UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Máster Universitario en Ingeniería de Telecomunicación

Trabajo Fin de Máster

Implementación de un sistema de videovigilancia usando FPGAs

Francisco Javier Reyes Torremocha

Julio, 2015

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Máster Universitario en Ingeniería de Telecomunicación

Trabajo Fin de Máster

Implementación de un sistema de videovigilancia usando FPGAs

Autor: Francisco Javier Reyes Torremocha

Fdo.:

Director: José María Granado Criado

Fdo:

Tribunal Calificador

Presidente:

Fdo.:

Secretario:

Fdo.:

Vocal:

Fdo.:

CALIFICACIÓN:

FECHA:

ÍNDICE GENERAL

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

RESUMEN

La tecnología de las FPGAs ha evolucionado permitiendo una gran flexibilidad en el diseño de todo tipo de sistemas y uno de sus principales mercados actualmente se sitúa en el procesamiento de imagen. En este proyecto se ha puesto en práctica una aplicación de tratamiento digital de imágenes para detectar móviles en un determinado escenario.

Para ello se ha implementado un sistema de extracción de fondo de la imagen que se capta utilizando una cámara, que permite una rápida detección y segmentación de los móviles.

Para hacer todo el procesamiento necesario se ha contado con una FPGA que ha sido convenientemente programada para, por un lado, poder comunicarse con los distintos dispositivos físicos implicados y, por otro, poder ejecutar un algoritmo que permita dotar al sistema de la funcionalidad deseada.

Como elemento de captación se ha contado con un sensor CMOS barato que, aunque no ofrece una elevada calidad, permite comprobar perfectamente toda la funcionalidad de sistema. Para la configuración del sensor se ha utilizado una placa Arduino como elemento auxiliar.

La FPGA ha tenido que configurarse para recibir imágenes del sensor CMOS, mostrar los resultados en una pantalla VGA y enviar alarmas utilizando una conexión Ethernet, por lo que ha sido necesario crear los interfaces físicos con todos estos dispositivos además de con algunos otros auxiliares, tales como, memorias, leds, etc.

El procesamiento de imagen se hace por software implementando una adaptación del algoritmo de extracción fondo de Karmann y von Brandt en C sobre 4 procesadores definidos al efecto. El software que implementa el algoritmo se ejecuta de forma paralela en los procesadores aumentando considerablemente el rendimiento del sistema.

Además, se ha implementado un quinto procesador al que se le ha dotado con los interfaces necesarios para, entre otras cosas, dotar de acceso a red IP al sistema para enviar alarmas por UDP a dispositivos remotos.

Como último punto de desarrollo, y para poder visualizar estas alarmas, se ha desarrollado una sencilla aplicación .Net para Windows que muestra los valores enviados por red por parte de la FPGA.

Este proyecto es un claro ejemplo de cómo, para llevar a cabo un desarrollo que conceptualmente puede ser muy sencillo, en ocasiones es necesario utilizar un gran número de tecnologías diferentes. En concreto aquí se ha utilizado como hardware una FPGA, un Arduino, un sensor de imagen y componentes de red Ethernet. En el lado del software se ha utilizado VHDL (aunque es un lenguaje de descripción hardware), C/C++ (para programar los procesadores embebidos y la placa Arduino) y C# (para la aplicación Windows).

Finalmente, se recogen una serie de conclusiones que ponen de manifiesto que el objetivo de desarrollar un sistema de detección de movimiento se ha conseguido y se proponen una serie de mejoras y funcionalidades adicionales para posibles trabajos futuros.

ABSTRACT

FPGA technology has evolved allowing a great design flexibility for all kind of systems and one of their main markets right now is image processing. This work describes a digital image processing application for detecting moving objects in a given environment.

For that purpose, a background extractor system has been implemented using the images captured using a camera. This allows a fast detection and moving objects segmentation.

For the required processing power a FPGA has been used. It has been programmed in order to, on the one hand, communicate with the different physical devices involved and, on the other hand, run an algorithm that gives the system the desired functionality.

As image capture device a cheap CMOS sensor has been used. This sensor does not provide a high image quality but it allows testing all the functionality of the system. For setting up the image sensor an Arduino board has been used as an auxiliary device.

The FPGA has been programmed to receive images from the CMOS sensor, to show the results in a VGA screen and to send alarms using an Ethernet connection. This made necessary creating all physical devices interfaces such us memories, leds, etc.

Image processing is made by software implementing an adaptation of the Karman and von Brandt background extraction algorithm in C, running in 4 processor defined for that purpose. The software implementing the algorithm is running in parallel improving highly the throughput of the whole system.

Moreover, a fifth processor has been implemented. This processor has all required interfaces for, among other things, give the system IP network access for sending UDP alarms to remote devices.

As a last development, and in order to watch this alarms, a simple .Net application has been created for Windows. This application shows all sending values from the FPGA.

This work is a clear example of how, for making a design that in concept can be very simple, there are situations in which the use of lots of different technologies is required. In this specific case, some hardware devices have been used, specifically an FPGA, an Arduino, an image sensor and network components. In the software side, it has been used VHDL (although it is a hardware description language), C/C++ (for embedded devices programming and Arduino board) and C# (for Windows application).

Finally, a set of conclusions are exposed showing that the target of developing a motion detection system has been reached and some improvements and additional functionalities for future works are proposed.

1 INTRODUCTION

The aim of this work is to implement a prototype of a video surveillance system with capabilities for working in an autonomous way so it can send alarms when the possibility of the existence of an intruder is detected. This system could be used for half-unattended surveillance tasks, as human observation only would be necessary whenever the system detected something estrange in the observation area. Also it could be used as an extension of a recording system, allowing the system to record in a more precise way that current systems that usually activate the recording when small changes in the scene are observed, even when these changes are produced by soft changes in ambient light with no relevant security meaning.

In order to make this implementation, a design using a FPGA was chosen because of it flexibility. The system was completed using cheap components that allow checking the utility of it.

It was desired to give some kind of intelligence to the system in order to make a good detection of moving objects. For this reason, a software based solution was selected. In this way, changing the logic of image processing would be easy. This processing would be made inside of processors defined inside the FPGA.

A development board with a Xilinx FPGA has been used. This limits the tools used to those created by the manufacturer of this FPGA. Although the board manufacturer (Digilent) gives design examples using Vivado Design Suite, the up to date suite from Xilinx, there was no available licenses what made necessary to use a previous version known as ISE Design Tools that merge, in several applications, hardware design for embedded processors, software for those processors and an integrated environment for those different applications.

Using those tools a Multiprocessor System on Chip (MPSoC) is designed inside the FPGA, creating interfaces to communicate with all external devices to it.

For those devices without an available IP-Core, it would be necessary to create the appropriate logic using VHDL for merging them into the design.

Because it is a research system, having an idea of what is happening inside the FPGA was considered important. This is the reason that made that one of the involved devices should be the VGA output available in the development board. Being able to watch, at real time, what is happening inside the system gives a very easy way to find design bugs and to evaluate the results of what has been developed.

In order to use such a system like the one to implement, it was considered interesting to have the possibility of making notifications in some way of the events generated by the system. The most versatile option in these cases is to notify using an IP network. For this reason, sending this kind of alarms was considered as an additional goal and, for using the least resources as possible inside the FPGA, it would be made using the simplest version, that is to say, sending notifications using UDP datagrams.

Applying the same argument that make necessary adding a VGA output to the system, it was decided to have some visibility of the information sent by network. A Windows application was created for that, in order to see the values sent by the video surveillance system.

The objective of this documentation is to reflect the work made to implement this system. In next chapter a description of the hardware that has been used (development board, FPGA, image sensor, etc.) is made.

Later, in chapter 2, the work that has been made in the FPGA is detailed. A description of the hardware defined inside the FPGA is made as well as the interaction with the other components of the design are explained.

Coming up next, in chapter 3, the software side of this work is shown. This includes the algorithms used in video processing, the coordination between processors and the additional software to have visibility of the system.

Finally, in chapter 4, the results obtained in this work are shown and in chapter 5 some reflections are made regarding what has been achieved, possible utilities and future lines of development.

1.1 Budget

Although it is not an example of how much could cost a final implementation of a system like this in an industrial scale, it has believed interesting to show a breakdown of the bill of materials. It has to be noted that the development board containing the FPGA was possible to be acquired using an especial price for students for being part of the educational program of the Universidad de Extremadura.

Component	**Provider**	**Units**	**Price**
Nexys4 development board	Digilent	1	160 €
Custom made PCB	Fritzing fab	2	49.84 €
OV7670 image sensor	Asian website	2	13 €
Arduino Pro Mini	Asian website	2	7 €
		Total:	**229.84 €**

Table 1 Project budget

In Table 1 the budget of the work can be seen. The table does not include the cost of small electronic components (resistors, capacitors, buttons, etc). Moreover, as can be seen, some of the components have been bought twice, sometimes because it small cost, others, as the PCB, in case there was some design mistake that made necessary making some adjustment in the board soldering wires over it. Fortunately, this was not necessary and the second board is available for future works.

COMPONENTES HARDWARE

1.2 FPGAs

Las FPGAs (Field-Programmable Gate Arrays) son un tipo de circuito integrado programable que está compuesto por recursos de procesamiento, recursos de interconexión y bloques de entrada y salida. Cuando se lleva a cabo un diseño hardware utilizando FPGAs, se pueden utilizar metodologías de diseño estándar (esquemáticos, VHDL,...) para, a continuación, plasmar dicho diseño sobre los recursos de la FPGA. Cuando se *programa* la FPGA, lo que se está haciendo en realidad es, por un lado, configurar los recursos de proceso para que hagan las operaciones necesarias y, por otro, utilizar los recursos de interconexión para enrutar las distintas señales entre las entradas y salidas, tanto del circuito integrado como de los recursos de proceso, para crear la funcionalidad completa. En la Figura 1 puede verse un esquema de la estructura de una FPGA.

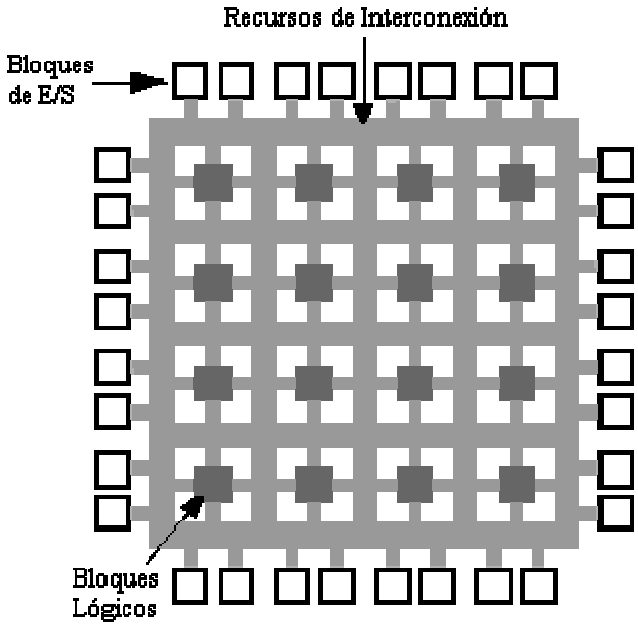

Figura 1 Esquema representativo de una FPGA

Una de las principales ventajas de las FPGAs es que pueden ser reprogramadas una y otra vez permitiendo hacer diseños hardware de forma rápida que pueden ser probados múltiples veces sin coste adicional. También es muy habitual que se utilicen como prototipos antes del diseño de circuitos ASIC.

Desde hace algún tiempo es bastante habitual encontrarse con dispositivos que han sido diseñados íntegramente utilizando FPGAs. El coste de las mismas ha bajado considerablemente haciendo posible un desarrollo de funcionalidades avanzadas, de muy alta potencia a precios muy competitivos. Además, los dispositivos basados en FPGAs tienen una ventaja muy importante y es que, al ser reprogramables, en caso de que un fallo en el diseño sea detectado cuando el producto ya está en manos del cliente, o si es necesario ofrecer una funcionalidad adicional, puede generarse una nueva versión de *firmware* haciendo las modificaciones oportunas. De esta manera, el tiempo de diseño se puede acortar considerablemente sin poner en riesgo la viabilidad del proyecto.

1.3 Elementos hardware utilizados

1.3.1 Nexys4 de Digilent

La plataforma Nexys4 de Digilent [1] es una tarjeta de desarrollo muy versátil por la cantidad de dispositivos de entrada/salida que tiene incluidos. Contiene una FPGA Artix-7 de Xilinx [2] (en concreto XC7A100T-1CSG324C [5]). Se ha elegido esta placa por ser una placa de bajo coste, con una cantidad de recursos reprogramables suficientes para el diseño y disponer de todas las conexiones de entrada/salida necesarias para el desarrollo del proyecto. Sin embargo, aunque la placa tiene todas las características necesarias, la reducida cantidad de bloques de memoria ha limitado la resolución de las imágenes capturadas, cosa que podría resolverse con soluciones de mayor coste. Las características principales de dicha FPGA son las siguientes:

- 15.850 slices lógicos, cada uno con cuatro LUTs de 6 entradas y 8 flip-flops
- 4.860 Kbits en bloques de RAM rápida

- 6 celdas de gestión de reloj, todas ellas con PLL
- 240 slices DSP
- Reloj interno con velocidades superiores a 450MHz
- Conversores analógico-digitales incluidos en el chip

Adicionalmente, la Nexys4 incluye los siguientes periféricos en la placa:

- 16 interruptores
- 16 LEDs
- 2 LEDs tricolores
- 2 displays de 7 segmentos de 4 dígitos
- Un puerto serie sobre USB
- Salida VGA de 12-bits
- Acelerómetro de 3 ejes
- 16 MB de RAM (CellularRAM)
- 16 MB de memoria flash (para datos o configuración)
- Salida de audio PWM
- Sensor de temperatura
- Micrófono PDM
- Interfaz físico de red Ethernet 10/100
- Puerto USB-HID para ratón y teclado
- Conector SD (para ficheros de configuración o sistemas de ficheros)

Además, se incluyen 4 puertos digitales con 8 señales cada uno y otro con 4 señales analógicas balanceadas. Como se verá más adelante, los puertos digitales han sido utilizados en el diseño para conectar la cámara. En la Figura 2 puede verse un diagrama de la placa de desarrollo junto con todos los dispositivos que incluye y número de pines que utiliza para conectarse a cada uno de ellos.

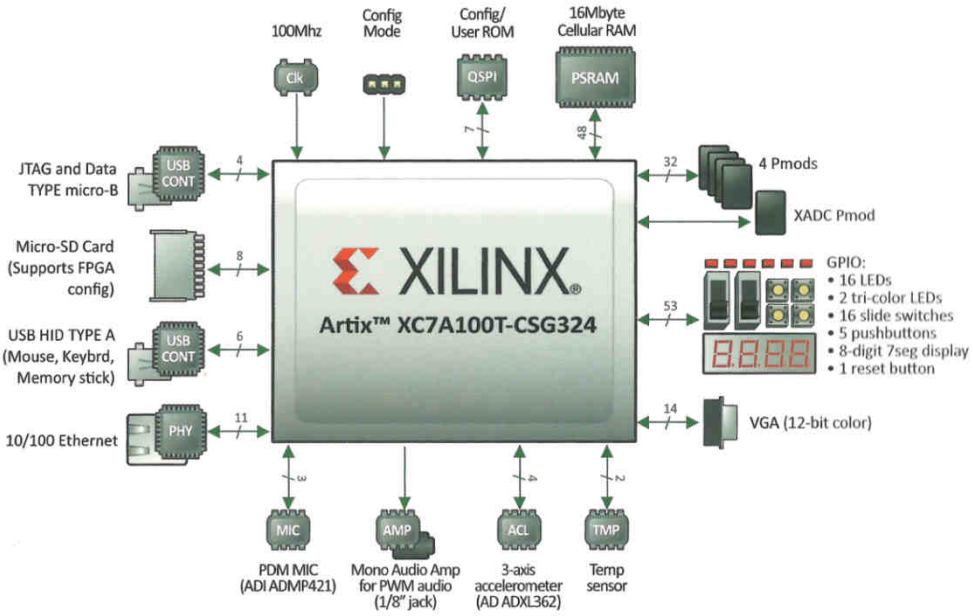

Figura 2 Diagrama de la Nexys4

Por último, a continuación, en la Figura 3 puede verse el aspecto físico de la placa de desarrollo.

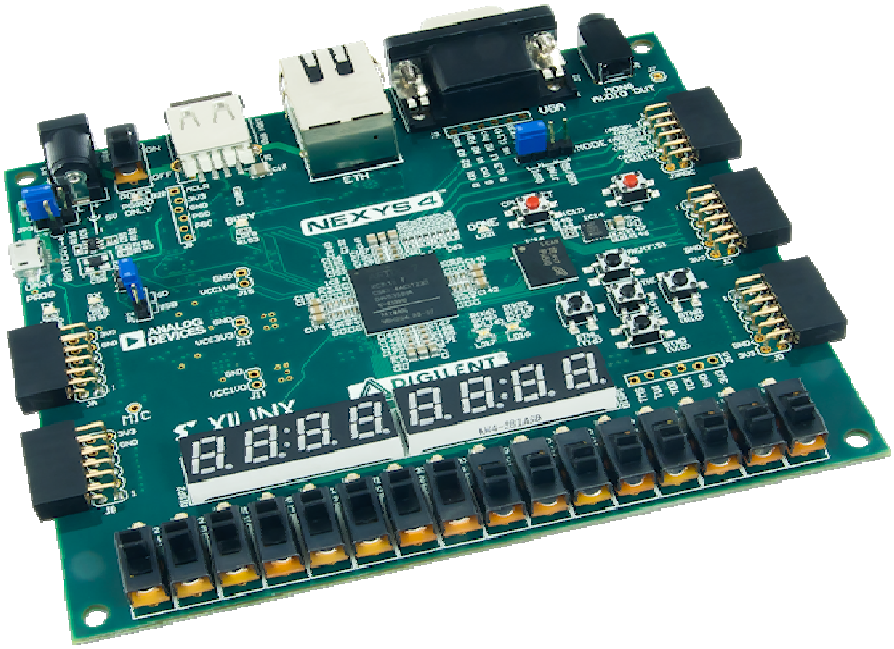

Figura 3 Tarjeta de desarrollo Nexys4

1.3.2 Sensor CMOS (OV7670)

Como elemento de captación de imagen se ha utilizado un sensor CMOS de bajo coste, concretamente el OV7670 de Omnivision [3] . Este sensor permite funcionar a una resolución de hasta 640x480 píxeles a 30 fotogramas por segundo.

Este sensor CMOS es del tipo conocido como CMOS de píxel activo (*active-pixel*). Los sensores de píxel activo utilizan un fototransistor o fotodiodo para captar la luz. En cada uno de los píxeles hay un amplificador haciendo de seguidor de tensión (de ahí el nombre de píxel activo) para hacer un primer acondicionamiento de la señal captada. Adicionalmente la celda de cada píxel contiene una puerta de reset para hacer el borrado de la carga captada y tomar una nueva imagen, y una puerta que lleva la tensión del amplificador del píxel a la salida para ser leído. En la Figura 4 puede verse el transistor seguidor de tensión (T1), el transistor que hace el reset del fotodiodo (T2) y el encargado de seleccionar el píxel para leerlo a la salida (T3).

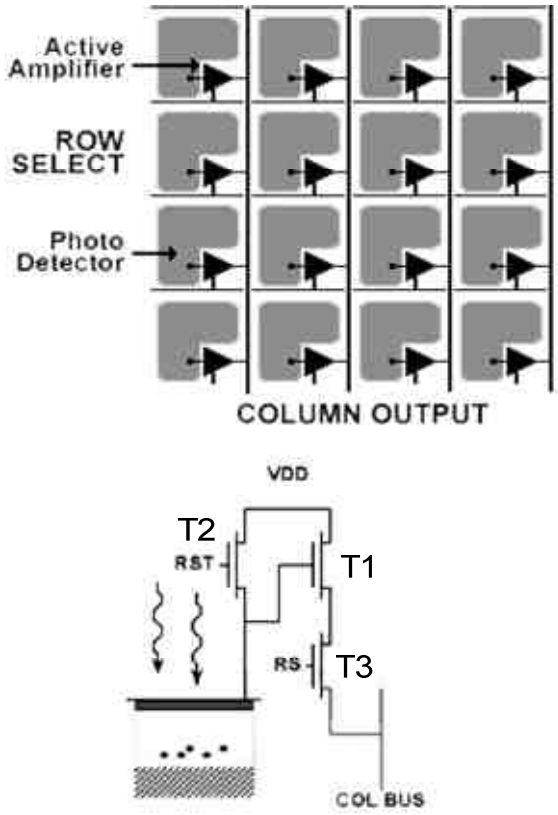

Figura 4 Estructura típica de una célula CMOS

Respecto al funcionamiento de estos sensores, hay que decir que se puede acceder a cada elemento de la matriz igual que se haría en una memoria SRAM. Esto ofrece una flexibilidad muy grande a la hora de usar el sensor, dado que permite múltiples combinaciones diferentes. Por ejemplo, se puede usar un sensor de una resolución determinada pero leer menos píxeles porque en ese momento sea necesaria una resolución menor, etc.

Este dispositivo se puede controlar utilizando el protocolo SCCB (Serial Camera Control Bus) [4] para acceder a sus registros internos. Este protocolo, en su versión sobre 2 hilos, es compatible con I2C (Inter-Integrated Circuit) siendo, por lo tanto, muy fácil adaptar cualquier diseño para poder configurarlo de forma adecuada.

