualizar en la ventana *Capture Parameters*. Para seleccionar cada señal debemos clicar en *Modify Connections*, donde podemos buscar las señales deseadas y hacer las conexiones.

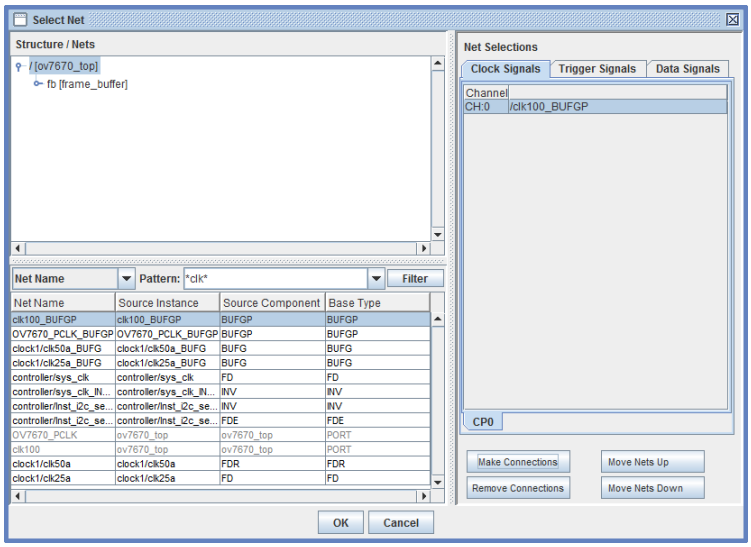


Figura 70. Selección de señales

Una vez seleccionadas todas las señales, el color de los puertos cambia a negro, como vemos en la Figura 71.

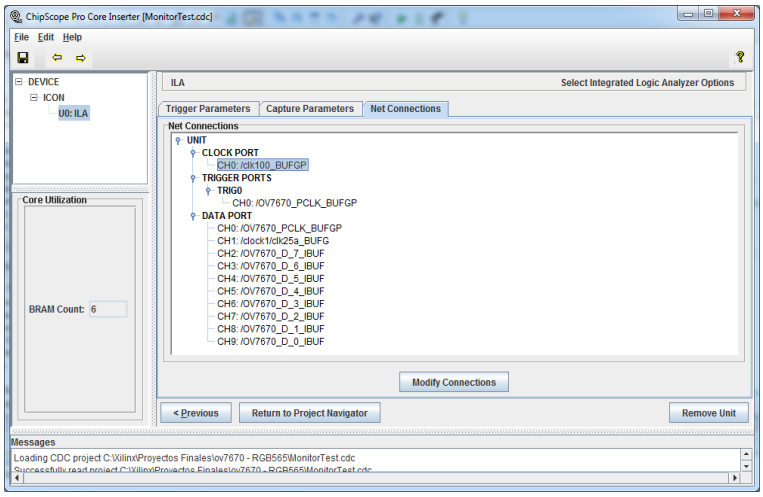


Figura 71. Configuración de ChipScope completa

Con la configuración finalizada se hace click en *Return to Project Navigator* y automáticamente volvemos a la interfaz de ISE y el módulo de *ChipScope* queda incluido en el proyecto.