
fcefn

Facultad de Ciencias Exactas, Físicas y Naturales
Universidad Nacional de San Juan

Tesis para obtener el grado de Magister en Informática

# Comparación de Enfoques de Desarrollo HDL y HLL en FPGA para Aplicaciones de Procesamiento de Imágenes

Autor: Ing. Roberto Millón
Director: Dr. Enzo Rucci
Codirector: Dr. Emmanuel Frati

San Juan, Argentina

2021

Universidad Nacional de San Juan
Facultad de Ciencias Exactas,
Físicas y Naturales


Tesis para obtener el grado de Magister en Informática

# Comparación de Enfoques de Desarrollo HDL y HLL en FPGA para Aplicaciones de Procesamiento de Imágenes

Autor: Ing. Roberto Millón
Director: Dr. Enzo Rucci
Codirector: Dr. Emmanuel Frati

San Juan, Argentina

2021

*Agradecimientos*

*Este trabajo es el fruto de años de dedicación junto al apoyo constante de mis seres queridos.*

*En primer lugar quiero agradecer a la Universidad Nacional de Chilecito (UNdeC) por darme la oportunidad y el tiempo necesarios para realizar esta investigación. A la Universidad Nacional de San Juan (UNSJ) por la comprensión con los alumnos de otras provincias, en especial a Daniela por su buena energía.*

*A todo el grupo de la maestría por la buena predisposición y compañerismo, en especial a Moni, Luz, Pablo y Mario con quienes compartí muchas tardes entre mates y risas.*

*Quiero agradecer a Sonia y Pimpo que me alivianaron la carga para seguir adelante y permitieron que finalice esta investigación.*

*A mis compañeros de oficina Euge, Lore, Franco, Donna, Maru y Mary por su calidez, amistad y ayuda permanente.*

*A mis amigos y hermanos de la vida Nano, Dani y Mariano por estar siempre en las buenas y en las malas.*

*A mis padres Roberto y Tedy, y mis queridos hermanos Diego, Ana, Andrés y Emma por ser incondicionales conmigo y apoyarme en cada decisión. Ellos han sido el sostén que me ha permitido alcanzar mis metas y objetivos.*

*A mis directores Enzo y Emma, que marcaron el norte para comenzar esta investigación y me ayudaron a recorrer este camino. Ellos no solo aportaron valiosos conocimientos sino que me brindaron su amistad y confianza. Gracias chicos por las horas de dedicación y esfuerzo que invirtieron en este trabajo.*

*Por último agradezco a mis dos amores, mi compañera de vida Naty y mi pequeña princesa Ana Paula. Ustedes son el motor que me impulsa a superarme en todo momento y el pilar que me mantiene en los momentos más dificiles. Sabemos el esfuerzo que significó trabajar en este tiempo de encierro, sin su cariño, apoyo y amor constante no imagino haber concluído esta tesis. Por todo eso, este trabajo también es de ustedes.*

# Resumen

Desde su invención a medidados de los 90, las FPGA han destacado por su gran poder de cómputo, bajo consumo energético y alta flexibilidad al reconfigurar su arquitectura interna para adaptarse a las aplicaciones. Esto convirtió a las FPGA en una excelente alternativa frente a los ASIC que presentaban una arquitectura fija y prolongados tiempos de diseño y fabricación, a pesar que contaban con eficiencia energética y rendimientos superiores a otras tecnologías. Una de las áreas que más se favoreció con la incorporación de FPGA es el procesamiento digital de imágenes, la cual consiste de técnicas y algoritmos computacionales que se aplican sobre una imagen para extraer información de ella o mejorar sus características.

A pesar de sus virtudes, las FPGA no tuvieron la aceptación esperada. Los lenguajes tradicionales de descripción de hardware (HDL) para programar FPGA como VHDL y Verilog resultaron poseer demasiadas palabras, ser complejos y propensos a errores. Esto, sumado a la creciente complejidad de los sistemas y los acotados tiempos de mercado, limitaron su uso.