La documentación del sensor incluye un listado completo de los registros que controlan el funcionamiento del mismo y una explicación de su funcionalidad. Para este diseño se ha utilizado la configuración por defecto como base y, sobre esta, se han hecho una serie de modificaciones para ajustar la funcionalidad al diseño. En la Tabla 2 se pueden ver todos los registros que han tenido que ser modificados.

Registro	**Valor por defecto**	**Valor configurado**	**Comentarios**
HSTART	0x11	0x25	Comienzo en el eje horizontal de la zona de exploración
HSTOP	0x61	0x4D	Fin en el eje horizontal de la zona de exploración
VSTRT	0x03	0x3F	Comienzo en el eje vertical de la zona de exploración
VSTOP	0x7B	0x7B	Fin en el eje vertical de la zona de exploración
DM_LNL	0x00	0xFD	Bytes alto y bajo del número de líneas *dummys* en cada frame. Usado para bajar la tasa de imágenes por segundo
DM_LNH	0x00	0x05	
COM8	0x8F	0x00	Solo activo al pulsar un pulsador. Desactiva el control automático de ganancia
SCALING_YSC	0x35	0xB5	Sólo activo al pulsar un pulsador. Activa el patrón de barras en el sensor

Tabla 2 Registros modificados en el sensor de imagen

1.3.2.1 Adaptación del sensor al sistema y configuración del mismo

Como se ha comentado anteriormente, el sensor se puede configurar utilizando el protocolo I2C. Durante las fases de análisis y diseño del sistema, se comprobó que iba a ser necesario acceder a la configuración de la cámara para adaptarla al diseño hw/sw de la FPGA.

Aunque inicialmente se pensó incluir un controlador I2C dentro del diseño, muy pronto se llegó a la conclusión de que era una mala idea ya que, ante cualquier cambio necesario en el sensor, se haría necesario recompilar el programa que se estaría ejecutando en los distintos procesadores o, lo que es peor, habría que volver a generar el hardware (proceso que en las etapas finales del diseño llevaba más de una hora).

Con el doble objetivo de facilitar el cambio de parámetros del sensor y buscar una forma adecuada de conectarlo a la placa de desarrollo se preparó un pequeño diseño sobre PCB (Printed Circuit Board) que incluía un zócalo para el sensor CMOS, otro para una placa Arduino y los pines necesarios para la conexión con la Nexys4.

Al ser un diseño sencillo, se llevó a cabo con la herramienta de diseño Fritzing [6] , que permite utilizar un entorno gráfico para añadir componentes de una librería relativamente completa (incluía los componentes necesarios como son un Arduino Pro Mini, zócalos, resistencias y switches) de una forma sencilla y, una vez terminado el diseño, enviarlo a fabricación con un precio y plazo de entrega razonable.

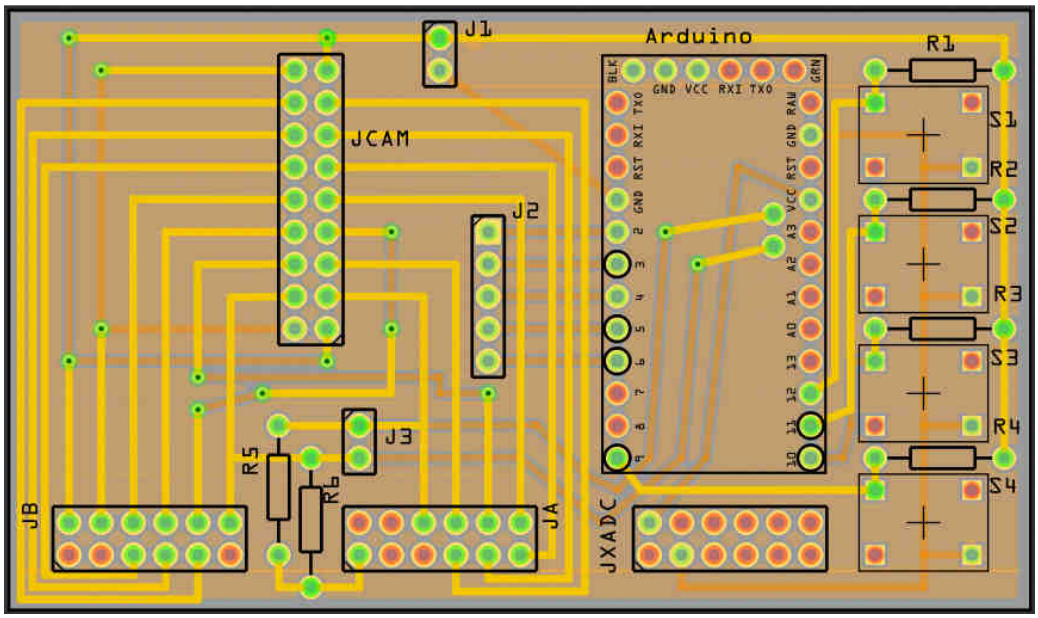

Figura 5 Desarrollo de la PCB dentro del programa Fritzing

En la Figura 5 puede verse el diseño de la placa tal y como lo muestra el interfaz gráfico de diseño de Fritzing. En la siguiente Figura 6 se puede observar ya la placa fabricada con los componentes soldados a ella.

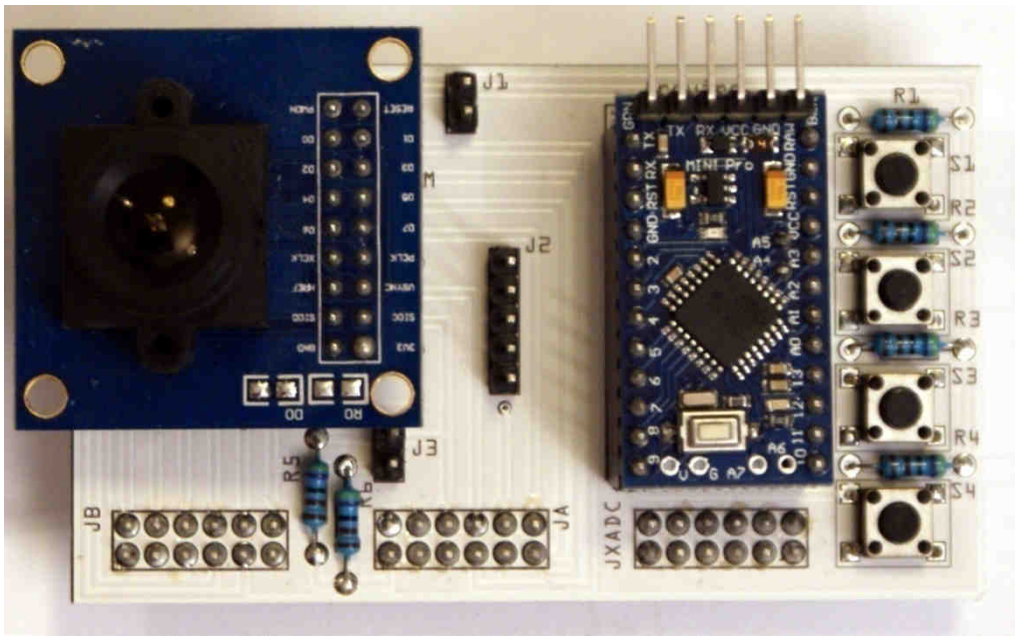

Figura 6 PCB soldada y con los componentes en su lugar

En la Tabla 3 se puede ver una descripción de todas las señales utilizadas del sensor CMOS así como a qué dispositivos están conectadas. Las señales relacionadas con los datos de píxel y sincronización de los mismos están conectadas a la FPGA mientras que las señales del bus I2C están conectadas al Arduino para la configuración.

Señal	Ancho	Sentido	Descripción
D(7..0)	8 bits	CAM->FPGA	Bus de datos para enviar los valores de los píxeles
HREF	1 bit	CAM->FPGA	Señal de sincronismo horizontal
VSYNC	1 bit	CAM->FPGA	Señal de sincronismo vertical
XCLK	1 bit	FPGA->CAM	Señal de reloj (debe ser 24 MHz)
PCLK	1 bit	CAM->FPGA	Reloj de píxel. Valida los valores del bus de datos
SIDC	1 bit	Arduino->CAM	Reloj del bus I2C
SIDD	1 bit	Arduino<->CAM	Señal de datos I2C

Tabla 3 Señales del sensor de imagen

1.3.2.2 Configuración del sensor usando una placa Arduino

Para simplificar la configuración del sensor y poder hacer cambios en la misma sin tener que rehacer todo el diseño se optó por utilizar una placa Arduino [7] para este fin.

Arduino es una plataforma hardware de computación open source basada en placas sencillas con un microcontrolador, y un entorno de desarrollo, también open source, para escribir software para las placas. Es un proyecto orientado a acercar los ordenadores al mundo físico y su diseño está, por tanto, muy enfocado a relacionarse con sensores, interruptores, motores, etc.

El lenguaje de programación de Arduino es una implementación de Wiring [14] que, a su vez, está basado en C/C++ y que simplifica las operaciones habituales de entrada y salida como simples asignaciones a variables.

Dentro del ecosistema Arduino hay muchas placas compatibles. Para este proyecto, y con el objetivo de optimizar espacio en PCB, se ha utilizado el modelo de placa Arduino Pro Mini [15] , que es una de las versiones de menor tamaño disponibles y contiene un microcontrolador ATmega328 [16] de Atmel. El único inconveniente es que esta placa no trae incorporado un conversor serie-USB para su programación con lo que hubo que utilizar uno externo.

El diseño que se ha creado sobre la PCB incorpora unos switches para permitir el cambio a configuraciones alternativas sin tener que cargar un nuevo programa en la placa. De esta manera, el código residente en el microcontrolador detecta las pulsaciones de los pulsadores y reconfigura la cámara. Además, si el cambio en la configuración no está implementado en los switches, a través del adaptador serie-USB se puede modificar el programa en segundos para estos cambios más importantes.

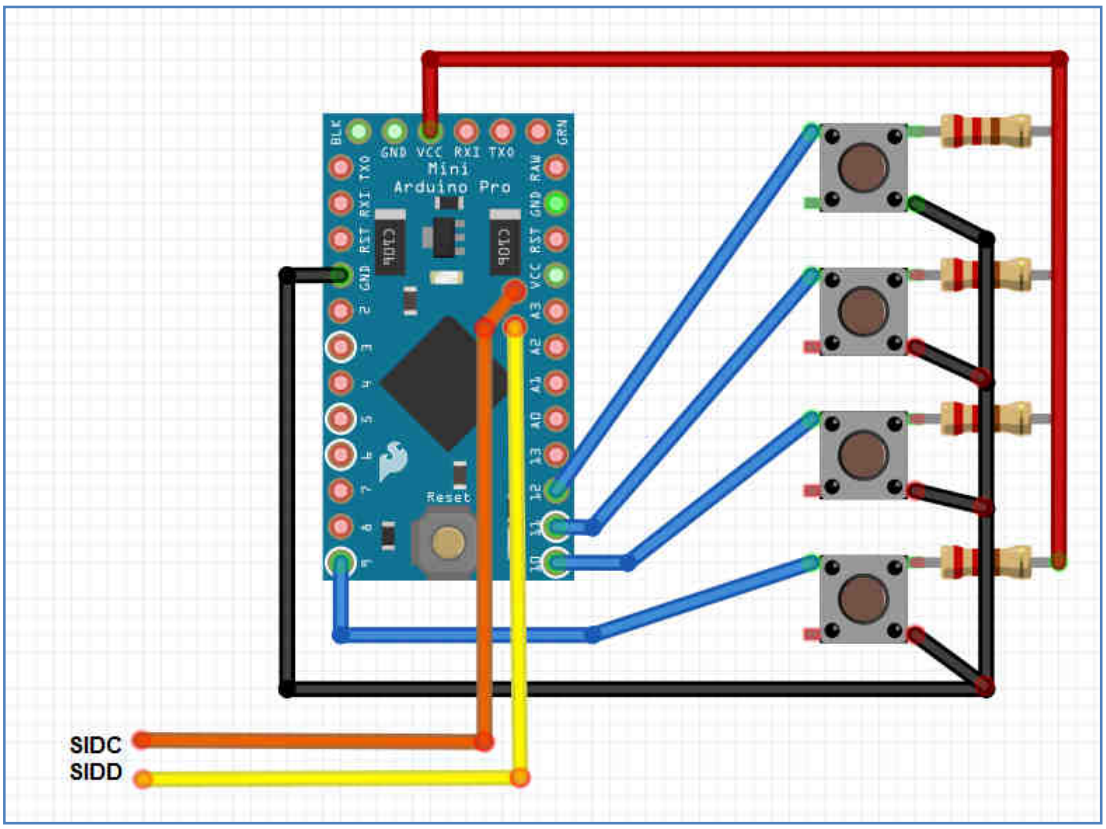

Figura 7 Esquema de la circuitería asociada al Arduino

En la Figura 7 se puede ver un esquema con toda la circuitería conectada a la placa Arduino. En Anexo I puede verse el código cargado en la placa Arduino. Dicho código está basado en la librería Wire, que permite controlar las comunicaciones sobre un bus I2C que, como se ha comentado, es compatible con el interfaz de control del sensor de imagen.

El sistema ejecuta un bucle infinito, reescribiendo los parámetros necesarios para el diseño de forma continua. Adicionalmente, ante la acción sobre alguno de los pulsadores se envía a la cámara alguna configuración alternativa (como, por ejemplo, para activar temporalmente el control automático de ganancia, la generación de barras de imagen, etc.).

2 DESARROLLO SOBRE FPGA

Para el diseño sobre la FPGA se ha utilizado el entorno de desarrollo ISE Design Tools 14.6 [2] . Este entorno está formado por una serie de herramientas integradas entre sí de manera que se puede hacer diseño hardware sobre esquemático o VHDL, integrar el hardware creado con otro entorno para diseño de procesadores embebidos y, finalmente, exportar todo a otro entorno para programar el código que se ejecutará sobre los procesadores anteriormente definidos.

El entorno de desarrollo incluye muchas herramientas de simulación, provisión de recursos dentro de la FPGA, etc. A continuación vamos a ir viendo cada una de las herramientas utilizadas en este trabajo así como el desarrollo hecho en las mismas.

2.1 Xilinx Platform Studio

Esta herramienta de diseño, conocida como XPS, permite crear sistemas complejos de procesadores embebidos dentro de una FPGA que interactúan con periféricos que pueden diseñarse dentro de la propia FPGA o bien externos a la misma. Ofrece un entorno de desarrollo gráfico que permite añadir los periféricos usando *drag-and-drop*, crear buses para interconectarlos y establecer dichas conexiones con simples clics de ratón.

En función de la familia de FPGAs que se vaya a utilizar, y las licencias disponibles, el entorno ofrece una serie de módulos IP (Intelectual Property) utilizables, o IP-Cores, que son elementos hardware predefinidos, habitualmente en VHDL, para interactuar con el resto de componentes a través de buses. Dentro la de arquitectura Artix 7, la que utiliza la placa de desarrollo Nexys4, el entorno ofrece el desarrollo utilizando variantes de un bus conocido como AXI [8] (Advanced eXtensible Interface) que permite interconectar las diferentes IPs entre sí.

Utilizando este entorno se han implementado cinco procesadores MicroBlaze [9] utilizando el asistente que a tal efecto tiene la herramienta de desarrollo. De estos cinco, cuatro son los encargados de comunicarse con la cámara y la salida VGA y llevar a cabo todo el procesamiento de imagen (en el diseño nos referiremos a ellos como *workers*), mientras que el quinto es el encargado de recibir la información de los anteriores y gestionarla para transmitir alarmas por IP e iluminando una serie de LEDs de la tarjeta de desarrollo.

Los cuatro *workers* son iguales en cuanto a diseño y, salvo los bloques de datos e instrucciones del bus local del procesador, tienen acceso a los mismos periféricos y mismas regiones de memoria. En la Tabla 4 pueden verse todos los dispositivos conectados al *worker* 1. Cabe destacar que hay 2 bloques de memoria local para el procesador, una de instrucciones y otra de datos, ya que, los MicroBlazes tienen arquitectura Harvard.

Periférico	**Bus**	**Descripción**
worker_1_d_bram_cntlr	LMB	Memoria local del procesador (datos)
worker_1_i_bram_cntlr	LMB	Memoria local del procesador (instrucciones)
mivideo2_0	AXI	Interfaz de salida a VGA
mailbox_1	AXI	Dispositivo para enviar información al procesador controlador
mi_cam_iface_0	AXI	Interfaz con la cámara

Tabla 4 Mapa de memoria del worker 1

Los periféricos han sido añadidos utilizando los asistentes del entorno de desarrollo, salvo los interfaces con la salida de vídeo y la cámara que, como se verá más adelante, han tenido que ser diseñados utilizando VHDL.

Si bien el diseño hardware de los *workers* es bastante sencillo, no se puede decir lo mismo del otro procesador (al que nos referiremos como *controller*), ya que este último interactúa con un número importante de periféricos, como se puede ver en la Tabla 5.

Periférico	**Bus**	**Descripción**
microblaze_0_d_bram_cntlr	LMB	Memoria local del procesador (datos)
microblaze_0_i_bram_cntlr	LMB	Memoria local del procesador (instrucciones)
LEDS	AXI	Puerto para acceder a los 16 LEDs de la tarjeta
mailbox_1	AXI	Dispositivo para recibir información del worker 1
mailbox_2	AXI	Dispositivo para recibir información del worker 2
mailbox_3	AXI	Dispositivo para recibir información del worker 3
mailbox_4	AXI	Dispositivo para recibir información del worker 4
axi_uartlite_0	AXI	Puerto serie para enviar información a consola
axi_bram_ctrl_0	AXI	Controlador para memoria interna adicional
axi_ethernetlite_0	AXI	Acceso a la capa física Ethernet
microblaze_0_intc	AXI	Controlador de interrupciones (necesario para el uso de la RED)
debug_module	AXI	Para depurar los programas en ejecución
axi_timebase_wdt_0	AXI	Temporizador
Generic_External_Memory	AXI	Controlador para la RAM externa de 16MB

Tabla 5 Mapa de memoria del Controller

Como se ha comentado antes, los procesadores definidos en el sistema se interconectan con cada uno de los dispositivos utilizando el propio entorno gráfico de manera que, de forma ágil, puede configurarse qué dispositivos pueden comunicarse entre sí y a través de qué buses. En la Figura 8 se pueden ver todas las conexiones entre los distintos módulos. Como se puede observar, cada uno de los procesadores tiene un par de buses LMB (Local Memory Bus) que los conectan a sus respectivos bloques de memoria local para datos e instrucciones.

Aparte de los buses LMB, se han definido 2 buses en el sistema. Los dos son buses AXI, los recomendados para utilizar con esta herramienta de desarrollo. Uno de los buses, utilizado en exclusiva por el procesador *controller* es del tipo AXI_Lite. Este bus sólo permite un maestro, y en este bus, están los dispositivos que son accedidos de forma exclusiva por este procesador.

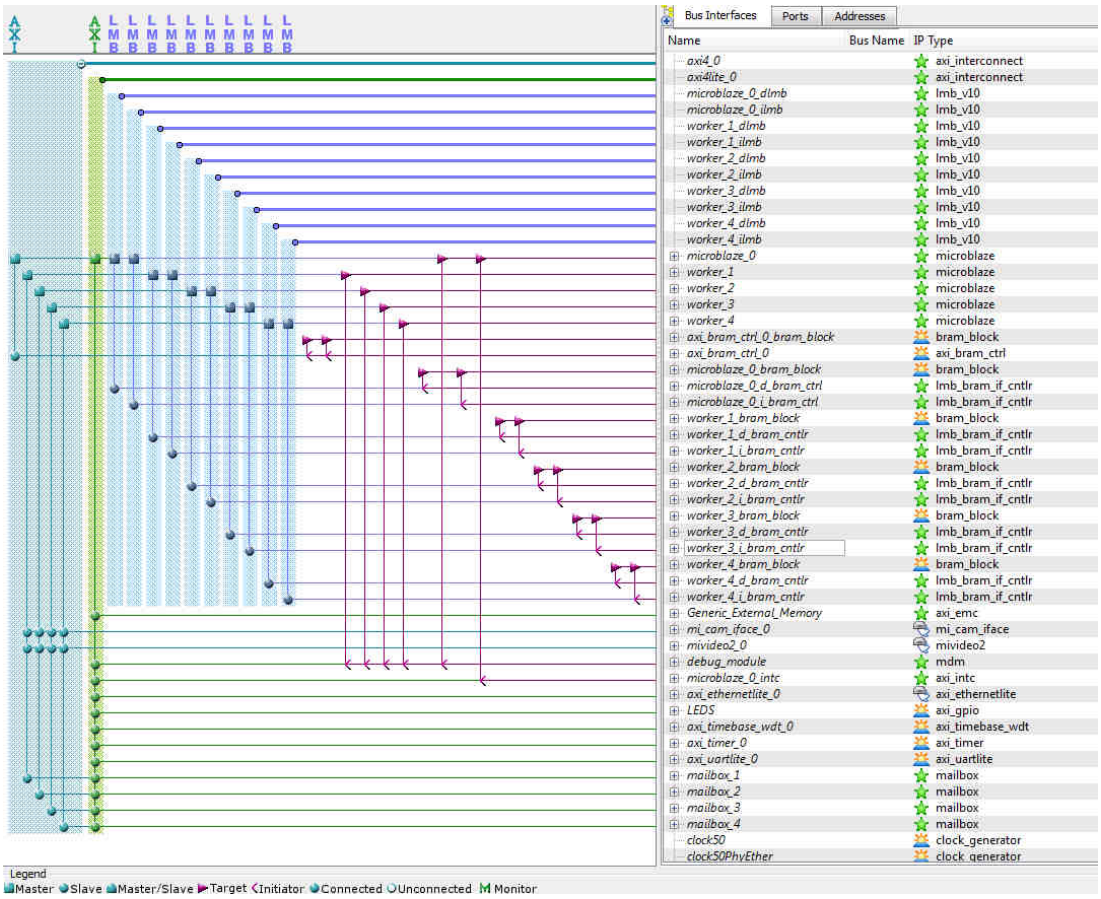

Figura 8 Diagrama de interconexión de buses

Sin embargo, todos los *workers* tienen que acceder a algunos dispositivos comunes (como es la cámara y la salida VGA). Esto implica que los cuatro procesadores han de actuar como maestros del bus (y el propio bus gestionar las posibles colisiones). Por este motivo, se ha definido el bus AXI estándar en el que se han conectado los dispositivos a los que tienen que acceder los *workers* así como el propio *controller* para acceder a los *Mailboxes*, que son los dispositivos que permiten la comunicación entre los *workers* y el *controller*.

