Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

Diseño Electrónico en FPGA con herramientas de Software Libre

Autor: Iván Ríos Santillán
Tutor/es: Ignacio Bravo Muñoz

2019

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENIERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

“Diseño Electrónico en FPGA con herramientas de Software Libre”

Iván Ríos Santillán

2019

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENIERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

“Diseño Electrónico en FPGA con herramientas de Software Libre”

Autor: Iván Ríos Santillán

TRIBUNAL:

Presidente: D. Pedro Martín Sánchez

Vocal 1: D Ignacio Fernández Lorenzo

Vocal 2: D. Ignacio Bravo Muñoz

FECHA:

Agradecimientos

Quisiera agradecer tanto a mi hermana como a mis padres todo el apoyo que me han brindado para que yo haya llegado hasta aquí sin más secuela que la pérdida de pelo por haberme estrujado la cabeza en cada trabajo y cada examen.

También a todos los compañeros de clase, los que se quedaron atrás, los que me rebasaron y los que han terminado conmigo, porque siempre estaban dispuestos a echarme un cable con cualquier problema que tuviera ya fuera personal o académico.

"¿Iván? Es un chico que cae muy bien"

Amigos que dominan el doble sentido.

Índice

Lista de Ilustraciones y Tablas

Glosario de acrónimos y abreviaturas

- **FPGA**: Field-Programmable Gate Array *(*matriz de puertas programables).
- **FOSS**: Free and Open Source Software (software libre y de código abierto).
- **ASIC**: Application-Specific Integrated Circuit (circuito integrado para aplicaciones específicas).
- **HDL**: Hardware Description Language (Lenguaje de descripción hardware).
- **DUT**: Design Under Test (Diseño bajo prueba).
- **NVCM**: Non-Volatile Configuration Memory (Memoria de configuración no volátil).
- **PLL**: Phase-Locked Loop (Bucle con bloqueo de fase).
- **GNU GPL**: GNU General Public License (Licencia Pública General de GNU)
- **SCCB**: Serial Camera Control Bus (Bus serie de control de cámara

Palabras Clave

FPGA

FOSS

Project IceStorm

Verilog

Sobel

Resumen

¿Cualquiera puede utilizar una FPGA? ¿Es asequible? Todos los proveedores de estos dispositivos obligan a utilizar su propio software, normalmente privativo y bajo licencia. El Proyecto IceStorm incluye varias herramientas FOSS para desarrollar todo el proceso, desde la compilación del diseño hasta su carga en chip, en ciertos modelos de FPGA.

Gracias a esto, prácticamente cualquier usuario con cierta idea de programar es capaz de utilizarlas con el único desembolso de la compra de una de estas FPGA.

En este trabajo se dará una completa explicación de su instalación, su uso y varios ejemplos de todo ese proceso utilizando únicamente las herramientas del "Proyecto IceStorm".

Abstract

Can anyone use an FPGA? Is it affordable? All providers of these devices require you to use their own software, normally proprietary and licensed. The "Project IceStorm" includes several FOSS tools to develop the entire process, from design compiling to download on chip, in certain FPGA models.

Thanks to this, practically any user with basic programming skills would be able to use them, reducing costs to just the acquisition of one of these FPGAs

This paper will give a complete explanation of its installation, its use and several examples of this whole process using only the tools of the "Project IceStorm".

Resumen Extendido

La comunidad FOSS, Clifford Wolf y su “Proyecto IceStorm”, el cual da una solución utilizando herramientas FOSS al problema de diseñar, compilar, testear y cargar un diseño de hardware en un a FPGA.

El “Proyecto IceStorm” se basa en las FPGAs de la familia Lattice iCE40 (de ahí su nombre) ya que al tener una estructura sencilla y fácil de entender un fueron perfectas para desarrollar herramientas libres para programarlas.

A pesar de que hay varios lenguajes de descripción de hardware (como VHDL, Verilog, ABEL u otros lenguajes propietarios) el “Proyecto IceStorm” se basa en Verilog por lo que todo lo que se vea en este trabajo irá enfocado a dicho lenguaje.

Verilog, *grosso modo*, es un HDL basado en lo que se definen como módulos (module), que básicamente son bloques, con sus entradas y salidas, compuestos por puertas lógicas que realizan una o varias funciones. Las funciones pueden ser tanto secuenciales como no secuenciales, por lo que cada bloque puede funcionar de forma similar a un microprocesador o como un circuito digital común, dando lugar a infinidad de formas de diseñar módulos para la resolución de problemas.

El “Proyecto IceStorm” utiliza varias herramientas, una para cada paso del proceso desde la compilación de errores hasta la implementación en placa. Todas ellas son FOSS, creadas y desarrolladas por diferentes personas:

- **Compilador y simulador de Verilog**: Icarus Verilog
- **Visualizador de señales**: Gtkwave
- **Sintetizador**: Yosys
- **Place & route**: Arachne-pnr
- **Utilidades y descarga en FPGA**: Icepack e Iceprog (Proyecto IceStorm)

Para la comprobación de dichas herramientas, este trabajo estará enfocado a la instalación, explicación de cada una de las herramientas además de una ligera introducción al lenguaje Verilog para que cualquier usuario sin conocimientos sobre este lenguaje o sobre las FPGAs en general sea capaz de seguirlo y comprenderlo.

Dicha introducción explicará la forma de proceder en el lenguaje Verilog (y en cualquier HDL), sus elementos propios como sus tipos de datos, sus sentencias básicas o su estructura, además de un par de ejemplos sencillos explicados minuciosamente. También se explicará lo que es un testbench (o banco de pruebas), su utilidad a la hora de simular y comprobar si lo diseñado funciona acorde con lo requerido y cómo se crea y configura para su utilización con las herramientas del “Proyecto IceStorm”.

Para finalizar este proyecto se llevará a cabo un diseño HDL complejo en base a las herramientas del "Proyecto IceStorm" para demostrar su viabilidad en proyectos más grandes y explorar los límites de estas herramientas.

El proyecto final será hacer un diseño real de un detector de bordes de una imagen en tiempo real utilizando el filtro Sobel aplicado a FPGAs. Para el desarrollo de dicho proyecto se utilizarán los siguientes dispositivos hardware:

- Tarjeta Alhambra 1.1 (con una FPGA compatible con el "Proyecto IceStorm")
- Tarjeta Arduino UNO
- Cámara OV7670

Se utilizará la tarjeta Arduino UNO para configurar la cámara de modo que el tamaño, formato y protocolo de comunicación de esta con la tarjeta Alhambra sean los adecuados.

La cámara OV7670 capturará y enviará una imagen tras otra (píxel a píxel en blanco y negro) y mientras, la tarjeta Alhambra las almacenará por filas (únicamente 4 filas simultaneas) que se irán sobrescribiendo con filas nuevas a medida que ya hayan sido utilizadas para el procesamiento de detección de bordes, creando así un procesamiento en tiempo real de la imagen, y enviando el resultado a un ordenador a medida que se vayan obteniendo los píxeles procesados.

Dicha cadena de píxeles se encuentra en formato binario, que será recogida en un archivo binario y transformada en una imagen de formato jpeg con MatLab.

1. Introducción

1.1 ¿Qué es una FPGA?

En el ámbito de los sistemas empotrados hay unos dispositivos, los cuales, están adquiriendo mucha relevancia debido a su velocidad y versatilidad: las FPGAs. [1]

Una FPGA es un dispositivo basado en tecnología de semiconductores formado por bloques lógicos físicamente diferenciados e interconectados que pueden combinados mediante programación, dando lugar a diseños electrónicos complejos. Internamente, las FPGAs se organizan como una red de puertas lógicas que son programadas por el desarrollador, utilizando lenguajes de descripción de hardware (HDL) en distintos niveles de abstracción: desde el bajo nivel, realizando interconexiones entre puertas lógicas, al alto nivel, mediante registros, procesos secuenciales, módulos…

1.1.1 Diferencias y similitudes con otros dispositivos

Las FPGAs son distintas a los microprocesadores o los ASICs, pero tienen algunas similitudes, por ello es necesario dejar bien claras las diferencias entre ellos. [2]

- Los microprocesadores son unos dispositivos programables que ejecutan órdenes pregrabadas secuencialmente, es decir, deben realizar una operación una detrás de otra (a no ser que sea un sistema con múltiples núcleos, en cuyo caso existen varios procesos ejecutándose al mismo tiempo). A su favor están su reducido coste (debido a la producción masiva), que pueden usar para dar solución prácticamente a cualquier problema de procesado de información y al definir su funcionamiento por software, éste se puede actualizar; con el inconveniente de que no funciona a grandes velocidades con varias tareas simultáneas debido a su comportamiento secuencial.

- En el caso de los ASICs, tanto sus entradas de información, como el procesado de estas y sus salidas están definidas desde la misma etapa de diseño pudiendo trabajar partes del diseño simultáneamente a otras. Son los más eficientes en términos de velocidad y consumo, pero debido a que sólo pueden realizar la función que se les asignó originalmente, para rehacer el diseño hay que fabricar otro circuito integrado diferente. Por ello es inviable usarlo en fases de desarrollo o en diseños que requieran ser actualizados, ya que conlleva un coste muy superior a las otras soluciones.

- Las FPGAs son también unos dispositivos programables que se usan para diseñar e implementar circuitos. Para ello se definen bloques lógicos (con sus entradas y salidas) que realizan las operaciones requeridas, trabajando todos éstos a la vez. Cada bloque está compuesto por numerosas puertas lógicas y/o flip-flops de memoria, por lo que pueden hacer desde operaciones lógicas de bits hasta almacenamiento en memoria de información.

1.2 Software propietario / software libre

Desde su creación, estos dispositivos han sido programados mediante software propietario (o privativo) del fabricante que, a excepción de periodos de prueba o versiones para estudiantes, suele ser de pago a través de licencias. Es decir, para programar una FPGA se debe pagar por adquirir una de estas licencias, siendo éste un coste extra para las empresas que pretendan desarrollar sus productos o soluciones en base a las FPGAs. Del mismo modo, suponen también una barrera considerable para un usuario particular que quiera utilizar estos dispositivos sin fines comerciales. Una solución para esto es que las FPGAs entraran en el mundo FOSS.

El término FOSS hace referencia a todo aquel software que pueda ser estudiado, modificado, utilizado con cualquier fin y distribuirlo con o sin mejoras; lo que se denominan las cuatro libertades del software libre: uso, estudio, mejora y distribución [3]. Aunque, no necesariamente todo el FOSS es gratuito, la mayoría de las veces sí lo es. Esto permite que el conocimiento se difunda y vaya mejorando, facilitando la labor de cada vez más personas.

Bajo esta idea se han formado numerosas comunidades FOSS, que están compuestas por muchísimas personas compartiendo conocimiento, soluciones e ideas. Las más conocidas son las de GNU Linux o Arduino. En los últimos años se ha creado una más, la de FPGAs.

1.3 Proyecto IceStorm – Estado del arte

Hasta hace unos años había herramientas libres que permitían realizar algunos de los pasos del proceso completo del diseño en FPGAs, como los de compilación y simulación de código Verilog que fueron desarrollados por Stephen Williams. [4]

En 2015, Clifford Wolf, profesor de electrónica en la Universidad de Artes Aplicadas de Viena, junto a Mathias Lasser y David Shah, desarrollaron un conjunto de herramientas software que daba una alternativa FOSS al desarrollo de FPGAs para una familia en concreto de ellas (Lattice iCE40). Este proyecto se denominó “IceStorm”.

Se basaron en las Lattice iCE40 ya que ellas tienen una estructura sencilla y bastante regular, además de no tener muchos tipos de celdas o unidades funcionales especiales. Esto las hacía perfectas para hacer un reconocimiento de su funcionamiento a través de ingeniería inversa y para crear las herramientas libres capaces de implementar diseños. [5] [6]

La familia de FPGAs Lattice iCE40 tiene varias gamas compatibles con el “Proyecto IceStorm” e incluso kits de evaluación de estos chips. Las características principales de las gamas compatibles aparecen en las Tablas 1, 2 y 3:

- De la iCE40 LP son compatibles las versiones LP384, LP1K, LP4K Y LP8 (Tabla 1)

Parámetros	LP384	LP1K	LP4K	LP8K
Celdas Lógicas	384	1280	3520	7680
NVCM	Sí	Sí	Sí	Sí
Consumo en estático	21 uA	100uA	250 uA	250 uA
Bits de RAM embebidos	0	32 K	80 K	128 K
PLL	-	1	2	2

Tabla 1: Parámetros de los chips ICE40 LP

- De la iCE40 HX son compatibles las versiones HX1K, HX4K y HX8K (Tabla 2)

Parámetros	HX1K	HX4K	HX8K
Celdas Lógicas	1280	3520	7680
NVCM	Sí	Sí	Sí
Consumo en estático	296 uA	1140 uA	1140 uA
Bits de RAM embebidos	64 K	80 K	128 K
PLL	1	2	2

Tabla 2: Parámetros de los chips ICE40 HX

- En su última actualización dieron también soporte para los dispositivos iCE40 UltraPlus UP3K y UP5K (Tabla 3).

Parámetros	UP3K	UP5K
Celdas Lógicas	2800	5280
NVCM	Sí	Sí
Consumo en estático	75 uA	75 uA
Bits de RAM embebidos	80 k	120 k
PLL	1	1

Tabla 3: Parámetros de los chips ICE40 UltraPlus

1.3.1 Proyectos similares

Actualmente se está trabajando en una herramienta que englobe todos los pasos (*Verilog to Bitstream*) no solo para las FPGAs de la familia ICE40, sino para muchas más. Esa herramienta se denomina SymbiFlow.

SymbiFlow engloba al "Proyecto IceStorm" además de dos proyectos paralelos éste, que son el "Proyecto Trellis" (destinado a FPGAs Lattice ECP5) y el "Proyecto X-Ray" (cuyo objetivo son las FPGAs de la Serie 7 de Xilinx). Ambos proyectos se beben del "Proyecto IceStorm" compartiendo el flujo *Verilog to Routing* para después crear en cada uno de ellos el bitstream necesario para cada su FPGA correspondiente. [8]

Actualmente, según [8], el "Proyecto Trellis" cubre la mayoría de las funcionalidades necesarias de las FPGAs Lattice ECP5 (como BRAM, multiplicadores, operadores lógicos, PLLs, puertos entrada/salida…).

En cambio, el "Proyecto X-Ray" aún está en una fase de desarrollo más temprana. Se está identificando y documentando el formato de bitstream que usan las FPGAs de la Serie 7 de Xilinx, en concreto de la Artix-7 xc7a50tfgg484-1. Cuando se complete el estudio de este dispositivo en concreto pretenden extender el estudio a todos los dispositivos Serie 7, UltraScale y UltraScale+.

1.3.2 Licencia ISC

El "Proyecto IceStorm" trabaja bajo la licencia ISC siendo ésta una licencia de software libre permisiva escrita por el Internet Systems Consortium (ISC). [9]

La licencia ISC dice, en su traducción al español:

> ***Se concede por la presente el permiso para usar, copiar, modificar y/o distribuir este software para cualquier propósito con o sin cargo, siempre y cuando el aviso de copyright anterior y este aviso de permiso aparezcan en todas las copias.***
>
> ***EL SOFTWARE SE PROPORCIONA "TAL CUAL" Y EL AUTOR RECHAZA TODAS LAS GARANTÍAS CON RESPECTO A ESTE SOFTWARE, INCLUIDAS TODAS LAS GARANTÍAS IMPLÍCITAS DE COMERCIABILIDAD Y ADECUACIÓN. EN NINGÚN CASO EL AUTOR SERÁ RESPONSABLE POR CUALQUIER DAÑO ESPECIAL, DIRECTO, INDIRECTO O CONSECUENTE, O CUALQUIER DAÑO QUE RESULTE DE LA PÉRDIDA DE USO, DATOS O BENEFICIOS, YA SEA EN UNA ACCIÓN DE CONTRATO, NEGLIGENCIA U OTRA ACCIÓN EXTRACONTRACTUAL QUE SURJA DE O EN CONEXIÓN CON EL USO O RENDIMIENTO DE ESTE SOFTWARE.***

1.3.3 Licencia GPL

Una de las licencias más utilizadas en el mundo FOSS es la licencia GNU GPL. El autor conserva los derechos de autor (copyright), y permite la redistribución y modificación bajo términos diseñados para asegurarse de que todas las versiones modificadas del software permanecen bajo los términos más restrictivos de la propia GNU GPL. Es decir, si se utiliza software bajo la licencia GPL para la creación de otro software, este último ha de estar bajo la licencia GPL también. [3]

En el diseño de la aplicación final se utiliza la licencia GPL ya que se ha modificado software bajo esta licencia.

1.3.4 Herramientas software necesarias

Las herramientas que se precisan para la realización de este proyecto son las siguientes:

- **Compilador y simulador de Verilog**: Icarus Verilog [4]
- **Visualizador de señales**: Gtkwave [10]
- **Sintetizador**: Yosys [11]
- **Place & route**: Arachne-pnr [12]
- **Utilidades y descarga en FPGA**: Icepack e Iceprog (Proyecto IceStorm) [5]

Simulador Verilog: Icarus Verilog

Icarus Verilog es una herramienta de simulación y síntesis de Verilog. Ésta funciona como compilador del código escrito con Verilog (IEEE-1364) obteniendo varios formatos de salida, como el formato .out (ejecutable para mostrar los datos de simulación por terminal) o el .vcd (archivo con las variaciones de las señales que puede ser leído por visualizadores como GTKwave).

La función principal de esta herramienta en este proyecto es la de compilar el código Verilog para detectar posibles errores en el código, y, si ha sido satisfactoria la compilación, crear el archivo VCD con toda la información de la simulación.

El compilador propiamente dicho está destinado a analizar y elaborar descripciones de diseño escritas en el estándar IEEE 1364 de 2005 [13]. Éste es un estándar bastante extenso y complejo, por lo que aún faltan por tener en cuenta muchas situaciones definidas en el estándar, pero poco a poco se irán agregando.

El autor principal de este compilador es Stephen Williams, ingeniero de software especializado en controladores para dispositivos y en sistemas embebidos, así como en diseño de hardware. Él mismo se define como “un ingeniero de software que escribe software para diseñadores de hardware”.

Debido a que esta herramienta es libre, muchas personas han participado ayudando a Stephen Williams creando parches, comprobaciones, e incluso partes del desarrollo. Esto, aparte de crear comunidad alrededor del proyecto, hace que avance constantemente y se solucionen los problemas que puedan surgir con bastante rapidez. De hecho, la propia herramienta permite extensiones de terceros.

Visualizador de señales: Gtkwave

GTKWave es un visualizador de señales basado en GTK+ con todas las funciones para Unix, Win32 y Mac OSX que lee archivos LXT, LXT2, VZT, FST y GHW, así como archivos Verilog VCD (obtenido al simular el testbench con Icarus Verilog) o EVCD permitiendo su visualización.

Con él, se podrá observar gráficamente el desarrollo temporal de las señales del diseño hardware para visualizar si todo funciona acorde a lo esperado.

Sintetizador: Yosys

Yosys es un entorno de trabajo para la síntesis RTL de Verilog. Actualmente tiene soporte hasta Verilog 2005 y proporciona un set básico de algoritmos de síntesis para obtener diversos ficheros:

- Procesa casi cualquier diseño sintetizable de Verilog-2005
- Conversión de Verilog a netlist BLIF / EDIF / BTOR / SMT-LIB / simple RTL Verilog / etc.
- Asignación a bibliotecas de células estándar ASIC (en formato de archivo Liberty)
- Asignación para Xilinx 7-Series y Lattice iCE40 FPGA.

Yosys permite añadir scripts de síntesis de terceros adicionales (algoritmos) según se necesite, pudiendo agregar alguna funcionalidad o solucionar algún bug o caso no contemplado.

Yosys es software libre bajo licencia de ISC, la cual concede "el permiso para usar, copiar, modificar y/o distribuir este software para cualquier propósito con o sin cargo, siempre y cuando el aviso de copyright anterior y este aviso de permiso aparezcan en todas las copias".

Place and Route: Arachne-pnr

Arachne-pnr implementa el paso de *Place and Route* en el proceso de compilación de hardware para FPGA. Acepta como entrada una *netlist* mapeada en tecnología en formato BLIF (obtenida por el conjunto de síntesis Yosys, por ejemplo) para dar a la salida una representación del bitstream en formato de texto (.txt). Actualmente se dirige a la familia de FPGA Lattice iCE40.

Yosys, Arachne-pnr e IceStorm forman una cadena de herramientas Verilog-to-bitstream completamente Open Source para las FPGA Lattice de la familia iCE40.

Durante agosto de 2019, David Shah anunció que *Arachne-pnr* dejó de tener mantenimiento debido a que hay otra herramienta similar, también desarrollada por él: *nextpnr*. *Nextpnr* adquiere todas las funcionalidades de *Arachne-pnr* y agrega nuevas, como la compatibilidad (por ahora experimental) con el "Proyecto Trellis" y en el futuro con el "Proyecto X-Ray". En este trabajo se utiliza *Arachne-pnr* ya que la aparición de *nextpnr* es posterior a la finalización de todo el desarrollo realizado. [14]

Utilidades y descarga en FPGA: Proyecto IceStorm

Las FPGAs de la familia iCE40 tienen una arquitectura muy minimalista con una estructura muy regular y además de no tener muchos tipos diferentes de mosaicos o unidades funcionales especiales. Esto las hace ideales para la ingeniería inversa y como una plataforma de referencia para el desarrollo de herramientas de FPGA de uso general.