Es por ello que desde el año 2000, los fabricantes comenzaron a ofrecer herramientas de desarrollo de alto nivel (HLS) como Catapult C, Bluespec y Autopilot para aumentar el nivel de abstracción y reducir la complejidad en el desarrollo de sistemas. Las HLS obtienen una descripción HDL del sistema a partir de un diseño en lenguaje de alto nivel (HLL) como C, C++ o System C. Esto permitió a los desarrolladores de software utilizar lenguajes de alto nivel para diseñar sistemas en FPGA y beneficiarse con las ventajas de esta tecnología.

A su vez, los fabricantes comenzaron a incorporar núcleos de procesamiento x86 y ARM a las FPGA lo que favoreció aún mas el uso de HLL. Estas arquitecturas heterogeneas denominadas Sistemas en Chip (SoC) permitieron balancear la carga computacional entre software y hardware utilizando el mismo algoritmo de alto nivel.

Sin embargo, los HLL fueron creados para desarrollar aplicaciones que ejecuten en procesadores secuenciales, por lo que presentaron algunos inconvenientes al ser utilizados en FPGA de naturaleza concurrente. Como consecuencia, los HLL incorporaron extensiones y directivas para suplir algunas falencias y permitir, entre otras cosas, sintetizar interfaces de comunicación, describir señales de cualquier longitud de bits y ejecutar tareas en forma concurrente. Por otro lado, se impusieron restricciones al lenguaje, por ejemplo, no permitir el uso de memoria dinámica.

Por lo tanto, al momento de elegir un lenguaje de programación para FPGAs resulta indispensable conocer sus fortalezas y debilidades. Es por ello que esta investigación tiene como objetivo general *comparar los enfoques de desarrollo HDL y HLL en FPGA con respecto a las prestaciones (rendimiento, uso de recursos y esfuerzo de programación) de sus implementaciones para aplicaciones de detección de contornos de imágenes.*

Para realizar el estudio comparativo, se eligió como caso de estudio un fil-

tro de detección de contornos Sobel, ya que al ser un filtro convolucional, resulta representativo de muchos otros. Luego, se desarrollaron dos implementaciones completas y funcionales siguiendo los enfoques HDL y HLL. Ambas propuestas fueron sometidas a pruebas sobre una plataforma SoC ZYBO usando cuatro imágenes extraídas de repositorios públicos. El diseño, implementación e integración de cada sistema se realizó con las herramientas de Xilinx Vivado 2019.1, Vivado HLS 2019.1 y XSKD 2019.1, sin el uso de librerías de imagen/video, sistema operativo o software adicional.
Los resultados obtenidos demostraron que la implementación HDL es levemente superior a la versión HLS en cuando a eficiencia en el uso de recursos y rendimiento. Respecto al primero, la versión HDL empleó menor cantidad de recursos aunque en ninguna de las implementaciones representó una restricción de diseño. En cuanto al rendimiento, la versión HDL obtuvo menor *throughput*, latencia y tiempo de ejecución. En el caso de este último, la versión en HDL alzanzó una aceleración mayor a 6× en la imagen de 512×512, aunque esa diferencia disminuyó al aumentar el tamaño de las imágenes obteniendo una aceleración de sólo 1.4× en la imagen de tamaño 1920×1080.
Sin embargo, el esfuerzo de programación en la implementación de HDL fue significativamente mayor. El diseño en HDL requirió especificaciones de bajo nivel, como la descripción de interfaces AXI4 y el solapamiento de tareas, que complejizaron el algoritmo, a diferencia del diseño de alto nivel que resolvió ambas especificaciones por medio de directivas de compilador. En consecuencia, la descripción en HDL no sólo aumentó el tiempo de desarrollo sino que los diseños tuvieron más líneas de código (SLOC), fue más difícil de programar y menos reutilizable respecto a la versión de alto nivel.
Luego del análisis de los resultados obtenidos se concluyó que el esfuerzo de programación requerido por HDL fue significativamente mayor respecto al de HLS y sólo se obtuvo una leve mejora en uso de recursos y rendimiento. En contextos similares a los de este estudio, HDL sólo resultaría conveniente en los diseños donde el uso de recursos y/o el tiempo de respuesta fueran parámetros de diseño sumamente críticos. De otra forma, HLS es una mejor opción que permite reducir significativamente el esfuerzo de programación y el tiempo de desarrollo.