Aparte de la interconexión de los buses dentro del mismo entorno de desarrollo, se conectan otras muchas señales como son las de reloj, reset y entradas y salidas de la FPGA.

Dentro de los periféricos añadidos cabe destacar 2 de ellos ya que, si bien el resto formaba parte del catálogo de Xilinx, éstos han sido creados específicamente para este proyecto. El entorno de desarrollo permite la creación, a través de un asistente, de una plantilla de un dispositivo que se conecta a uno de los buses del sistema (una caja negra). Al ejecutar este asistente se generan los ficheros VHDL necesarios para acceder a un dispositivo basado en registros o regiones de memoria y se crean todos los mecanismos de arbitrio del bus necesarios.

Para los dispositivos que acceden a la cámara y la salida VGA se ha utilizado este asistente. Mediante esta herramienta se pueden elegir dos modelos, acceso a valores de registro o mapeo de regiones de memoria. En este caso se ha utilizado la opción de mapeo de regiones de memoria. Estas regiones de memoria serán los píxeles captados por la cámara y el buffer de vídeo de la salida VGA. Todo el diseño de estos dos periféricos se explica en la siguiente sección.

2.1.1 Diseño de interfaces de cámara y VGA

Para el diseño de estos dos módulos se ha utilizado la plantilla generada por el XPS y se ha cargado en un entorno de desarrollo de Xilinx que permite trabajar directamente sobre VHDL. Este entorno es el ISE Design Tool (en adelante ISE). Este programa permite abrir el fichero de proyecto generado anteriormente por el XPS, permitiendo hacer modificaciones en el código VHDL para adaptarlo a las necesidades concretas.

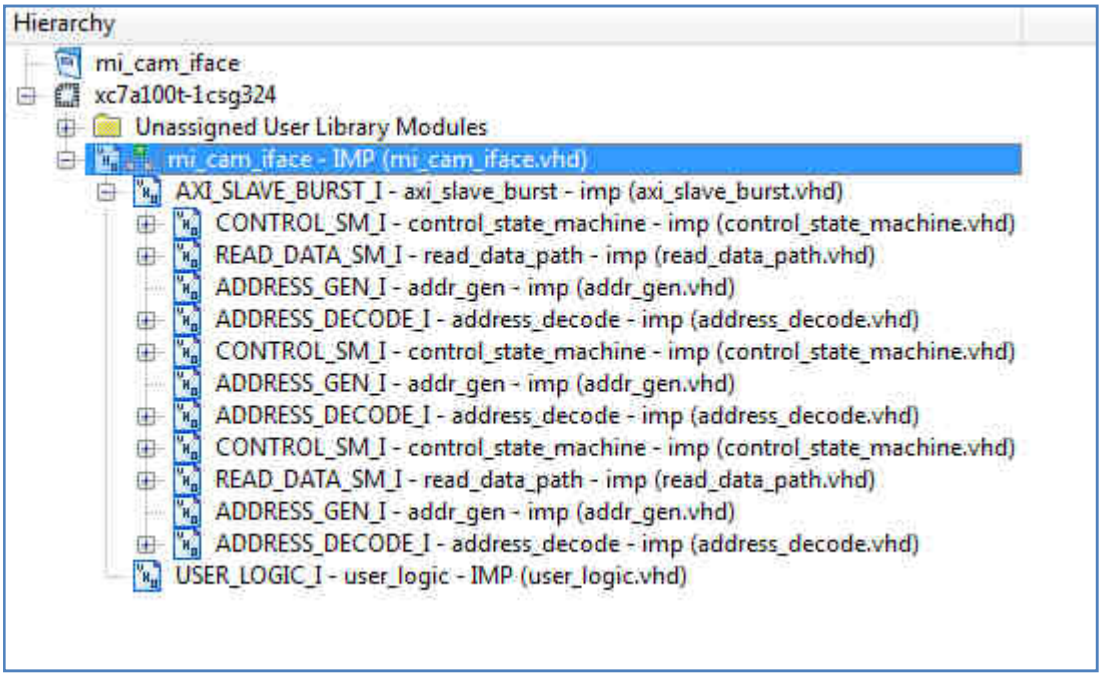

Figura 9 Ficheros generados por el XPS

En la Figura 9 puede verse un ejemplo de los ficheros con código VHDL generados por el XPS como plantilla. Se incluye un fichero global (mi_cam_iface.vhd), un fichero donde se implementará la lógica de usuario (user_logic.vhd) y una serie de ficheros adicionales que son los encargados de implementar la lógica de control del bus.

Desde el punto de vista del desarrollador, lo aconsejable es modificar solo el fichero global para incluir en él los puertos de entrada y salida necesarios en el módulo y conectar dichos puertos al módulo user_logic.vhd, donde se hará la implementación de la lógica necesaria para la funcionalidad deseada. Vemos a continuación en detalle cómo se han desarrollado los interfaces con la cámara y con el VGA.

2.1.1.1 Interfaz con la cámara

Los procesadores van a acceder a la información del vídeo del sensor a través del puerto AXI leyendo los datos de los píxeles. Siendo así, se ejecutó el asistente de

creación de dispositivos basados en mapeo de bloque de memoria. La funcionalidad del módulo se puede ver en la Figura 10.

Figura 10 Diagrama de funcionalidad del interfaz con la cámara

Previo al diseño propiamente dicho, se hizo un estudio de la capacidad de memoria de la FPGA de forma que se eligiese una resolución adecuada en la cámara. La FPGA instalada en la Nexys4 contiene 4.860 Kbits de RAM interna, o lo que es lo mismo, unos 607 KB. Inicialmente se intentó hacer un diseño utilizando la imagen completa VGA de la cámara OV7670 (640x480), pero esto requeriría 1 byte para cada píxel (como se verá más adelante 8 bits en escala de grises), es decir, 300 KB sólo para la cámara. Si a esto añadimos la memoria necesaria para la salida VGA (al menos otros 300 KB), se comprueba que la FPGA no tiene RAM suficiente para estas regiones de memoria. Además, hay que tener en cuenta que los bloques de memoria de los procesadores y algunos dispositivos también utilizan parte de esta RAM *on-chip*.

Por lo visto anteriormente, se optó por hacer funcionar el sensor de imagen a un cuarto de VGA (320x240 píxeles). Esto hace necesario utilizar para el módulo de comunicación con la cámara 320x240 = 75 KB, una cifra que es bastante razonable teniendo en cuenta las necesidades del resto de componentes. Este es uno de los motivos que han hecho trabajar en 8 bits y en escala de grises. Trabajar utilizando imagen en color habría hecho necesario multiplicar por 3 (1 byte para cada color) la memoria necesaria o habría forzado una reducción de resolución aún mayor. El diseño presentado en este trabajo podría adaptarse fácilmente a resoluciones o número de colores mayores, simplemente usando FPGAs con más memoria *on-chip*, pero también más costosas. En cualquier caso, como se verá más adelante, la solución aquí presentada, usando una FPGA de bajo coste, ha sido satisfactoria incluso con baja resolución de imagen.

Con esto como requisito se dimensionó la memoria a utilizar y se enrutaron dentro del módulo las señales correspondientes de la cámara. En la Tabla 6 se pueden ver las señales implicadas en este módulo.

Señal sensor	**Descripción**
clk_cam	Señal de reloj a suministrar al sensor (24 MHz)
pix_clk_cam	Reloj de píxel. Valida la información disponible en el bus de datos
data_cam (7..0)	Bus con los datos de imagen de los píxeles
hsync_cam	Indicación de estar en zona de vídeo activa
vsync_cam	Indicación de estar en la zona de barrido vertical

Tabla 6 Señales del sensor de imagen

El asistente genera en el fichero *user_logic.vhd* la lógica necesaria para acceder, tanto en lectura como en escritura, a 256 palabras de 32 bits. El código ha sido modificado para que acceda a 75 KB poniendo a cero los 3 bytes más significativos de cada palabra. Además, se introdujeron cambios para que, desde el bus, esa región de memoria sólo fuera de lectura, ya que la escritura se haría con los valores leídos de la cámara. En el código que se implementaría en los *workers* esto fue tenido en cuenta para hacer solo peticiones de lectura de bytes.

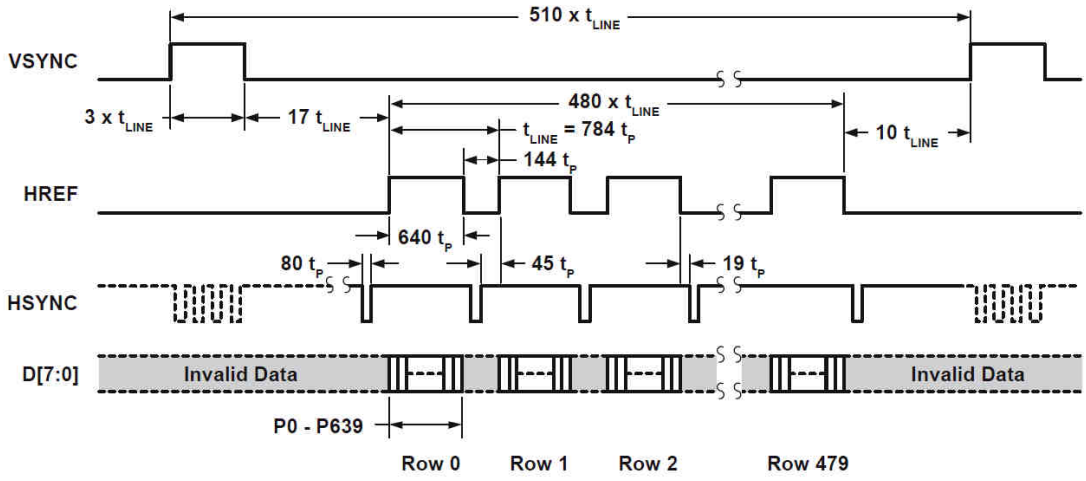

Figura 11 Temporización de un frame VGA

Para hacer la lógica de llenado de la memoria se utilizó un contador de píxel sincronizado con las señales de sincronismo horizontal y vertical y el reloj de píxel. En la Figura 11 puede observarse la temporización de las señales de sincronismo vertical (en el flanco de bajada de VSYNC se inicia el frame) y de sincronismo horizontal (cuando estamos en área de vídeo activo HREF está a nivel alto). En la Figura 12, puede verse como el bus de datos tiene información válida cuando, estando HREF a nivel alto, llega el flanco de subida del reloj de píxel PCLK.

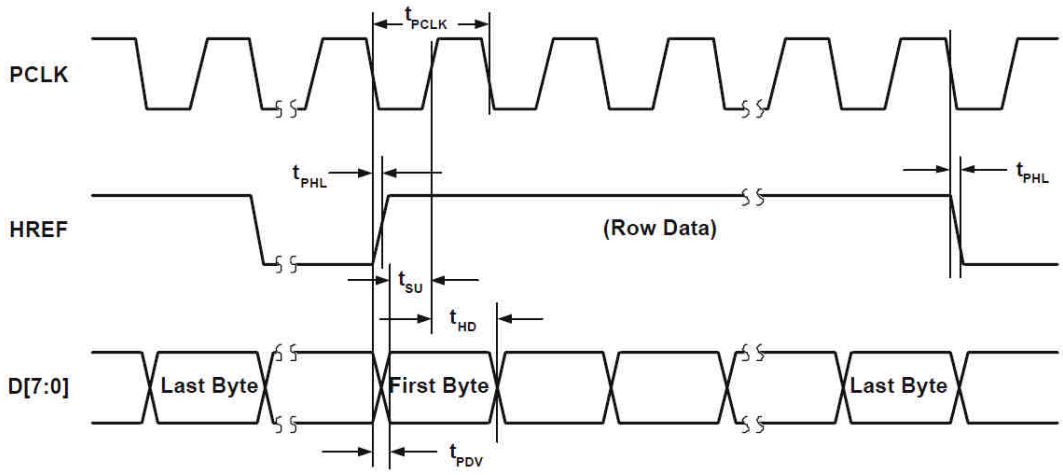

Figura 12 Detalle de la sincronización respecto al reloj de píxel

Dentro del código VHDL se ha definido la lógica que inicializa el contador de píxel de acuerdo con la señal VSYNC, lo incrementa de acuerdo con lo que marcan HREF y PCLK y, este contador, se utiliza para direccionar la RAM e ir llenando de valores cada una de las posiciones.

En la Figura 13 se deja un esquema del módulo. Aunque el diseño se ha hecho utilizando VHDL, se deja este esquema que, aunque no es exhaustivo en cuanto a diseño, permite comprender de forma sencilla el funcionamiento del módulo creado.

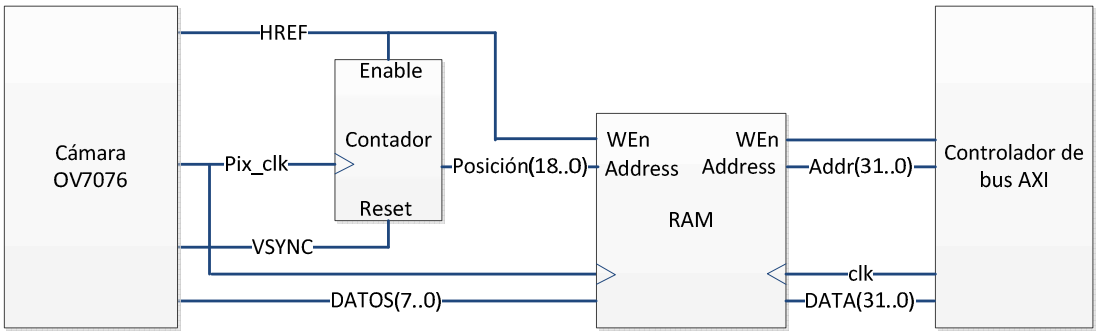

Figura 13 Esquema del módulo que controla la cámara

Cabe indicar que además, al ser un sensor en color, en el modo de configuración por defecto se van alternando consecutivamente valores de crominancia y luminancia en el bus de datos. Debido a esto, en el código se tiene en cuenta si estamos en dato par o impar para ir descartando la crominancia y guardando la luminancia.

Aunque todas las gráficas de temporización mostradas anteriormente (Figura 11 y Figura 12) son para cuando el sensor está configurado en resolución VGA, también es válida para resoluciones menores ya que, en este caso, hemos usado el *windowed-mode*, que no es más que definir un área rectangular de interés dentro de la región VGA y sacar al exterior sólo los valores correspondientes a ese área. En la Figura 14 puede verse cómo se comporta la señal HREF cuando se está en este modo.

Por último, durante el desarrollo de este trabajo, surgió un problema de sincronización entre la cámara y los *workers*, y es que estos últimos no tenían forma de saber qué píxel de la cámara se estaba cargando en la RAM. Si la tasa de fotogramas era muy alta, entonces el *worker* procesaría varias veces la misma imagen. Si por el contrario la tasa era muy baja, el *worker* no terminaría de procesar los datos antes de que se empezaran a meter datos dentro de la RAM.

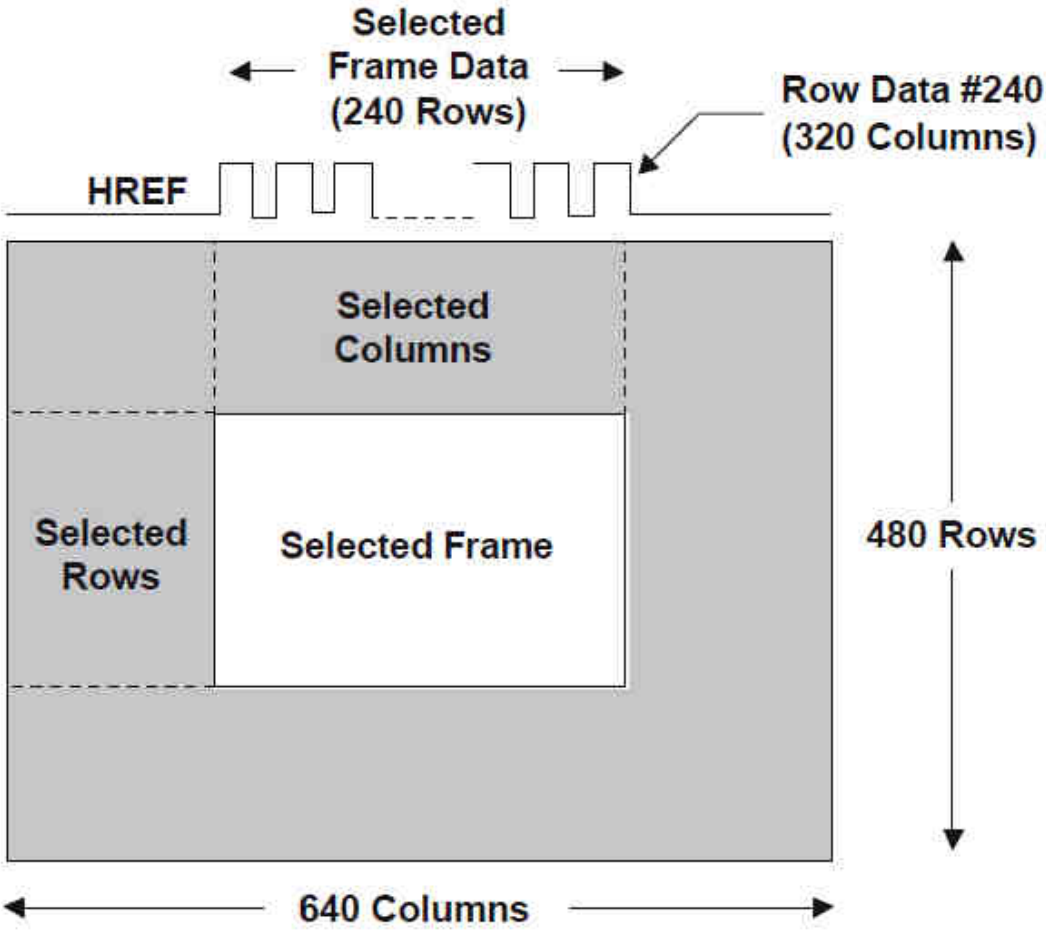

Figura 14 Selección de un área utilizando *windowed-mode*

Como se verá más adelante, la región de memoria en la que se guardan los datos se dividió en 4 secciones, de forma que cada *worker* procesara una región independiente. Para sincronizar la cámara con los *workers*, se ha utilizado el primer píxel de cada región de forma que, para este píxel, en lugar de llenar la RAM con el dato de la cámara se va llenando con un contador cíclico, que al desbordar vuelve a cero. De esta manera, el *worker* no empezará a procesar el área que le corresponde hasta que no detecte que el valor de ese píxel de sincronización ha variado. En la Figura 15 se puede ver el tamaño de cada una de las áreas en las que se ha dividido la imagen así como la posición de los píxeles de sincronización.

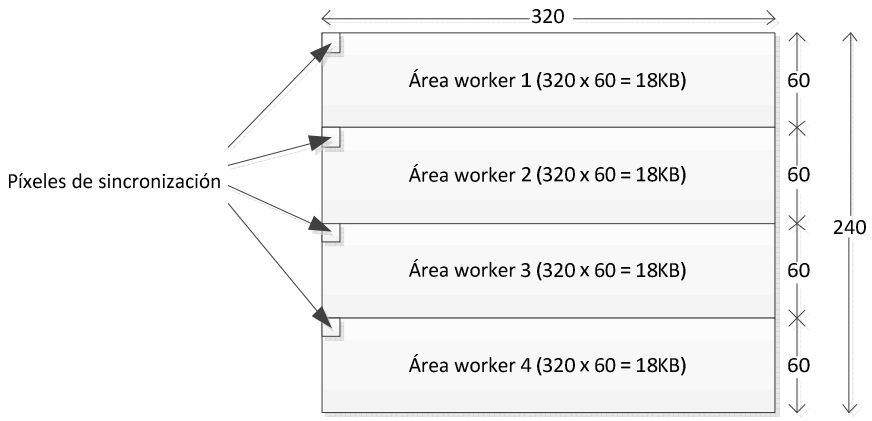

Figura 15 Píxeles de sincronización

2.1.1.2 Interfaz con el monitor VGA

En el diseño llevado a cabo se ha utilizado el conector VGA que lleva incorporada la placa Nexys4. A nivel físico, se utilizan 14 señales de la FPGA para crear el puerto VGA con una profundidad de 4 bits por color (en este diseño se utilizará sólo en escala de grises pero el cableado viene soldado en placa) y las dos señales de sincronismo horizontal y vertical. Las señales de color utilizan un circuito por divisor de tensión a base de resistencias que, junto con la impedancia de entrada de un monitor VGA (75 Ω) permite crear 16 niveles diferentes para cada color (rojo, verde y azul). Este circuito, mostrado en la Figura 16, produce señales de color de vídeo que van de 0 a 0,7 Voltios tal y como contempla el estándar VGA.