Las herramientas que incluye el Proyecto IceStorm son: Icepack (para diversas funcionalidades, principalmente la traducción del bitstream de texto a binario) y Iceprog (utilizado para la descarga en la FPGA del diseño)

En definitiva, el proceso establece el siguiente diagrama de flujo:

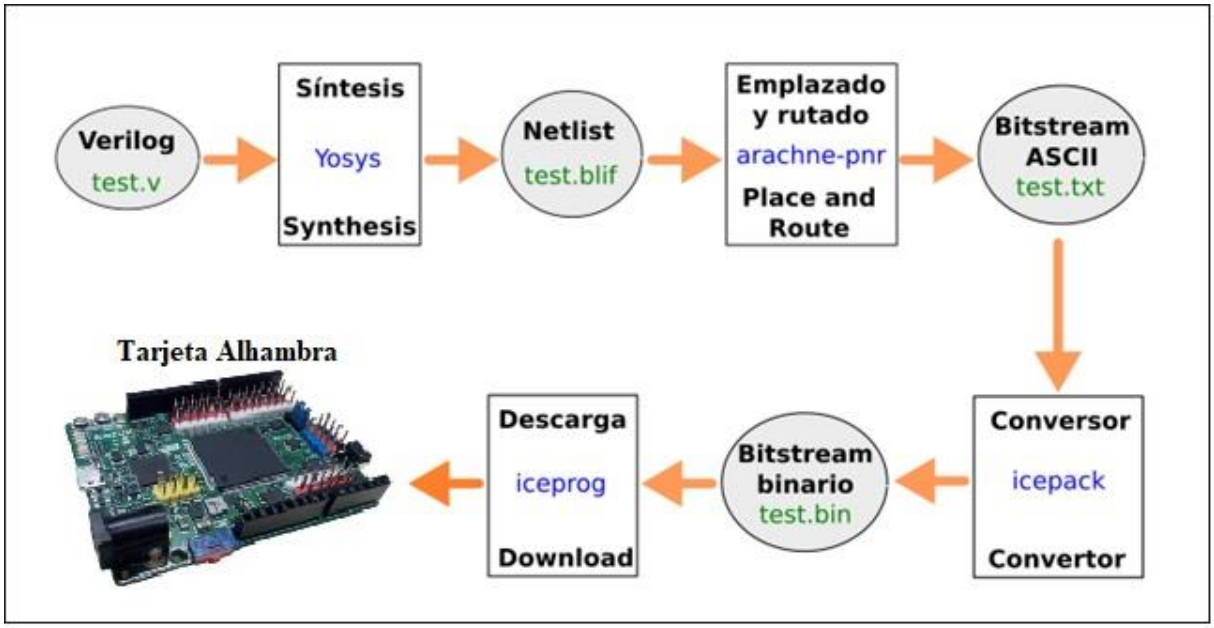


Ilustración 1: Flujograma de las herramientas del "Proyecto IceStorm"

1.4 Objetivos

Este trabajo está enfocado en la explicación y utilización de las herramientas libres anteriormente mencionadas para que un usuario, desde cero hasta la creación de un proyecto con HDL funcional.

Por ellos los objetivos principales son:

- Dar una ligera introducción al mundo FOSS y de las FPGAs
- Explicar algunas de sus utilidades y realizar pequeños diseños con Verilog.
- Implementarlos en la tarjeta Alhambra.
- Desarrollar una aplicación compleja que demuestre si las herramientas expuestas en este trabajo tienen un nivel de desarrollo y utilidad suficiente además de explorar posibles carencias.

2. Hardware

En el presente trabajo se ha decidido trabajar con la tarjeta Alhambra 1.1, la cual tiene como núcleo la FPGA (iCE40 HX1K), la cámara OV7670 y una tarjeta Arduino UNO para la configuración inicial de la cámara. El conexionado para el diseño final se explicará en el apartado 6.1.6

2.1 Alhambra 1.1

Esta tarjeta ha sido diseñada y fabricada por Mareldem Technologies, una pequeña empresa de consultoría de proyectos electrónicos de Pinos del Valle (Granada), donde trabaja Eladio Delgado (Diseñador de las tarjetas Alhambra y Director de Ingeniería de Mareldem Technologies) con el apoyo Juan González (Universidad Rey Juan Carlos), entre otros, con el fin de crear un dispositivo hardware que pudiera darle una dimensión práctica y accesible al diseño en FPGA con herramientas FOSS. [15]

Ilustración 2: Tarjeta Alhambra 1.1

Las principales características de la Alhambra 1.1 son las siguientes:

- Open hardware
- Tarjeta de desarrollo FPGA (ICE40HX1K-TQ144 [16] de Lattice)
- Compatible con la Opensource IceStorm Toolchain, de Clifford Wolf
- Multiplataforma: Linux / Mac / Windows
- Tarjeta estilo Arduino: similar pinout al del Arduino ONE / BQ zum.
- Compatible con shields diseñados para Arduino/Zum
- Oscilador MEMS de 12 MHZ
- Switch ON/OFF Hardware
- Alimentación: 6 - 17v
- Máxima corriente de entrada: 3A
- 20 input/output 5v pins
- 8 input/Output 3.3V pins
- USB micro-B para programar la FPGA desde PC

- El dispositivo FTDI 2232H USB permite programar la FPGA y comunicación UART con PC
- Pulsador de reset
- 8 leds programables
- 2 pulsadores programables
- 4 entradas analógicas a través del bus I2C
- Protección hardware contra cortocircuitos, cambios de polaridad, etc.

2.2 Cámara OV7670

Para comprobar la utilidad de las herramientas software antes descritas se plantea hacer un diseño complejo para la demostración de su funcionalidad. Se decide hacer una aplicación simple de visión artificial: un detector de bordes. Para la obtención de imágenes se necesita una cámara y se decide elegir la cámara OV7670 [17] debido a su reducido coste, sus conexiones, y a su familiaridad con el mundo FOSS ya que tiene numerosos proyectos con Arduino.

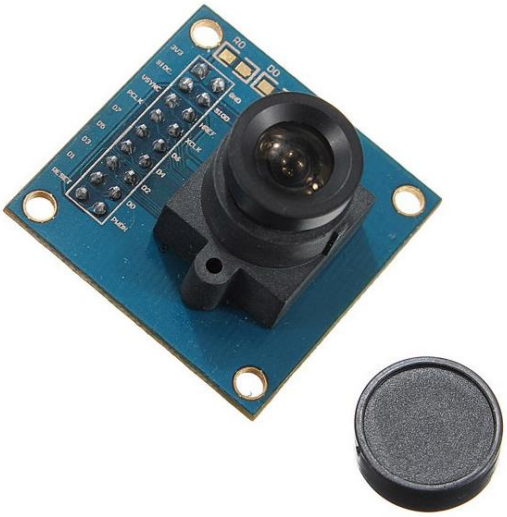


Ilustración 3: Cámara OV760

La cámara cuenta con las especificaciones principales:

- Voltaje de operación de 3.3V en DC
- Consumo de 60mW/15fpsVGAYUV
- Corriente de Standby <20µA
- Transmisión de 8 bits en paralelo
- Lente de 1/6’’
- Ángulo de visión de 25º
- Resolución máxima de VGA (640/480)
- Sensibilidad de 1.1V / (Lux-sec)
- Ratio Señal-Ruido (SNR) de 46 dB
- Modo de vista progresivo

- Lente de alta calidad (F1.8/6mm)
- Framerate máximo de 30 fps para VGA
- Formatos de Salida: Raw RGB (8 bits), RGB (GRB 4:2:2, RGB 565/555/444), YUV (4:2:2) y YCbCr (4:2:2)
- Necesita de una señal de frecuencia mínimo de entrada de 8MHz mínimo

2.3 Arduino UNO

Aunque en un principio se iba a utilizar únicamente para comprobar que la cámara funcionaba correctamente, al final, la Arduino UNO se incorporó en el diseño final de la aplicación completa. Su cometido es el de configurar la cámara OV7670 ajustando el tamaño, el formato y la velocidad de envío de la imagen.

Se elije la Arduino UNO debido a que ya hay aplicaciones libres funcionales diseñadas para Arduino y la OV7670, además de ser un dispositivo barato, fácil de conseguir, su lenguaje de programación es bastante sencillo y cuenta con muchísima información en internet.

En concreto, se utiliza el modelo Arduino UNO R3 CH340G SMD [18] ya que ya se contaba con uno y no hizo falta comprar otra tarjeta. Sus características técnicas principales son:

- Microcontrolador: ATMEGA328P
- Alimentación: 5V
- Alimentación recomendada: 7-12V
- Límite de alimentación: 6-20V
- Pines digitales (entrada / salida): 14 (6 con PWM)
- Entradas analógicas: 6
- Corriente máxima por pin: 20mA
- Memoria flash: 32KB
- SRAM: 2KB
- Reloj: 16MHz

Ilustración 4: Arduino UNO CH340G SMD

2.4 Adaptadores de tensión

Como después se aclarará en el apartado 6.4 para adaptar en tensión las señales de salida de la cámara hacia la Alhambra se han usado unos adaptadores de tensión "cd40109b-q1" de Texas Instrument. [19]

3. Instalación de las herramientas software

Este trabajo ha sido desarrollado en una distribución basada en "Ubuntu 16.04.5 LTS", por lo que toda esta explicación está verificada para Ubuntu 16.04 aunque también está soportada oficialmente para las versiones de Ubuntu 14.04, 15.10 y para Fedora 22. [20]

3.1 Instalación automática

David Cuartielles, uno de los cofundadores de Arduino, también ha apoyado al "Proyecto IceStorm", creando un instalador que realiza todo el proceso de descarga de herramientas, compilación e instalación de todas las herramientas necesarias tanto para Ubuntu (14.4, 15.10, 16.4) como Fedora 22. [21]

Los comandos de consola para la instalación automática son:

```
$ git clone https://github.com/dcuartielles/open-fpga-install.git
$ cd open-fpga-install
$ sudo bash install.sh
```

3.2 Instalación manual

Si ocurriera algún error en la instalación automática de alguno de las herramientas o librerías, también se puede utilizar la instalación manual de cada una de ellas, o directamente instalarlo todo manualmente, uno a uno. Para ello, se deben lanzar por terminal los siguientes comandos para cada parte: [20]

- Lo primero será instalar las **dependencias** necesarias, es decir, aquellas aplicaciones y bibliotecas que no se usan directamente, pero son necesarias para que funcionen correctamente el resto de las aplicaciones del "Proyecto IceStorm":

```
$ sudo apt-get install build-essential clang bison
flex libreadline-dev gawk tcl-dev libffi-dev git
mercurial graphviz  xdot pkg-config python python3
libftdi-dev
```

- Instalación de **IceStorm Tools** (icepack, icebox, iceprog):

```
$ git clone https://github.com/cliffordwolf/icestorm.git
icestorm
$ cd icestorm
$ make -j$(nproc)
$ sudo make install
$ cd ..
```

- Instalación de **Arachne-PNR** (the place&route tool):

```
$ git clone https://github.com/cseed/arachne-pnr.git arachne-
pnr
$ cd arachne-pnr
$ make -j$(nproc)
$ sudo make install
$ cd ..
```

- Instalación de **Yosys** (Verilog synthesis):

```
$ git clone https://github.com/cliffordwolf/yosys.git
yosys
$ cd yosys
$ make -j$(nproc)
$ sudo make install
$ cd ..
```

- Instalación de **Icarus Verilog** y **GTKwave**

```
$ sudo apt-get install gtkwave iverilog
```

3.3 Configuración de la descarga en placa

Para la descarga en placa, ya sea en la IceStick o en la Alhambra (ambas con la FPGA iCE40LP/HX1K), se necesita la biblioteca *libftdi*, por ello es necesario tener permisos de acceso. Esos permisos se pueden conseguir utilizando "sudo" antes del comando de la descarga en placa (`$ sudo iceprog test.bin`), o bien configurar el *udev* (el gestor de dispositivos que usa el kernel Linux) para que, al conectar la placa al USB, el usuario tenga permisos.

El problema de la opción de *sudo* es que a cada cierto tiempo se pedirá introducir la contraseña para dar los permisos de ejecución y puede ser muy molesto cuando se van a hacer muchas descargas en placa para su comprobación. Por ello, en este trabajo se ha optado por la opción de configurar el *udev* y tener los permisos directamente [19]. Para ello hay que hacer lo siguiente:

- Crear el archivo “/etc/udev/rules.d/80-icestick.rules” con el siguiente contenido: ACTION=="add", SUBSYSTEM=="usb", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6010", OWNER="user", GROUP="dialout", MODE="0777"

- La forma más sencilla de crearlo es ejecutar este comando por la terminal porque el archivo que se necesita debe estar en una carpeta privilegiada del sistema:

```
$ sudo sh -c "echo 'ACTION=="add", SUBSYSTEM=="usb", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6010", OWNER="user", GROUP="dialout", MODE="0777"' >>/etc/udev/rules.d/80-icestick.rules"
```

- Ejecutar este comando para relanzar el administrador de udev y cargue la nueva regla:

```
$ sudo udevadm control --reload-rules && sudo udevadm trigger
```

Ahora ya se podría hacer la descarga en placa del archivo binario deseado (el cual se explicará su obtención en el siguiente apartado) con el siguiente comando por terminal :

```
$ iceprog ejemplo_bitstream.bin
```

En Ubuntu es necesario reiniciar la máquina para que funcione correctamente.

3.3 Otras herramientas software

3.3.1 Icestudio IDE

El equipo detrás del diseño de la tarjeta Alhambra también ha diseñado una interfaz gráfica para el diseño de sistemas electrónicos, como un HDL, mediante bloques: Icestudio IDE (Ilustración 5). Esta herramienta cuenta con las siguientes características: [22]

- Interfaz sencilla
- Compatible con todas las FPGAs pertenecientes al “Proyecto IceStorm”
- Multiplataforma (GNU/Linux, Windows y Mac OS DMG)
- Exporta diversos tipos de archivos: Verilog, PCF, Testbench, GTKWave, BLIF, ASC y Bitstream.

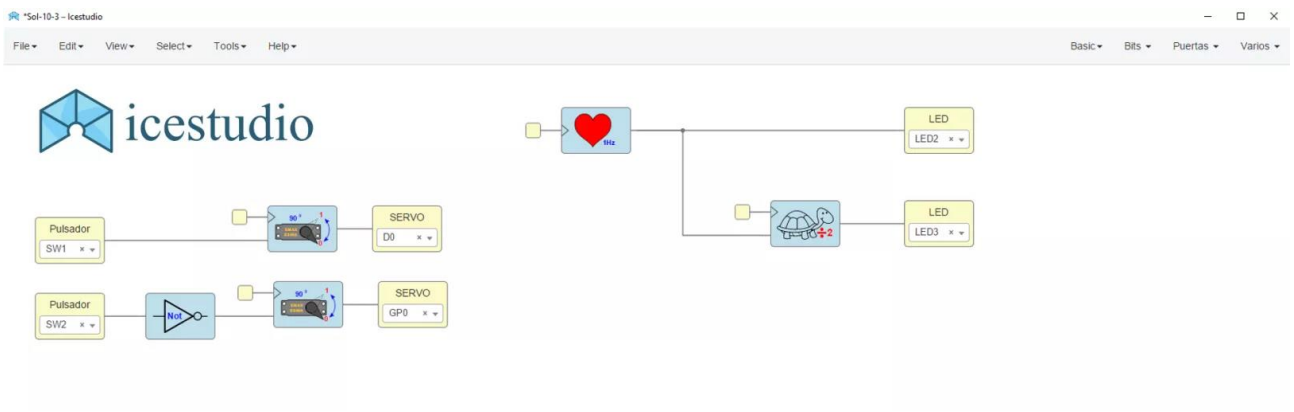


Ilustración 5:Captura herramienta IceStudio

3.3.2 Editor de texto

Para la creación y modificación de los archivos Verilog del diseño hardware es necesario un editor de textos. Prácticamente vale cualquiera, pero se recomienda uno que tenga apoyo para Verilog, que remarque las funciones propias del lenguaje, los pares de paréntesis…

Hay muchos: *Notepad++*, *Emacs* o *gedit* (entre otros). En este trabajo de ha decidido utilizar *gedit* ya que se trabaja en GNU/Linux y viene predeterminado en muchas de las distribuciones libres de GNU/Linux.

3.3.3 Terminal de puerto serie.

Se necesita un terminal de puerto serie para recibir la información que se enviará desde la FPGA en la aplicación del desarrollo final. Hay numerosos programas que se pueden utilizar como GtkTerm, Hyperterminal o usando comandos a través del terminal.

Se ha decidido utilizar GtkTerm debido a que su interfaz gráfica en Ubuntu hace simple su uso (configurar los baudios, transformar la información recibida en un archivo binario) además de ser gratuito.

Para instalarlo en Ubuntu únicamente hay que lanzar por terminal:

```
$sudo apt-get install gtkterm
```

3.3.4 Editor de texto binario/hexadecimal

Una vez recibido el archivo binario a través del terminal de puerto serie, se necesita un editor de archivos binarios para extraer y organizar la información necesaria de él.

Se ha optado por usar Bless HEX Editor, ya que fue el primer editor encontrado que se adaptaba a las necesidades del proyecto (más información en el apartado 6.3.8)

Para instalarlo en Ubuntu:

```
$ sudo apt-get install bless
```

4. Uso de las Herramientas del Proyecto IceStorm

A continuación, se detallarán cómo utilizar cada una de las herramientas del "Proyecto IceStorm":

4.1 Cómo Usar YOSYS

La herramienta Yosys la utilizaremos de sintetizador del diseño HDL para obtener las netlist en el formato .blif a partir de todos los archivos en formato Verilog.

Escribiendo en la línea de comandos:

```
$ yosys -p "synth_ice40 -blif test.blif" test.v
```

La opción "-p" ejecuta el comando, de las librerías de Yosys, incluido entre las comillas. En este ejemplo es "synth_ice40" seguido de "-blif test.blif", lo cual significa que se va a hacer una síntesis basada en las FPGAs de la familia ice40 y se va a obtener a la salida un archivo .blif de nombre test.blif .

Después de todos las opciones y comandos, al final de la sentencia se nombran la ruta de los .v los cuales se quieran sintetizar, en este caso, solo "test.v"

Hay numerosas opciones y comandos, por ello es interesante tenerlas en cuenta. Si se pone en la línea de comandos:

```
$ yosys -h
```

Se muestran las diversas opciones que tiene la herramienta y una pequeña descripción de cada una. Por ejemplo, se puede comprobar que la opción "-p" se encarga de ejecutar comandos.

Con "-H" (en vez "-h") se obtiene una lista de todos los comandos ejecutables y una pequeña descripción de los mismos.

Si se quisiera conocer exactamente cómo funciona un comando bastaría con escribir por terminal:

```
$ yosys -p 'help comando'
```

4.2 Cómo usar Arachne-pnr

Arachne-pnr es la herramienta de emplazado y rutado a usar y, para este proyecto, se utiliza de forma normal con la siguiente estructura:

```
$ arachne-pnr -d 1k -p test.pcf -o test.txt test.blif
```

Las opciones que se utilizan en este caso son “-d 1k” , “-p test.pcf” y “-o test.txt”, seguido del archivo de entrada “test.blif”.

- -La opción “-d” selecciona la FPGA concreta a la que se le realiza el emplazado y rutado, en este caso, se selecciona “1k” haciendo referencia a la Lattice Semiconductor iCE40LP/HX1K.
- Con “-p” , se le suministra como entrada el archivo .pcf, el cual relaciona los pines físicos de la FPGA con los cables de salida del módulo superior del diseño.
- Por último, “-o” nombra el archivo de salida (de texto) de las operaciones realizadas por Arachne-pnr, en este caso “text.txt”, el cual contiene el bitstream en formato .txt

Al igual que con Yosys, la herramienta Arachne-pnr, dispone de un menú de ayuda:

```
$ arachne-pnr -h
```

Arachne-pnr es una herramienta más sencilla por lo que solo precisa de un único menú de ayuda.

4.3 Cómo usar Icepack

Aunque de la herramienta Arachne-pnr obtenemos un bitstream, su formato es de texto y para programar definitivamente la FPGA necesitamos un conversor a binario (.bin), éste será la herramienta icepack, ejecutándose de la siguiente forma:

```
$ icepack test.txt test.bin
```

Éste no requiere prácticamente de opciones (aunque también tiene disponibles), únicamente se introduce en primer lugar el nombre del bitstream de texto a convertir, y al final el nombre del bitstream binario, test.txt y test.bin respectivamente.

De igual forma, para conocer todas las opciones de las que dispone Icepack, hay que usar:

```
$ icepack -h
```

4.4 Cómo usar Iceprog

Una vez obtenido el bitstream binario, únicamente queda cargarlo en la FPGA. Para ello se ha de conectar vía USB la tarjeta que incorpore la FPGA, asegurarse que el sistema la ha reconocido, y ejecutar la herramienta Iceprog suministrándole el bitstream binario deseado:

```
$ iceprog test.bin
```

Una vez más, la herramienta Iceprog tiene su propio menú de ayuda que muestra varias opciones (especificar el dispositivo USB el cual se quiere programar, modo de borrado, modo de escritura…) aunque esta vez se ejecuta de la siguiente forma:

```
$ iceprog --help
```

A pesar de haber varias opciones, las que están por defecto funcionan perfectamente, simplemente se echa mano de ellas cuando el diseño requiere algo especial.

4.5 Cómo usar Icarus Verilog

Icarus Verilog hace las veces de simulador del diseño completo en Verilog además de servir de simulador no gráfico (mostrando mensajes en la terminal predefinidos para comprobar su funcionamiento)

Para ejecutar la herramienta en este ejemplo simplemente hay que seguir la siguiente estructura:

```
$ iverilog test.v test_tb.v -o test_tb.out
```

- Primero se incluyen todos los archivos en verilog del diseño (librerías propias incluidas), en este caso solo "test.v"
- Después el archivo testbench el cual programa entradas para el módulo a testear y puede leer las salidas de éste.
- Por último, la opción "-o" seguido de "test_tb.out" que define el archivo que incluye el ejecutable de la simulación.

Icarus Verilog también tiene un menú de ayuda, el cual se ejecuta de la siguiente forma:

```
$ iverilog -h
```

Este menú no tiene demasiada información, únicamente listados algunas opciones, pero sin descripción alguna. Esto se soluciona mirando los manuales que tiene la propia herramienta o alguna página web que los tenga recopilados [23]

Una vez compilado el diseño, se procede a la simulación con el ejecutable:

```
$ ./test_tb.out
```

Al ejecutarlo, empieza la simulación que se ha definido en el testbench imprimiendo por pantalla los mensajes que hayamos definido en él, dependiendo de los valores de las variables o si funciona como se esperaba.

Sobra decir que ésta es la simulación funcional, ya que no se tiene en cuenta la FPGA específica que se va a usar ni los retardos en los cambios de las señales.

Si en el testbench se han incluido las sentencias "$dumpfile("test_tb.vcd");" y "$dumpvars(0, test_tb);" al simular con el ejecutable anterior, con primera sentencia se creará el archivo "test_tb.vcd" y con la segunda guardará cada variación de las señales del diseño en el archivo creado (que se verá en el apartado 5.2.1).

El archivo “test_tb.out” a pesar de que se pueda leer con un editor de texto, es bastante difícil leer la información que contiene (al fin y al cabo, es un ejecutable), pero al archivo .vcd si que se le puede hacer alguna lectura: (hay ejemplos de ambos en el Anexo)

- Para empezar, se pueden ver la fecha de su creación (es decir de la simulación), o la escala de tiempo a la que se va a mostrar la simulación:

$date	*$timescale*
Mon Sep 16 18:06:15 2019	*1s*
$end	*$end*

- Después se pueden ver las instanciaciones de los módulos y de las señales incluidas en ellos:

$scope module app $end
$var wire 1 " PCLK $end
$var reg 1 g PCLK_ant $end
$var wire 7 % addr_read0 [6:0] $end
…
$enddefinitions $end

$scope module indica que se abre el módulo *app* y que las siguientes sentencias pertenecen a este módulo. *$var* indica en la sentencia que se trata de una variable del tipo *reg* o *wire*, seguido del número de bits asociados a esa variable. Después hay un carácter que indica su posición dentro de la lista de variables (en formato ASCII) dentro de su módulo y, por último, el nombre de la variable.