# Abreviaturas

| | |
|---|---|
| **ALU** | Unidad Aritmética Lógica |
| **AMBA** | Arquitectura de Buses para Microcontroladores Avanzada |
| **ARM** | Máquinas RISC Avanzadas |
| **ASIC** | Circuito Integrado de Aplicación Específica |
| **AXI** | Interfaces Extensibles Avanzadas |
| **BRAM** | Bloques de RAM |
| **CLB** | Bloques Lógicos Configurables |
| **CPU** | Unidad Central de Procesamiento |
| **DMA** | Acceso Directo a Memoria |
| **DP** | Productividad de Diseño |
| **FF** | *Flip-Flop* |
| **FPGA** | Arreglo de Compuertas Programables en Campo |
| **GPU** | Unidad de Procesamiento Gráfico |
| **HDL** | Lenguaje de Descripción de Hardware |
| **HLL** | Lenguaje de Alto Nivel |
| **HLS** | Síntesis de Alto Nivel |
| **ILA** | Analizador Lógico Integrado |
| **IOB** | Bloques de Entrada y Salida |
| **LUT** | Tabla de Búsqueda |
| **MPEG** | *Moving Picture Experts Group* |
| **MUX** | Multiplexor |
| **NRE** | Ingeniería No-Recurrente |
| **PL** | Lógica Programable |
| **PS** | Sistema de Procesamiento |
| **RAM** | Memoria de Acceso Aleatorio |
| **RTL** | Nivel de Transferencia de Registro |
| **SLOC** | Líneas de Código Fuente |
| **SoC** | Sistemas en Chip |
| **SRAM** | RAM Estática |
| **VHDL** | *Very high speed integrated circuit Hardware Description Language* |

# Índice

# Índice de tablas

# Índice de figuras

# Introducción

Este capítulo inicia en la sección Antecedentes donde se describen estudios comparativos entre HDL y HLS para FPGA. En la sección denominada Formulación del Problema y Justificación del Tema se detallan los inconvenientes en el uso de los HDL y el surgimiento de los HLS. El capítulo continúa enunciando los Objetivos de esta investigación y la Metodología aplicada para llevarla a cabo. Luego, se describen los aportes de esta tesis en Contribuciones y, finalmente, en la sección Publicaciones se enuncian los trabajos científicos que avalan la presente tesis.

## 1.1. Antecedentes

Los arreglos de compuertas programables en campo (FPGA, Field Programmable Gate Array) se han consolidado como una excelente alternativa para la aceleración de aplicaciones de diversas áreas, en gran medida debido a las posibilidades de paralelizar la ejecución de algoritmos y el bajo consumo energético que presentan respecto a otras tecnologías como las unidades centrales de procesamiento (CPU, Central Processing Unit) y las unidades de procesamiento gráfico (GPU, Graphics Processing Unit). Sin embargo, la complejidad de los lenguajes de descripción de hardware (HDL, Hardware Description Language) utilizados para desarrollar en las FPGA ha limitado su uso.

En las últimas dos décadas, un conjunto de herramientas de síntesis de alto nivel (HLS, High-Level Synthesis) para FPGA han sido optimizadas para permitir a los diseñadores realizar implementaciones en lenguajes de alto nivel (HLL, High Level Language) como C y C++, los cuales resultan más habituales para los programadores de aplicaciones. Sin embargo, los HLL fueron desarrollados para escribir aplicaciones que serían ejecutadas en máquinas de Von Neumann bajo el concepto de programa almacenado y ejecución secuencial de instrucciones. Aunque se desempeñan muy bien en CPU y GPU (Windh y cols., 2015), al utilizarlos en FPGA suelen ser menos eficientes respecto de los desarrollos HDL.

Para comparar las implementaciones hardware en FPGA diseñadas con HDL respecto de los HLL, la comunidad científica ha realizado numerosas investigaciones. Los resultados obtenidos en los estudios demuestran que HLL suele disminuir los tiempos de desarrollo por el nivel de abstracción mayor y la "familiaridad" de los diseñadores con los lenguajes de este tipo. Sin embargo, los HDL usualmente emplean los recursos de las FPGA de mejor manera y obtienen velocidades mayores de operación en los diseños.