A pesar de que la Nexys4 está cableada para trabajar en color, todo el procesado que se ha llevado a cabo en este proyecto se ha hecho en blanco y negro. Al usar sólo 4 bits para cada uno de los colores y llevar la componente de luminancia a cada uno de ellos, el resultado es que la visualización en pantalla cuenta con 2^4=16 niveles de grises. Como se puede comprobar en la pantalla VGA, eso genera problemas de *banding* por falta de profundidad de bit. Este problema es una limitación de la salida VGA cableada en la Nexys, pero sólo a nivel de visualización ya que, internamente, todo el procesamiento se lleva a cabo a 8 bits de profundidad.

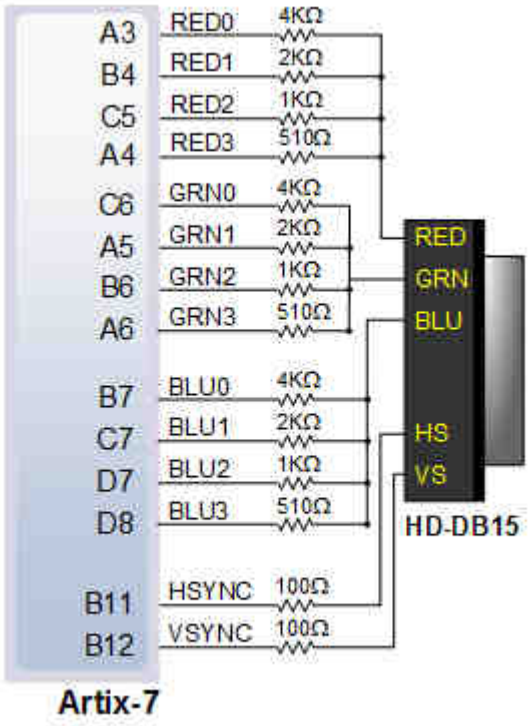

Figura 16 Circuito de generación de la salida VGA

Al igual que con el interfaz de la cámara, se ha utilizado el asistente de generación de periféricos y se ha adaptado para implementar una salida VGA. En este caso, a las señales creadas por el asistente se han añadido las correspondientes a la capa física de la señal VGA que están recogidas en la Tabla 7.

Señal VGA	**Descripción**
pxl_clk	Señal de reloj de píxel
vga_red_o (3..0)	Bus de 4 bits para el color rojo
vga_green_o (3..0)	Bus de 4 bits para el color verde
vga_blue_o (3..0)	Bus de 4 bits para el color azul
vga_hs	Señal de sincronismo horizontal
vga_vs	Señal de sincronismo vertical

Tabla 7 Señales añadidas al interfaz VGA

La salida VGA se ha utilizado con una doble función. Por un lado, se muestra la imagen que se está recibiendo de la cámara y, por otro, se muestra el fondo generado y que se utiliza como patrón para la detección de intrusos (esto se explicará en el apartado siguiente). Además, el fondo ha de ser almacenado en algún sitio para futuras iteraciones y, ya que la memoria es un bien relativamente escaso en la FPGA, se utiliza la propia memoria de la salida VGA precisamente para eso.

Por lo anteriormente expuesto, y dado que la RAM utilizada en este interfaz es escrita por los *workers* para luego volver a ser leída por estos mismos, se ha implementado una segunda lógica de lectura de la misma dentro del fichero *user_logic.vhd* correspondiente. Esta segunda lógica va leyendo los datos correspondientes tanto a la imagen como al fondo y poniéndolos a la salida en el bus de datos VGA de acuerdo con la temporización mostrada en la Tabla 8.

Temporización de línea			**Temporización de frame**		
Sección de la línea	**Píxeles**	**Tiempo(µs)**	**Sección del frame**	**Líneas**	**Tiempo(ms)**
Área visible	640	25,422	Área visible	480	15,253
Pórtico frontal	16	0,6355	Pórtico frontal	10	0,317
Pulso de sincronización	96	3,813	Pulso de sincronización	2	0,063
Pórtico posterior	48	1,906	Pórtico posterior	33	1,048
Línea completa	800	31,777	Frame completo	525	16,683

Tabla 8 Temporización de la señal VGA

Analizando la información de la Tabla 8 se deducen la frecuencia de píxel (25,175 MHz) y la velocidad de refresco vertical (31,468 KHz) para una tasa de refresco de 60 Hz.

En la Figura 17 se esquematiza el funcionamiento del módulo. Como se puede observar, se han ido generando los píxeles en la salida VGA de acuerdo con lo que se ha volcado en la RAM desde los *workers*. Dado que en la cámara se trabajaba a 320x240, se ha podido trabajar en pantalla con 2 imágenes completas. Como se muestra en la Figura 18, en la primera se puede ver la imagen que en ese momento está captando la cámara y en la otra puede verse el fondo generado. Esto deja la

mitad inferior de la imagen sin utilizar. Para ahorrar memoria de la FPGA, en el diseño en VHDL se pone a cero (negro) todo valor que no esté en la RAM.

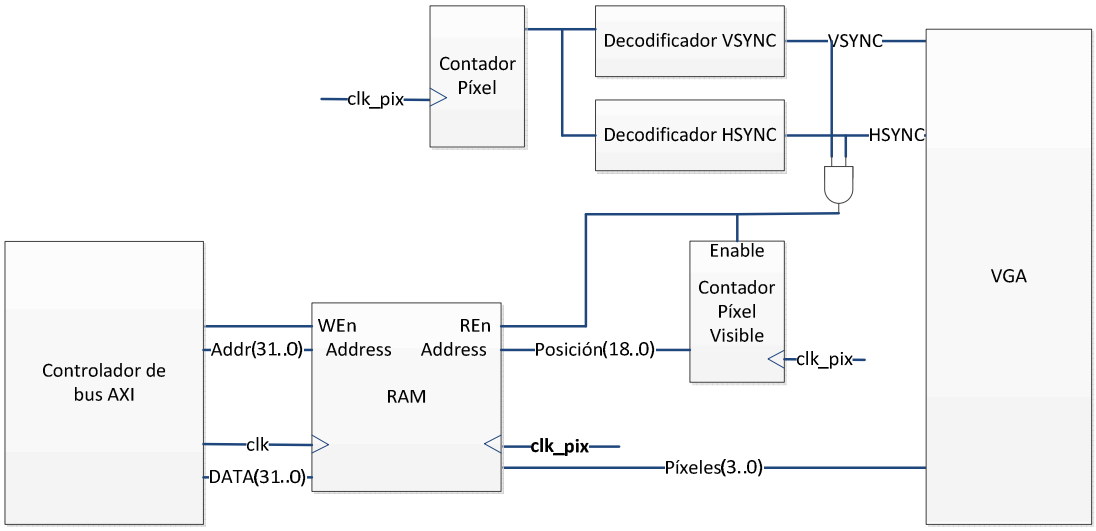

Figura 17 Esquema del interfaz con la salida VGA

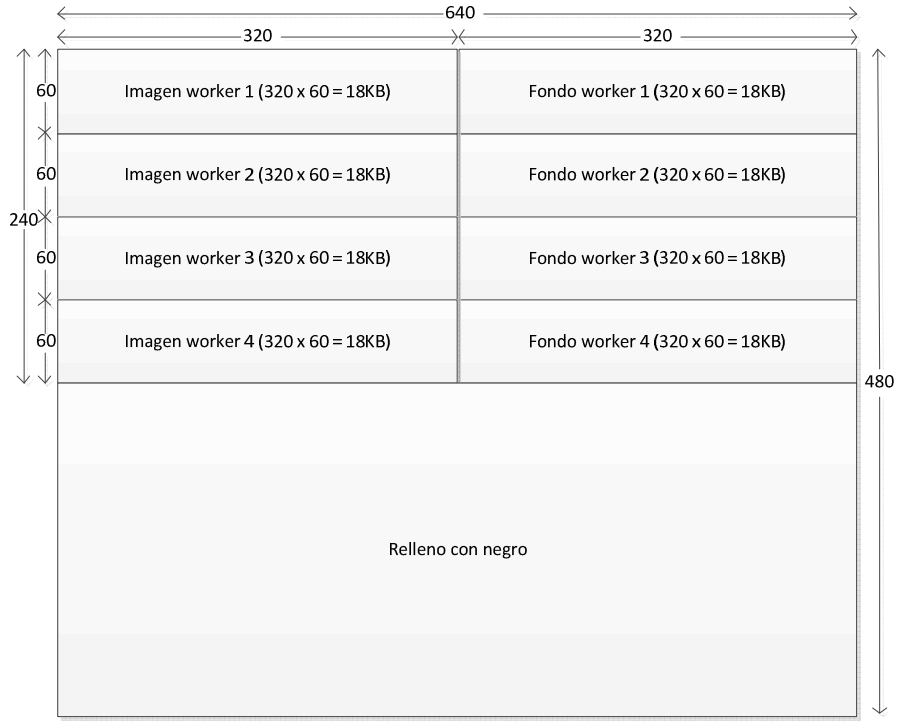

Figura 18 Distribución de la imagen VGA

El resultado final de la imagen en la pantalla VGA puede verse en la Figura 19, donde se muestra la imagen captada por la cámara a la izquierda, mientras que a la derecha, se ve el fondo en el que, en ese momento, se va incorporando el intruso. Más adelante se explicará todo este proceso.

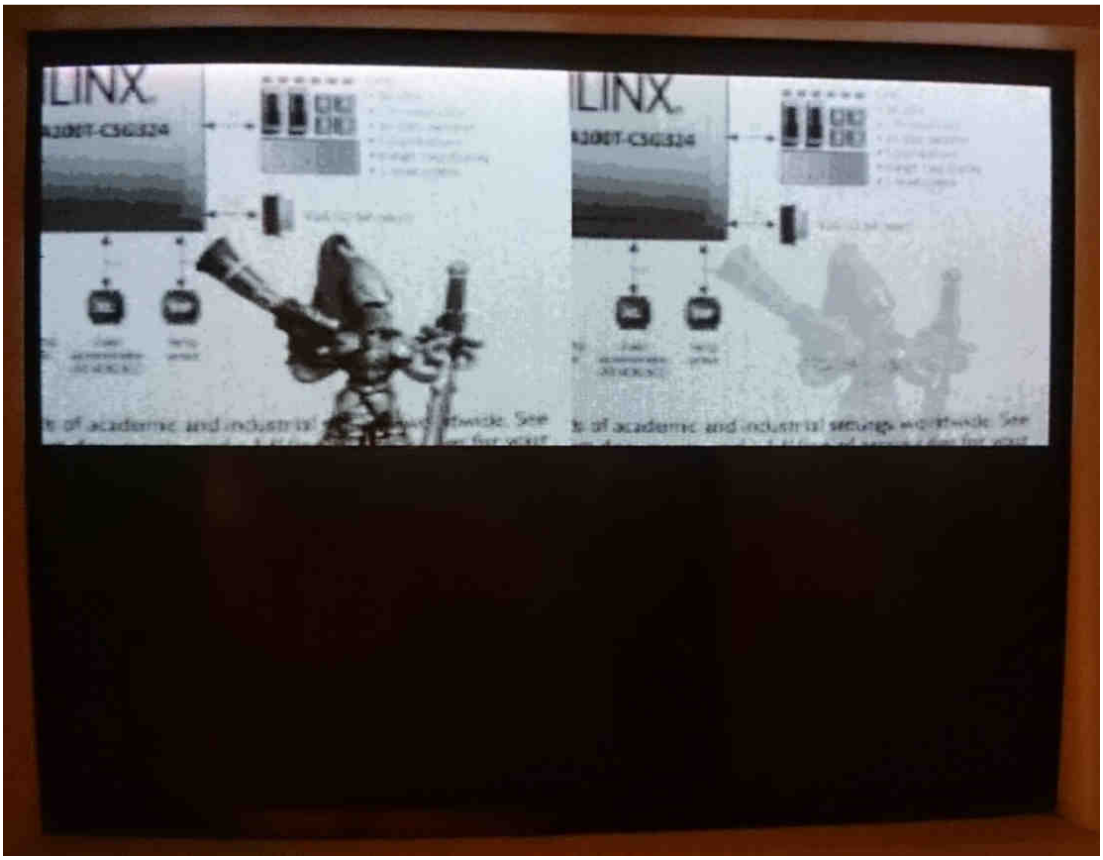

Figura 19 Fotografía de un monitor VGA

2.1.1.3 Configuración de los demás dispositivos de la placa Nexys4

La configuración de la mayoría de los dispositivos incluidos en la placa de desarrollo Nexys4 ha podido ser hecha de forma sencilla, tan solo arrastrando el dispositivo correspondiente al procesador dentro del entorno XPS, conectándolo a los buses adecuados y llevando a la salida los pines de entrada y salida.

Aún así, la configuración casi nunca pudo ser directa ya que el fabricante de dicha placa suministra plantillas para usarla dentro del entorno de diseño Vivado

(última plataforma de desarrollo de Xilinx) pero no se contaba con una licencia disponible para dicho entorno, con lo que tuvo que hacerse utilizando el XPS. Debido a esto, todos los dispositivos tuvieron que ser configurados a base de consultar hojas de características para parametrizarlos correctamente en función de la temporización de cada dispositivo y su pinado.

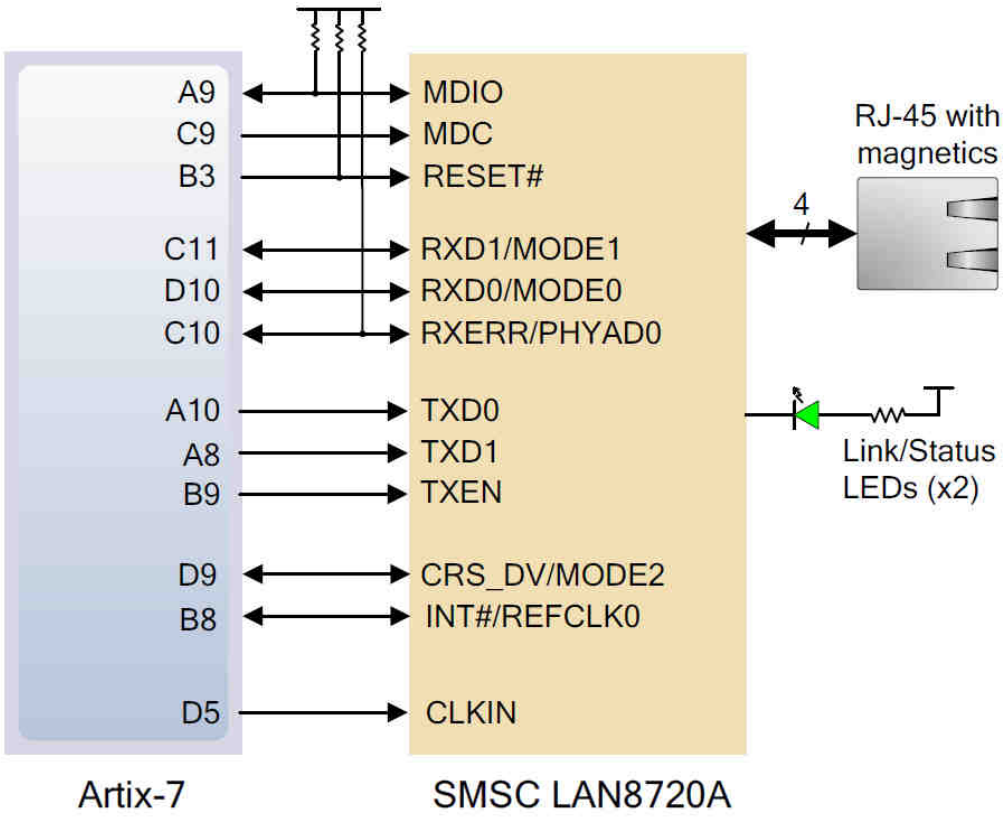

Figura 20 Conexiones físicas del controlador Ethernet

Especial mención requiere la configuración de la conexión Ethernet. La placa Nexys4 incluye un chip controlador de la capa física Ethernet 10/100 (SMSC LAN8720A en Figura 20) implementando el interfaz RMII (Reduced Media Independent Interface). Para poder controlarlo hubo que utilizar el IP core *axi_ethernetlite* que implementa la capa MAC de la conexión Ethernet. El problema viene de que dicho IP-core está preparado para conectarse a un interfaz físico MII (Media Independent Interface). Para solucionar dicho problema se tuvo que utilizar

un IP-core adicional que permite adaptación de señales de MII a RII, lo que complicaba bastante el diseño ya que, entre otras cosas, hacía necesario utilizar dos relojes de 50MHz a cada IP core pero con un desfase de 45º para que todo funcionara correctamente.

Otro de los dispositivos que fueron problemáticos fue la RAM de 16MB externa. Aunque inicialmente la intención era no utilizarla, al implementar una pila TCP/IP dentro de uno de los procesadores, la RAM interna resultó insuficiente (aunque todos los bloques de RAM hubiesen sido utilizados de forma exclusiva para el procesador lo habría sido también), con lo que hubo que hacer una configuración correcta de temporización del controlador de memoria para conseguir que el conjunto funcionara.

En concreto, el chip de RAM (Micron part number M45W8MW16) puede operar como una memoria asíncrona SRAM (en realidad PSRAM) con ciclos de lectura y escritura de 70 ns, o como una memoria síncrona con un bus de 104 MHz. Dado que el uso de esta memoria viene motivado por lo grande que es el código de la pila TCP/IP utilizada y no interviene en el procesamiento de vídeo, se ha optado por configurarla en modo asíncrono que es mucho más sencillo en cuanto a temporización.

3 DISEÑO SOFTWARE

El entorno de desarrollo de Xilinx ofrece la posibilidad de, una vez finalizado el diseño hardware en el Xilinx Platform Studio, exportar el diseño, junto con el fichero de programación, a un entorno de desarrollo software (conocido como Xilinx Software Development Kit o Xilinx SDK) para programar los procesadores definidos en la herramienta anterior. Una vez hecho esto, se generan de forma automática los drivers de cada uno de los dispositivos de forma que están disponibles para utilizar dentro del entorno de desarrollo software.

Este entorno de desarrollo utiliza como base el popular entorno de desarrollo Eclipse [18] , sobre el que el fabricante ha introducido herramientas específicas para desarrollo sobre FPGA, como llamadas a utilidades para programar los dispositivos, etc. Por lo demás, utiliza toda su potencia como entorno de desarrollo coloreando la sintaxis, facilitando el acceso a la ayuda, etc.

Para programar dentro de la FPGA el entorno ofrece diferentes posibilidades. Se puede programar tanto en C como en C++ directamente sobre el hardware de los procesadores con la ayuda de una serie de funciones de control de los dispositivos que genera el propio entorno.

Otra opción es programar las aplicaciones que se ejecutarán sobre un sistema operativo que residirá en la FGPA. En este caso, el SDK genera una distribución Linux que se ejecutará en los procesadores embebidos, o bien, un kernel específico de Xilinx (Xilkernel [13]) que proporciona cierto nivel de abstracción e hilos posix.

En este caso, y debido a que el código que se ejecuta en cada uno de los procesadores embebidos tiene una funcionalidad muy concreta, es posible abordar todo el diseño de forma directa sin la necesidad del soporte de un sistema operativo que, en nuestro caso, podría conducir a indeterminaciones en la temporización cuando asumiera el control de los procesadores.

3.1 Detección de fondo

Muchos sistemas de visión diseñados para funcionar en el mundo real requieren detectar cambios en la escena observada. Sistemas que analizan flujos de tráfico, hacen seguimiento de objetos en movimiento, monitorizan entornos industriales, etc., están basados en algoritmos de procesamiento de imagen cuyo primer paso es a menudo utilizar un algoritmo que detecte cambios en la escena.

Los métodos que emplean algoritmos basados en la diferencia entre imágenes, como el ejemplo de la Figura 21, realizando sustracciones entre imágenes consecutivas de una secuencia, son a menudo incapaces de detectar cambios reales en la imagen. Objetos moviéndose lentamente son difícilmente detectables. En general, estas técnicas conducen a una segmentación insatisfactoria de los objetos móviles y, por lo tanto, llevan a una incorrecta extracción de la verdadera silueta de los objetos debido, en parte, porque la diferencia de valores se manifiesta tanto en la posición actual del móvil como en la anterior.

Figura 21 Detección por sustracción

Algunos autores [10] han propuesto métodos de detección de cambios basados en la sustracción de una imagen de referencia, representando el fondo de la

escena dinámica de la imagen de entrada, como se observa en la Figura 22. La habilidad de estimar y usar la información de fondo lleva a algunas ventajas: procesamiento simple de datos, eficiente detección de cambios en la escena y robusta segmentación dinámica de los objetos que se mueven.