Al final de todas variables aparece $*enddefinitions* indicando que ya no hay más definiciones.

- A partir de este punto, parece ser la descripción de las señales que varían en cada instante, iniciándolo con #x y finalizándolo con $end. Lo que deberían ser las variaciones son practicamente ilegibles para un humano, pero las interpreta perfectamente la sigueinte herramienta: GTKWave.

#1
bx B
bx D$
bx N$
bx @$
bx M$
…
$end

4.6 Cómo usar GTKWave Analyzer

GTKWave es el visualizador de las señales que se han simulado con Icarus Verilog. El archivo el cual ejecuta esta herramienta es un .vcd (value change dump) el cual crea Icarus Verilog.

Dicho archivo documenta los cambios de las señales producidos en la simulación y los hace legibles para GTKWave, de forma que se puedan visualizar y analizar de forma cómoda en una interfaz gráfica

La ejecución básica por terminal de esta herramienta es la siguiente:

```
$ gtkwave test_tb.vcd
```

En la Ilustración 6 aparece una captura de un proyecto más avanzado, buffer_tb.vcd, la cual servirá para explicar la interfaz.

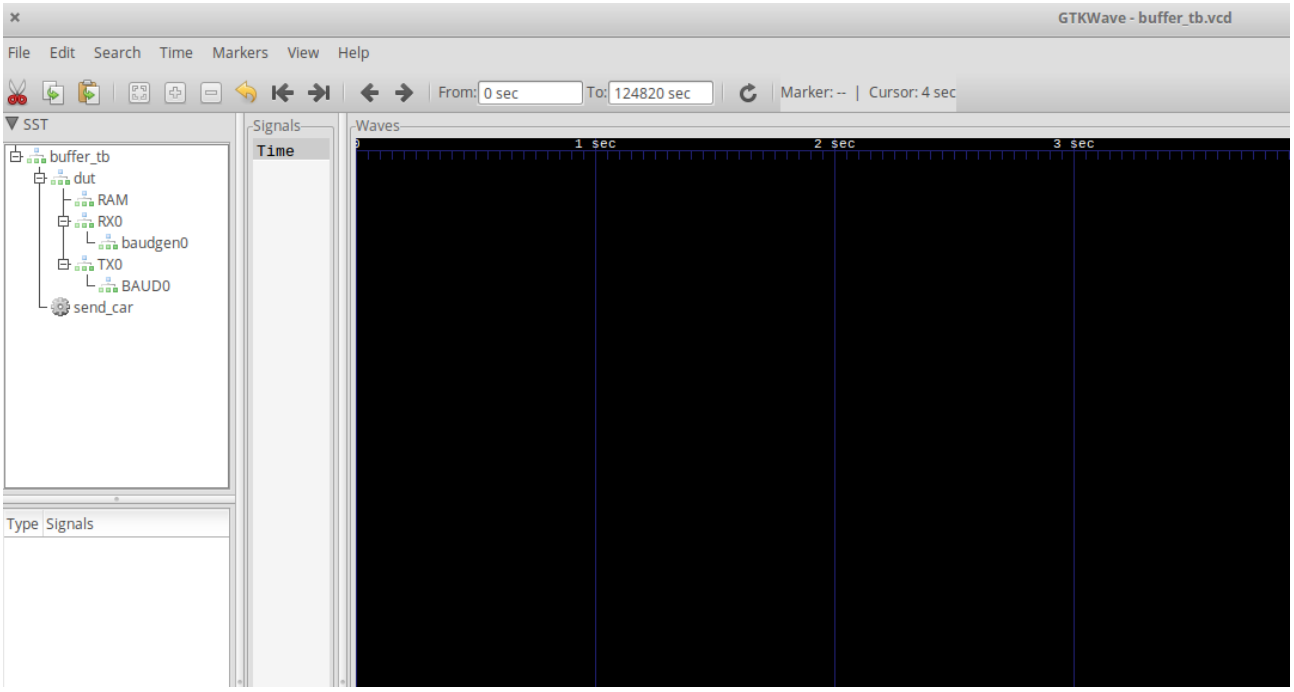


Ilustración 6:GTKWave mostrando la jerarquía del DUT

Al inicio no aparecerá ninguna señal en la zona “Waves”, hay que cargarla. Para ello se selecciona, en la ventana “SST”, el módulo del cual se quieren observar las señales, tanto de registros como de cables, que aparecerán en la ventana inferior “Type | Signals”. Al arrastrarlas a la ventana “Signal” aparecerán su representación temporal en la ventana “Waves”, como se puede ver en la Ilustración 7.

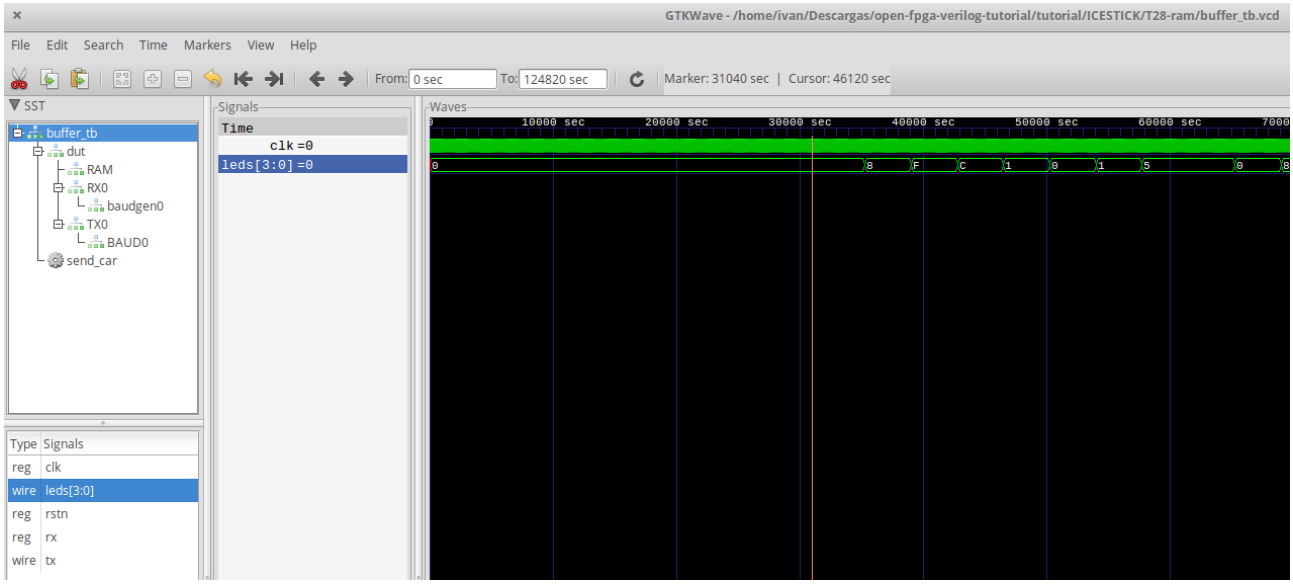


Ilustración 7: GTKWave representando "Signals" en la ventana "Waves"

Para visualizarlas, simplemente hay que arrastrar el cable o el registro desde la ventana inferior a la zona de "Signals". Presionando el segundo botón del ratón sobre la señal que se desee de la zona "Signals" podemos seleccionar algunas opciones como el formato en el que se muestra la información, el color o incluso removerla del panel (con "Cut").

Se debe de tener muy claro que GTKWave no simula las señales, simplemente representa la información de simulación obtenida por Icarus Verilog, es decir, no puedes mostrar más tiempo de simulación del que se ha definido en el testbench o ha adquirido la herramienta Icarus Verilog.

4.7 Uso del comando de terminal "make"

En el diseño final, debido al gran número de archivos y de comandos por terminal, se decidió utilizar un archivo "Makefile" (el cual se puede consultar en el Anexo) para realizar las tareas descritas en punto 4.

El archivo "Makefile" es un script para la terminal de Linux el cual tiene escritas en su interior todas las sentencias necesarias para poder ejecutarlas, todas o algunas de ellas, escribiendo por terminal únicamente "make" seguido de la opción que se ha programado, por ejemplo:

```
$ make sint
```

El archivo "Makefile" se puede dividir en 3 partes:

- **Declaración de archivos:** En esta parte se declaran dos macros distintas, NAME (que incluye el nombre del archivo de más alto nivel del proyecto) y DEPS (que incluye los nombres de todos los archivos necesarios para la simulación y síntesis

del mismo) de forma que sea más sencilla e intuitiva la programación de los comandos.

```
NAME = imagen_uart

DEPS = control_uart.v  uart_tx.v baudgen.v
```

- **Configuración de opciones disponibles:** Aquí se agrupan los comandos necesarios para cada una de las opciones (en el ejemplo de Anexo, *sim*, *sint* y *clean*)

La forma de agruparlos es la siguiente:

```
sim: $(NAME).v $(DEPS) $(NAME)_tb.v

        #-- Compilar
        iverilog $^ -o $(NAME)_tb.out

        #-- Simular
        ./$(NAME)_tb.out

        #-- Ver la simulacion con gtkwave
        gtkwave $@ $(NAME)_tb.gtkw &
```

Como se puede observar, la opción *sim* incluye las sentencias de compilación, simulación y visualización que se han descrito anteriormente pero, para realizarlo todo de un solo paso y simplificar la escritura se incluye todo dentro.

Por ejemplo, al aparece *"$(NAME)_tb.out*" se hace referencia al archivo *"imagen_uart_tb.out"*, el ejecutable de simulación.

Cuando se pone "*$^*" se quiere incluir en la sentencia de *iverilog* todos los archivos incluidos en la parte superior *($(NAME).v $(DEPS) $(NAME)_tb.v)*

De esta forma se ejecutan consecutivamente y se hace mucho más amena las simulaciones que se realicen en poco tiempo debido a que se han realizado correcciones de código.

Se ha incluido también la opción de borrado de los archivos resultado de cada una de las opciones, ya que si se quiere ejecutar una simulación de un archivo ya existente (sin ninguna modificación) a través de la opción "*sim*", aparecerá por terminal que no se han producido cambios y no la ejecutará. Por ello, es mucho más cómodo limpiar todos los archivos resultado y volver a ejecutar "*make sim*" que ejecutar el comando de visualización con el nombre del archivo concreto.

```
#-- Limpiar todo

clean:

        rm -f *.bin *.txt *.blif *.out *.vcd *~
```

5. Tutoriales: Verilog, base, medio, completo.

Antes de realizar los primeros diseños de hardware, lo primero es entender la estructura que tiene Verilog y cómo funciona este lenguaje.

5.1 Introducción a Verilog

Los lenguajes HDL se basan en diseño modular, en el cual se establecen "módulos" o cajas con sus entradas y salidas, que realizan una o más operaciones. Con cada módulo, una vez se ha verificado su comportamiento, puede ser integrado en diseños mayores como si de una librería se tratara. En un diseño con jerarquía superior, no habrá que preocuparse en qué es lo que hace exactamente por dentro dicho módulo, únicamente en qué entradas recibe y si las salidas sean las correspondientes.

5.1.1 Sentencia "module"

Los módulos se definen con la sentencia "*module"* seguido del nombre con el cual se va a asignar y de la lista de entradas y salidas que va a tener dicho modulo. Una vez definido todo el comportamiento interno de éste se cierra la definición del mismo con "*endmodule*".

```
module nombre (input wire entrada, output wire salida);
        ....
endmodule
```

Los HDL suelen seguir la siguiente norma: incluir un único módulo en cada archivo .v. Así a la hora de almacenar archivos .v se sabe perfectamente que incluye cada uno y es mucho más sencillo a la hora de compilar también. Por ello los archivos de Verilog se suelen nombrar igual que el nombre del módulo que se ha diseñado en él.

5.1.2 Tipos de dato

Internamente los módulos trabajan con varios tipos de datos:

- Nets: Representan las uniones estructurales entre componentes. No tienen capacidad de guardar información. Hay muchos tipos distintos de *nets*, pero los más utilizados son los *wire.*
- Registros: Representan variables con capacidad de almacenar información. Los más representativos son *reg* e *integer* (este último únicamente en la creación de testbench)
- Macros: Se pueden definir constantes (a través de *'define MACRO),* sin tipo de dato o tamaño, con fin de clarificar el código
- Parámetros: *paramenter* y *localparameter.* Ambos se suelen utilizar para almacenar el valor de constantes. Mientras que las macros no pueden tener ni tipo ni tamaño predefinido, los parámetros sí. La diferencia entre las dos es que *localparameter* sólo es accesible dentro del módulo en el que es declarada (constante local).

Los datos que se vayan a utilizar en el módulo se han de declarar, y por visibilidad suele ser al principio. Para ello basta con indicar el tipo de dato y el nombre que se le quiere dar, además del número de bits que va a incluir. También se suele aprovechar para dar el valor inicial.

5.1.3 Tipos de procesos

Los módulos pueden usar varios tipos de procesos para definir sus funciones, los tres más representativos son: *inital, always y assign:*

- El proceso *inital* se ejecuta a parir del instante inicial y puede encerrar una o una serie de instrucciones que se realizan una detrás de otra (lógica secuencial).
- El proceso *always*, que se activa cuando varia uno de los datos incluidos en la lista sensible, puede también incluir una o una serie de instrucciones que se realizan una detrás de otra.
- El proceso *assign* responde a la lógica combinacional que, a diferencia de los dos procesos anteriores, se ejecuta de forma continua y no precisa de lista sensible. Se utiliza para asignar únicamente datos tipo *net*.

Cuando un proceso *initial* o *always* tienen más de una asignación procedural (=) o una estructura de control (if-else, cause, for…), éstas deben de estar contenidas entre *begin* y un *end*.

5.1.4 Estructuras de control básicas

Dentro de los procesos *initial* y *always* se pueden utilizar varias estructuras de control básicas, algunas son:

- **If-else**. Si se da la condición del if ejecuta las instrucciones asociadas al if , si no, comprueba por orden las de los siguientes else if para ejecutar las instrucciones correspondientes, y, si ninguna de las anteriores se ha dado, ejecuta las instrucciones asociadas al else. Es imprescindible que siempre se empiece esta estructura de control con la sentencia if , pero las sentencia else if y else no son obligatorias.

```
if (enable == 0'b1)
     a<= 0;

else if (enable == 1'b1 && rst == 1'b1)
begin
     a<= 1;
     b<= 1;
end

else
```

```
begin
     a<= 1;
     b<= 0;
end
```

- **Case.** Se compara una variable con distintos valores y se ejecutan las sentencias asociadas con dicho valor.

```
     case (variable)
          <caso1>: sentencia;
          <caso2>: begin
               sentencia;
               sentencia;
               end
          ....
          <default>: sentencia;
     endcase
```

- **For.** Igual que en C, pero con una sintaxis algo distinta.

```
for (i=0, i<64, i=i+1)) //Inicialización de memoria RAM
     ram[i] = 0;
```

Al igual que en el resto de los casos, cuando en una estructura de control se han de encadenar varias sentencias, se utilizan los delimitadores *begin-end*.

5.2 Ejemplo Base

Para entenderlo mejor, se verá todo con un ejemplo sencillo:

```
     module div3(input wire clk_in, output wire
clk_out);

     reg [1:0] divcounter = 0;

     //-- Contador módulo 3
     always @(posedge clk_in)

               if (divcounter == 2)
                  divcounter <= 0;

               else
                  divcounter <= divcounter + 1;

     //-- Sacar el bit más significativo por clk_out
     assign clk_out = divcounter[1];

     endmodule
```

En este caso, el módulo "*div3"* tiene un bit de entrada "*clk_in"* y una salida *"clk_out"*. Su función es hacer de divisor de frecuencia de la señal "c*lk_in"* entre 3, aprovechando el segundo bit de "*divcounter"* para la señal de salida ya que tiene un paso de 0 a 1 (flanco de subida) por cada tres flancos de subida de la señal de entrada.

Primero se define un registro "*divcounter*", de dos bits, y se inicializa a 0. Después se inicia un proceso "*always*" el cual se activa únicamente cuando se da un flaco de subida (posedge) de la señal "*clk_in*", tal y como se ve en su lista de sensibilidad.

Dentro del proceso "*always*" se utiliza una estructura de control de tipo if-else por lo que, cada vez que se dé un flanco de subida en "*clk_in*" comprobará el valor de "*divconunter*"; si éste es igual a 2, variará su valor a 0, si no lo es, incrementa en una unidad su valor. El proceso "*always*" sólo tiene una estructura de control (cada grupo de if-else cuenta como una única estructura de control) y las sentencias if y else solo tienen una asignación cada una, por lo que ninguna precisa de los delimitadores *begin-end*.

Al final del módulo se utiliza una asignación procedural mediante el proceso "*assign*" el cual asigna el valor de "*divcounter[1]"* al de "*clk_out*".

5.2.1 Testbench del ejemplo base

Los testbench son los archivos que se utilizan para simular el comportamiento de los módulos que ya han sido diseñados. Los testbench también están escritos en HDLs (Verilog en este caso) y se suelen nombrar con el mismo nombre del diseño a testear (DUT) , seguido de "_tb". Por ejemplo, el módulo "divisor" está contenido en el archivo "divisor.v" y será comprobado con el testbench "divisor_tb.v"

Un testbench es un módulo de una jerarquía superior al DUT, es decir, que el DUT estará instanciado en el testbench.

Hasta ahora, al diseñar en Verilog se ha tenido siempre en mente que todo lo diseñado debe de ser implementable en una FPGA real. Por ello, no se han utilizado numerosas opciones como esperas o comandos de escritura que en el testbench si se pueden utilizar ya que únicamente servirá para hacer las simulaciones del DUT.

Tomando el ejemplo base, se va a realizar un testbench para hacer su simulación. Quedaría un testbench de la siguiente forma:

module div3_tb();

```
//-- Registro para generar la señal de reloj

    reg clk = 0;
    wire clk_out;

    //-- Instanciar el divisor
    div3
      dut(
        .clk_in(clk),
        .clk_out(clk_out)
      );

    //-- Generador de reloj. Periodo 2 unidades
    always #1 clk = ~clk;

    //-- Proceso al inicio
    initial begin

      //-- Fichero donde almacenar los resultados
      $dumpfile("div3_tb.vcd");
      $dumpvars(0, div3_tb);
      # 30 $display("FIN de la simulación");
      $finish;

     end

endmodule
```

Al inicio se definen todos los datos que se necesitan para el testbench, en este caso, únicamente un registro, el cual se va a ir modificando para hacer las veces de señal de reloj de entrada, y un *wire* para representar la señal de salida.

Por lo general en los testbench siempre se definen como *reg* los datos de entrada y como *wire* los de salida ya que para testear el comportamiento del DUT se tienen que modificar sus señales de entrada para comprobar el correcto funcionamiento (por medio de registros) y observar las salidas como si de cables se trataran.

Lo siguiente a hacer es instanciar el DUT, nombrando primero el nombre del módulo que queremos testear, seguido del sobrenombre que se le quiere asignar (útil para cuando usas un mismo módulo para distintas cosas en un mismo proyecto, por ejemplo, RAMs), y por último, asignar las señales de entrada y salida del DUT con su correspondiente *reg/wire* del testbench de la forma que está representada en el ejemplo anterior.

Por último se añaden las variaciones que se quieren hacer a las señales de entrada para comprobar su funcionamiento:

- Primero, y prácticamente lo más importante porque se utiliza en casi todos los testbench para cualquier DUT, es establecer la señal de reloj. La forma más sencilla, limpia y útil para ello es crear un proceso *always* con un retardo de una unidad de tiempo (*#1)* para que varíe el *clk* (una unidad de tiempo a nivel alto, otra a nivel bajo… y así indefinidamente)

- Después se suele crear un proceso *initial* (que no se repite) para establecer la lista de cambios de las señales de entrada que se quieren realizar. En el ejemplo anterior no hay ninguna (pero en los siguientes ejemplos se verán más claramente) ya que únicamente se necesita la variación del *clk.* Para el proceso de simulación que se va a dar en este trabajo, como ya se explicó en el apartado 4.6, se han de incluir al inicio del proceso initial las sentencias: *$dumpfile("div3_tb.vcd") y $dumpvars(0, div3_tb)* para la creación del archivo de simulación y para el volcado de los resultados de la simulación en él respectivamente.

- Al final de la lista de cambios en las entradas del DUT dentro del proceso *initial* siempre se ha de cerrar con *$finish;* para señalar donde termina la simulación y no se quede simulando infinitamente sin obtener un archivo de salida.

5.3 Ejemplo medio

En los HDL, al igual que en el resto de los lenguajes de programación, se tiende a hacer módulos o funciones simples las cuales puedan formar parte de procesos más complejos con el objetivo de no programar varias veces lo mismo.

Siguiendo esta idea, en el siguiente ejemplo se instanciará dentro de un módulo otro de menor jerarquía: una modificación del divisor del ejemplo anterior como módulo instanciado y un contador de segundos como módulo de jerarquía superior (en la siguiente página).

Módulo M-divisor (divider.v)

```
`include "divider.vh"
module divider(input wire clk_in, output wire clk_out);

     parameter M = `F_1Hz;
     localparam N = $clog2(M);

     reg [N-1:0] divcounter = 0;

     //-- Contador módulo N
     always @(posedge clk_in)

          if (divcounter == M-1)
             divcounter <= 0;

          else
              divcounter <= divcounter + 1;

    //-- Sacar el bit más significativo por clk_out
    assign clk_out = divcounter[N-1];

endmodule
```

Al inicio de divider.v, se incluye una librería (*divider.vh)* en la cual hay distintos valores de constantes, una de ellas es *`F_1Hz* que, como se puede ver en el archivo divider.vh, su valor es 12000000 (los pulsos a 12Mhz en un segundo)

La función *$clog2(M)* obtiene el número de bits necesarios para representar el valor de M para ser guardados, en este caso en N. Con esto se puede un divisor para cualquier valor de una forma sencilla.

Librería divider.vh

```
//-- Hertzios (Hz)
`define F_2Hz   6_000_000
`define F_1Hz   12_000_000

`define F_05Hz  24_000_000
`define F_10Hz  1_200_000
```

Como se puede ver, simplemente se utiliza *'define MACRO X* para declarar la macro y asignarle un valor. Las barras bajas entre números no las tiene en cuenta el compilador, son simplemente para visualizar bien números grandes como éstos.

Módulo contador de segundos (countsec.v)

```
`include "divider.vh"

module countsec(input wire clk, output wire [3:0]
data);

     //-- Parámetro del divisor. Fijarlo a 1Hz
          parameter M = `F_1Hz;

     //-- Señal de reloj de 1Hz. Salida del divisor
     wire clk_1HZ;

     //-- Contador de segundos
     reg [3:0] counter = 0;

     //-- Instanciar el divisor
     divider #(M)
       DIV (
         .clk_in(clk),
         .clk_out(clk_1HZ)
       );

     //-- Incrementar el contador en cada flanco de
subida de la señal de 1Hz
     always @(posedge clk_1HZ)
     counter <= counter + 1;

     //-- Sacar los datos del contador hacia los leds
     assign data = counter;

endmodule
```

El módulo funciona de forma simple, recibe una señal *clk* a 12Mhz que se introduce en el divisor *divider* y de éste sale una señal *clk_1HZ.* La cual, cada vez que da un flanco de subida, aumenta *counter* en uno, y el valor que tiene este registro se asigna a *data* (salida del módulo superior) que se conectará a los leds para ver la cuenta de segundos.