Las afirmaciones anteriores apoyan una corriente de pensamiento que anticipa un reemplazo de HLL respecto a HDL en FPGA, del mismo modo que

lenguajes de mayor nivel de abstracción (como por ejemplo, el lenguaje C) reemplazaron al lenguaje Assembler para programar en CPU. Entre las investigaciones que están alineadas con estas afirmaciones, se encuentra el estudio comparativo en (Hiraiwa y Amano, 2013) para acelerar un sistema embebido de visión en tiempo real, el cual puede ser utilizado en cámaras de vigilancia o cámaras vehiculares. En esta investigación, aunque las latencias alcanzadas por ambas implementaciones fueron similares, los autores observaron un aumento en el uso de los recursos de la FPGA por parte de la implementación de alto nivel (entre 3 y 4 veces más respecto a HDL). La mayor diferencia se observó en los tiempos de desarrollo, con una demora de 15 días para obtener el diseño de bajo nivel, y sólo 3 días para obtener el mismo sistema utilizando herramientas y un lenguaje de alto nivel.
Otra investigación que obtiene resultados similares fue realizada por Zwagerman (Zwagerman, 2015) también en el ámbito del procesamiento de imágenes. El autor implementó un filtro de desenfoque tanto en HDL como en HLL. Las métricas elegidas para comparar ambos diseños fueron: eficiencia de uso de recursos de la FPGA, frecuencia máxima de operación y costo de ingeniería no-recurrente (NRE, Non-Recurring Engineering). Con esta última métrica incorporó en su análisis el tiempo de desarrollo de cada implementación. Sus resultados muestran que la implementación en HLL demandó la mitad de tiempo de desarrollo respecto a la de HDL, pero fue menos eficiente (utilizó más recursos de la FPGA). En cuanto a la frecuencia de operación de ambas implementaciones, el diseño HDL alcanzó velocidades mayores de operación comparado al diseño HLL, aunque sin resultar significativa esa diferencia.
Otros dos estudios realizados en procesamiento de imagen como (Pelcat, Bourrasset, Maggiani, y Berry, 2016) y (Gurel, 2016) obtuvieron resultados acordes a esta tendencia. En el primero, se utilizó como caso de estudio el formato de compresión de video MPEG (Moving Picture Experts Group) y se evaluó la productividad de diseño (DP, Design Productivity) de las implementaciones en HDL y HLS. Se concluyó que al sintetizar a alto nivel se obtiene una ganancia en DP de 2x respecto de HDL. Los autores de este trabajo propusieron la métrica de comparación PD basada en NRE y la calidad de resultados (QoR, Quality Of Results). Por otro lado, el estudio realizado por Gurel en (Gurel, 2016) utilizó la implementación de varios filtros de imagen en bajo y alto nivel. Sus resultados indicaron que, en general, las implementaciones en HLL consumieron más cantidad de recursos, pero demandaron menos tiempo de desarrollo. Sin embargo, las implementaciones de alto nivel obtuvieron buenos rendimientos en cuanto a *throughput*, incluso superando a la implementación en HDL para el caso del filtro Gausiano. El autor considera que esto se debe a que las implementaciones en HDL dependen en gran

medida de la experticia del diseñador, mientras que en los desarrollos de alto nivel muchos detalles en la arquitectura de la aplicación quedan a cargo del compilador HLS (por ejemplo las señales de control, la cantidad de puertos de Entrada/Salida, entre otros). A su vez, la optimización en los diseños de alto nivel se obtiene a través de directivas que no modifican la lógica del algoritmo.

Se puede mencionar también el trabajo (Stanciu y Gerigan, 2017), en el cual los autores compararon un diseño sintetizado en bajo y alto nivel para realizar operaciones de multiplicación y división en campos de galois [1]. Las métricas de comparación fueron la eficiencia en el uso de los recursos de la FPGA, la cantidad de líneas de código y la frecuencia de operación. Los resultados indican que la multiplicación sintetizada con HLL demanda aproximadamente el doble de recursos de la FPGA respecto a la implementación con HDL. Sin embargo, la cantidad de líneas de código en HLL son un 74 % menos que en HDL (40 líneas de código en HLL en comparación a 250 líneas de código en HDL). Respecto a la frecuencia de operación, ambas implementaciones alcanzaron la misma velocidad.