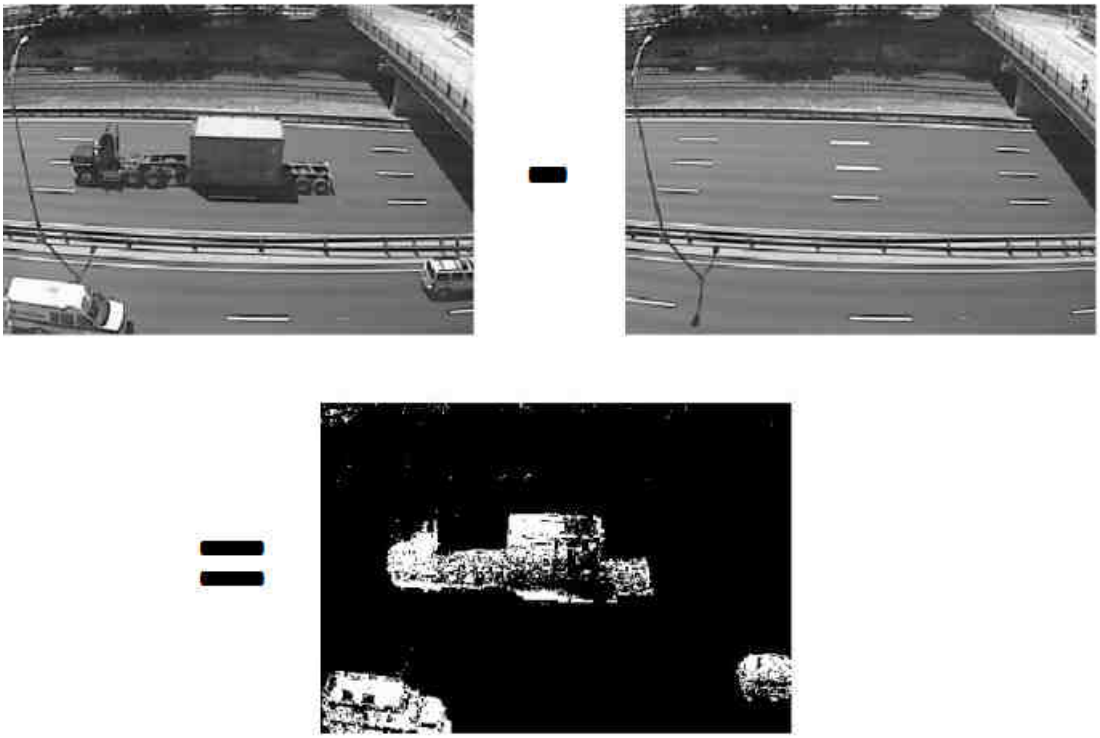

Figura 22 Detección de movimiento por sustracción respecto a un fondo

Una amplia familia de algoritmos estiman los niveles de gris del fondo utilizando métodos adaptativos basados en filtrado paso bajo. En este caso, algunos problemas pueden llegar con cambios inesperados en las condiciones de iluminación o con alteraciones estructurales de la escena observada. Otros problemas afectan a la estimación del fondo de imagen cuando objetos móviles están presentes en la escena y cubren el fondo por largos intervalos de tiempo.

Para pasar por encima este problema, una estrategia común consiste en llevar un contador para cada píxel indicando durante cuantas imágenes consecutivas ha habido algún cambio en éste [11] . De esta forma, se incorporarán cambios en el fondo sólo si éstos han persistido durante un tiempo elevado. Este enfoque trae

asociado el problema de necesitar un contador para cada píxel, lo que hace necesaria una cantidad importante de memoria. Como alternativa está la posibilidad de hacer una estimación de donde hay móviles y adaptar rápidamente los cambios en las regiones de fondo y lentamente los que sucedan en las regiones asociadas a los móviles o al primer plano.

El éxito de estos métodos depende directamente de la habilidad de detectar adecuadamente los móviles de la escena. De esta forma, detecciones erróneas pueden acumularse en el tiempo y el proceso total puede resultar insatisfactorio a la hora de realizar una robusta segmentación de móviles y fondo. Otros métodos basados en filtros más complejos son sensibles a errores en el modelo y requieren una fase crítica de ajuste de parámetros.

3.2 Algoritmo de extracción de fondo de Karmann y von Brandt

El modelo de actualización de fondo utilizado en el proyecto es una versión adaptada del algoritmo de Karmann y von Brandt [12] , que consiste en una relación recursiva para introducir cambios en el fondo a la velocidad que marcan dos factores (uno rápido y otro lento) diferentes, F_t y S_t (Fast y Slow) tal y como se muestra en la ecuación (1)

$$B_{i,j,t} = B_{i,j,t-1} + (F_t - S_t \cdot M_{i,j,t}) \cdot (I_{i,j,t} - B_{i,j,t-1}) = B_{i,j,t-1} + (F_t - S_t \cdot M_{i,j,t}) \cdot D_{i,j,t}$$

$$\text{donde} \quad M_{i,j,t} = 1, \ \text{si} \left| D_{i,j,t} \right| > T_t; \quad M_{i,j,t} = 0 \ \text{en otro caso.} \tag{1}$$

$$\text{Si } M_{i,j,t} = 1 \Rightarrow B_{i,j,t} = B_{i,j,t-1} + (F_t - S_t) \cdot D_{i,j,t}$$

$$\text{Si } M_{i,j,t} = 0 \Rightarrow B_{i,j,t} = B_{i,j,t-1} + F_t \cdot D_{i,j,t}$$

En este algoritmo, $B_{i,j,t}$ representa el nivel de gris de un píxel de fondo en el instante t, $I_{i,j,t}$ el nivel de brillo correspondiente al píxel de la imagen de entrada, y $M_{i,j,t}$ es una máscara binaria definida en relación a un umbral T_t, tal y como se indica en la ecuación (2).

$$M_{i,j,t} = 1,\ si \left| D_{i,j,t} \right| > T_t\ ;\quad M_{i,j,t} = 0\ en\ otro\ caso. \qquad (2)$$

El sentido de $M_{i,j,t}$ es el siguiente: si el valor absoluto de la diferencia entre la imagen actual y el fondo anterior (VAD) es menor del umbral T_t, se asume que es debido a cambios de iluminación y, en tal caso, el fondo será modificado de acuerdo con el factor rápido F y si el cambio permanece tras un corto número de imágenes, se incorporará al fondo. Por otro lado, si el VAD es mayor que ese umbral, lo que se asume es que un objeto móvil ha entrado en la región de interés y está ocultando temporalmente el fondo, por lo que éste será actualizado con el factor más lento (F_t–S_t). De esta forma se mantiene el VAD más tiempo y causa la diferenciación del objeto durante un intervalo más largo, aunque si permanece por un periodo suficientemente largo, acabaría incorporado en el fondo.

El punto crítico llega cuando se dan dos casos: pequeños cambios de brillo pueden deberse a objetos en movimiento con poco contraste con el fondo. En realidad, la visión humana podría tener un problema similar (camuflaje, mimetismo, etc.). Por otro lado, cambios bruscos pueden ser causados por violentos cambios como sombras, reflexiones de la luz, etc., en lugar de por objetos móviles, especialmente en escenarios externos. Además, en ciertas condiciones de iluminación las sombras proyectadas por los móviles pueden confundirse con móviles en sí.

Como se puede deducir, el sistema opera con cuatro entradas: imágenes $I_{i,j,t}$ y controles dados al sistema T_t, F_t y S_t, estos tres relacionados con el contexto de la escena y, por lo tanto, sujetos a ser cambiados en función del entorno para mantener la eficiencia en un nivel aceptable basándose en dos salidas, el fondo actualizado y la máscara de móviles, que permite ajustar Ft y St a los rangos de velocidades esperadas. Este proceso se describe en la sección siguiente.

3.2.1 Ajuste de los parámetros del extractor de fondo

Para conseguir un correcto funcionamiento del sistema, es crítica una elección acertada de los parámetros que controlan el proceso de extracción de imágenes de fondo, ya que si no se hace adecuadamente se puede llegar a puntos en los que el

fondo se actualice demasiado rápido y los móviles no se reconozcan como tal o todo lo contrario, es decir, que un móvil que se para no llegue a ser nunca parte del fondo como lógicamente debe hacer.

3.2.1.1 Ajuste del umbral T_T

Este parámetro es el que determina si un píxel, y por ende una región, es considerada móvil o no. Si el valor absoluto de la diferencia entre el píxel actual y el píxel del fondo anterior (VAD) es mayor que este umbral (T_t), se determinará que en ese píxel hay un móvil. De esta forma, cuanto más bajo sea el umbral más selectiva será la detección, y a un mínimo cambio del nivel de gris se detectará la existencia de un píxel de una región móvil.

Esto puede llevar asociado un problema y es que, si el umbral es demasiado bajo, el mismo ruido generado en el sensor de imagen puede inducir a pensar que hay un móvil donde no lo hay, lo que puede hacer fallar la detección. Por otra parte, si es demasiado alto, móviles que se diferencien poco del entorno (colores que generen tonos de gris parecidos, etc.) serán interpretados de forma incorrecta como fondo de la imagen y serán incorporados rápidamente a este llevando a un error en la actualización del fondo y, por tanto, a la larga, a errores de detección.

En la práctica, la elección del umbral dependerá de la intensidad de iluminación de la escena y el contraste existente entre el fondo y los móviles, a fin de que los móviles sean correctamente detectados. Valores adecuados son los comprendidos en un margen de entre 15 y 50 para una imagen codificada en blanco y negro con 256 niveles de gris.

3.2.1.2 Ajuste de factores rápido y lento F_T y S_T

Estos factores (F_t y S_t) regulan la velocidad a la que se actualiza el fondo en función de si el objeto ha sido reconocido como móvil o como fondo conforme al resultado que haya determinado la aplicación del umbral.

Estos dos factores están íntimamente relacionados con la dinámica de la escena, es decir, tendrán que ser necesariamente diferentes los utilizados para una escena con cambios súbitos (una carretera con coches circulando a gran velocidad) y los usados en una escena con cambios más lentos, como pueden ser peatones paseando.

En primer lugar se considerará el factor rápido (F_t). Este factor se utiliza para actualizar la parte de fondo de la escena de forma rápida, de tal manera que las variaciones en la iluminación sean incorporadas al fondo lo antes posible para tener un fondo representativo actualizado. Este factor depende del entorno en el que se esté utilizando el sistema, de forma que, si estamos en un escenario exterior, los cambios de iluminación son de esperar. Por ello, el factor debe ser suficientemente alto como para conseguir que la oclusión del sol por una nube, etc. no haga fallar al sistema considerando el cambio de iluminación como un móvil que ocupa toda la escena. Esto se soluciona incorporando los cambios de brillo rápidamente al fondo, es decir, aplicando un factor F_t alto. Un factor rápido de alrededor de 0.25 en esta situación puede considerarse como adecuado para este escenario.

En escenarios interiores, con iluminación artificial o leve exterior difusa, este factor puede ser mucho más bajo, ya que la iluminación se puede considerar constante lo que hará que los cambios en el fondo, si existieran, sean más debidos a las reflexiones de la iluminación en otros objetos. Esto genera menores diferencias y se podrá utilizar un factor rápido más bajo. Para estas circunstancias un valor de aproximadamente 0.04 puede considerarse apropiado.

Respecto al factor lento, S_t, hay que tener en cuenta que este factor es el encargado de incorporar al fondo partes de la imagen que dejan de ser móviles, ya sea porque un objeto ha llegado a la escena permaneciendo parado en algún lugar de ésta, o bien porque un objeto que estaba parado en la escena se ha marchado dejando un hueco que ha de ser incorporado al fondo.

En realidad, más que de valor, para el factor lento se ha de hablar del valor que debe tomar la diferencia entre el rápido y el lento (F_t-S_t) ya que es así como se aplica en el algoritmo (ver ecuación (1)). Para esta diferencia, un valor que se puede considerar adecuado es un octavo de las cantidades utilizadas para el factor rápido, es decir, entre aproximadamente 0.03 y 0.005. La cantidad realmente utilizada dependerá lógicamente del escenario utilizado. Si los móviles son rápidos, el factor debe ser lo mayor posible para determinar rápidamente cuando un móvil se para y se incorpora al fondo, pero si los objetos son lentos debe ser un factor pequeño para evitar que sea incorporado al fondo cuando en realidad se está moviendo lentamente.

3.2.2 Dependencia con la tasa de fotogramas

Por último, hay que indicar que, dado que todos los parámetros aquí utilizados son aplicados para cada frame que llega del sensor de imagen, existe una fuerte dependencia del comportamiento del sistema con la tasa de fotogramas. Como se verá más adelante (sección 4.2), la implementación llevada a cabo no pudo implementarse a 25 fps con lo que finalmente hubo que hacer un pequeño ajuste empírico sobre dichos parámetros.

3.3 Reparto de tareas

Como se ha comentado anteriormente, en el diseño hardware se han definido5 procesadores con diferentes periféricos, 4 de ellos llevarán a cabo el proceso. En la Figura 24 puede verse un diagrama del sistema completo. Como puede apreciarse, los *workers* son los únicos que tienen acceso a los dispositivos de imagen (cámara y salida VGA). Cada uno de ellos estará haciendo el procesamiento de su región para generar la imagen fondo y la detección de intrusos en paralelo. Cada vez que terminan de procesar un frame, pasan la información relativa a intrusos al procesador c*ontroller* utilizando un dispositivo de comunicación inter-procesador tipo MailBox.

Este tipo de dispositivo cuenta con un doble interfaz de bus AXI como se ve en la Figura 23. Para habilitar la comunicación entre procesadores sólo hay que conectar cada uno de los buses de este IP core a el bus AXI de cada procesador. A

partir de ese momento se puede usar el driver de comunicación con el MailBox que permite enviar un dato al mismo. Por otro lado, el mismo driver puede utilizarse para recibir un dato.

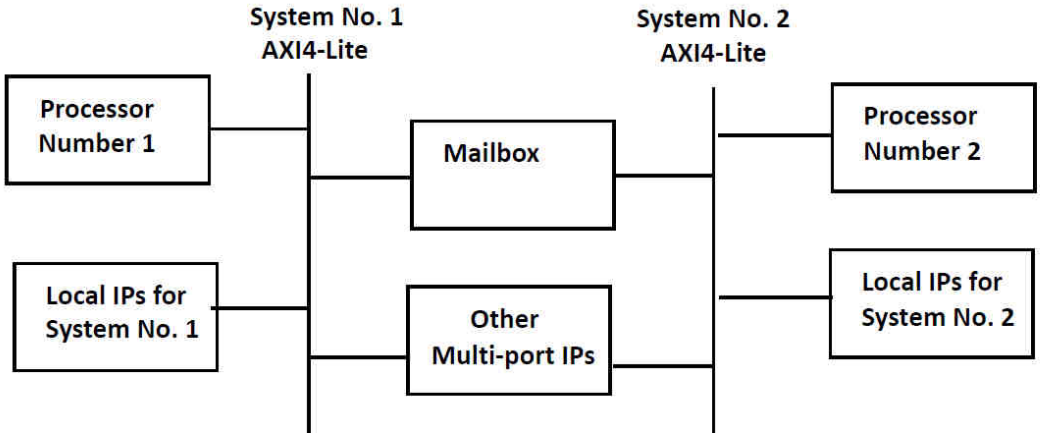

Figura 23 Dispositivo MailBox

Internamente, cada dispositivo MailBox dispone de una memoria FIFO para cada uno de los sentidos de comunicación e implementa control de *overflow* y *underflow* de la misma. Además, el driver generado para el desarrollo software permite acceder al estado de las FIFOs así como hacer el vaciado de las mismas.

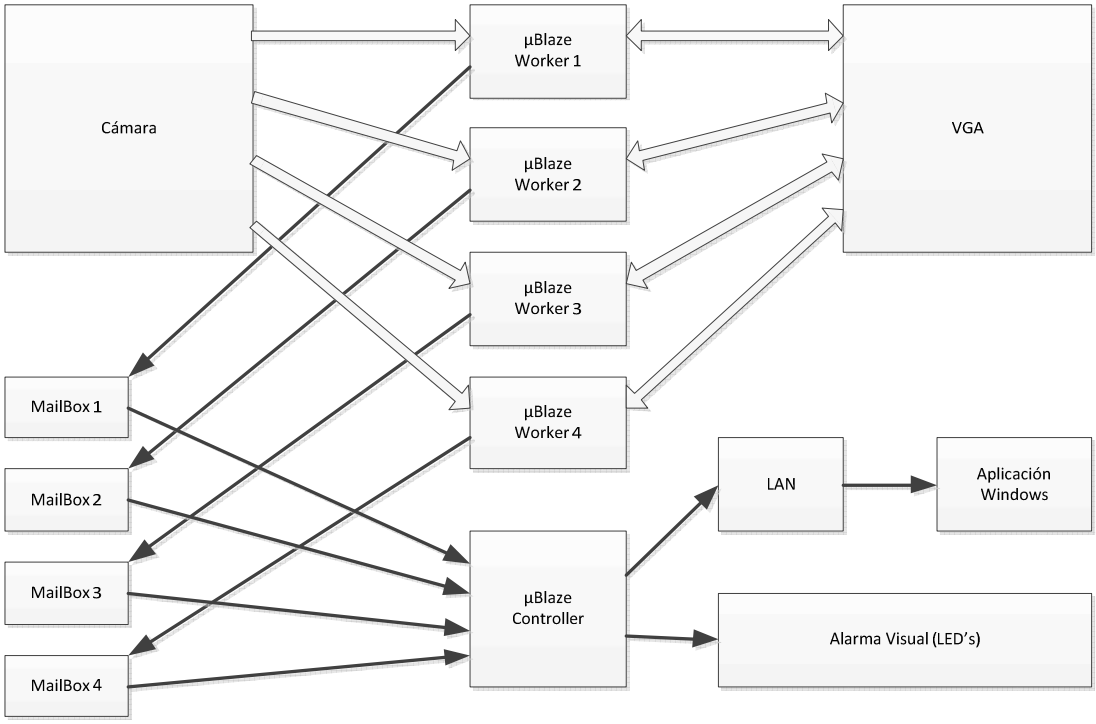

Figura 24 Diagrama funcional del sistema completo

Por su parte, el *controller* permanece en espera hasta que recibe la información oportuna de cada uno de los *workers*. Cuando la recibe, combina la información de los cuatro y, por un lado, actualiza el valor de los LEDs para generar una alarma visual y, por otro, interactúa con el interfaz Ethernet para generar una alarma por UDP hacia una dirección de red.

Por último, se ha diseñado una sencilla aplicación Windows que permite recibir los comandos que envía el *controller* para mostrar en pantalla un aviso en función del nivel de alarma recibido.

A continuación se va a analizar el código que se está ejecutando en cada uno de los procesadores.

3.3.1 Código en los *workers*

Los *workers* son los encargados de realizar todo el procesado de vídeo. Es por esto por lo que tienen acceso a las zonas de memoria que tienen mapeados los datos

de imagen tanto de la cámara como de la salida VGA. Además, los 4 *workers* hacen exactamente el mismo tipo de procesamiento, salvo que cada uno de ellos accede a una parte determinada de la imagen.

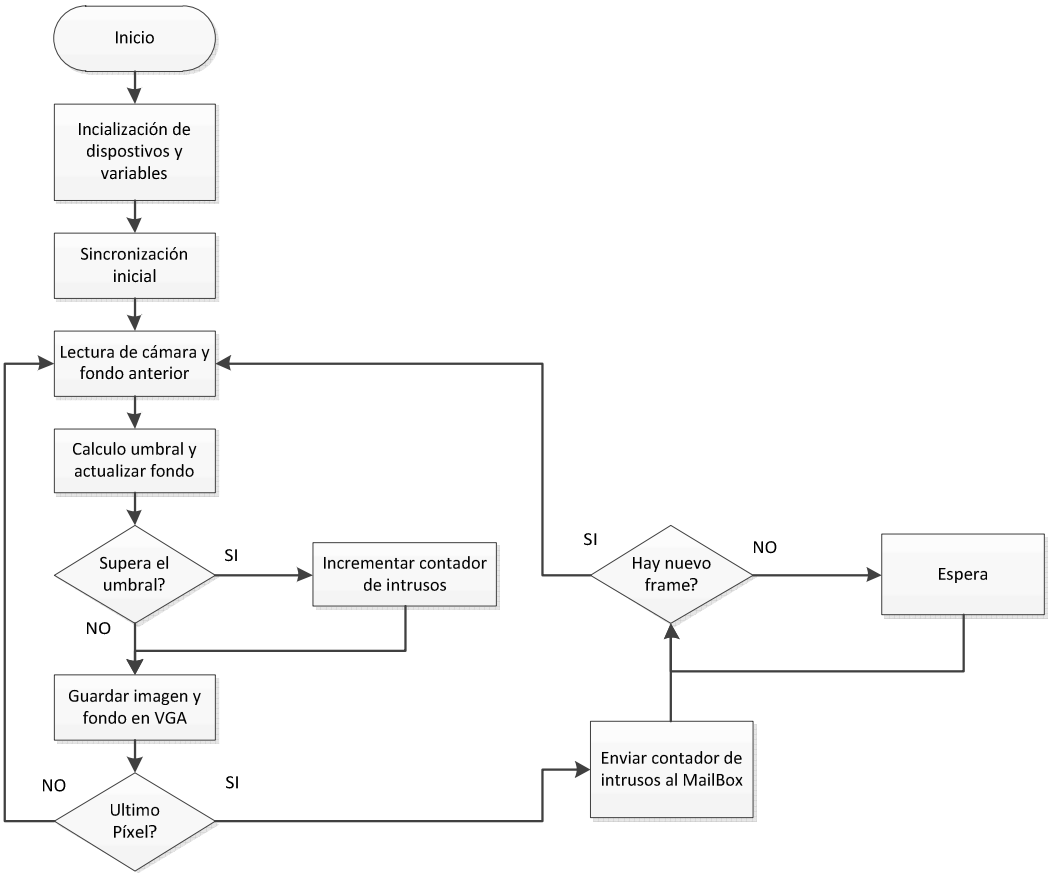

Figura 25 Diagrama de flujo del código de los *workers*

Con esto último en mente, y con el objetivo de poder reutilizar el código generado para uno de los *workers* en los otros tres, se ha hecho un diseño en función de un parámetro correspondiente al procesador en concreto dentro de la propia FPGA. De esta forma, el mismo código se ejecuta en los 4 *workers* pero accediendo a regiones diferentes de memoria.