Lo importante de este ejemplo es cómo se instancia un módulo en otro. Como se puede apreciar, no sólo se ha instanciado como se hizo en el anterior testbench, sino que también se ha añadido *#(M)* después del nombre del módulo inferior. Con ello lo que se hace es forzar a que el valor que tenga el parámetro M en el módulo inferior sea el que se le asigne en el módulo superior. En este caso, M, es igual en ambos módulos, pero si fuera cualquier otro (de la librería o no) prevalecería el del módulo superior.

6. Diseño final

Los ejemplos anteriores han servido de introducción al lenguaje, pero no como una comprobación de la funcionalidad y eficiencia de las herramientas descritas en el Apartado 4 . Para ello se ha realizado un proyecto bastante más grande y con una funcionalidad concreta: detección de bordes en una imagen.

El objetivo de este diseño final es, recoger la imagen que capta la cámara OV7670 (la cual se envía píxel a pixel), almacenar únicamente la información de blanco y negro de la imagen (luminancia del formato YUV) en RAMs creadas mediante HDL en la FPGA para aplicarle un filtro Sobel para la detección de bordes de dicha imagen, y enviar la imagen procesada a un ordenador para su visionado.

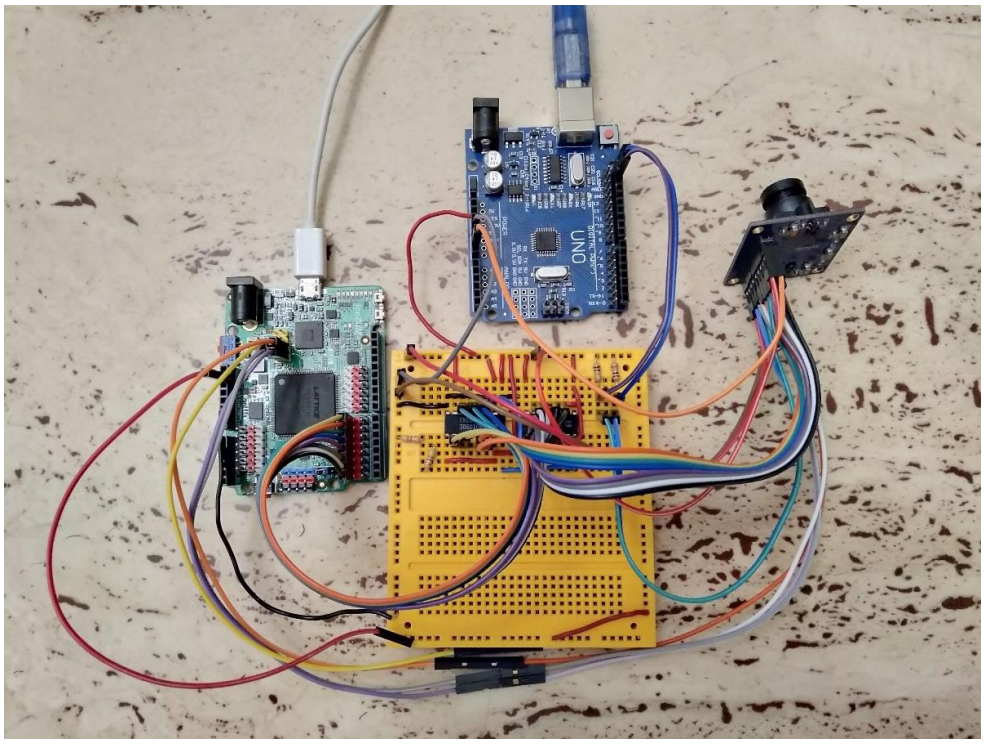


Ilustración 8: Conexionado real del desarrollo final

Como la memoria BRAM que tiene la FPGA de la tarjeta Alhambra no es capaz de almacenar una imagen completa, se procederá al procesado Sobel a medida que se va recibiendo los píxeles, es decir, en tiempo real.

Para exponer todo el proceso del diseño se ha dividido esta parte en varias explicaciones desde las consideraciones iniciales, las descripciones esquemáticas iniciales de los módulos de la FPGA, la descripción final y los problemas que se han ido encontrando realizándolo.

6.1 Consideraciones y problemas iniciales

A la hora de empezar con el diseño, se han detectado algunas dificultades iniciales, las cuales se describen a continuación junto a las soluciones que se han encontrado.

6.1.1 Filtro Sobel

El filtro Sobel se utiliza en Visión Artificial para la detección de bordes en una imagen, obteniendo como resultado una imagen en escala de grises donde las zonas más oscuras representan que no hay cambios de tonalidad significativos en la imagen original (como puede ver en la Ilustración 8) y las zonas más claras donde sí las hay (como un cuadro colgado en esa pared). Esto se suele utilizar para la identificación de objetos en una imagen para después poder sacar información más fácilmente de ella.

Ilustración 9: Ejemplo de Filtro Sobel

Técnicamente lo que se está calculando es el gradiente de intensidad de cada uno de los píxeles, por lo que a mayor diferencia de intensidad entre los píxeles izquierda/derecha y arriba/abajo, mayor será el valor del píxel resultado. El conjunto de todos los píxeles resultado da lugar a la imagen procesada por el filtro Sobel.

Para aplicar este filtro se utilizan máscaras, que no son más que la aplicación de ciertas operaciones a pixeles concretos de una imagen dependiendo de su ubicación respecto al píxel referencia. La máscara normalmente suele tener forma cuadrada, de dimensiones N_{impar} x N_{impar} para que siempre exista un píxel central de referencia. El tamaño más utilizado es el 3x3, aunque también es habitual usar un 5x5 o un 7x7, dependiendo del estilo de imagen (detalles, calidad, fondo, contraste…).

Para aplicar este filtro a una imagen con una máscara 3x3 se coloca la máscara centrada en el primer píxel en el que se puede operar (con una máscara 3x3, en el píxel [1 1]). Se realizan los cálculos pertinentes y el resultado obtenido es el gradiente de intensidad que tiene ese píxel en esa posición, por lo que se almacena en una *imagen resultado* en la misma posición que tenía el píxel referencia en cuando se aplicó la máscara. Después se centra la máscara en el siguiente pixel a la derecha (el [1 2]), y así sucesivamente hasta que termine de procesar esa fila y pase a la siguiente.

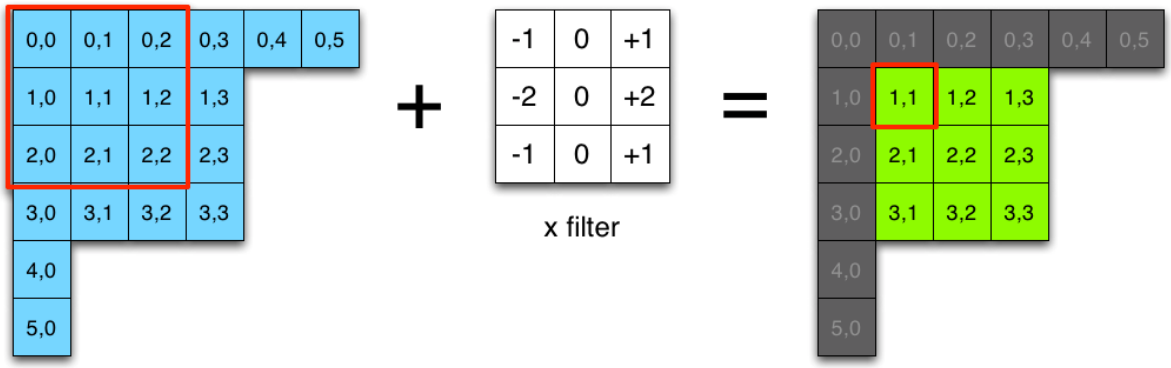


Ilustración 10: Aplicación de máscara Sobel

Obviamente, al utilizar este método, la primera y última fila, y la primera y última columna no tendrán ningún valor por lo que, o toman valores cero o se recorta la imagen dejando únicamente los píxeles que han sido procesados.

En el ejemplo "pared-cuadro", independientemente del tono de la pared, mientras sea uniforme, en la imagen resultado se obtendrán pixeles negros (cuanto más uniforme sea el tono de la pared, más cercano al valor cero del píxel resultado). Mientras que, cuando la máscara se ubique en una zona donde haya una variación (horizontal y/o vertical), se obtendrá un valor alto para el píxel resultado, tal y como se puede ver en la Ilustración 8.

En cuanto al proyecto realizado en este trabajo, se utilizará la máscara 3x3 y la imagen resultado se verá reducidas sus dimensiones al no incluir los bordes de la imagen.

En la descripción del módulo Sobel en el apartado 6.3.6 se explicará toda la operativa que conlleva.

6.1.2 Memoria RAM de la FPGA

Para este TFG se ha decidido utilizar la tarjeta Alhambra 1.1 la cual lleva implementada un chip FPGA Lattice iCE40 HX1K debido a su precio, organización de pines (muy familiar ya que es casi igual a la distribución que tiene la Arduino UNO) y que el resto de las tarjetas de evaluación que incluían chips compatibles con el proyecto IceStorm eran en formato pen drive, con el mismo chip, y menos leds y pines I/O.

Pero las características de dicha tarjeta no encajaban perfectamente con el planteamiento inicial, ya que la BRAM disponible en dicho chip es de 256x16 (64Kbits), insuficiente para guardar una imagen completa por lo que se propuso hacer el procesado en tiempo real mientras se está mandando la imagen. Para ello únicamente era necesario tener 3 filas almacenadas (para aplicar el filtro Sobel con una máscara 3x3) y una cuarta fila que se vaya rellenando mientras se procesa la primera.

Teniendo en cuenta que, mientras se está procesando con el filtro Sobel, es necesario tener disponible simultáneamente 3 pixeles consecutivos por fila se opta por almacenar cada fila en 3 RAMs distintas de forma intercalada, es decir, el primer pixel se almacena en la posición cero de la primera RAM, el segundo pixel en la posición cero de la segunda, el

tercero en la posición cero de la tercera, el cuarto en la posición uno de la primera… y así hasta completar la fila. De esta forma siempre van a estar disponible cualquier combinación de pixeles por fila de las que se requieren para la máscara.

La siguiente imagen sirve de apoyo visual para entender mejor el planteamiento:

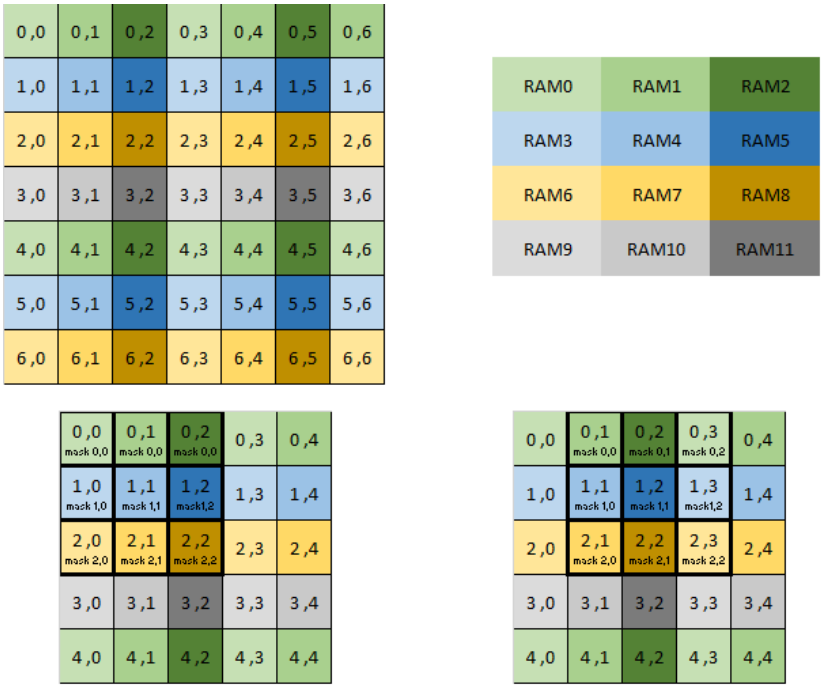


Ilustración 11: Distribución de los píxeles en las RAMs y paso de la máscara Sobel

Se valoró también incluir memorias Flash externas para almacenar la imagen para después procesarla, pero se vio en la propuesta anterior una solución mucho más rápida y eficiente de resolver el problema.

6.1.3 Inicialización de la cámara OV7670 (SCCB)

El protocolo de comunicación de la cámara para su configuración es el SCCB de dos cables (SIOC y SIOD), qué es prácticamente igual a I2C excepto que, en el caso de SCCB de dos cables sólo puede haber un maestro y un esclavo. De hecho, se pueden comunicar ambos dispositivos mediante el protocolo I2C (teniendo en cuenta la excepción anterior). [24]

En un principio se tenía pensado controlar la cámara, procesar la imagen y enviar el resultado desde la FPGA. Pero con el objeto de entender el funcionamiento de la cámara, se empezó a utilizar una tarjeta Arduino UNO con un programa compartido en varios foros FOSS, el cual configuraba la cámara en blanco y negro, el tamaño de la imagen, la recibía y la enviaba a una aplicación externa para su visualización. El uso de la Arduino UNO iba a ser únicamente en la fase de desarrollo hasta que la aplicación de procesado

fuera completamente funcional para después implementar el módulo de comunicación especial que tiene la cámara (protocolo SCCB) en Verilog y configurarla desde la FPGA.

Cuando ya estaba suficientemente avanzado el proyecto, se quiere implementar el módulo de comunicación SCCB, ya programado, pero no es posible debido a que el sintetizador requiere de más celdas lógicas de las que dispone el chip de la FPGA.

Por ello se decide mantener la tarjeta Arduino únicamente para configuración inicial de la cámara, la cual está disponible en el Anexo.

6.1.4 Adaptación de señales

Muchos de los cables de conexión entre placas, se pueden conectar directamente, pero otros han necesitado una adaptación por lo que he ha utilizado una placa de prototipado para estas conexiones auxiliares (el conexionado está descrito más adelante):

- La cámara trabaja con tensiones de I/O de 3.3 V mientras que la FPGA sólo tiene 8 pines a 3.3 V (cuando precisan, mínimo, de 12 pines conectados para llevar a cabo la aplicación), teniendo, aparte, 20 pines de 5V. Por ello, se han instalado unos adaptadores de tensión para poder recibir esa información de los pines de 3.3V en los pines de 5V de la FPGA. (Ilustraciones 13, 14 y 15)
- Se han incorporado unas resistencias de pull up (Ilustración 13) en la conexión de comunicación entre el Arduino y la cámara (comunicación SCCB).

6.1.5 Formato YUV.

El filtro Sobel requiere de una imagen de un solo canal, por lo que lógicamente, se emplean imágenes en blanco y negro. La cámara OV7670 tiene varios espacios de color disponibles, como RGB, Raw o YUV, pero únicamente uno de ellos tiene la información de "imagen en blanco y negro" sin realizar ninguna operación adicional: el formato de color YUV.

En el YUV 4:2:2, que es el formato incluido en la cámara, cada píxel de color tiene 3 canales, uno de luminancia (mayor o menor luminosidad), y dos de crominancia. El formato de envío elegido dentro de la configuración de la cámara es el YUYV (el resto son muy similares), el cual se envía según la siguiente tabla:

	1er ciclo	2º ciclo	3er ciclo	4º ciclo	5º ciclo	6º ciclo	7º ciclo	8º ciclo
Dato[7:0]	Y_0[7:0]	U_{01}[7:0]	Y_1[7:0]	V_{01}[7:0]	Y_2[7:0]	U_{23}[7:0]	Y_3[7:0]	V_{23}[7:0]

Tabla 4:Orden de bytes enviados en formato YUV 4:2:2

Como se puede ver, el valor de luminancia se envía cada dos ciclos, mientras que cada uno de crominancia se envía cada 4. Esto es porque en los espacios de color YUV cada

píxel tiene su propio valor de luminancia, pero se comparten los valores de crominancia cada par de pixeles.

	Valor de Y	Valor de U	Valor de V
1er Píxel	Y_0	U_{01}	V_{01}
2º Píxel	Y_1	U_{01}	V_{01}
3er Píxel	Y_2	U_{23}	V_{23}
4º Píxel	Y_3	U_{23}	V_{23}

Tabla 5:Valores de luminancia y crominancia por pixel

En la Imagen X se puede comparar el formato YUV 4:2:2 con el 4:4:4, que para cada píxel hay un valor de Y, U y V propio:

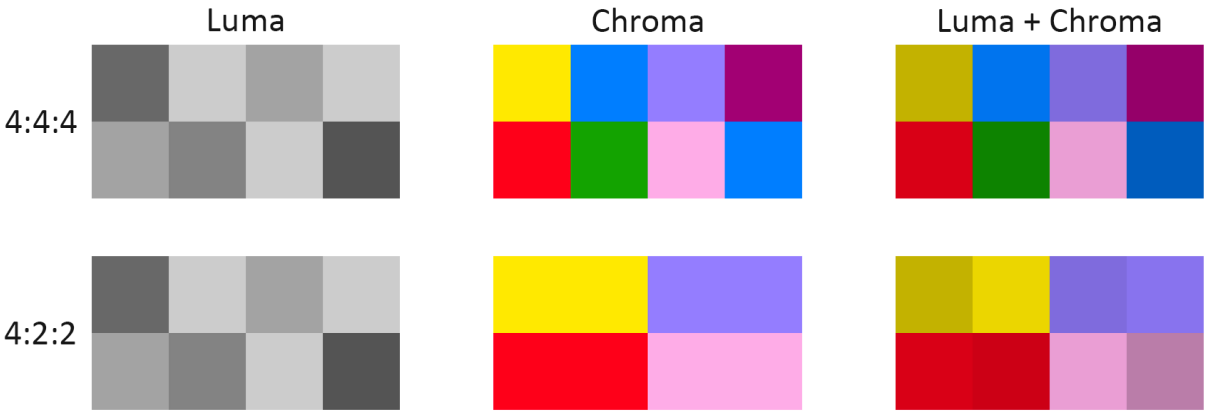


Ilustración 12: Diferencias de color entre YUV 4:4:4 y YUV 4:2:2

En este proyecto no se van a utilizar los canales de crominancia, ya que lo que se precisa es únicamente la imagen en blanco y negro, y el canal de la luminancia tiene toda la información necesaria. Por todo esto se descartará en el proyecto uno de cada dos bytes recibidos desde la cámara, y el otro (el de luminancia) se almacenará como único valor del píxel correspondiente.

6.1.6 Conexionado

Para el conexionado se ha necesitado la utilización en una placa de prototipado ya que tanto los chips CD40109B-Q1, las resistencias de pull-up, como las diversas conexiones a masa y a las distintas alimentaciones (3,3V y 5V) se necesitaban muchos nudos.

Debido a la gran cantidad de cables y que no se aprecian las conexiones, se procederá a la explicación del conexionado a través de los siguientes esquemas:

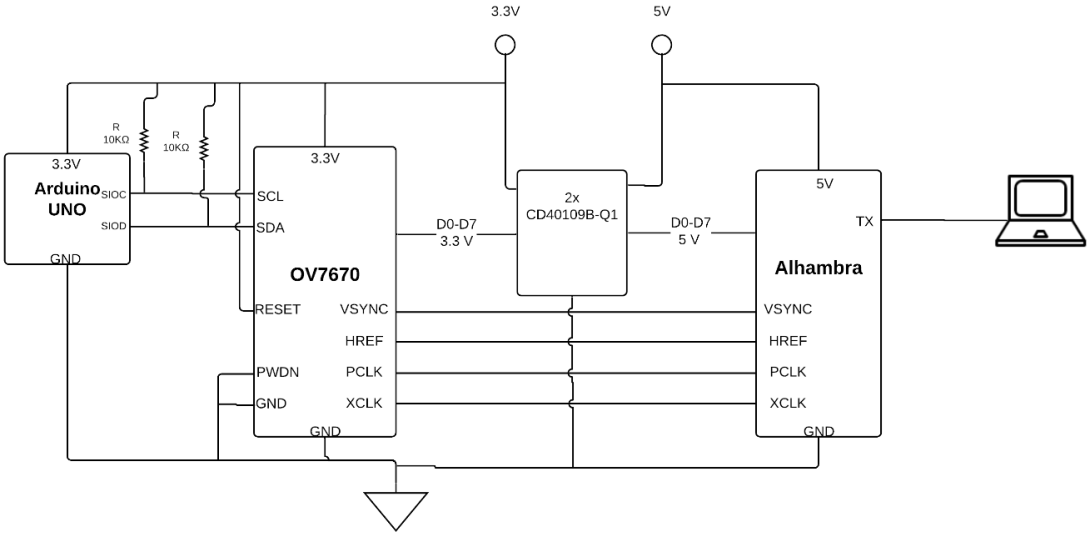


Ilustración 13: Esquema de conexionado del desarrollo completo

A continuación, se explicarán cada una de las partes importantes:

- Las alimentaciones de 3,3V y de 5V del sistema las proporcionan el Arduino UNO y la Alhambra respectivamente, y éstas, de los puertos USB del PC (tanto Arduino como Alhambra). Por lo tanto, el Arduino alimenta a 3,3V tanto a la OV7670 como a bloque de adaptación de tensión, y la Alhambra alimenta a 5V al bloque de adaptación.
- Las conexiones a masa de cada uno de los dispositivos están unidas entre ellas para que la referencia de cada señal sea la misma y así la lectura de las señales sea correcta.
- Se instalan dos resistencias de pull-up para asegurar el correcto funcionamiento del protocolo SCCB entre el Arduino y la cámara.
- Los terminales RESET y PWDN se llevan a 3,3V y a masa respectivamente. En el caso del RESET, porque es activo a nivel bajo (no se producirá ningún reset por este modo) y en el del PDWN porque así se mantiene el modo normal y no se activa el "Power Down Mode".
- Las conexiones directas entre la cámara OV7670 y la tarjeta Alhambra (VSYNC, HREF, PCLK y XCLK) han de ir conectados a los pines IO de la tarjeta Alhambra de 3,3V (tiene 8) ya que la cámara trabaja con estas tensiones.
- La señal TX para la comunicación serie entre la tarjeta Alhambra y el PC se produce a través de la conexión USB por lo que no habrá que conectar nada extra.

- Por último, el bloque de adaptación de señales con dos chips CD40109B-Q1 se encarga de convertir las señales de datos de la cámara (D0-D7) de 3,3 V a 5V para la Alhambra ya que no se disponen de suficientes pines de 3,3V y se utilizan los de 5V (dispone de 20).

  En cuanto a la configuración interna del bloque, los pines 1 y 16 de cada bloque son las referencias a nivel alto de la entrada y la salida (3,3V y 5V) y el pin 8 corresponde a masa.

  Los pines de entrada son el 3, el 6, el 10 y el 14, y los de salida el 4, el 5, el 11 y el 13 respectivamente.

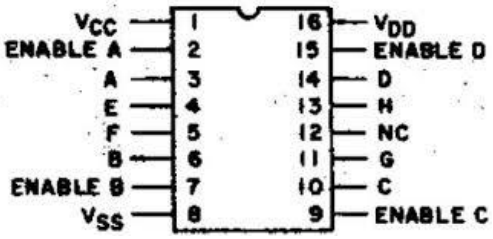


Ilustración 14: Esquema de pines del CD40109B-Q1

Los pines 2, 7, 9 y 15 son los habilitadores de cada par de pines entrada-salida de conversión. Es decir, para que funcione la conversión de los pines 3-4 ha de tener un 1 lógico en 2 (3,3V), para el par 6-5 el 7, para el par 10-11 el 9 y para el par 14-13 el 15, tal y como se muestra en la siguiente imagen:

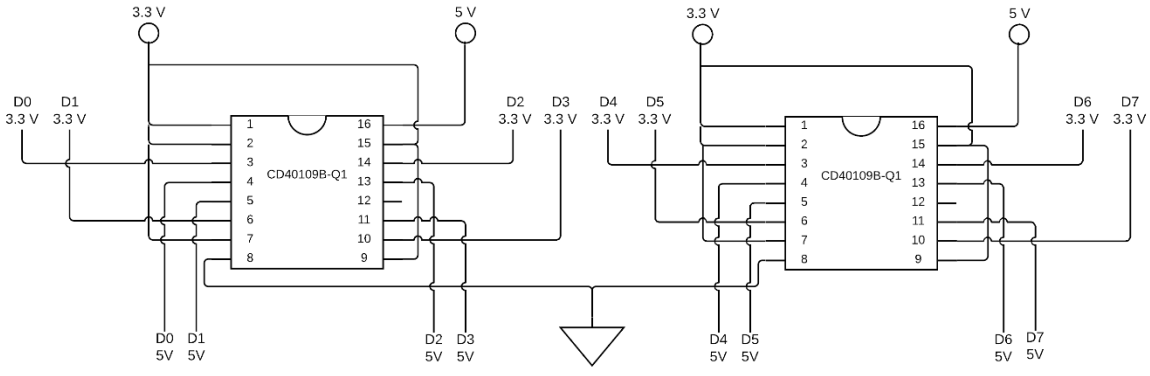


Ilustración 15:Esquema de conexionado de los CD40109B-Q1

6.2 Descripción esquemática del proyecto

El método que se ha seguido en el desarrollo del proyecto completo es el de segmentación por partes. Primero se plantearon esquemáticamente los módulos (sobre todo sus funcionalidades) y cómo iban conectados entre ellos, para después ir de uno en uno programando una versión sencilla de su funcionamiento, simularlo, probarlo en placa, aumentar su complejidad, volver comprobarlo, hasta que se obtiene la versión deseada del módulo y comprobada perfectamente. Este proceso se ha aplicado a cada módulo, de uno en uno, para finalmente instanciarlos poco a poco siguiendo el orden del procesado.

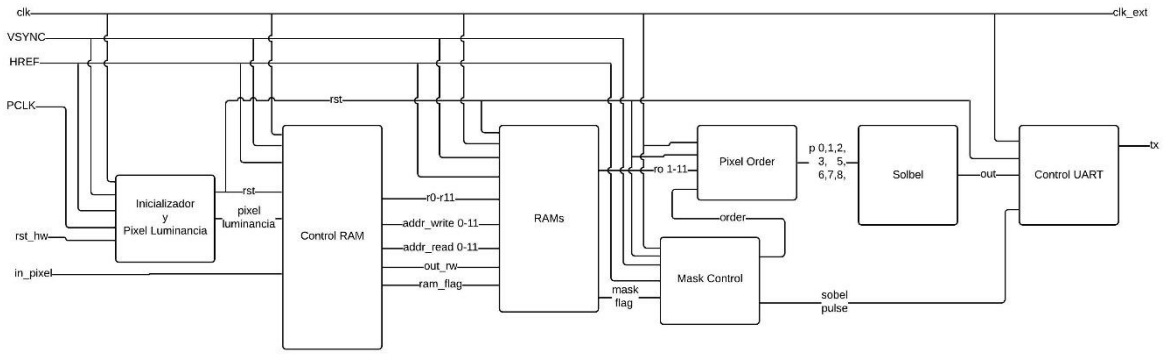


Ilustración 16: Diagrama de bloques de los módulos de la FPGA

Conceptualmente se agruparon los módulos de la siguiente manera:

Inicializador: Este módulo simplemente se encarga de, al alimentar la tarjeta, se tenga una señal de reset activa para asegurar que el resto de los módulos inician con los valores deseados. Esta señal de reset también se activa al pulsarse el botón asignado como reset. Además, se le ha añadido la funcionalidad de activar la señal la señal pixel luminancia cada dos flancos de PCLK ya que solo se quiere la información en blanco y negro.

Control RAM: Funciona como controlador tanto del guardado de los pixeles en las RAMs, siguiendo método descrito para el Filtro Sobel como de selector del valor de salida de cada una de las RAM para ser procesado correctamente por el filtro Sobel.

RAMs: En él se instancias las 12 RAMs necesarias para el proyecto. Estas RAMs están diseñadas con doble entrada de dirección, una de escritura y otra de lectura, es decir, dependiendo de si la entrada *r/w* está a cero o a uno, guarda en la posición indicada por la entrada de dirección de escritura o saca a la salida el valor guardado en la posición introducido por la entrada de dirección de lectura. Hay un total de 12 RAMs, 3 para almacenar cada fila, por lo tanto, se puede guardar información simultáneamente de 4 filas. Mientras 3 de las filas (9 RAMs) se están utilizando para el procesado del filtro Sobel 3x3, se está almacenado la siguiente fila en las RAMs restantes.

Mask Control y Pixel Order: La función de estos módulos no es más que recibir las 12 salidas de las RAMs y, dependiendo en qué momento se esté del procesado, seleccionar

8 de ellas y colocarlas a la salida de forma adecuada para llevar a cabo el filtro Sobel de forma correcta.

Sobel: Tal como su nombre indica, este módulo se encarga de realizar el filtro Sobel 3x3, tomando a la entrada los 8 pixeles ordenados dependiendo de su posición dentro de la máscara (el píxel central de la máscara no se opera) y a la salida saca el píxel resultado.

Control UART: Este último módulo se encarga de recibir los píxeles resultado y enviarlos a través del protocolo UART al terminal del ordenador. Los píxeles se envían de forma binaria uno detrás de otro, ordenados, pero sin distinciones de ubicación entre ellos. Por ello, y por ser un procesamiento en tiempo real, cada vez que se empiece a enviar una nueva imagen se enviará una cabecera claramente identificable para diferenciar una imagen de otra.

Con la implementación de estos módulos, el desarrollo software ya estaría terminado, únicamente faltaría traducir esos archivos binarios recibidos por terminal a una imagen, lo cual se puede hacer fácilmente con Matlab preparando un script teniendo en cuenta el tamaño final de la imagen resultado.

6.3 Descripción funcional del proyecto.

Una vez definidas cada una de las partes en las que se quiere dividir el proyecto, comienza su implementación. Lo que en principio eran ideas algo superficiales, se van concretando en módulos reales, mucho más complejos y robustos de lo que en un principio se planteaba.

Una vez realizada la síntesis, los recursos empleados de la FPGA son los siguientes:

```
After packing:
IOs             15 / 96
GBs              0 / 8
    GB_IOs       0 / 8
LCs            813 / 1280
    DFF          339
    CARRY        97
    CARRY, DFF   5
    DFF PASS     267

 CARRY PASS 13
BRAMs            8 / 16
WARMBOOTs        0 / 1
PLLs             0 / 1
```

```
After placement:
PIOs          15 / 96
PLBs         140 / 160
BRAMs          8 / 16
```

Al final se logra un diseño que cumple perfectamente con su función (código incluido en el Anexo), cuyo funcionamiento se describe con detalle a continuación:

6.3.1 Funcionamiento de la cámara y sus señales

La cámara, una vez programada con el Arduino, adquiere la siguiente configuración: imagen en YUV 4:2:2, unos 400kHz de frecuencia de píxel (PCLK) y dimensiones de imagen de 320x240, entre otros.

Se elige el formato YUV 4:2:2 porque, de sus 3 componentes (luminancia, y dos de crominancia) la primera representa a la perfección una foto en blanco y negro, lo viene perfecto para nuestro filtro Sobel.

Los 400kHz se obtienen al dividir la frecuencia a la que se alimenta la cámara, los 12 MHz sacados directamente de la FPGA, entre 15 (necesario para el correcto envío de la imagen y evitar el efecto "scrambled") y entre 2 (que al desactivar el PLL digital de la cámara demedia la frecuencia de alimentación). Todo ello se selecciona en el mismo registro, el CLKRC (0x11), con una escritura de un 0x0E en él.

Se elije el tamaño de 320x240, o QCIF, para no exigir demasiado a nivel de BRAM a la FPGA, siendo a la vez una resolución lo suficientemente buena para comprobar los resultados del filtro.

En cuanto a la lógica que sigue al envío de los píxeles es muy sencilla:

- Cada vez que la cámara va a enviar una nueva imagen, la señal de VSYNC pasa de 0 a 1 para avisar de un nuevo envío y pasa a 0 para iniciar su envío (Ilustración 17). Para evitar envíos en falso, si no se ha recibido un flanco de subida de VSYNC la aplicación se queda esperándola sin hacer nada. (Ilustración 19)

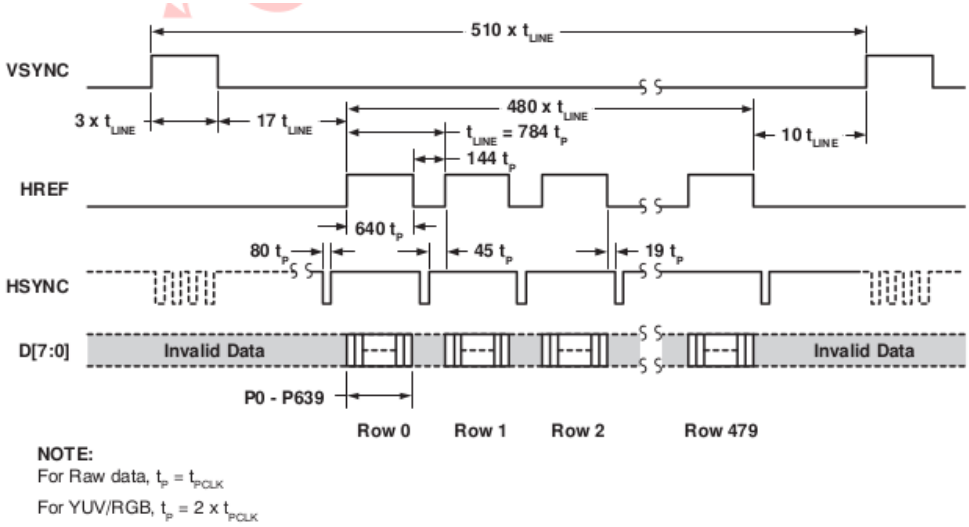


Ilustración 17: Diagrama temporal VSYNC-HREF de la cámara OV7670

- La señal HREF se encarga de avisar de que se están enviando píxeles y de que se ha cambiado a transmitir la fila siguiente de la imagen (Ilustración 17). Una vez se ha dado el pulso de VSYNC, HREF pasa de 0 a 1 para indicar que se inicia el envío de pixeles de la primera fila y pasa de 1 a 0 para avisar de que se ha terminado el envío de la fila. Después vuelve a pasar de 0 a 1 para mandar la segunda fila, pasa de 1 a 0 para indicar el fin de fila… y así sucesivamente hasta que envía la imagen completa.

- Por último, la señal PCLK que con cada uno de sus flancos de subida (con HREF a nivel alto) indica que el dato que hay en los pines D0-D7 es un píxel correcto (Ilustración 17). PCLK está configurado para que únicamente esté funcionando cuando se ha dado el pulso de VSYNC y HREF está en alto, es decir, cuando realmente se está enviando información. En la configuración por defecto siempre está funcionando, como un CLK continuo, pero por seguridad y robustez se ha elegido esta configuración, aunque el diseño está preparado para la configuración por defecto. En simulación (Ilustración 19) aparece continuo ya que era mucho más sencillo de programar, pero en la configuración de la cámara en Arduino está así como se puede ver en el archivo de Arduino en el Anexo.

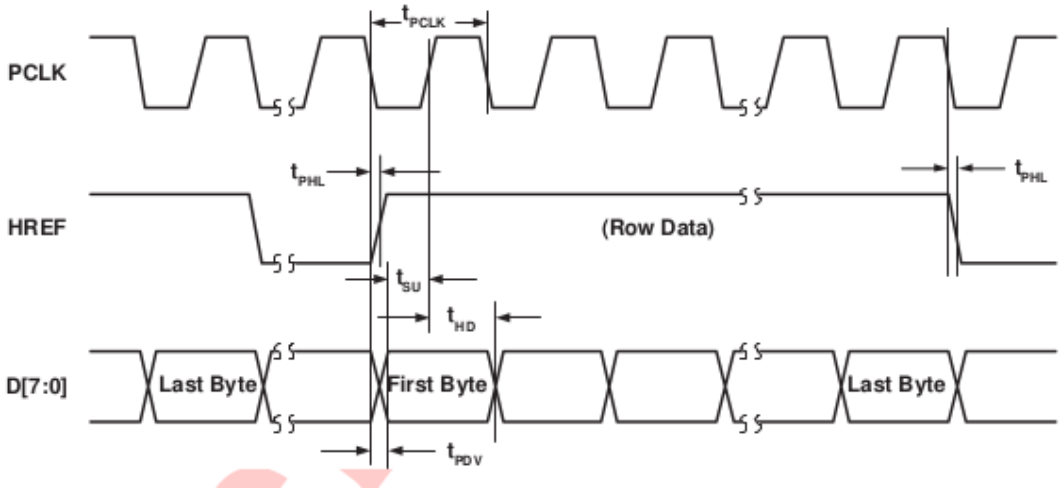


Ilustración 18: Diagrama temporal HREF-PCLK de la cámara OV7670

En cuanto al envío de los píxeles en YUV 4:2:2, como se aclara en el apartado 6.1.5, se tendrán en cuenta únicamente los píxeles de luminancia (el primero, el tercero… de cada fila) ignorando los valores de crominancia, por lo que, además se reduce la frecuencia efectiva de PCLK a la mitad al reducir la información necesaria del envío de la cámara.

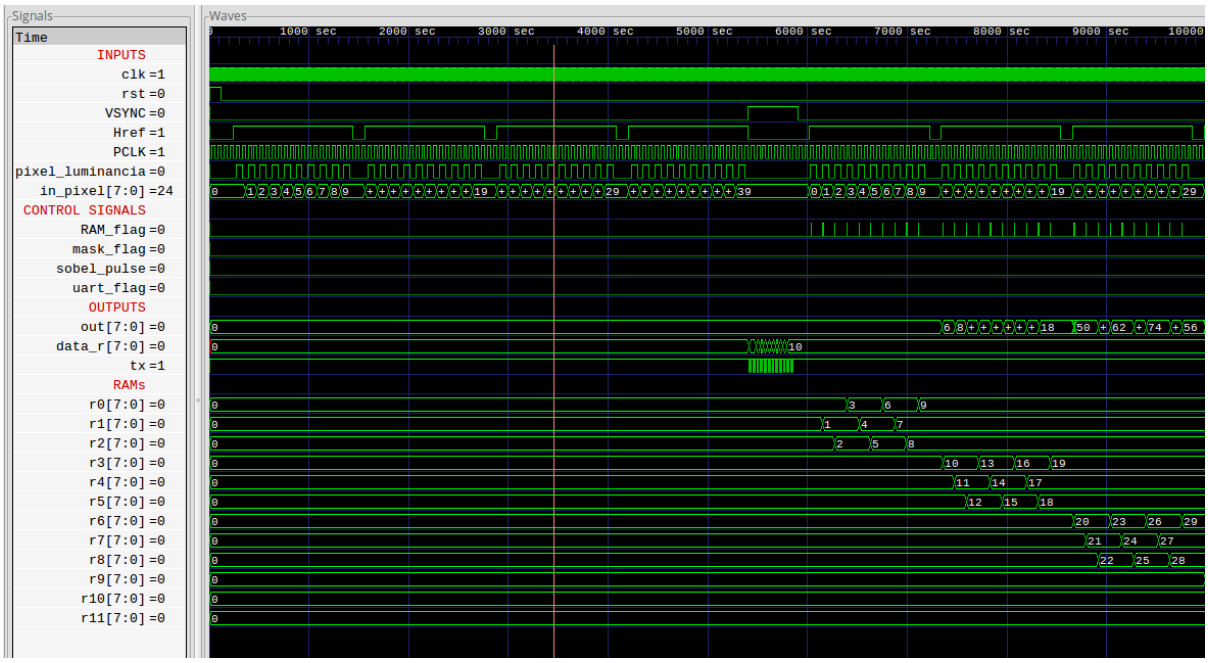


Ilustración 19: Inicio simulación funcional del desarrollo final

6.3.2 Inicialización y pixeles de luminancia

En el módulo de más alto nivel, en el cual están el resto de los módulos instanciados, se han incluido dos procesos simples, los cuales no se veía necesario crear un módulo concreto para ellos, que son el proceso del reset y el proceso luminancia/crominancia.

El proceso de reset, simplemente, inicia el valor de un registro *rst* (que está conectado todos los módulos) a 1 para que se tomen los valores por defecto en todo el proyecto en el primer ciclo de reloj para que al siguiente ya tome el valor que tiene el botón asignado al reset. De esta forma se reinician registros y variables al iniciar y cuando se pulse dicho botón.

El proceso luminancia/crominancia se encarga básicamente de activar la recepción por parte de *Control RAM* únicamente de los bytes de luminancia. Para ello da un pulso en el registro *pixel_luminancia* y no lo da, alternativamente, por cada flanco de subida de la señal externa PCLK. Para asegurar que siempre se cogen los bytes correctos, se inicializan sus registros en cada reset, nivel alto de VSYNC y en cada nivel bajo de HREF

En la Ilustración 19 se puede apreciar que la frecuencia de *PCLK* es el doble que la de *pixel_luminancia*.

6.3.3 Control RAM

El funcionamiento del módulo *Control RAM* se divide en tres procesos:

Proceso de selección r/w de las RAMs.

Como se va a llevar a cabo escrituras en unas RAMs mientras se lee en las otras, conviene tener un bus de 12 bits de *r/w* para tener perfectamente controladas cuales son las RAMs que deben de ser escritas y cuales leídas. De ello se encarga este proceso, cambiando el valor que tiene cada módulo de RAM de *r/w* (Ilustración 20) dependiendo de la fila que esté enviando la cámara (atendiendo a cada flanco de subida de la señal *Href).*

El proceso interpreta la fila que está siendo enviada a través del registro *fila* el cual varía de 0 a 3 (Ilustración 20). Si a la hora del flanco de subida de *Href* , *fila=0,* se seleccionan las RAMs 0, 1 y 2 en modo escritura y el resto en modo lectura y *fila* pasa a 1. Cuando se da el flanco y *fila=1* las RAMs 3, 4 y 5 son las únicas en modo escritura y fila pasa a 2, y así sucesivamente.

En él se establece un estado inicial (*fila=0 y rw=000_000_000_000*) de "todo escritura" en cada flanco de bajada de la señal *VSYNC* o cada *reset (Ilustración 20).*

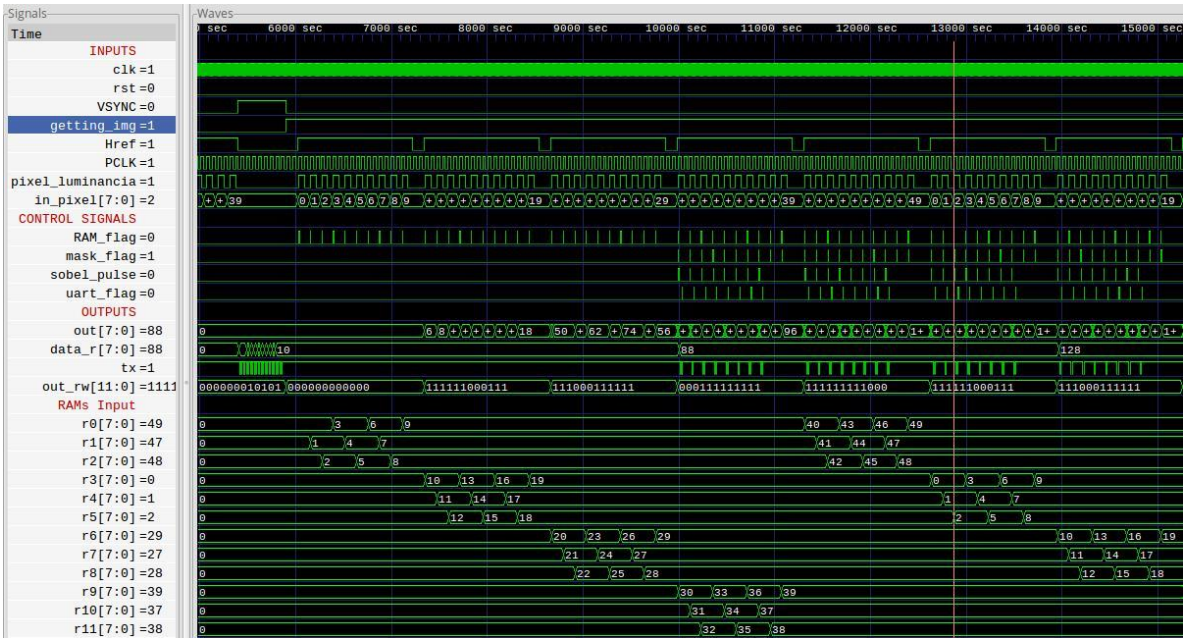


Ilustración 20: Simulación funcional del inicio de recepción de una imagen (pulso VSYNC)

Además, a este proceso se le ha añadido una funcionalidad más, la de bloquear cualquier escritura o procesado si no se ha recibido la señal de nueva imagen (pulso en *VSYNC*) con el registro *getting_img (Ilustración 20),* el cual está a cero tras *reset* o nivel alto de *VSYNC* y a uno si se da un flanco de bajada en *VSYNC*. Este registro ha de estar en nivel alto para que las funciones de escritura y procesado estén operativas.

Proceso de almacenamiento.