Como se muestra en el diagrama de la Figura 25, el código está dividido en dos partes. Primero se inicializa todo el sistema y, una vez hecho esto, se entra en un bucle infinito que se encarga del procesamiento propiamente dicho. En primer lugar

se lee el valor de un píxel de la cámara y el fondo anterior. Con estos dos datos se actualiza el fondo utilizando el algoritmo de extracción de fondo (sección 3.2) y se incremente el contador de intrusos en caso de que se haya superado el umbral de detección.

Este proceso se repite para todos los píxeles de área asignada a ese *worker* para, a continuación, enviar el número de píxeles alarmados hacia el procesador *controller* utilizando el MailBox que tiene asignado.

Por último, el procesador espera hasta que su correspondiente píxel de control cambia de valor para proceder al procesamiento del siguiente frame.

El propio entorno de desarrollo permite crear programas de ejemplo que interactúan de forma básica con los distintos periféricos. En este caso se ha partido de un programa de ejemplo que hace un test de memoria de todas las regiones del sistema. Esto es muy conveniente en este caso ya que el interfaz con la cámara y con la salida VGA utiliza mapeo de memoria para intercambio de datos. Adicionalmente a eso, se le han añadido las rutinas necesarias para controlar los dispositivos tipo MailBox y poder enviar hacia el procesador *controller* la información necesaria.

3.3.2 Código en el *controller*

El procesador *Controller* es el responsable de procesar la información que le pasan los *workers* y de gestionar las alarmas. Aunque debido al tipo de dispositivos con los que interactúa, el código que ejecuta este procesador es relativamente complejo, desde un punto de vista funcional se puede hacer un diagrama muy simple que representa su funcionalidad, tal y como se muestra en la Figura 26.

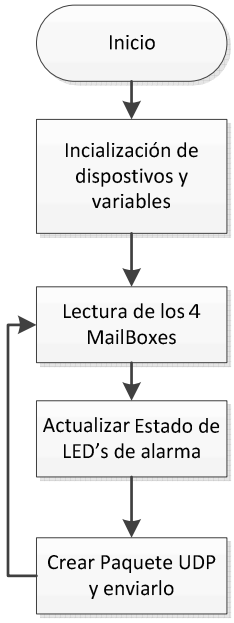

Figura 26 Diagrama de flujo del código del *controller*

En esta ocasión se ha vuelto a utilizar uno de los ejemplos que ofrece la plataforma de desarrollo para probar los dispositivos asociados al procesador. Concretamente, el código se ha programado partiendo el ejemplo de un servidor de eco sobre TCP. Este ejemplo, creado para ilustrar el funcionamiento de la pila TCP/IP LightweightIP, asigna dirección MAC al interfaz físico, le asigna una IP y, a continuación, pone en escucha un socket TCP en el puerto 7.

Cuando dicho puerto recibe una conexión, el programa la acepta y, a partir de ese momento, todo lo que se recibe por el socket es a su vez enviado de vuelta implementando un eco simple. Para el código del proyecto se ha sustituido toda la parte TCP por una sección de código que crea un paquete UDP y, posteriormente, lo envía hacia una IP prefijada.

Previo a la notificación por red, el código lee los cuatro MailBoxes que interconectan el procesador con los *workers* (ver Figura 24) para obtener el número de píxeles alarmados en cada una de las áreas de observación y poder totalizarlos para poner dicha información en el paquete UDP que se enviará.

3.4 Ajuste del sistema

Cuando se comenzó con el diseño resultaba muy complicado poder predecir el comportamiento del sistema en cuanto a velocidad de proceso. Las estimaciones se podían hacer utilizando la información disponible en ese momento. La información que debía procesar cada *worker* viene determinada por el número de píxeles a procesar. Para una imagen a tiempo real a 25 fps serían 480.000 píxeles/seg, como se ve en la ecuación (3).

$$\frac{25 * frames}{segundo} * \frac{(320 * 240)}{4} = 480.000 \frac{píxeles}{seg} \quad (3)$$

Además, revisando el algoritmo aplicado para la actualización del fondo, el procesado completo lleva un número determinado de operaciones:

- Cálculo de posición de memoria del píxel
- Lectura de píxel
- Cálculo de posición de memoria del fondo anterior
- Lectura de fondo anterior
- Cálculo del valor absoluto de la diferencia
- Comparación con el umbral
- Aplicación del factor de actualización correspondiente
- Cálculo de posición de memoria del píxel
- Escritura del píxel
- Cálculo de posición de memoria del fondo actualizado
- Escritura del fondo actualizado
- Incremento de contador de píxel
- Comparación de fin de fotograma

Se llevó a cabo un estudio rápido de la carga computacional que exige la aplicación del algoritmo teniendo en cuenta que el propio procesador vería su comportamiento en función de la estrategia de optimización que se utilice al definirlo

en el XPS que, en este caso, se optó por la optimización en área. Estos cálculos eran bastante sencillos para las operaciones básicas que se realizaban dentro del procesador y su memoria local, pero no era tan sencilla la estimación para los accesos a memoria de los dispositivos.

Como se comentó en el correspondiente apartado, el interfaz con la cámara y la salida VGA se ha implementado como dispositivos de memoria mapeada. Se hace necesario acceder a esas posiciones de memoria a la hora de leer y escribir los píxeles. Además, al menos en el interfaz con la cámara, había que desactivar la caché de lectura, ya que, si no se hacía, el procesador podría cachear un dato que ya hubiera sido escrito por parte de la cámara. Todo esto, unido a la incertidumbre de los tiempos de acceso a esta memoria mapeada (que se lleva a cabo a través de un bus AXI compartido para otros procesadores y otros dispositivos) hacía poco fiable cualquier estimación que se pudiera hacer a priori.

En cualquier caso, sí que se podía llegar a una conclusión, y es que dado el número de píxeles a procesar por cada *worker* (480.000 pixel/seg) y a una frecuencia de trabajo de los mismos de 100 MHz, hay disponibles 208 ciclos de reloj por píxel lo que indica que, el cuello de botella del sistema no se localiza en el tiempo de proceso necesario, sino en los accesos a memoria.

Visto todo lo anterior, se hizo patente que el ajuste y optimización del sistema tendría que hacerse de forma empírica, y así se hizo cuando el sistema estuvo compilado.

Al haber preferido hacer la configuración de la cámara con un dispositivo externo de fácil programación (una placa Arduino), se pudo ir reprogramando dicho dispositivo para que configurara a tasas de fotograma diferentes hasta encontrar la adecuada.

El procedimiento está basado en prueba y error. Si la tasa de fotogramas era demasiado alta para poder ser procesada, la sincronización entre la cámara y el MicroBlaze fallaría, ya que, para cuando se quisiera procesar el siguiente frame, el

contador cíclico de sincronización habría cambiado en más de uno, con lo que habría que esperar a que este contador desbordara. Esto, en la pantalla VGA, se mostraba como un congelado de imagen durante un tiempo determinado. Si por el contrario, la imagen progresaba de forma más o menos fluía significaba que se estaba por debajo de la tasa de fotogramas máxima.

Tras varias pruebas se llegó a una tasa de 4 frames por segundo. Aunque para un sistema de detección de intrusión una tasa de este orden podría ser aceptable, resultaba bastante más bajo de lo esperado.

Viendo el resultado empírico y teniendo en cuenta lo expuesto anteriormente sobre los accesos al bus AXI, se analizó cuidadosamente el código, encontrando un cuello de botella creado por el mecanismo de sincronización. Dado que el primer píxel de cada área se utiliza para sincronizar la cámara con los *workers*, cuando se ha terminado el procesado del fondo, el programa se dedica a leer una y otra vez el píxel de sincronización hasta que este cambia de valor. Este procedimiento causa que se esté accediendo de forma continua al bus y, al ser un bus compartido para todos los *workers*, se genera un tráfico excesivo en el mismo y, por tanto, un cuello de botella.

Para evitar esto, se introdujo dentro del código un pequeño delay entre comprobaciones del píxel de sincronización para aliviar la carga del bus. Simplemente con esta medida y tras volver a buscar empíricamente el punto óptimo de funcionamiento, se pudo subir el funcionamiento a 10 fps, es decir, más del doble con introducir un sencillo mecanismo de prevención de congestión del bus.

3.5 Aplicación Windows receptora

Dentro de los objetivos del proyecto se marcó el poder enviar alarmar a través de la red, de forma que el sistema se pudiera comportar como un detector remoto de intrusión. Para ver qué era lo que estaba enviando la placa a la red, inicialmente se utilizó la herramienta de monitorización de red Wireshark, que permite analizar todo el tráfico del interfaz de red seleccionado como se ve en Figura 27.

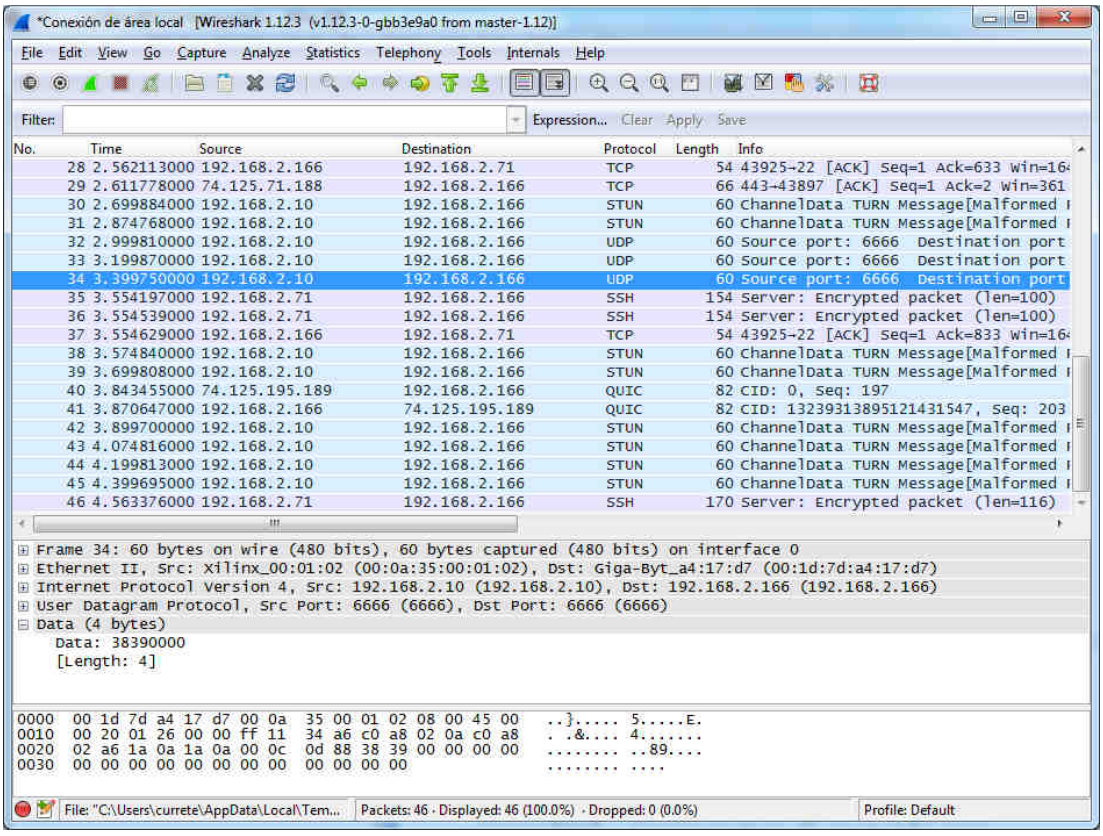

Figura 27 Escaneo de red con Wireshark

Aunque Wireshark permite crear filtros sobre el tráfico de red, y dado que no era algo demasiado complejo, se consideró más operativo desarrollar una pequeña aplicación Windows que permitiera, mediante un interfaz muy básico, monitorizar la información enviada por la Nexys y visualizarla en pantalla.

Para ello, se ha utilizado el Microsoft Visual Studio 2010 y se ha desarrollado la aplicación .NET que se muestra en la Figura 28. Esta aplicación se activa pulsando el botón "Conectar". Una vez hecho esto, en la barra se muestra el porcentaje del total de píxeles de la imagen que están alarmados. Con el deslizador se puede elegir el umbral de alarma que hará que el botón "Conectar" tome el color rojo y, por último, en el cuadro "Píxeles Alarmados" se muestra el valor numérico de píxeles en los que se estima que hay un intruso.

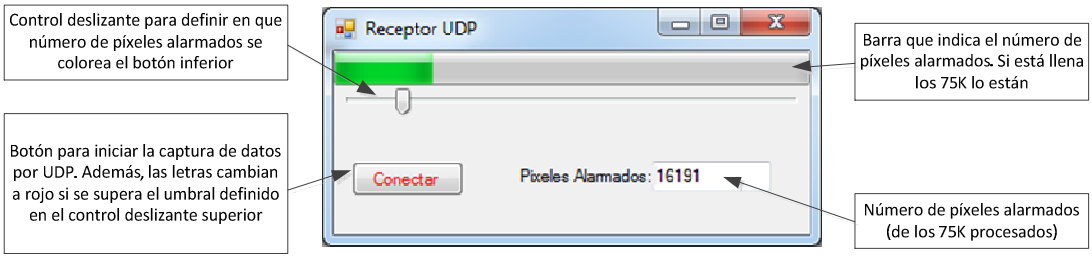

Figura 28 Aplicación receptora de alarmas

En la Figura 29 se puede ver un diagrama con la funcionalidad implementada en esta aplicación. Se ha obviado en el diagrama el proceso asíncrono de modificar el valor del control deslizante que define el nivel de alarma.

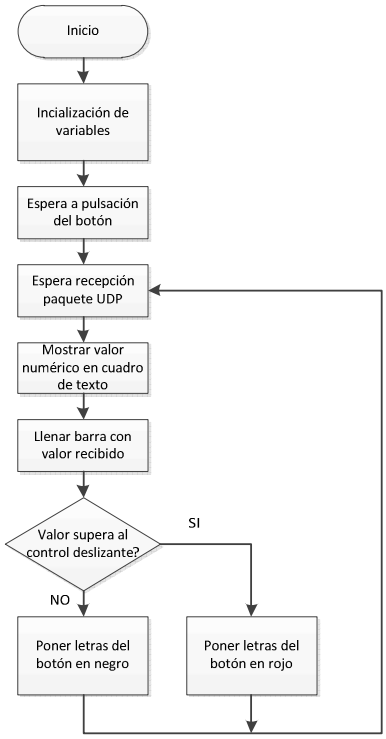

Figura 29 Diagrama de flujo de la aplicación Windows

4 RESULTS

This section shows the results obtained during the development of this work, both software and hardware.

4.1 FPGA implementation results

Now we will analyze the results obtained in the hardware design that has been made. First of all we are going to analyze the inputs and outputs of the FPGA what will give us an idea of controlled devices and, after that, we will see the chip occupation where some of the design decisions will be shown.

4.1.1 Global input and outputs

As it has been told previously, here we are going to see the global inputs and outputs of the FPGA. In Xilinx design suite those are defined in an ".ucf" format file. Although here is not include that file, in Table 9 all signals involved in the design can be seen, grouped by the device they are connected to.

Device	Signal	Description
Clock	clk	On-board clock 100 MHZ
Reset	Reset	On-board button to reset it
Leds	Led(0..15)	On-board multipurpose Leds. Shows alarm level
UART	RsRx	Uart receive (used for debugging)
	RsTx	Uart receive (used for debugging)
RAM Memory	RamCLK	RAM clock
	RamADVn	(Address Valid) Address bus validation signal
	RamCEn	Memory Chip Enable
	RamCRE	Signal to access to control register of the memory
	RamOEn	Output enable
	RamWEn	Write enable
	RamBEN(0..1)	Byte access selection

	MemDB(0..15)	Data bus
	MemAdr(0..23)	Address bus
Network interface	PhyMdc	Management interface clock
	PhyMdio	Management interface data bus
	PhyRstn	MAC layer reset
	PhyCrs	Carrier detection
	PhyRxErr	Receive error
	PhyRxd(0..1)	Data receive
	PhyTxEn	Transmit enable
	PhyTxTxd(0..1)	Data transmit
	PhyClk50Mhz	Physical clock (50 MHz)
VGA	vga_Red_o(0..3)	VGA red component signal
	vga_Blue_o(0..3)	VGA blue component signal
	vga_Green_o(0..3)	VGA green component signal
	vga_hs	Horizontal sync signal
	vga_vs	Vertical sync signal
Camera	data_cam(0..7)	Camera data bus
	hsync_cam	Horizontal sync signal
	vsync_cam	Vertical sync signal
	pix_clk_cam	Camera pixel clock
	clk_cam	Camera clock (24 MHz)

Table 9 Hardware design global signals

105 FPGA input or output pins have been used in implemented design and, although most of the components are soldered to the board, for the camera interface considering which signal was routed to each pin for a correct design. Because of the internal design of the FPGA, not all pins can be used for any kind of signal in an optimal way. As an example, only some of the pins are connected to clock inputs of the flip-flops for an optimal route (regarding delay and timing). Care was taken with pixel clock signal as this is the signal that latches into the register the data in the camera bus.

4.1.2 FPGA utilization

We could say that the FPGA included in Nesys4 board is quite big for a board designed mainly for an educational environment. In fact, as can be seen in Table 10, available resources are far to be fully used despite having implemented 5 embedded processors on it. Regarding input and output signals, the chip is far from being fully used.

Device Utilization Summary (actual values)			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	7,778	126,800	6%
Number of Slice LUTs	10,788	63,400	17%
Number of occupied Slices	4,720	15,850	29%
Number of bonded IOBs	105	210	50%
Number of RAMB36E1/FIFO36E1s	106	135	78%

Table 10 FPGA utilization summary

Nevertheless, the same cannot be said regarding FPGA internal memory capacity. In Table 10 can be seen that 78% of internal memory has been used. This is mainly because of the amount memory used in video buffers, both in camera and VGA output interfaces. In fact, optimal resolution that allows the image sensor (640x480) could not be used and, instead, a quarter resolution (320x240) was used in order to fit all necessary data inside the FPGA

Besides, during design process, hardware generation tools found some problems while routing signals because an optimal path could not be found for some of them inside the FPGA, as not all timing restrictions could be satisfied. After studding the case and appropriately tuning those tools configuration, hardware could be correctly generated and after that to follow into the design of the software that would be running in the embedded processors.

In Annex IV a full breakdown of the components used in the FPGA can be seen.

4.2 System functionality

In Figure 30 a full system diagram of this work can be seen. This diagram includes all hardware, both internal and external to FPGA, as well as auxiliary elements (Arduino, image sensor, etc.).

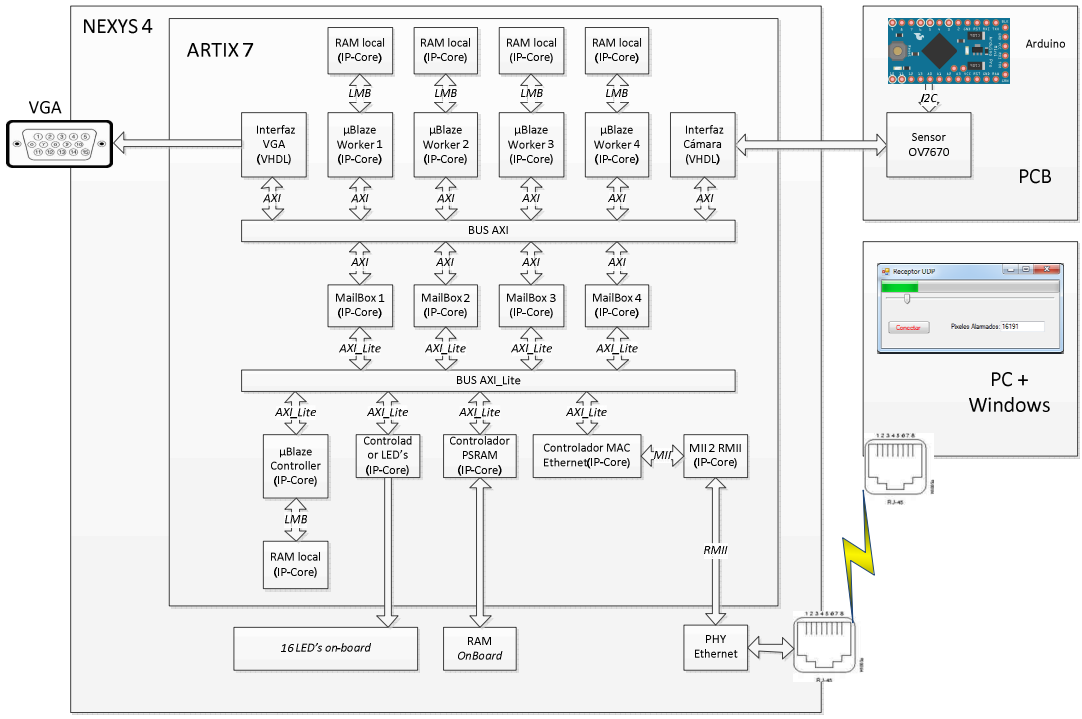

Figure 30 Full system diagram

Using the hardware previously defined in the FPGA, background generation and movement detection algorithms as well as code for processing results and sending them through the network has been implemented.