Una vez establecidos los pixeles que se deben guardar (luminancia) y en qué RAMs, solo queda guardarlos. Para ello se atiende al valor del registro *fila* para seleccionar en qué grupo de RAMs se van a guardar los datos (de 0 a 2, de 3 a 5, de 6 a 8 o de 9 a 11, como se expuso en el apartado anterior). Todos los cambios descritos en este apartado son visibles en la Ilustración 21.

Después, y en cada flanco de subida de *pixel_luminancia,* se almacena el valor de las entradas *in_pixel* siguiendo la siguiente secuencia:

Primero se almacenan los 3 primeros pixeles en las posiciones cero de las RAM_0, RAM_1 y RAM_2 respectivamente, después en la posición uno, luego en la dos… así hasta el final de la fila (como se pueden ver en los registros $addr\_write_x$ de la Ilustración 21), con el flanco de bajada de *Href* (observar los cambios de las señal r0, r1, r2 por cada flanco de subida de *pixel_luminancia* y que corresponden a los valores de *in_pixel*) .

Como medidas de robustez del proceso se comprueba que cada vez que se da un flanco de subida de pixel_*luminancia* estén tanto la señal *Href* como *getting_img* a nivel alto. Y para asegurar la correcta inicialización de los registros, cada vez que se da un *reset* o un nivel bajo de *Href*, se hagan cero todos los registros usados en el proceso (*addr*, *RAM_flag y state).*

Cada vez que se da un flanco de subida acepado de *pixel_luminancia,* se hace que la señal *RAM_flag* también de otro flanco de subida para la activación en cadena de procesos de otros módulos.

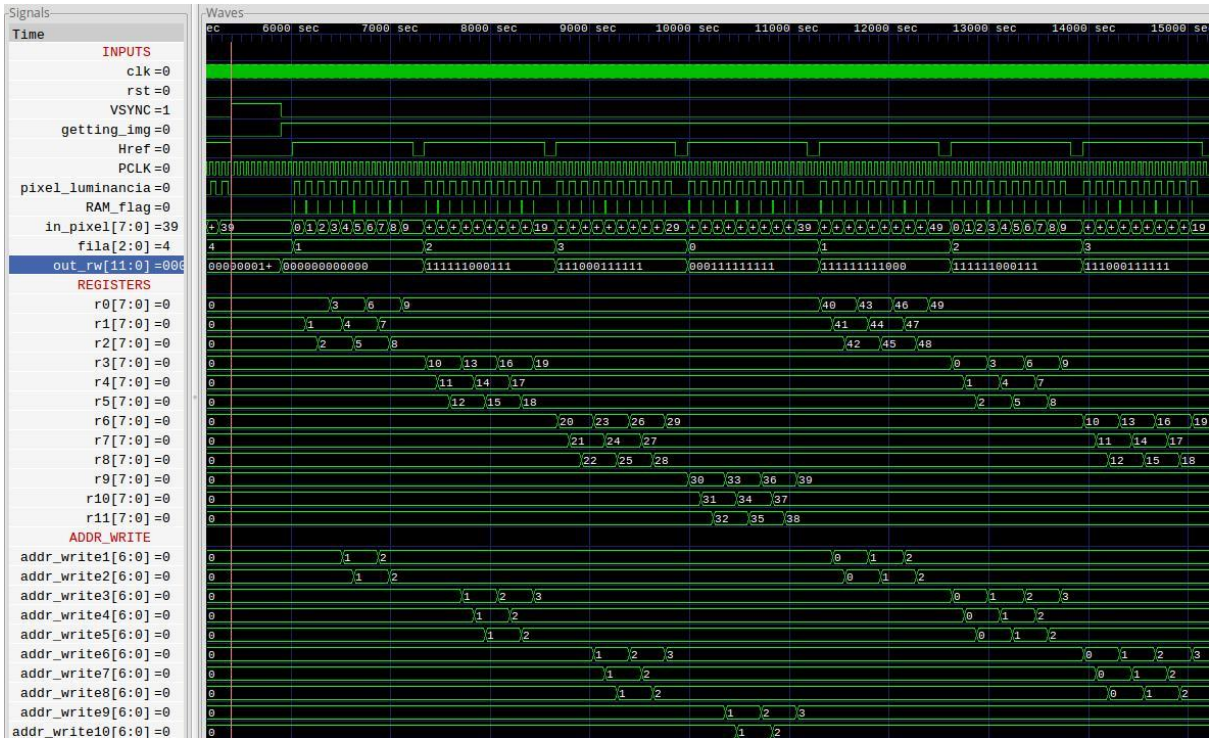


Ilustración 21: Simulación funcional de la escritura en RAM de la imagen recibida

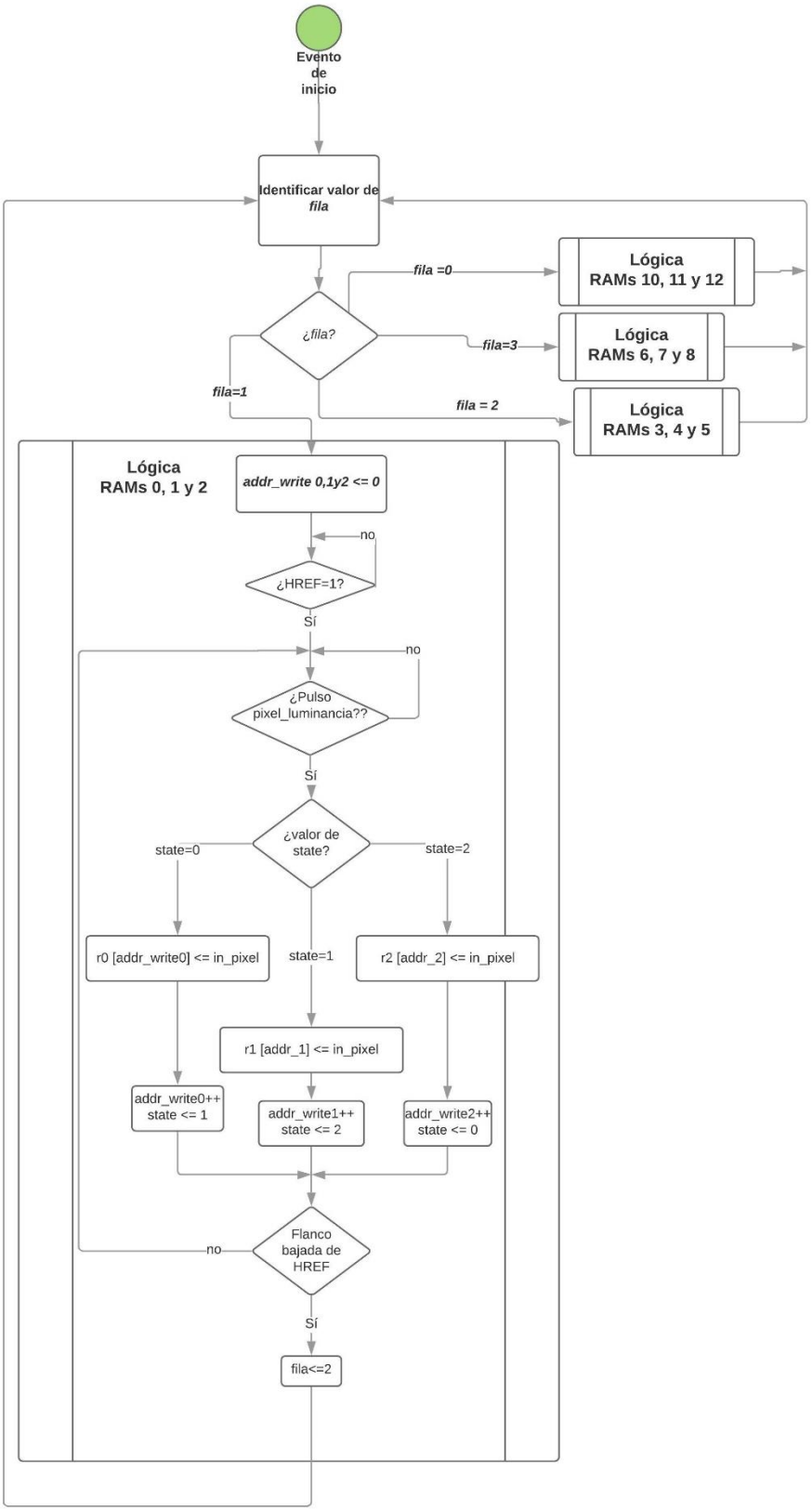


Ilustración 22: Flujograma de escritura en RAM

Proceso de selección de direcciones de memoria para el filtro Sobel.

Ésta es, quizá, la parte más abstracta de todo el proyecto: el control de direcciones de lectura de las RAMs. Por ello, la mejor forma de exponer el funcionamiento de este proceso es mediante un ejemplo, siendo este visible en la Ilustración 23.

Se suponen ya almacenadas en las memorias RAM 0, 1 y 2 la primera fila, en las memorias 3, 4 y 5 la segunda fila y en las memorias 6, 7 y 8 la tercera fila. Ahora, y a la misma frecuencia a la que se va recibiendo la cuarta fila (*pixel_luminancia)* comienza el primer procesado Sobel de la imagen.

Primer flanco de *pixel_luminancia*: Se van a seleccionar en las RAMs (de la 0 a la 8) los tres primero píxeles de cada fila, los cuales están en las posiciones de memoria cero de cada una de las RAMs. Se manda un flanco de subida en la señal *RAM_flag* para avisar al siguiente módulo que las salidas de las RAMs están listas para el procesado.

Segundo flanco de *pixel_luminancia*: Ahora la máscara se ha desplazado un píxel a la derecha en la imagen por lo que los píxeles que se necesitan para la máscara son el segundo, el tercero y el cuarto de cada fila, los cuales están almacenados en la posición cero de la RAM_1, la posición cero de la RAM_2, y la posición uno de la RAM_0 (la distribución se conserva en las otras RAMs). Se vuelve a mandar el flanco de subida de *RAM_flag*.

Tercer flanco de *pixel_luminancia*: Como en el caso anterior, se desplaza la máscara un píxel a la derecha por lo que se necesitan los pixeles tercero, cuarto y quinto de cada fila, (posición cero RAM_2, posición uno RAM_0 y posición uno de la RAM_1). Se vuelve a mandar el flanco de subida de *RAM_flag*.

El proceso descrito en este apartado se puede ver en la Ilustración 23 a partir del cuarto pulso de HREF. Siendo el registro *rw* (*out_rw*) el encargado de indicar la lectura (1) o escritura (0) de cada una de las RAMs, siendo el bit menos significativo el correspondiente a la RAM_0 y el más significativo el de la RAM_{11}. Los ro_x son las salidas de cada una de las RAMs y $addr\_read_x$ los registros que indican las direcciones para las RAMs que deben ser leidas (prestar atención a que las que se están escribiendo en ese momento, no varía su valor de $addr\_read_x$ porque no se está utilizando)

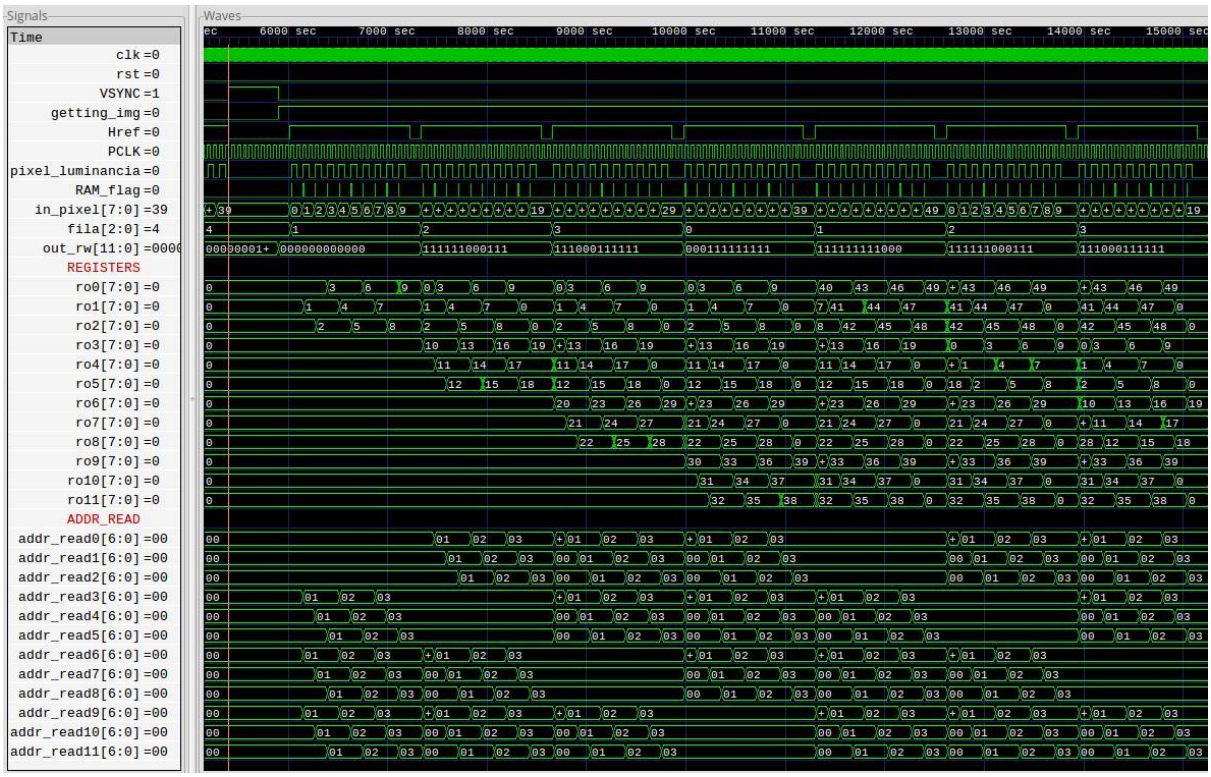


Ilustración 23: Simulación funcional de la lectura en RAM de la imagen recibida

Como se puede ver en la Ilustración 23, lo que en realidad se está haciendo es, partiendo de la posición 0 de todas las memorias RAM, ir sumando uno a la dirección de memoria de las RAMs colocadas en la parte izquierda de la máscara en la siguiente iteración.

Cuando termine de recibirse la cuarta fila se empezará a recibir la quinta fila. Como ya se ha terminado el procesamiento de las tres primeras filas y la primera fila no se va a volver a utilizar en ningún procesado, la RAM_0 pasará a almacenar la quinta fila.

Llegados a este punto, mientras se está recibiendo la quinta fila se va a ir procesando la segunda, la tercera y la cuarta que están localizadas en las RAMs de la 3 a la 11. Y así continuamente.

Como es obvio, hasta que no se hayan almacenado las 3 primeras filas de la imagen no se puede empezar el procesado, por ello es innecesario que este proceso esté modificando las direcciones de las RAMs que estén en modo lectura atendiendo a la lógica del procesado. Aun así, se ha decidido mantener así por motivos de simplicidad, limpieza y evitar el exceso de variables de control innecesarias.

En el siguiente flujograma se puede seguir la lógica descrita en este apartado:

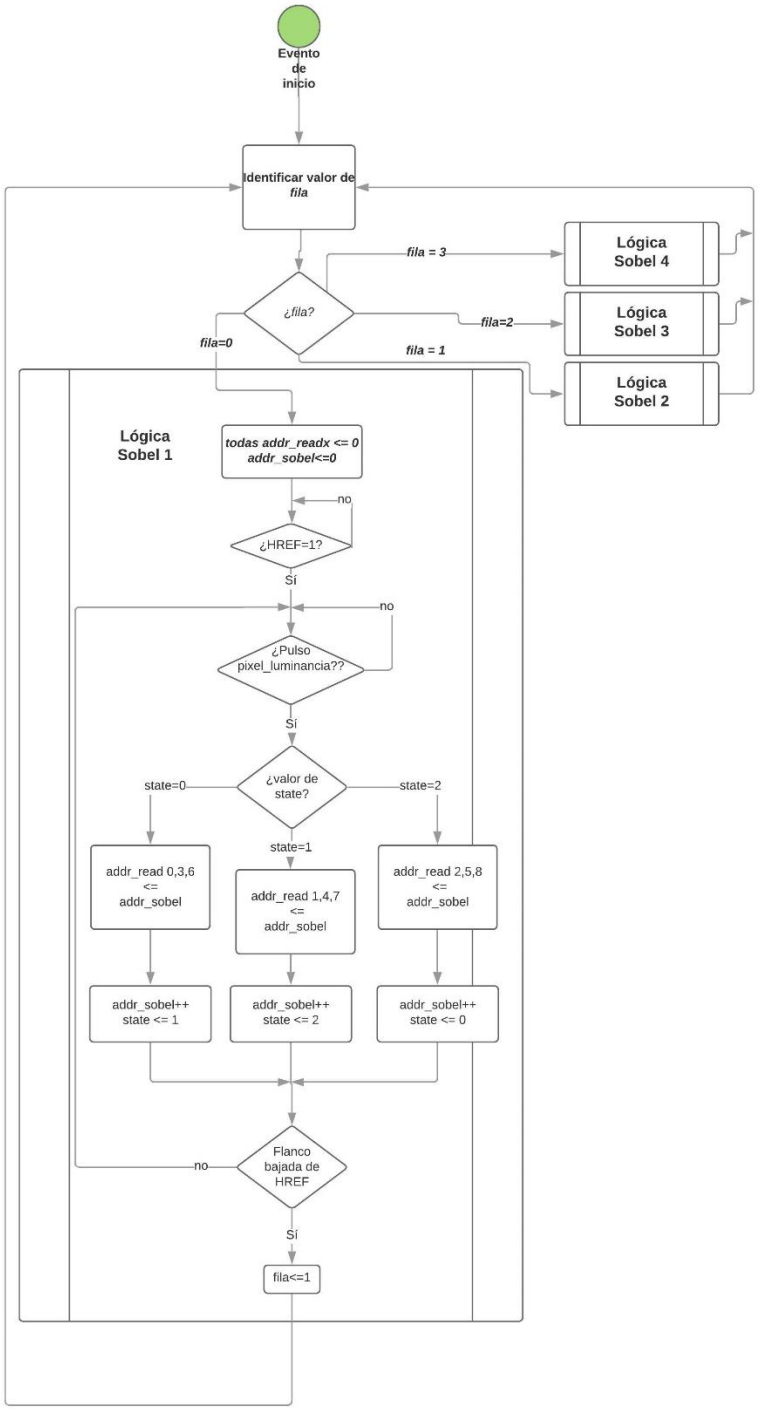


Ilustración 24: Flujograma de lectura en RAM

6.3.4 RAMs

Este módulo únicamente tiene las instanciaciones de los 12 módulos de RAMs de doble dirección (lectura y escritura) para facilitar su gestión y su testeo y un pequeño proceso que se encarga de replicar la señal RAM_*flag* en la señal *mask_flag*, pero si aún no se han completado la recepción de 3 de las filas (3 flancos de bajada de *Href*), se mantiene a 0. Esta señal *mask_flag* activa la siguiente fase de módulos.

6.3.5 Mask Control y Pixel Order (Antiguo Selector)

En el módulo de Control RAM se han seleccionado qué RAMs están guardando una nueva fila y en cuáles de sus posiciones se almacena cada píxel, y también qué RAMs están siendo utilizadas en el procesado y cuáles son las posiciones de esas memorias que se necesitan. Pero el trabajo del procesado no se queda ahí, también hay que colocar esos 9 valores de las RAMs de escritura de forma correcta en la máscara Sobel.

Siguiendo un poco con el ejemplo anterior, cuando se dio el primer flanco para el procesado de las 3 primeras filas (todas las RAMs en la posición cero) la colocación dentro de la máscara era la siguiente:

RAM_0	RAM_1	RAM_2
RAM_3	X*	RAM_5
RAM_6	RAM_7	RAM_8

Ilustración 25: Posicionamiento de las RAMs dentro de la máscara en el primer paso

*El valor del píxel de referencia no influye en el filtro Sobel por lo que no se incluye

Para cuando se dio el segundo flanco para el procesado, la colocación de las RAMs era distinta debido al desplazamiento a la derecha (las RAMs 0, 3 y 6 tenían como posición de memoria uno en vez de cero):

RAM_1	RAM_2	RAM_0
RAM_4	X*	RAM_3
RAM_7	RAM_8	RAM_6

Ilustración 26: Posicionamiento de las RAMs dentro de la máscara en el segundo paso

Debido a estos cambios de colocación dentro de la máscara se requiere una forma de coger las 12 salidas de las RAMs para elegir cuales son las necesarias para el procesado y en qué posiciones. Teniendo en cuenta las 12 RAMs, el tamaño de máscara, y bajo qué lógica se mueve, hay un total de 12 posiciones distintas en las que se puede colocar las RAMs y que tengan sentido a la hora de realizar el filtro.

Para realizar esta administración se han diseñado los módulos: *Mask Control* y *Píxel Order*, y toda la operativa que se explica a continuación puede ser comprendida mejor gracias a la Ilustración 25.

Mask Control: es el módulo encargado en decir cuál de las 12 posibles opciones de colocación de las RAMs de lectura dentro de la máscara Sobel es la adecuada. Para ello utilizan dos registros: *pos_mask_v* (con valores comprendidos entre 0 y 3) y *pos_mask_h* (con valores entre 0 y 2). El primero indica cuantos desplazamientos verticales ha realizado la máscara o pasadas del filtro completadas (aumenta en uno con un flanco de

bajada en *Href*) y el segundo indica cuantos desplazamientos horizontales ha realizado (aumenta en uno por cada *mask_flag)* siendo ambas visibles en la Ilustración 25.

Con las distintas combinaciones de estos dos registros obtenemos las 12 posibles colocaciones de las RAMs, y da lugar, en otro proceso, a la señal *order*, cuyos valores varían de 0 a 11 y se conecta directamente con el otro módulo, *Pixel Order*.

El tamaño de la fila es de 320 píxeles, el tamaño de la máscara es 3x3 por lo tanto el número de iteraciones distintas que hace la máscara en una pasada es de 318, dos menos que las de valores enviados por la cámara. Por ello, y al utilizar la frecuencia del *pixel_luminancia*, tanto para los procesos de almacenamiento como los de procesado de la imagen, se ha incluido un contador en este módulo para que cuando llegue a esos 318 (8 en simulación) deje de enviar pulsos *sobel_flag* y, por lo tanto, deje de procesar las filas.

Pixel order: este módulo tiene a la entrada cada una de las salidas de las RAMs y la señal *order* , y como salida los 8 valores necesarios y bien colocados en cada una de las posiciones p_x para realizar el procesado Sobel. Para hacerlo, simplemente, dependiendo del valor de la señal *order*, asigna a cada una de las posiciones de la máscara el valor que le corresponde (consultar el archivo pixel_order.v del Anexo para ver las posiciones).

Inicialmente se tenía planteado un sólo módulo como para realizar la función descrita en *Selector* pero por problemas, que se describirán en los apartados de problemas y de conclusiones, se tuvo que partir en dos.