Because of the limitations in existing hardware, a maximum working speed of 10 frames per second was obtained. Giving the system more computing power it could easily perform at 25 frames per second (fps), what in video is considered as

real time, but, in this kind of applications it is not necessary. In fact, there are implementations, as one from Nair, Laprise and Clark [17] that run a people detection algorithm at 2.5 fps.

In a system like this, that could be used as an early alarm in an alarm center or to control a video recorder, the result is very satisfying with the frame rate that has been achieved.

Figure 31 Working system current consumption

Despite being a quite complex system with many peripherals used, a very low power system has been design. In Figure 31 total current consumption of working Nexys4 board, Arduino and image sensor is about 70 mA. This reduced consumption, using 5 Volts only needs 350 mW and allows a simple installation, giving the possibility of using, in a final implementation, POE (*Power Over Ethernet*) in order to install surveillance cameras that could include all image processing on them.

4.3 System limitations

All artificial systems, starting in the very specifications, have a limited range of application that, in part, is determined by those specifications or their elements characteristics.

Assuming the limitations of a system and the reason of them is the most appropriate approach for, not only know in which environment it can work, but also, to decide different ways of improving its capacities and characteristics.

The system object of this work has been mainly developed for showing a simple and fast choice for detecting quite small moving objects, compared with the field of vision or observed scene. Basic components have been used (FPGA, printed circuit board, sensors), and processing have been made by software. The goals of the project have been greatly achieved and now is time of making an analysis of observed limitations and justify the origin of them, as a previous step for future works with better and more versatile characteristics.

As limitations of the design we could point to some fails system functionality. The main of them is the fact that, under some particular environment and conditions, there are moving objects that are poorly detected. This detection failure occurs when, over a certain background, moving objects with similar grey level pass over it. There are times that, as an example, over a red background, a green object moves that is not detected properly. Anyway, this is a behavior that rarely happens as, even if this color combination appears, the system usually detects it because moving objects does not have a solid color and it evolves lightly because of the reflection of ambient lights during its movement. Those small differences, or just the shadow they create, usually are enough for their detection.

However, this problem must be reviewed: even for human vision system there are situations in which we do not see movement at all if moving objects achieve to mimic the environment. This is a strategy used by some life forms (chameleon, coelenterates, etc.) and in some human activities (camouflage).

Also can be observed, sometimes, a light flicker in image frames. This is caused by the fact that the video memory that feed the VGA output is the same used for updating background data. This makes that processors making background extraction might be updating data in the same moment that VGA interface is reading them. This little flicker is caused because processors use asynchronous clocks between them and different read and write speeds. This could have been solved using *double buffering* techniques in order to update the background in a memory region while other, previously updated, is read. This, however, had made necessary duplicate the amount or RAM used in VGA interface something that was not an option with the available FPGA. Besides, in a final implementation, camera signal could be sent directly to an alarm so using VGA interface for monitoring will not be used.

Finally, must be said that the quality of the image sensors used is not very high. They were bought in Asian websites with prices under 7€. Probably with the same resolution but a better manufacturing process, mainly in the optics, the images obtained would be more clear and, therefore, more reliable.

5 CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

This master work has consisted in the implementation of a background image extraction system for using it for detecting moving objects. Those backgrounds are generated using the sequence of images taken using a CMOS camera at a frame rate of 10 frames per second. For this camera configuration, a printed circuit board has been manufactured and an Arduino board has been properly configured for that.

This background extractor has been implemented by software using a FPGA in which 4 embedded processors has been defined in order to make a parallel video processing achieving speeds up to 10 frames per second.

In addition, within the FPGA itself, a fifth processor has been defined. This last processor would be the one for gathering the information generated by the others, merging it and sending it using an Ethernet network to generate alarms.

As a complement for the hardware design made, a Windows application has been implemented. It will allow seeing those generated alarms after comparing with extracted background.

Moreover, the design itself feed a VGA output that shows the pictures captured by the camera as well as the background that has been generated.

5.2 Future works

Trying to improve the system, several paths remain open. Not only to improve performance but also to give the system more capabilities. Here some of them are shown:

- Using floating point processing for the background extraction algorithm. This would allow a more precise updating of it.

- The utilization of color sensors could end with any possible error when detecting moving objects of different color but that generate similar grey level when using black and white sensors.
- Higher resolution. Increasing resolution both in image and, what is more important, in the background, may allow a better object detection or, maybe, with the use a wide angle optic, covering a bigger scene having the same effectiveness. For making this, it would be necessary having an FPGA with much memory on it.
- Something interesting, but probably complex, would be trying to use adaptive F and S factors not only in time but also in space. The idea would be to find a way to adjust those parameters dynamically depending of changing conditions of observed scene.
- Making a full hardware implementation of the background extraction algorithm. If software processing is removed from the system, probably, a background extraction could be done at real time. In the other hand, the complexity of the design could highly increase.
- Finally, using a FPGA with more resources (on-chip memory, processing resources and working frequency) in order to process higher resolution images, or widen the area under surveillance. In fact, all previous propositions summarizes in this one as all of them would need higher processing power.

BIBLIOGRAFÍA

[1] Digilent Inc. http://www.digilentinc.com/

[2] Xilinx Inc. http://www.xilinx.com/

[3] Documentación del sensor de imagen utilizado. http://www.voti.nl/docs/OV7670.pdf

[4] Definición del protocolo SCCB. http://www.ovt.com/download_document.php?type=document&DID=63

[5] Documentación de la FPGA utilizada. http://www.xilinx.com/support/documentation/user_guides/ug475_7Series_Pkg_Pinout.pdf

[6] Fritzing http://fab.fritzing.org/fritzing-fab

[7] Proyecto Arduino http://www.arduino.cc/

[8] AXI Reference guide www.xilinx.com

[9] Referencia Microblaze Soft Processor Core http://www.xilinx.com/tools/microblaze.htm

[10] Alksej Makarov, “Comparison of background extraction based intrusion detection algorithms”, Int. Conf. Image Processing, 1996.

[11] D. Helper, H. Li, "Analysis of uncovered background prediction for image sequence coding”, Picture Coding Symposium, 1987

[12] K. Karmann, A. Brandt., “Moving object recognition using an adaptive background memory”, Time-varying Image Processing and Moving Object Recognition, Vol. 2, Elesevier, 1990.

[13] Xilinx, Documentación Xilkernel.
http://www.xilinx.com/ise/embedded/edk91i_docs/xilkernel_v3_00_a.pdf

[14] Proyecto Wiring http://wiring.org.co/

[15] Documentación Arduino Pro Mini
http://arduino.cc/en/Main/ArduinoBoardProMini

[16] Referencia de la familia de microcontroladores de Atmel.
http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf

[17] V. Nair, P.O. Laprise, J.J. Clark.,"An FPGA-based people detection system", EURASIP Journal on Applied Signal Processing, Volume 2005.

[18] Proyecto Eclipse. http://www.eclipse.org/eclipse/

Anexo I. Código ejecutándose en la placa Arduino

```
// Master I2C sobre arduino
// para controlar CAM OV-7670
// Francisco Reyes
// Este script configura la cámara en QVGA (usando el windowed mode)
//

// Modificado 29/12/2014

#include <Wire.h>// Librería Wire (I2C)

// Según el PCB los pines utilizados para los switches son
// de arriba hacia abajo: 12, 11, 10, 9

int led = 13; // usaremos este led sólo para confirmar que el
programa está en ejecución
int agc_off = 11; // pin para activar/desactivar el AGC
boolean lecturaAGC, previoAGC, estadoAGC;

int barras_off = 12; // pin para poner/quitar barras.
                     // Hay que tener en cuenta que estamos en
windowded mode
                     // con lo que no veremos las 8 barras de color
boolean lecturaBAR, previoBAR, estadoBAR;


byte sensor_addr = 0x42; // Dirección I2C de los registros de
escritura de la cámara
voidsetup()
{
  //Configuración inicial
  Wire.begin(2); // join i2c bus (address optional for master)
  pinMode(led, OUTPUT);
  pinMode(agc_off, INPUT);
  pinMode(barras_off, INPUT);
  //Serial.begin(9600);

// Reset de todos los registros de la cámara en el power up
  Wire.beginTransmission(sensor_addr >> 1);
  Wire.write(0x12); // Esto escribe el valor 0x80 en el registro
0x12
  Wire.write(0x80); // poniendo todos los registros en su posición
inicial
  Wire.endTransmission();
  delay(200);

// Inicio para cuarto de pantalla
// Se utiliza el "windowed mode"
// De esta forma se utiliza como imagen activa una porción del senso
r
  Wire.beginTransmission(sensor_addr >> 1); //
  Wire.write(0x17);
  Wire.write(0x25); //0x11 (default) = VGA ; 0x25 = QVGA
```

```
  Wire.endTransmission();

  Wire.beginTransmission(sensor_addr >> 1); //
  Wire.write(0x18);
  Wire.write(0x4d); //0x61 (default) = VGA ; 0x4d = QVGA
  Wire.endTransmission();

  Wire.beginTransmission(sensor_addr >> 1); //
  Wire.write(0x19);
  Wire.write(0x3f); //0x03 (default) = VGA ; 0x3f = QVGA
  Wire.endTransmission();

  Wire.beginTransmission(sensor_addr >> 1); //
  Wire.write(0x1a);
  Wire.write(0x7b); //0x7B (default) = VGA ; 0x7b = QVGA
  Wire.endTransmission();
// Fin para cuarto de pantalla
  delay(200);

  // Inicio de secuencia para poner la CAM a 4 fps
  Wire.beginTransmission(sensor_addr >> 1); // transmit to device #4
  Wire.write(0x92);
  Wire.write(0xFD); //0x66 = 25fps; 0xCC = 1 fps ; 0xF4 = 4 fps ;
0xFD = 10 fps
  Wire.endTransmission();

  Wire.beginTransmission(sensor_addr >> 1); // transmit to device #4
  Wire.write(0x93);
  Wire.write(0x05); //0x00 = 25fps ; 0x39 = 1fps ; 0x0C = 4 fps ;
0x03 = 10 fps
  Wire.endTransmission();
  // Fin para 25 fps

}



voidloop()
{
  // Codigo para que los pulsadores se comporten como interruptores
  // Para el AGC
  lecturaAGC = digitalRead(agc_off);
  if (lecturaAGC == LOW&& previoAGC == HIGH ) {
    if (estadoAGC == HIGH)
      estadoAGC = LOW;
    else
      estadoAGC = HIGH;
  }
  previoAGC = lecturaAGC;

  // Para las barras
  lecturaBAR = digitalRead(barras_off);
  if (lecturaBAR == LOW&& previoBAR == HIGH ) {
    if (estadoBAR == HIGH)
```

```
      estadoBAR = LOW;
    else
      estadoBAR = HIGH;
  }
  previoBAR = lecturaBAR;

  if (estadoAGC) { //Solo activo si se presiona el botón
    Wire.beginTransmission(sensor_addr >> 1); // transmit to device
#4
    Wire.write(0x13); // Registro COM8 donde se habilita el AGC
    Wire.write(0x8F); // Default 0x8F (El AGC está activo por
defecto)
    Wire.endTransmission();
  }
  if (!estadoAGC){
    Wire.beginTransmission(sensor_addr >> 1); // transmit to device
#4
    Wire.write(0x13); // Registro COM8 donde se habilita el AGC
    Wire.write(0x00); // 0x00 desabilitar AGC (forma de trabajo
habitual en este diseño)
    Wire.endTransmission();
  }
  delay(200);

  if (estadoBAR) { //Solo activas al presionar el botón
    Wire.beginTransmission(sensor_addr >> 1); // transmit to device
#4
    Wire.write(0x71); // Registro SCALIN_YSC
    Wire.write(0xb5); //  Para activar Barras 0xb5
    Wire.endTransmission();
  }
  if (!estadoBAR){
    Wire.beginTransmission(sensor_addr >> 1); // transmit to device
#4
    Wire.write(0x71); // Registro SCALIN_YSC
    Wire.write(0x35); // Default 0x35 (Sin barras)
    Wire.endTransmission();
  }
  delay(200);

  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage
level)
  delay(200);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the
voltage LOW
  delay(200);               // wait for a second
  //reg++;

}
```

Anexo II. Código de los *Workers*

```
/*
 * Código implementado para el procesamiento de los píxeles captados
 * por la cámara. El mismo código se utiliza en los 4 procesadores
 * "Workers". Se diferencian las posiciones de memoria a las que
 * acceden utilizando su CPU_ID
 *
 * Se ha utilizado como plantilla el código autogenerado por la
 * herramienta de desarrollo (Xilinx SDK) para hacer un test
 * a las regiones de memoria presentes en el diseño hardware.
*
 */

#include <stdio.h>
#include "xparameters.h"
#include "xil_types.h"
#include "xstatus.h"
#include "xil_testmem.h"

#include "platform.h"
#include "memory_config.h"
#include "mbox_header.h"
#include "xmbox.h"

/*
* Utilizaré las CPU_ID para parametrizar las posiciones de memoria
* a las que hay que acceder.
* Worker 1 => CPU_ID = 0
* Worker 2 => CPU_ID = 1
* Worker 3 => CPU_ID = 2
* Worker 4 => CPU_ID = 3
**/

#define CPU_ID XPAR_CPU_ID
#define BASE_CAM XPAR_MI_CAM_IFACE_0_S_AXI_MEM0_BASEADDR //
0xf0000000
#define BASE_VGA XPAR_MIVIDEO2_0_S_AXI_MEM0_BASEADDR // 0x40000000

/////// !!
#define MBOX_DEVICE_ID        XPAR_MAILBOX_1_IF_0_DEVICE_ID 0
#define ID_MAIL_BOX_APP 0 //XPAR_MAILBOX_1_TESTAPP_ID

static XMbox Mbox;      /* Instance del driver del MailBox para
comunicarse con el procesador principal */


void putnum(unsignedint num);
void print(char*ptr){
      return0;

}
```

```
void espera(int cuenta){// Rutina básica para producir pausas en la
ejecución
	int i_cont;
	int basura;
	for(i_cont=0; i_cont>cuenta; i_cont++){
		basura= i_cont + cuenta;
	}
}

int main()// Programa principal. Terminará en bucle una vez
inicializado el sistema
{

    init_platform();

	XMbox_Config *ConfigPtr;// Puntero a la configuración del
MailBox
	u16 MboxDeviceId = ID_MAIL_BOX_APP;
	int Status;

	/* Se van a coger los datos de configuración del MailBox de la
tabla
	 * de configuraciciones y a continuación se inicializa.
	*/

	ConfigPtr = XMbox_LookupConfig(MboxDeviceId );
	if(ConfigPtr ==(XMbox_Config *)NULL){
		print("FALLO EN LOOKUP CONFIG MailBox \r\n");
		return XST_FAILURE;
	}
	else{
		print("OK EN LOOKUP CONFIG  MailBox \r\n");
	}

	/*
	 * Resto de la inicialización
	 */

	Status = XMbox_CfgInitialize(&Mbox, ConfigPtr, ConfigPtr-
>BaseAddress);
	if(Status != XST_SUCCESS){
		print("FALLO EN INITIALIZE MailBox \r\n");
		return XST_FAILURE;
	}
	else{
		print("OK EN INITIALIZE  MailBox \r\n");
	}

	u32 datosEnviar,datosEnviados;
	datosEnviar=0xffffffff;
	Status = XMbox_Write(&Mbox,   &datosEnviar,4,&datosEnviados);
	putnum(datosEnviados);

print("---- --- -Hola Mundo -- -- - - -- \n\r");
```

```
    u16 pixel_img, pixel_fondo, pixel_fondo_ant ;
/* pixel_img .- pixel capturado por la cámara
     * pixel_fondo .- nuevo fondo calculado
     * pixel_fondo_ant .- pixel de la imágen fondo anterior
     */
    u8 *puntero_cam,*puntero_vga_img,*puntero_vga_bg;
/* puntero_cam .- posición de memoria del píxel de la cámara a
procesar
     * puntero_vga_img .- posición del buffer vga con la imagen
actual
     * puntero_vga_bg .- posción del buffer vga con el fondo
     */
int linea_actual;// línea que se está procesando (240 líneas por
imágen)
    u8 contador_frame;// utilizado para sincronizar la imágen de la
cámara
    contador_frame =0;  // y no procesar más de una vez cada imágen
int i,j;// j .- contador de fotograma; i .- contador de píxel
    u32 contador_detectados;// Cálculo de cuantos píxeles han
superado el umbral en el fotograma actual
      int diferencia;// Aquí se almacenará la diferencia entre el
píxel actual y el fondo anterior
      int umbral;// Umbral que determina si hay "intruso"
      umbral=0x1f;
    contador_frame =0;

for(j =0;j<10000000; j++){
      j =1;// Comentar para que el bucle no sea infinito.
      /* Inicio código de sincronización Cámara-Worker */

      puntero_cam = BASE_CAM +19200*4*  CPU_ID;// Posición del píxel
de sincronización
      pixel_img =*puntero_cam;
      while(((contador_frame - pixel_img)&0x0f))// Se espera hasta
que en el buffer de la cámara ya está la imagen a procesar.
      {// Esta espera se lleva a cabo para no bloquear el bus con
peticiones recurrentes.
            pixel_img =*puntero_cam;
            espera(100);
      }
      contador_frame++;

      /* Fin código de sincronización Cámara-Worker */

      contador_detectados =0;// Se limpia del fotograma anterior

      /* Inicio código de procesado píxeles y fondo */

      for(i =19200* CPU_ID; i <19200*(CPU_ID +1); i++){// < 19200 en
lectura; 38400 en escritura
                  puntero_cam = BASE_CAM + i*4;// Posición de inicio
de píxeles (se multiplica por 4 porque el bus es de 32 bits)
                  pixel_img =*puntero_cam;// Se lee un píxel de la
cámara
```

```
                    linea_actual = i/320;
                    puntero_vga_img = BASE_VGA + i*4+(linea_actual
*320*4);
                    *puntero_vga_img = pixel_img;// Se copia el píxel
a la VGA
                    puntero_vga_bg = BASE_VGA + i*4+(linea_actual
*320*4)+320*4;
                    pixel_fondo_ant =*puntero_vga_bg;// Se recupera el
píxel anterior de la VGA
                    diferencia=abs((int) pixel_fondo_ant -(int)
pixel_img);// Se calcula la diferencia entre píxel actual y fondo
anterior
                    if((diferencia)< umbral){// Se actualizará más o
menos rápido en función de si la diferencia supera el umbral
definido;
                          pixel_fondo =((12* pixel_fondo_ant +4*
pixel_img)+8)/16;
                    }
                    else{
                          pixel_fondo =((122* pixel_fondo_ant
+6*pixel_img)+64)/128;
                          contador_detectados++;// Se incrementa la
cuenta de píxeles "alarmados"
                    }
                    *puntero_vga_bg = pixel_fondo;// Se actualiza el
fondo en la VGA
              }

      /* Fin código de procesado píxeles y fondo */

      Status = XMbox_Write(&Mbox,
      &contador_detectados,4,&datosEnviados);//Se envía al
procesador principal el número de píxeles alarmados
              putnum(contador_detectados);
              putnum(Status);
              putnum(CPU_ID);
              print("<===det|Sta|CPU\n\r");
}
return0;
}
```

Anexo III . Código del *Controller*

```
/*
 * Código implementado para recibir el resultado del proceso de los
 * píxeles por parte de los "Workers". Se leen los mailboxes que
comunican
 * con cada Worker, se suma todo y se envía el resultado por UDP.
Además,
 * se iluminan los LEDs de la placa de desarrollo para comprobar el
 * funcionamiento sin software externo.
 *
 * Se ha utilizado como plantilla el código autogenerado por la
 * herramienta de desarrollo (Xilinx SDK) para hacer un test
 * de la pila TCP/IP que implementa un eco TCP.
*/

#include <stdio.h>

#include "xparameters.h"

#include "netif/xadapter.h"

#include "platform.h"
#include "platform_config.h"
#ifdef __arm__
#include "xil_printf.h"
#endif

#include "mbox_header.h"
#include "xmbox.h"
#include "lwip/err.h"
#include "lwip/udp.h"
#include "xgpio.h"

#define MBOX_DEVICE_ID1       XPAR_MAILBOX_1_IF_0_DEVICE_ID 0
#define ID_MAIL_BOX_APP1      0 //XPAR_MAILBOX_1_TESTAPP_ID
#define MBOX_DEVICE_ID2       XPAR_MAILBOX_1_IF_1_DEVICE_ID 1
#define ID_MAIL_BOX_APP2      1 //XPAR_MAILBOX_1_TESTAPP_ID
#define MBOX_DEVICE_ID3       XPAR_MAILBOX_1_IF_2_DEVICE_ID 2
#define ID_MAIL_BOX_APP3      2 //XPAR_MAILBOX_1_TESTAPP_ID
#define MBOX_DEVICE_ID4       XPAR_MAILBOX_1_IF_3_DEVICE_ID 3
#define ID_MAIL_BOX_APP4    3 //XPAR_MAILBOX_1_TESTAPP_ID

/* defined by each RAW mode application */
void print_app_header();
int start_application();
int transfer_data();

/* missing declaration in lwIP */
void lwip_init();

staticstruct netif server_netif;
struct netif *echo_netif;
```

```
/* Instancias de los drivers de los Mailboxes */
static XMbox Mbox1,Mbox2,Mbox3,Mbox4;
XMbox_Config *ConfigPtr;// Puntero a la configuración del MailBox.
                                    // Se reutiliza en
configuraciones sucesivas
u32 pixelesAlarmados,pixAlm1,pixAlm2,pixAlm3,pixAlm4;// Píxeles
alarmados en cada una de las
// regiones de proceso y la suma de los cuatro

XGpio GpioOutput;
int Status;


void inicializaMailBoxes()
{

      u16 MboxDeviceId = ID_MAIL_BOX_APP1;
      int Status;
      /* Se van a coger los datos de configuración del MailBox de la
tabla
       * de configuraciciones y a continuación se inicializa.
       * Luego se repetira para cada MailBox
       */

      print("\r\nInicializando MAILBOX_1...\r\n");
      ConfigPtr = XMbox_LookupConfig(ID_MAIL_BOX_APP1);
      if(ConfigPtr ==(XMbox_Config *)NULL){
            print("FALLO EN LOOKUP CONFIG \r\n");
            return XST_FAILURE;
      }
      else{
            print("OK EN LOOKUP CONFIG \r\n");
      }

      /*
       * Resto de la inicialización
       */

      Status = XMbox_CfgInitialize(&Mbox1, ConfigPtr, ConfigPtr-
>BaseAddress);
      if(Status != XST_SUCCESS){
            print("FALLO EN INITIALIZE \r\n");
            return XST_FAILURE;
      }
      else{
            print("OK EN INITIALIZE \r\n");
      }

      /*
       * Se prepite para los otros 3 MailBoxes
       */

      print("\r\nInicializando MAILBOX_2...\r\n");
```

```
	ConfigPtr = XMbox_LookupConfig(ID_MAIL_BOX_APP2);
	if(ConfigPtr ==(XMbox_Config *)NULL){
		print("FALLO EN LOOKUP CONFIG \r\n");
		return XST_FAILURE;
	}
	else{
		print("OK EN LOOKUP CONFIG \r\n");
	}
	Status = XMbox_CfgInitialize(&Mbox2, ConfigPtr, ConfigPtr-
>BaseAddress);
	if(Status != XST_SUCCESS){
		print("FALLO EN INITIALIZE \r\n");
		return XST_FAILURE;
	}
	else{
		print("OK EN INITIALIZE \r\n");
	}


	print("\r\nInicializando MAILBOX_3...\r\n");
	ConfigPtr = XMbox_LookupConfig(ID_MAIL_BOX_APP3);
	if(ConfigPtr ==(XMbox_Config *)NULL){
		print("FALLO EN LOOKUP CONFIG \r\n");
		return XST_FAILURE;
	}
	else{
		print("OK EN LOOKUP CONFIG \r\n");
	}
	Status = XMbox_CfgInitialize(&Mbox3, ConfigPtr, ConfigPtr-
>BaseAddress);
	if(Status != XST_SUCCESS){
		print("FALLO EN INITIALIZE \r\n");
		return XST_FAILURE;
	}
	else{
		print("OK EN INITIALIZE \r\n");
	}


	print("\r\nInicializando MAILBOX_4...\r\n");
	ConfigPtr = XMbox_LookupConfig(ID_MAIL_BOX_APP4);
	if(ConfigPtr ==(XMbox_Config *)NULL){
		print("FALLO EN LOOKUP CONFIG \r\n");
		return XST_FAILURE;
	}
	else{
		print("OK EN LOOKUP CONFIG \r\n");
	}
	Status = XMbox_CfgInitialize(&Mbox4, ConfigPtr, ConfigPtr-
>BaseAddress);
	if(Status != XST_SUCCESS){
		print("FALLO EN INITIALIZE \r\n");
		return XST_FAILURE;
	}
	else{
```

```
			print("OK EN INITIALIZE \r\n");
		}

		// A continuación se vacían los MailBoxes para limpiarlos de
datos no actualizados.

		XMbox_Flush(&Mbox1);
		XMbox_Flush(&Mbox2);
		XMbox_Flush(&Mbox3);
		XMbox_Flush(&Mbox4);

}

u32 calculaPixelesAlarmados(){
		/* Esta rutina lee los 4 MailBoxes y devulve la suma de total
		 * de píxeles alarmados en las 4 regiones de proceso.
		*/

		u32 datosRx,bytesRx;
		int Status;
		Status = XMbox_Read(&Mbox1,   &datosRx,4,&bytesRx);
		if(Status != XST_SUCCESS){
			print("El buffer estaba vacio\r\n");
		}
		else{
			pixAlm1 = datosRx;
		}

		Status = XMbox_Read(&Mbox2,   &datosRx,4,&bytesRx);
		if(Status != XST_SUCCESS){
			print("El buffer estaba vacio\r\n");
		}
		else{
			pixAlm2 = datosRx;
		}

		Status = XMbox_Read(&Mbox3,   &datosRx,4,&bytesRx);
		if(Status != XST_SUCCESS){
			print("El buffer estaba vacio\r\n");
		}
		else{
			pixAlm3 = datosRx;
		}

		Status = XMbox_Read(&Mbox4,   &datosRx,4,&bytesRx);
		if(Status != XST_SUCCESS){
			print("El buffer estaba vacio\r\n");
		}
		else{
			pixAlm4 = datosRx;
		}

		return pixAlm1 + pixAlm2 + pixAlm3 + pixAlm4;

}
```

```
void
print_ip(char*msg,struct ip_addr *ip)
//Función que estrutura 4 número para su impresión
{
	print(msg);
	xil_printf("%d.%d.%d.%d\n\r", ip4_addr1(ip), ip4_addr2(ip),
			ip4_addr3(ip), ip4_addr4(ip));
}

void
print_ip_settings(struct ip_addr *ip,struct ip_addr *mask,struct
ip_addr *gw)
// Función para imprimir la configuración IP de la tarjeta
{

	print_ip("Board IP: ", ip);
	print_ip("Netmask : ", mask);
	print_ip("Gateway : ", gw);
}

int main()
{
	struct ip_addr ipaddr, netmask, gw, ip_destino;

	/* Dirección MAC de la tarjeta */
	unsignedchar mac_ethernet_address[]=
	{0x00,0x0a,0x35,0x00,0x01,0x02};

	echo_netif =&server_netif;

	init_platform();

	/* Configuración IP de la tarjeta */
	IP4_ADDR(&ipaddr,192,168,2,10);
	IP4_ADDR(&netmask,255,255,255,0);
	IP4_ADDR(&gw,192,168,2,1);

	// Inicializo la IP destino de los paquetes UDP
	IP4_ADDR(&ip_destino,192,168,2,166);

	// Se imprime la configuración IP para tenerla como referencia
	print_ip_settings(&ipaddr,&netmask,&gw);

	//Inicialización de la pila TCP/IP
	lwip_init();

	/* Se añadir el interfaz de red a la lista netif_list
	 * y se configura como interfaz por defecto  */
	if(!xemac_add(echo_netif,&ipaddr,&netmask,
					&gw, mac_ethernet_address,
					PLATFORM_EMAC_BASEADDR)){
		xil_printf("Error adding N/W interface\n\r");
		return-1;
	}
```

```
	netif_set_default(echo_netif);


	/* Se habilitan las interrupciónes */
	platform_enable_interrupts();

	/* Se habilita el interfaz */
	netif_set_up(echo_netif);

	//Vamos a inicializar los 4 mailboxes
	inicializaMailBoxes();

	// Se inicializan los LEDS
	 Status = XGpio_Initialize(&GpioOutput, XPAR_LEDS_DEVICE_ID);
	if(Status != XST_SUCCESS){
		return XST_FAILURE;
	}

	// Inicializamos variables
	pixAlm1 =0; pixAlm2 =0; pixAlm3 =0; pixAlm4 =0;


	struct udp_pcb *pcb;// Dato estructurado que recoge el estado
del  socket UDP
struct pbuf *pb;// Buffer que contendrá los datos a enviar
	err_t err;
	unsigned port =7;


int varLeds;
	int contador_bucle;
	for(contador_bucle =0; contador_bucle <1000000;
contador_bucle++){
		contador_bucle =1;// Comentar para que el bucle no sea
infinito

		pixelesAlarmados= calculaPixelesAlarmados();// Se leen
los píxeles alarmados

		// Se imprime el resultado
		print("W1: 0x");putnum(pixAlm1);
		print("  W2: 0x");putnum(pixAlm2);
		print("  W3: 0x");putnum(pixAlm3);
		print("  W4: 0x");putnum(pixAlm4);
		print("  Total: 0x");putnum(pixelesAlarmados);
		print("\r\n");


		// Se va a escribir en los leds el nivel de alarma en
una escala logarítmica
		varLeds=0;
		while(varLeds<pixelesAlarmados){
			varLeds=(varLeds *2)+1;
		}
		varLeds= varLeds /2;
```

```
            XGpio_DiscreteWrite(&GpioOutput,1, varLeds);

      // Se incializa el PCB
      pcb= udp_new();
      if(!pcb){ xil_printf("Error creando PCB. Out of
Memory\n\r");return;}
      err= udp_bind(pcb,&ipaddr,6666);
      if(err != ERR_OK){ xil_printf("Unable to bind to
port\n\r");return;}
      err= udp_connect(pcb,&ip_destino,6666);
      if(err != ERR_OK){ xil_printf("Unable to
connect\n\r");return;}
      pb= pbuf_alloc(PBUF_TRANSPORT,4, PBUF_POOL);
      if(!pb){ xil_printf("Error while allocating
pbuf\n\r");return;}
      memcpy(pb->payload,&pixelesAlarmados,4);

      // Se envían el paquete
      err= udp_send(pcb, pb);
      if(err == ERR_MEM){
                  xil_printf("Error enviando el paquete UDP. Out of
memory\n\r");
      }elseif(err == ERR_RTE){
      xil_printf("No hay ruta al destino\n\r");
      }elseif(err != ERR_OK){
                  xil_printf("Error sending packet - %d\n\r", err);
      }else{
                  xil_printf("Packet sent\n\r");
      }
          pbuf_free(pb);
          udp_remove(pcb);// Se libera el PCB

      // Se activa el proceso de envio y recepción de paquetes.
            xemacif_input(echo_netif);

      }



      return0;
}
```

Anexo IV: Resultado de la generación del hardware

Project Status (04/07/2015 - 19:51:51)			
Project File:	TestRamEthernet.xmp	**Implementation State:**	Programming File Generated
Module Name:	TestRamEthernet	• **Errors:**	
Product Version:	EDK 14.6	• **Warnings:**	

XPS Reports				[-]
Report Name	**Generated**	**Errors**	**Warnings**	**Infos**
Platgen Log File	mar 7. abr 19:27:25 2015	0	139 Warnings (85 new)	91 Infos (1 new)
Simgen Log File				
BitInit Log File	lun 26. ene 19:04:25 2015			
System Log File	mar 7. abr 19:48:10 2015			

XPS Synthesis Summary (estimated values)					[-]
Report	**Generated**	**Flip Flops Used**	**LUTs Used**	**BRAMS Used**	**Errors**
TestRamEthernet	mar 7. abr 19:29:40 2015	11638	13300	106	0
TestRamEthernet_clock50_wrapper	mar 7. abr 19:26:46 2015		1		0
TestRamEthernet_clock50phyether_wrapper	mar 7. abr 19:26:46 2015		1		0
TestRamEthernet_clock_generator_0_wrapper	mar 7. abr 19:26:46 2015		1		0
TestRamEthernet_worker_4_i_bram_cntlr_wrapper	mar 7. abr 19:26:46 2015	2	6		0
TestRamEthernet_worker_4_ilmb_wrapper	mar 7. abr 19:26:46 2015	1			0
TestRamEthernet_axi4_0_wrapper	mar 7. abr 19:07:28 2015	1141	2137		0
TestRamEthernet_axi4lite_0_wrapper	mar 7. abr 19:07:28 2015	229	613		0
TestRamEthernet_axi_bram_ctrl_0_bram_block_wrapper	mar 7. abr 19:07:28 2015			4	0
TestRamEthernet_axi_bram_ctrl_0_wrapper	mar 7. abr 19:07:28 2015	418	506		0
TestRamEthernet_axi_ethernetlite_0_wrapper	mar 7. abr 19:07:28 2015	607	697	2	0

TestRamEthernet_axi_timebase_wdt_0_wrapper	mar 7. abr 19:07:28 2015	96	111		0
TestRamEthernet_axi_timer_0_wrapper	mar 7. abr 19:07:28 2015	221	313		0
TestRamEthernet_axi_uartlite_0_wrapper	mar 7. abr 19:07:28 2015	90	121		0
TestRamEthernet_debug_module_wrapper	mar 7. abr 19:07:28 2015	136	182		0
TestRamEthernet_generic_external_memory_wrapper	mar 7. abr 19:07:28 2015	530	813		0
TestRamEthernet_leds_wrapper	mar 7. abr 19:07:28 2015	110	90		0
TestRamEthernet_mailbox_1_wrapper	mar 7. abr 19:07:28 2015	174	322		0
TestRamEthernet_mailbox_2_wrapper	mar 7. abr 19:07:28 2015	174	322		0
TestRamEthernet_mailbox_3_wrapper	mar 7. abr 19:07:28 2015	174	322		0
TestRamEthernet_mailbox_4_wrapper	mar 7. abr 19:07:28 2015	174	322		0
TestRamEthernet_mi_cam_iface_0_wrapper	mar 7. abr 19:07:28 2015	184	284	20	0
TestRamEthernet_microblaze_0_bram_block_wrapper	mar 7. abr 19:07:28 2015			32	0
TestRamEthernet_microblaze_0_d_bram_ctrl_wrapper	mar 7. abr 19:07:27 2015	2	6		0
TestRamEthernet_microblaze_0_wrapper	mar 7. abr 19:07:28 2015	1867	1772	5	0
TestRamEthernet_microblaze_0_dlmb_wrapper	mar 7. abr 19:07:27 2015	1			0
TestRamEthernet_worker_2_wrapper	mar 7. abr 19:07:28 2015	1214	938		0
TestRamEthernet_worker_3_wrapper	mar 7. abr 19:07:28 2015	1214	938		0
TestRamEthernet_worker_4_wrapper	mar 7. abr 19:07:28 2015	1214	938		0
TestRamEthernet_microblaze_0_i_bram_ctrl_wrapper	mar 7. abr 19:07:27 2015	2	6		0
TestRamEthernet_microblaze_0_ilmb_wrapper	mar 7. abr 19:07:27 2015	1			0
TestRamEthernet_microblaze_0_intc_wrapper	mar 7. abr 19:07:27 2015	80	124		0
TestRamEthernet_mii_to_rmii_0_wrap	mar 7. abr	79	34		0

per	19:07:27 2015				
TestRamEthernet_mivideo2_0_wrapper	mar 7. abr 19:07:27 2015	199	344	35	0
TestRamEthernet_proc_sys_reset_0_wrapper	mar 7. abr 19:07:27 2015	69	56		0
TestRamEthernet_worker_1_bram_block_wrapper	mar 7. abr 19:07:27 2015			2	0
TestRamEthernet_worker_1_d_bram_cntlr_wrapper	mar 7. abr 19:07:26 2015	2	6		0
TestRamEthernet_worker_1_dlmb_wrapper	mar 7. abr 19:07:26 2015	1			0
TestRamEthernet_worker_1_i_bram_cntlr_wrapper	mar 7. abr 19:07:26 2015	2	6		0
TestRamEthernet_worker_1_ilmb_wrapper	mar 7. abr 19:07:26 2015	1			0
TestRamEthernet_worker_1_wrapper	mar 7. abr 19:07:27 2015	1214	938		0
TestRamEthernet_worker_2_bram_block_wrapper	mar 7. abr 19:07:26 2015			2	0
TestRamEthernet_worker_2_d_bram_cntlr_wrapper	mar 7. abr 19:07:26 2015	2	6		0
TestRamEthernet_worker_2_dlmb_wrapper	mar 7. abr 19:07:26 2015	1			0
TestRamEthernet_worker_2_i_bram_cntlr_wrapper	mar 7. abr 19:07:26 2015	2	6		0
TestRamEthernet_worker_2_ilmb_wrapper	mar 7. abr 19:07:26 2015	1			0
TestRamEthernet_worker_3_bram_block_wrapper	mar 7. abr 19:07:26 2015			2	0
TestRamEthernet_worker_3_d_bram_cntlr_wrapper	mar 7. abr 19:07:26 2015	2	6		0
TestRamEthernet_worker_3_dlmb_wrapper	mar 7. abr 19:07:25 2015	1			0
TestRamEthernet_worker_3_i_bram_cntlr_wrapper	mar 7. abr 19:07:25 2015	2	6		0
TestRamEthernet_worker_3_ilmb_wrapper	mar 7. abr 19:07:25 2015	1			0
TestRamEthernet_worker_4_bram_block_wrapper	mar 7. abr 19:07:25 2015			2	0
TestRamEthernet_worker_4_d_bram_cntlr_wrapper	mar 7. abr 19:07:25 2015	2	6		0
TestRamEthernet_worker_4_dlmb_wrapper	mar 7. abr 19:07:25 2015	1			0

Device Utilization Summary (actual values)				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	7,778	126,800	6%	
Number used as Flip Flops	7,735			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	43			
Number of Slice LUTs	10,788	63,400	17%	
Number used as logic	9,274	63,400	14%	
Number using O6 output only	6,959			
Number using O5 output only	289			
Number using O5 and O6	2,026			
Number used as ROM	0			
Number used as Memory	1,348	19,000	7%	
Number used as Dual Port RAM	328			
Number using O6 output only	0			
Number using O5 output only	1			
Number using O5 and O6	327			
Number used as Single Port RAM	4			
Number using O6 output only	4			
Number using O5 output only	0			
Number using O5 and O6	0			
Number used as Shift Register	1,016			
Number using O6 output only	1,015			
Number using O5 output only	1			
Number using O5 and O6	0			
Number used exclusively as route-thrus	166			
Number with same-slice register load	97			
Number with same-slice carry load	30			
Number with other load	39			
Number of occupied Slices	4,720	15,850	29%	
Number of LUT Flip Flop pairs used	12,638			
Number with an unused Flip Flop	5,251	12,638	41%	
Number with an unused LUT	1,850	12,638	14%	
Number of fully used LUT-FF pairs	5,537	12,638	43%	

Number of unique control sets	650			
Number of slice register sites lost to control set restrictions	2,614	126,800	2%	
Number of bonded IOBs	105	210	50%	
Number of LOCed IOBs	105	105	100%	
IOB Flip Flops	88			
Number of RAMB36E1/FIFO36E1s	106	135	78%	
Number using RAMB36E1 only	106			
Number using FIFO36E1 only	0			
Number of RAMB18E1/FIFO18E1s	0	270	0%	
Number of BUFG/BUFGCTRLs	8	32	25%	
Number used as BUFGs	8			
Number used as BUFGCTRLs	0			
Number of IDELAYE2/IDELAYE2_FINEDELAYs	0	300	0%	
Number of ILOGICE2/ILOGICE3/ISERDESE2s	22	300	7%	
Number used as ILOGICE2s	22			
Number used as ILOGICE3s	0			
Number used as ISERDESE2s	0			
Number of ODELAYE2/ODELAYE2_FINEDELAYs	0			
Number of OLOGICE2/OLOGICE3/OSERDESE2s	50	300	16%	
Number used as OLOGICE2s	50			
Number used as OLOGICE3s	0			
Number used as OSERDESE2s	0			
Number of PHASER_IN/PHASER_IN_PHYs	0	24	0%	
Number of PHASER_OUT/PHASER_OUT_PHYs	0	24	0%	
Number of BSCANs	1	4	25%	
Number of BUFHCEs	0	96	0%	
Number of BUFRs	0	24	0%	
Number of CAPTUREs	0	1	0%	
Number of DNA_PORTs	0	1	0%	
Number of DSP48E1s	3	240	1%	
Number of EFUSE_USRs	0	1	0%	
Number of FRAME_ECCs	0	1	0%	

Number of IBUFDS_GTE2s	0	4	0%	
Number of ICAPs	0	2	0%	
Number of IDELAYCTRLs	0	6	0%	
Number of IN_FIFOs	0	24	0%	
Number of MMCME2_ADVs	3	6	50%	
Number of OUT_FIFOs	0	24	0%	
Number of PCIE_2_1s	0	1	0%	
Number of PHASER_REFs	0	6	0%	
Number of PHY_CONTROLs	0	6	0%	
Number of PLLE2_ADVs	0	6	0%	
Number of STARTUPs	0	1	0%	
Number of XADCs	0	1	0%	
Average Fanout of Non-Clock Nets	4.48			

Performance Summary			[-]
Final Timing Score:	0 (Setup: 0, Hold: 0, Component Switching Limit: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Translation Report	Current	mar 7. abr 19:35:49 2015	0	16 Warnings (16 new)	3 Infos (3 new)
Map Report	Current	mar 7. abr 19:42:19 2015			
Place and Route Report	Current	mar 7. abr 19:44:28 2015	0	22 Warnings (22 new)	2 Infos (2 new)
Post-PAR Static Timing Report	Current	mar 7. abr 19:45:28 2015	0	0	4 Infos (4 new)
Bitgen Report	Current	mar 7. abr 19:48:09 2015	0	22 Warnings (22 new)	0

Secondary Reports			[-]
Report Name	**Status**	**Generated**	
WebTalk Log File	Current	mar 7. abr 19:48:10 2015	

Date Generated: 04/07/2015 - 19:51:52