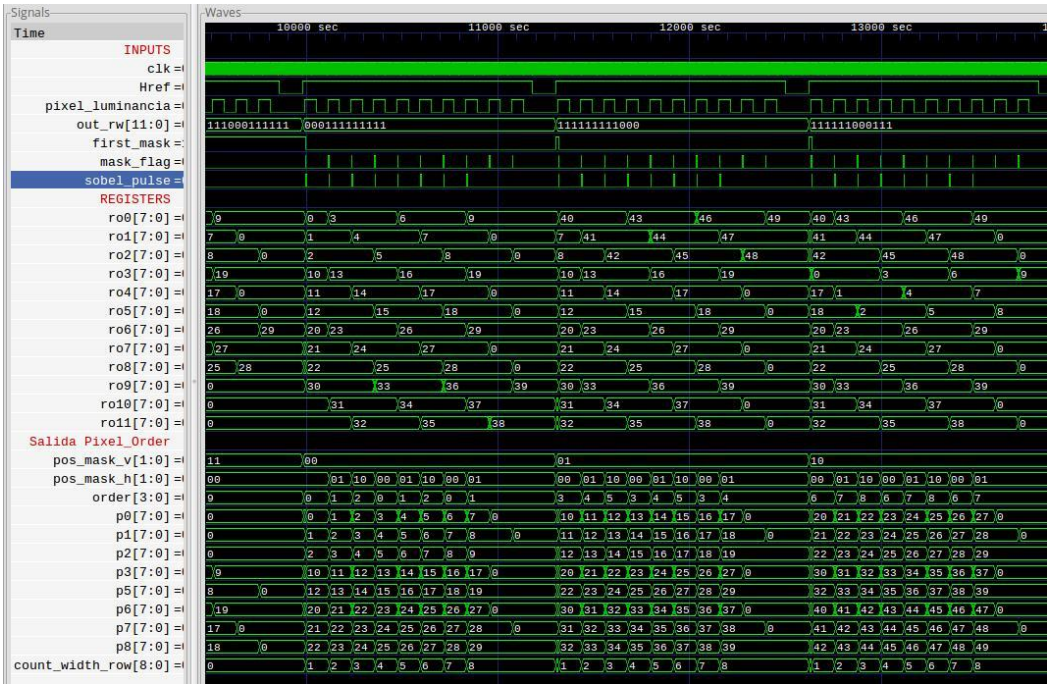


Ilustración 27: Orden de los elementos de las RAMs dentro de la máscara a cada instante.

6.3.6 Sobel

En el apartado 6.1.1 se ha descrito el cualitativamente el funcionamiento y la utilidad del filtro Sobel, ahora, se va a explicar su funcionamiento cuantitativamente.

Aunque hasta ahora se ha hablado de "la máscara del filtro Sobel" en realidad hay dos: máscara para la detección de bordes verticales, y la de detección de bordes horizontales. Su forma es la misma pero girada:

-1	0	1
-2	0	2
-1	0	1

Ilustración 28: Gx, Máscara de detección bordes verticales

-1	-2	-1
0	0	0
1	2	1

Ilustración 29: Gy, Máscara de detección de bordes horizontales

Para aplicar cada máscara a los pixeles de entrada simplemente hay que multiplicar elemento a elemento los valores. Por ejemplo, para la máscara de bordes verticales (Ilustración 26) se multiplica (-1) por el valor del pixel en la posición (0,0) o p_0, por (-2) el valor del pixel en la posición (1,0) o p_1… etc y luego sumar todos los resultados, obteniendo el gradiente horizontal (bordes verticales) y el gradiente vertical (bordes horizontales) correspondiente al píxel referencia: Gx y Gy respectivamente (cuyas máscaras correspondientes se representan en las Ilustraciones 26 y 27)

p0	p1	p2
p3	0	p5
p6	p7	p8

Ilustración 30: Nomenclatura de posiciones dentro de las máscaras

Una vez aplicadas ambas máscaras sobre un mismo conjunto de píxeles y por lo tanto obtenidos los gradientes verticales y horizontales del píxel, se deberían combinar para hallar la magnitud del gradiente calculando la raíz cuadrada de la suma de los cuadrados de ambos gradientes. Como al estar en lenguaje HDL operaciones como raíces cuadradas no son posibles, se recurre a la suma de los valores absolutos de los módulos de los gradientes.

A pesar de que el tamaño de los píxeles es de 8 bits, los tamaños tanto de *gx* como de *gy* se han establecido en 11 bits y en formato *signed* ya que pueden ser negativos y con las multiplicaciones tener valores mucho mayores a 255.

Una vez realizada la suma de los valores absolutos, se limita el resultado, es decir, si es mayor a los 255 que se pueden tener con 8 bits, a la salida del módulo habrá un 255, y si es menor, el valor que sea.

Ilustración 31: Simulación funcional filtro Sobel con Gx, Gy y valor final

En la Ilustración 29 se puede ver como para cada conjunto de valores de $p_{0\text{-}8}$ en cada uno de los procesamientos. Los valores de Gx y Gy no cambian debido a que los valores de simulación estan seleccionados de tal forma para que sea sencillo observar si el filtro está funcionando bien (En el Anexo existe una hoja de cálculo en la cual está aplicado el filtro sobel en la imagen ficticia que se recibe por simulación para conocer los valores resultado de Gx y Gy)

El registro out es la suma de los valores absolutos de Gx y Gy, y además se observa en el cuarto pulso de HREF que el valor de out es mas alto que los anteriores. Eso se debe a que los píxeles de $p_{6\text{-}8}$ son valores de decenas cero mientras que los otros son de decenas 3 y 4, es decir, ha encontrado un pequeño borde horizontal (valor de Gy mayor).

En este módulo todas las operaciones se han realizado con asignaciones, por lo que la variación de cualquier entrada hará variar prácticamente al instante la salida.

6.3.7 UART Control

Este es el último módulo utilizado en el proyecto, cuya función es enviar por puerto serie a la terminal del ordenador los píxeles ya procesados.

Las velocidades de transmisión más comunes en este protocolo no eran válidas porque si se quiere hacer un procesado y envío en tiempo real, la velocidad de envío ha de ser bastante superior a la frecuencia a la que se envían los píxeles nuevos. Por ello se configuró el terminal del puerto serie (utilizando el programa GTKTerm en Ubuntu) a una velocidad de 12.000.000 baudios, alimentados directamente del reloj de la FPGA. El resto de parámetro de configuración sí que siguieron una configuración más estándar: sin paridad, envíos de 8 bits, con un bit de stop y sin control de flujo.

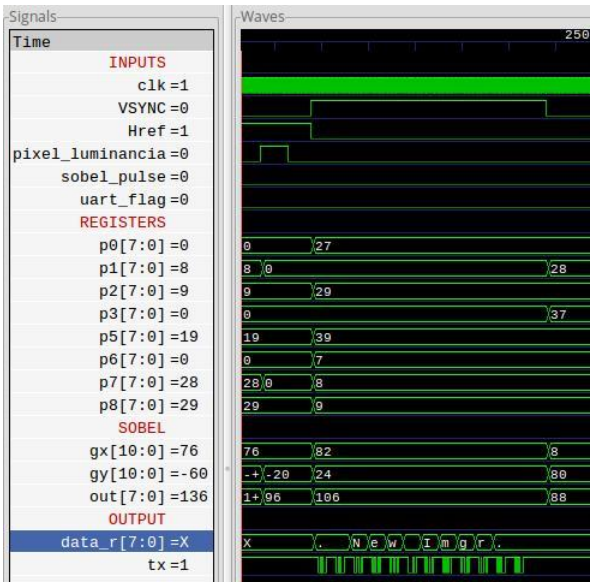


Ilustración 32:Simulacion Funcional envío de cabecera por UART

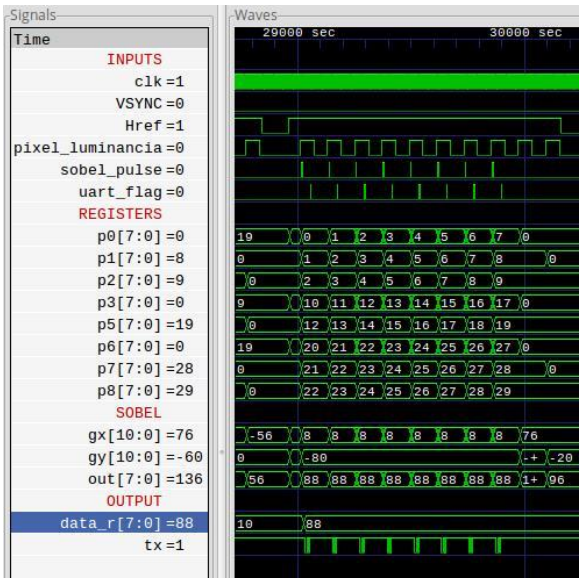


Ilustración 33: Simulación funcional envío de imagen procesada por UART

En cuanto a desarrollo del módulo, se instanció un módulo de envío UART con algunas modificaciones para adecuarlo al proyecto, como el envío de una cadena de caracteres cada flanco de subida de *VSYNC* para diferenciar en el archivo recibido al PC una imagen de otra y la transmisión de bytes por pulso (Ilustración 30).

En la Ilustración 31 se ve que cada vez que se recibe un flanco de la señal *sobel_pulse*, se inicia el envío del valor que hay en *out* que es copiado al registro *data_r*, el cual es el valor exacto que se está enviando. Se ha considerado importante el visionado de *data_r* porque entre que se da la señal de envío hasta que se adquiere el valor internamente dentro del módulo de la UART, hay un par de ciclos de reloj que se tiene que mantener *out* fijo, y así se comprueba que ambos coinciden.

El envío serie se puede ver en la señal *tx* y el final de envío se marca con el flanco de subida de *uart_flag* (posteriormente utilizado para medir la frecuencia máxima de operación)

Con todo esto se da fin a la aplicación de la FPGA. Como resumen, la aplicación espera a recibir un flanco de subida de VSYNC, va almacenando filas de la imagen y cuando empieza a recibir la cuarta, aplica filtro sobel a las 3 filas anteriores, siguiendo así hasta la finalización del envío por parte de la cámara.

Como se puede ver en la simulación de la Ilustración 32, la distancia entre la detección del flanco de subida de PCLK hasta la finalización del envío serie (con el registro *uart_reg)* es de 22 ciclos de reloj, por lo tanto, la mayor frecuencia a la que se podría recibir los píxeles de una imagen sería, aproximadamente, unos 545kHz (a 12MHz de frecuencia de reloj maestro)

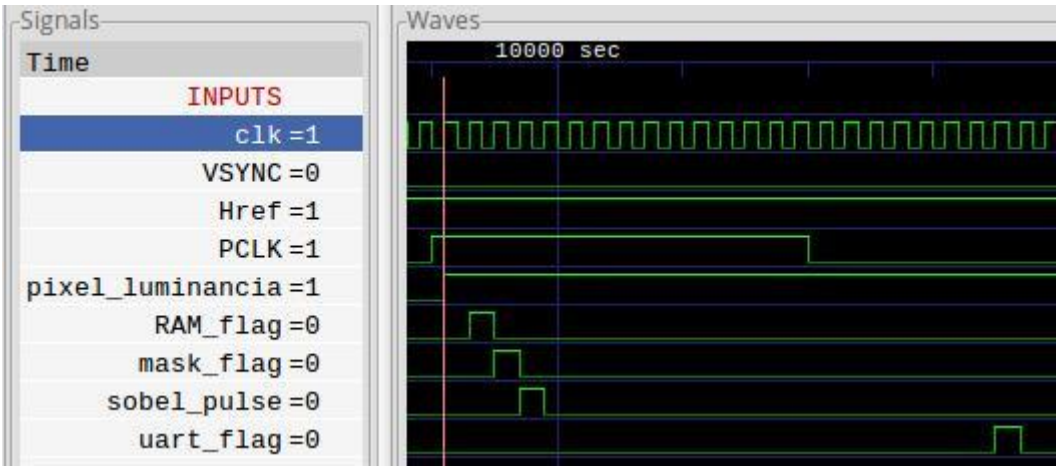


Ilustración 34: Simulación funcional de los ciclos de reloj necesarios para procesar y enviar un pixel

6.3.8 Recepción por terminal de imágenes en binario y traducción.

GtkTerm:

Una vez GtkTerm recibe perfectamente la información con la configuración planteada en el punto anterior, simplemente hay que almacenar en un archivo binario la información que envía la FPGA.

La comunicación se debe configurar como se precisa en el apartado anterior: 12.000.000 baudios, sin paridad, envíos de 8 bits, con un bit de stop y sin control de flujo.

Después se selecciona la opción *"Log > To File..."* se crea el archivo binario en el cual se quiere guardar la información recibida, para después iniciar el envío. Cuando ya se crea que suficiente información ha sido recibida, se ha de detener el guardado de información, accediendo a *"Log > Stop"*.

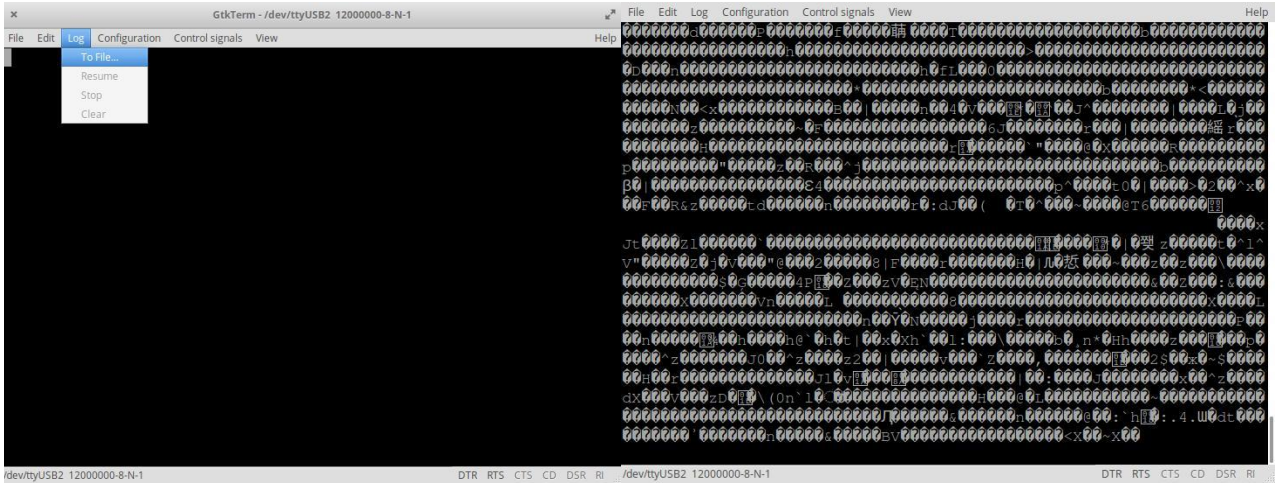


Ilustración 35 : GtkTerm para crear el archivo de recepción y la recepción binaria de la imagen

Bless Hex Editor:

Una vez obtenido el archivo binario con varias imágenes en su interior se ha recurrir a un editor de archivos binarios, en este proyecto, al realizarse en un entorno Linux se ha utilizado “Bless HEX Editor” para crear un archivo binario por cada imagen.

Para ello se ha de dividir toda la información que haya entre dos cabeceras (definidas en el módulo Control UART) y pegarlas en un archivo nuevo.

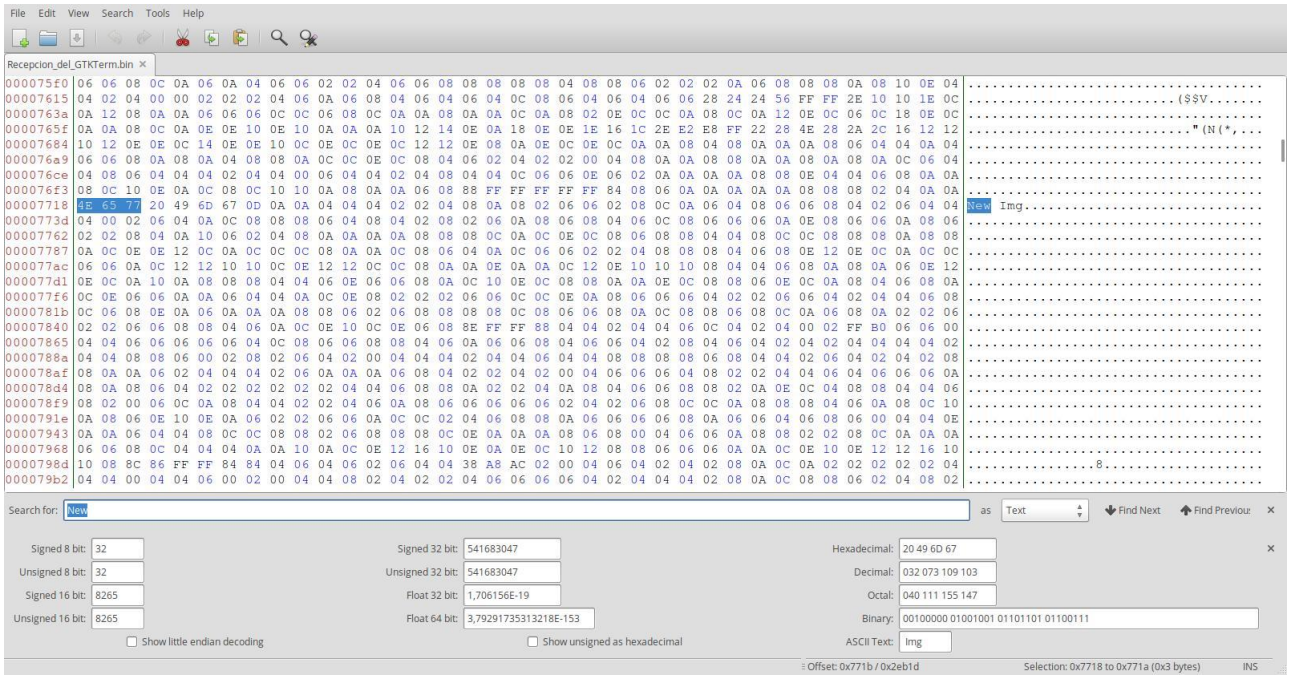


Ilustración 36:Detección de cabeceras con el buscador de Bless Hex Editor

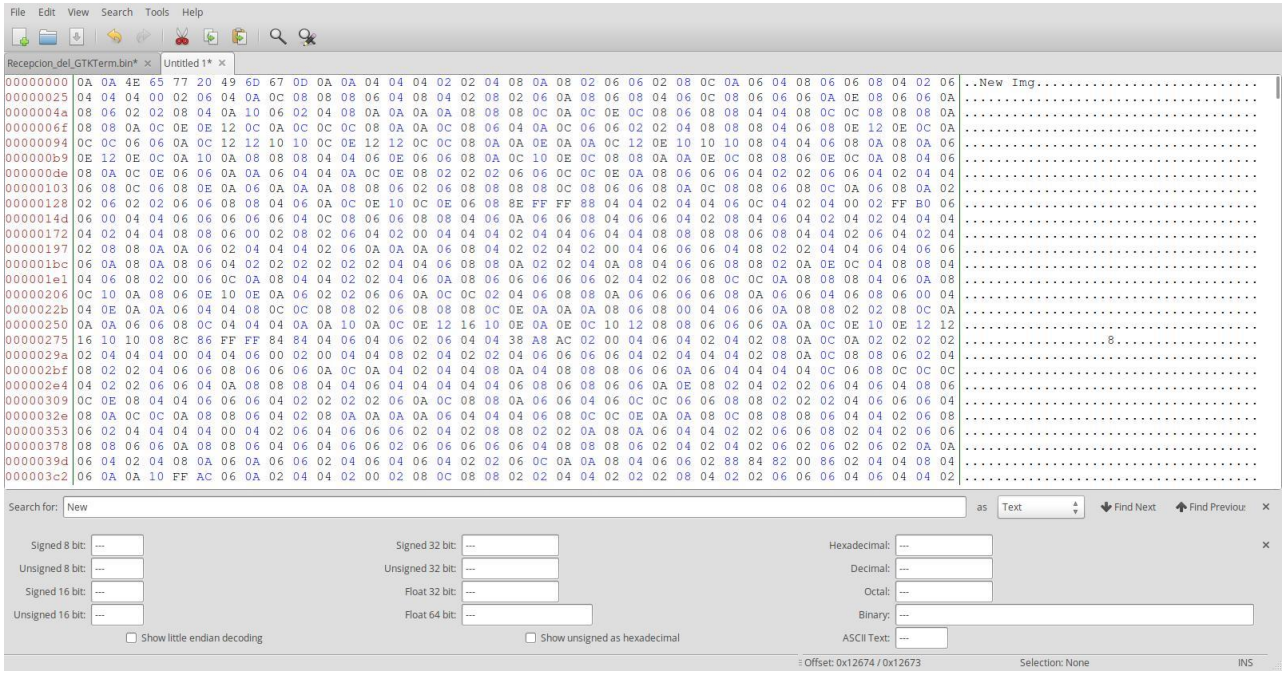


Ilustración 37: Creación de un nuevo archivo con solo una imagen en su interior

Una vez ya se tienen esos archivos binarios de las imágenes por separado, se deben de quitar la cabecera ya que el script en Matlab está preparado para procesar únicamente la información. En este caso la cabecera, en hexadecimal es: 0A 0A 4E 65 77 20 49 6D 67 0D 0A 0A

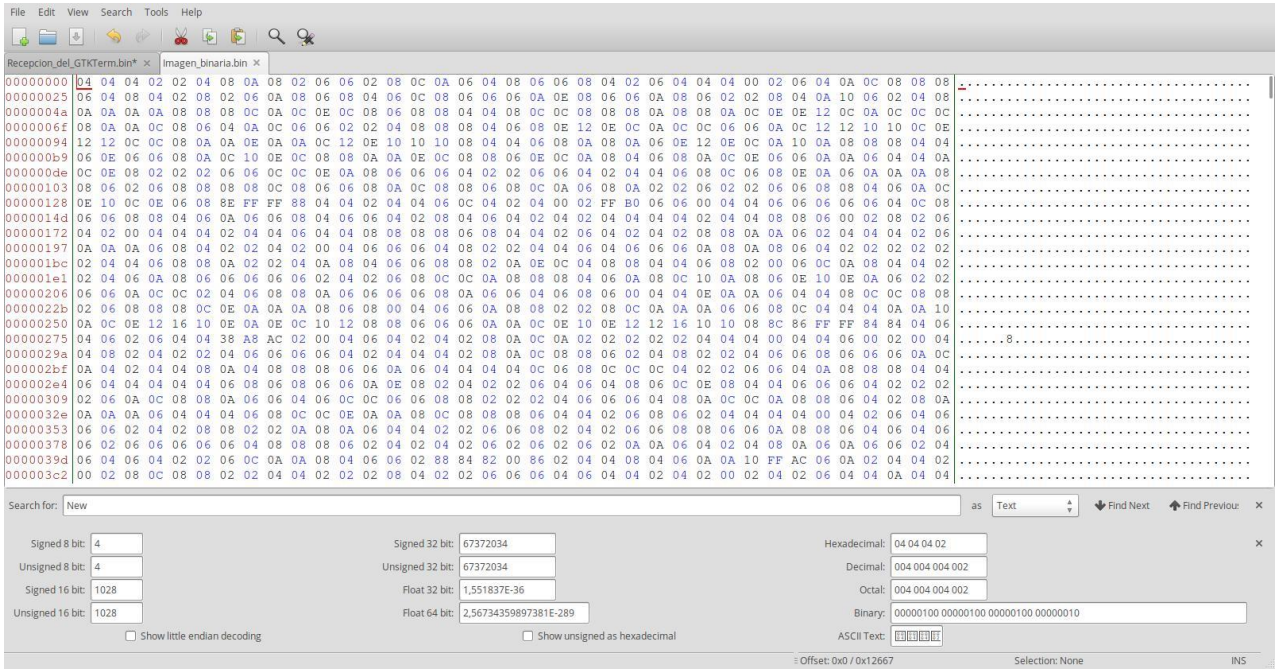
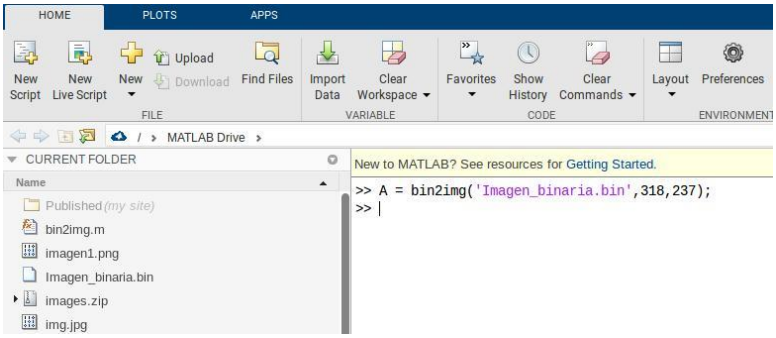


Ilustración 38: Eliminación de la cabecera de la imagen

Matlab:

Para la traducción del archivo binario a imagen en blanco y negro se ha desarrollado un script para Matlab con el cual, llamando a la función descrita en él y dándole como entrada el nombre del archivo a traducir, el ancho y el alto de la imagen, se obtiene como salida la imagen en blanco y negro (archivo img.jpg)



No hay que olvidar dos cosas respecto a las dimensiones. Ya se ha hablado de que el ancho de la imagen es de 318 en vez de 320 ya que no todos los píxeles pueden ser centrales o de referencia y por la misma razón las filas no van a ser 240 (que son las que

se envían) sino 237, 3 menos, por las dos filas que no tienen píxeles referencia (la primera y la última) y porque cuando la cámara termina de enviar la última fila y terminar el envío (no está funcionando PCLK) el sistema no realiza la última pasada con la máscara que es capaz de hacer (la de las tres últimas filas.

6.3.9 Resultado

Se han capturado imágenes de 3 objetos distintos para comprobar la eficacia del proyecto: un objeto esférico sobre fondo (Ilustraciones 39-40), un objeto plano sobre fondo (Ilustraciones 41-42) y una imagen plana con detalles (Ilustraciones 43-44). Como se puede observar se obtienen unos resultados bastante satisfactorios.

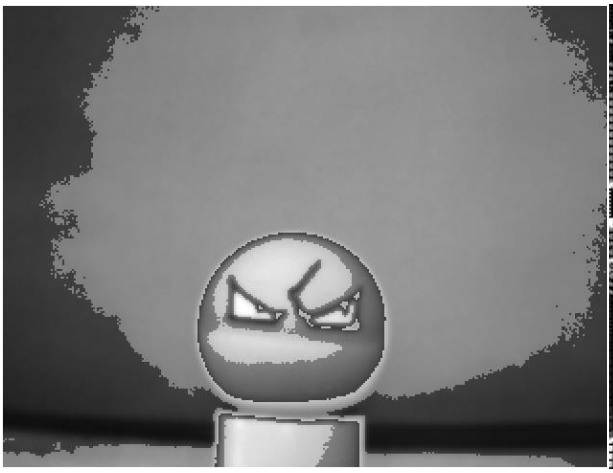


Ilustración 39: Captura 1 de la cámara previo filtro Sobel

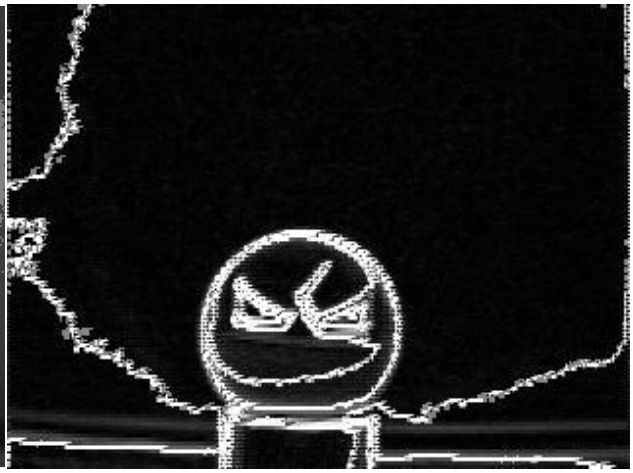


Ilustración 40: Procesado Sobel de la Captura 1

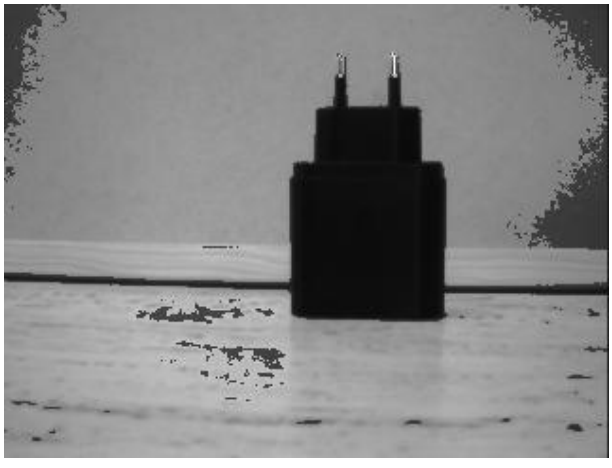


Ilustración 41: Captura 2 de la cámara previo filtro Sobel

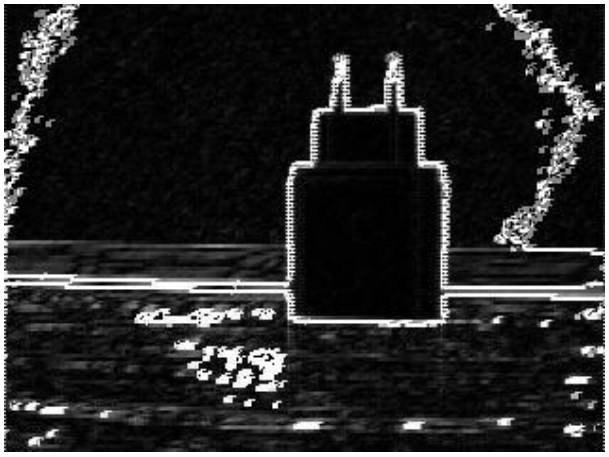


Ilustración 42: Procesado Sobel de la Captura 2

Ilustración 43: Captura 3 de la cámara previo filtro Sobel

Ilustración 44: Procesado Sobel de la Captura 3

Las zonas que tienen una intensidad similar, al aplicar el filtro Sobel, pasan a ser oscuras. Y en las zonas en las que se produce un cambio (por ejemplo, en el brillo del fondo o en el cambio del fondo a la esfera en primer plano) aparecen en tonos claros.

Se aprecian algunos pixeles negros en las zonas de píxeles de borde. Esto, en visión artificial, se le llama ruido "sal y pimienta" que son píxeles negros que deberían ser blancos y viceversa. Para solucionar este tipo de errores se suelen aplicar las técnicas de "erode" y "dilate", las cuales, de forma muy simple, convierten los píxeles que tengan en cercanía $\sqrt{2}$ un píxel negro, en negro (erode) y los que estén a $\sqrt{2}$ de un píxel blanco, en blanco (dilate).

Aplicando primero el "dilate" y después el "erode", desaparecerían los puntos negros dentro de los bordes.

Otra imperfección que tiene la imagen resultado son los distintos tonos de negro y de blanco (en menor medida). Esto se solucionaría aplicando umbrales superior e inferior, para que los tonos oscuros sean totalmente negros y los más claros, totalmente blanco.

Aplicar estas soluciones en este ejemplo, tal cual está pensado, es imposible ya que para aplicarlas hay que tener la imagen procesada entera almacenada en memoria y, de hecho, no se almacena ningún píxel resultado debido a se hace el envío en tiempo real (no hay memoria RAM suficiente ni para la imagen original).

6.4 Problemas y modificaciones durante la evolución del proyecto

Como ya se ha adelantado en algún apartado anterior, el proyecto ha sufrido de algunas modificaciones desde su concepción inicial hasta la finalización del mismo, como la permanencia del Arduino para la configuración de la cámara.

El objetivo de este apartado no es otro que el de reflejar las distintas formas de las que se enfocó el desarrollo de los módulos y por qué se descartaron, sirviendo muchas de ellas como prueba e introducción para el siguiente punto, el de conclusiones de las herramientas.

6.4.1 Control Maestro

La idea inicial era que tanto el control de la escritura/lectura en RAM, como del procesado y el envío recayera sobre un módulo de control maestro, que fuera el encargado de decir cuándo y cómo se hace cada tarea. De esta forma se podría incluso separar en procesos completamente paralelos la recepción de información del envío del resultado. De hecho, la primera versión completa simulada con éxito fue de esta forma. Con esta versión no existía el problema de la falta de procesamiento de las últimas tres filas ya que era el control maestro el que daba la orden de iniciar el procesado de filas.

El problema surgió al implementarlo en placa, que los resultados no tenían nada que ver con lo que se obtenía en las simulaciones, las cuales funcionaban perfectamente. Los testbench utilizados estaban diseñados con gran fidelidad respecto a la secuencia de señales que explican en el datasheet de la cámara OV7670 e incluso forzando frecuencias mucho más altas de las que la cámara da en realidad.

Después de examinar en placa cada una de las partes del diseño, se encuentra la zona de código en la que no funciona igual la implementación real a la simulada (la sincronización entre el módulo de Control RAM y el de Control Maestro) pero después de muchas modificaciones y pruebas en placa de esta última parte se decide prescindir del Control Maestro y enfocarlo de una forma más lineal e independiente.

6.4.2 Mask Control y Píxel Order por Selector

En el proyecto en el que estaba incluido el *Control Maestro*, con el fin de clarificar el diseño se dividió lo que originalmente era el módulo selector en los que se han descrito antes como *Mask Control y Pixel Order*. Cuando el proyecto anterior falló, y se volvió a rehacer, se optó por volver a juntar los dos módulos solamente en uno.

La programación de ambos es prácticamente idéntica, únicamente que mientras que en *Mask Control se* seleccionaba el valor de la señal *order* con los registros *pos_mask_h y pos_mask_v* en el segundo proceso, en *Selector*, en ese mismo proceso, se hacían las asignaciones de las RAMs a las entradas al módulo *Sobel* directamente (ahorrando bastantes registros y señales).

Cuando se realiza el desarrollo de este proyecto con el módulo *Selector*, en la última síntesis, con todos los módulos incorporados y con las simulaciones funcionales perfectas, la compilación actúa de forma extraña y da error por mucha falta de espacio 1760/1280 LCs , cuando hasta justo antes de incluir la versión final de *Selector* la síntesis no ocupaba más de 600 LCs.

Simplemente con sustituir, en este proyecto, el módulo *selector* por los de *Mask Control y Pixel Order*, las síntesis pasaron a ocupar 700-800 LCs y funcionando perfectamente en placa.

7. Conclusiones.

Las herramientas son funcionales, sin demasiados problemas para instalarlas (los cuales los propios desarrolladores suelen haber resuelto ya esos problemas en foros o en sus propias páginas) y con bastantes ejemplos sencillos e implementables en placa de lenguaje Verilog para empezar a meterte en el mundo desde prácticamente cero.

Pero, a este nivel de desarrollo, no son competencia clara de las herramientas profesionales tanto en opciones como en sencillez como optimización. Se hecha mucho de menos la simulación temporal, aunque a estos niveles de complejidad, frecuencias y tamaño de proyecto, no debería suponer una gran diferencia, pero cuando compila, simula funcionalmente, sintetiza y enruta, pero no funciona en placa… Ese apoyo podría ser la solución.

El compilador no funciona del todo bien, el error que más se ha dado, y ha pasado desapercibido tanto por el compilador como por el sintetizador, es el de al instanciar un componente y olvidar conectar algún cable o registro, y que compile y sintetice sin mostrarte ningún aviso.

Aún me queda la duda de si los fallos en las distintas versiones del proyecto completo han sido error mío al programar algo y no ser reconocido por el compilador/sintetizador o que he dado con varios de los “callejones sin salida” que aún no han sido parcheados.

En definitiva, para pequeños proyectos funcionan suficientemente bien y si, en simulación funciona perfectamente y falla en la ejecución en placa, prepárate para darle la vuelta a tu diseño para hacerlo funcionar.

8. Futuros Trabajos

Como resultado de la investigación y desarrollo descritos en este trabajo, se considera que las herramientas usadas son aptas para profundizar en el tema y avanzar.

En un futuro se pueden realizar varias tareas:

- Sustituir la herramienta *Arachne-pnr* por *nextpnr*, ya que la fecha a la que se tuvo consciencia de ella fue en la fase de escritura de la memoria con toda la investigación y el desarrollo realizado.
- Mantener bajo vigilancia la página SymbiFlow, ya que el acceso a FPGAs con mayor potencia como pueden ser las Serie 7 mediante herramientas FOSS puede ser muy interesante.
- Investigar la versión II de la tarjeta Alhambra [15], compararla con la v1.1 y posibles proyectos con ella teniendo en cuenta la diferencia de características entre los dos modelos.
- Basándose en los diseños de las tarjetas Alhambra, se podría hacer una tarjeta incluyendo una FPGA más potente y ciertos complementos para hacerla más completa en comparación a los modelos de Alhambra y diseñar una aplicación utilizando las herramientas del "Proyecto IceStorm"
- Posibles proyectos software con hardware más potente que la tarjeta Alhambra 1.1 podrían ser: la integración de todo el proceso en la FPGA (prescindiendo del Arduino), buscar formas de sacar la imagen procesada lo más limpia posible (ya sea con postprocesado Sobel, mejor configuración de OV7670…), ejecución de un filtro más complejo a una imagen recibida por un OV7670, etc.
- Crear una aplicación para Linux, Windows, MacOS o incluso para Android (comunicación por bluetooth) para recibir la información directamente de la FPGA y que la aplicación muestre por pantalla el resultado, sin tener que recurrir a editores de archivos binarios o Matlab.
- Otra opción sería comprobar si la aplicación, tal como está diseñada, es compatible con el envío de la imagen a un monitor a través de un cable VGA, una pantalla LCD en un shield de Arduino o algo parecido.
- Uso y creación de un curso para estudiantes desde secundaria, similar a lo que se hace con Arduino, para acercar la programación con HDL y programar un robot, por ejemplo.

9. Presupuesto

Debido a que este trabajo está basado en herramientas FOSS, el presupuesto para desarrollarlo no es elevado. Simplemente se ha de disponer de un ordenador con una distribución GNU/Linux, y la adquisición de la cámara y las tarjetas.

Hardware	Coste
Tarjeta Alhambra 1.1	49.90 + IVA = 60.37 €
Arduino UNO	4 - 8 €
Cámara OV7670	2.5 - 9 €
2x CD40109B-Q1	2 €
Protoboard, cables, etc.	5 €
Ordenador con distro GNU/Linux e internet	-
	Coste Hardware **85 €**
Recursos humanos	**Coste**
20 semanas de trabajo en investigación y diseño (4 meses)	4 x 1400 = 5600 €
TOTAL	**5685 €**

Excepto la Tarjeta Alhambra que la fabrica una única empresa y que actualmente no hay un dispositivo de evaluación que contenga una FPGA Lattice compatible con el "Proyecto IceStorm" con mejores prestaciones, el resto de los componentes son bastante económicos. Además hay que tener en cuenta que tanto la cámara como el Arduino tienen un abanico de precios muy amplio, ya que hay muchas tiendas online que las venden, tanto en España como fuera de ella.

10. Pliego de Condiciones

Para poder llevar a cabo este trabajo se han utilizado los siguientes recursos:

Hardware:

- Tarjeta Alhambra
- Tarjeta Arduino UNO
- 2x Cables USB
- Cámara OV7670
- 2x CD40109B-Q1
- 2x Resistencias de 10 kΩ
- Protoboard
- Ordenador con distribución GNU/Linux

Software:

- Icarus Verilog [4]
- Gtkwave [10]
- Yosys [11]
- Arachne-pnr [12]
- Icepack e Iceprog (Proyecto IceStorm) [5]
- Gedit (o cualquier editor de texto para programación)
- GtkTerm (o cualquier terminal serie)
- Bless Hex Editor (o cualquier editor de archivos binarios)
- Matlab (o cualquier forma de conversión de binario a imagen)

11. Manual de Usuario

Dado que el trabajo está expuesto como una guía de cada uno de los pasos necesarios para poder replicar o utilizarlo como base para otros proyectos.

Por ello a continuación se enlazarán a cada una de las partes en las que se describen dichos pasos:

- Instalación de las herramientas software del “Proyecto IceStorm”.
- Cómo utilizarlas por terminal.
- Introducción a la programación de Verilog con un ejemplo base y otro medio.
- Cómo se utilizan los programas GtkTerm, Bless Hex Editor y Matlab.
- Explicación detallada de cómo está concebido el diseño final y su funcionamiento.

12. Anexo de Código

Todo el código utilizado ha sido subido a la plataforma GitHub, tanto la aplicación final, como la aplicación de recepción de la imagen en blanco y negro sin procesar y el código de la Arduino UNO para configurar la OV7670.

Enlace: https://github.com/LordRios/Sobel-Filter-with-Project-IceStorm

Bibliografía:

[1] S. M. Trimberger “Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology” [En línea]. Disponible en: https://ieeexplore.ieee.org/document/7086413/ [Accedido: 5-sep-2019]

[2] S. Suresh, S. F. Beldianu y S. G. Ziavras “FPGA and ASIC Square Root Designs for High Performance and Power Efficiency”
[En línea]. Disponible en: https://www.academia.edu/Documents/in/Asic_Design_and_Fpga/ [Accedido: 5-sep-2019]

[3] “Software Libre” *Wikipedia 2019,* [En línea]. Disponible en: https://es.wikipedia.org/wiki/Software_libre/ [Accedido: 5-sep-2019]

[4] S. Williams “Icarus Verilog” [En línea]. Disponible en: http://iverilog.icarus.com/ [Accedido: 5-sep-2019]

[5] C. Wolf “Project IceStorm” [En línea]. Disponible en: http://www.clifford.at/icestorm [Accedido: 5-sep-2019]

[6] Betajet “IceStorm: Reverse-Engineering the Lattice iCE40 Bitstream” *EETimes 2015* [En línea]. Disponible en: https://www.eetimes.com/author.asp?section_id=36&doc_id=1327061 [Accedido: 5-sep-2019]

[7] “iCE40 LP/HX/LM” *Lattice* [En línea]. Disponible en: https://www.latticesemi.com/Products/FPGAandCPLD/iCE40 [Accedido: 5-sep-2019]

[8] “SymbiFlow - open source FPGA tooling for rapid innovation*”* [En línea]. Disponible en: https://symbiflow.github.io/#about [Accedido: 5-sep-2019]

[9] “Licencia ISC” *Wikipedia 2019,* [En línea]. Disponible en: https://es.wikipedia.org/wiki/Licencia_ISC [Accedido: 5-sep-2019]

[10] “GTKWave*”* [En línea]. Disponible en: http://gtkwave.sourceforge.net/ [Accedido: 5-sep-2019]

[11] C. Wolf “Yosys” [En línea]. Disponible en: http://www.clifford.at/yosys/ [Accedido: 5-sep-2019]

[12] D. Shah “Arachne-pnr”, *GitHub 2019,* [En línea]. Disponible en: https://github.com/YosysHQ/arachne-pnr/ [Accedido: 5-sep-2019]

[13] “IEEE Standard for Verilog Hardware Description Language” [En línea]. Disponible en: http://staff.ustc.edu.cn/~songch/download/IEEE.1364-2005.pdf [Accedido: 6-sep-2019]

[14] D. Shah “nextpnr -- a portable FPGA place and route tool”, *GitHub 2019,* [En línea]. Disponible en: https://github.com/YosysHQ/nextpnr [Accedido: 5-sep-2019]

[15] “Alhambra Bits” [En línea] Disponible en: https://alhambrabits.com/ [Accedido: 6-sep-2019]

[16] “ICE40HX1K-TQ144” [En línea] Disponible en: https://github.com/Obijuan/open-fpga-verilog-tutorial/raw/master/tutorial/doc/iCE40LPHXFamilyDataSheet.pdf [Accedido: 6-sep-2019]

[17] “Datasheet OV7670” [En línea] Disponible en: https://www.voti.nl/docs/OV7670.pdf [Accedido: 6-sep-2019]

[18] “Arduino UNO R3 compatible CH340” [En línea] Disponible en: https://www.e-ika.com/arduino-uno-r3-compatible-ch340-cable-usb [Accedido: 6-sep-2019]

[19] “CD40109B-Q1” [En línea] Disponible en: http://www.ti.com/product/CD40109B-Q1# [Accedido: 6-sep-2019]

[20] “Instalación” [En línea] Disponible en: https://github.com/Obijuan/open-fpga-verilog-tutorial/wiki/Cap%C3%ADtulo-0%3A-you-are-leaving-the-privative-sector [Accedido: 6-sep-2019]

[21] “open-fpga-install”,*GitHub 2016* [En línea] Disponible en: https://github.com/dcuartielles/open-fpga-install [Accedido: 6-sep-2019]

[22] “Icestudio IDE” [En línea] Disponible en: https://alhambrabits.com/software/ [Accedido: 6-sep-2019]

[23] “Iverilog Flags” *Fandom 2019*, [En línea]. Disponible en: https://iverilog.fandom.com/wiki/Iverilog_Flags [Accedido: 6-sep-2019]

[24] “Serial Camera Control Bus Functional Specification”, [En línea]. Disponible en: http://www4.cs.umanitoba.ca/~jacky/Teaching/Courses/74.795-LocalVision/ReadingList/ov-sccb.pdf [Accedido: 6-sep-2019]