La comparación de implementaciones realizadas en FPGA con el uso de HLL y HDL se extiende también al campo de las finanzas, como se observa en (Stamoulias, Kachris, y Soudris, 2017). En este trabajo, se implementaron soluciones para análisis de riesgos, concluyendo que HLL permite disminuir los tiempos de desarrollo e incluso obtener resultados más precisos por la utilización de punto flotante a costa de un aumento en el uso de los recursos de la FPGA.

También se incluye el estudio realizado en (Marc-André, 2018) donde los autores implementaron dos módulos para tratamiento de medicina nuclear por imagen en HLL y HDL. El diseño de alto nivel demandó menos tiempo de desarrollo pero resultó menos eficiente en el uso de los recursos y obtuvo velocidades menores de operación respecto al HDL. Si bien el autor reconoce las ventajas de utilizar HLL en FPGA por demandar menos tiempo de desarrollo y obtener diseños más sencillos de modificar/actualizar, enfatiza la necesidad del diseñador de entender los principios de desarrollo en hardware reconfigurable para obtener resultados óptimos. Esto último se debe a que HLL requiere modificaciones para ser utilizado en FPGA.

En sentido contrario a las investigaciones anteriores, un estudio reciente demostró que HDL podría resultar superior para desarrollar diseños parametrizables respecto de HLL, cuando el diseño tiene características particula-

[1] Campos de Galois (CG) Conjunto finito de elementos donde el número de elementos debe ser primo o una potencia de un número primo. Los CG se utilizan en diferentes áreas como criptografía, detección/corrección de errores, cifrado de datos, entre otros.

res (Aledo, Schafer, y Moreno, 2019). Empleando como caso de estudio una red de neuronas artificiales, los autores determinaron que HDL dispone de herramientas más potentes respecto a HLL para obtener diseños genéricos que sean altamente parametrizables, lo que eleva el esfuerzo de programación del segundo. Este trabajo permite suponer que los HDL podrían ser una mejor opción para algunos desarrollos, o partes de ellos, respecto a HLL al utilizar sentencias genéricas.
En general, las comparaciones entre HLL y HDL en FPGA parecen concluir de forma similar en lo que refiere al esfuerzo de programación. Hasta donde llega el conocimiento del tesista, el uso de HLL respecto a HDL está en aumento, ya que los tiempos menores de desarrollo en alto nivel permiten hacer frente a los exigentes tiempos de mercado. Sin embargo, no hay tendencias claras en lo que refiere al rendimiento y el uso de recursos por parte de cada uno de estos dos enfoques, estando fuertemente influenciados por las características del problema a resolver, las herramientas utilizadas y las especificaciones de los diseños.

## 1.2. Formulación del Problema y Justificación del Tema

Las FPGA son dispositivos electrónicos configurables ampliamente utilizados en la industria electrónica desde su invención a mediados de los 90. Estas fueron desarrolladas como una alternativa a los Circuitos Integrados de Aplicación Específica (ASIC, Application Specific Integrated Circuit) que demandaban alto costo de NRE, elevado tiempo de desarrollo y un diseño fijo no reconfigurable, pero que ofrecían la mejor eficiencia energética y altas velocidades de operación (Trimberger, 2015). En contraposición a los ASIC, se encontraban los desarrollos software en CPU de propósito general como la alternativa más flexible, con la posibilidad de reprogramar la aplicación ejecutada pero con mayor consumo energético. Las FPGA llegaron para ofrecer una solución intermedia respecto a los ASIC y CPU, con la flexibilidad de reconfigurar su arquitectura de acuerdo a la aplicación y con gran eficiencia energética (Putnam y cols., 2015). Estas características convirtieron a las FPGA en una propuesta muy interesante para disminuir el consumo energético, que ha adquirido gran relevancia en la industria en los últimos años. Según el estudio de (Wu y cols., 2014) en el año 2010 los centros de datos consumían 1,2 % de la electricidad global, con un aumento del 56 % desde 2005 hasta 2010.
A pesar de ello, las FPGA no han tenido la aceptación esperada, principal-

mente por la complejidad de los desarrollos. Durante años la configuración de una FPGA se realizó exclusivamente con HDL. Estos lenguajes son muy rígidos, verbosos, poco reutilizables y demandan amplia experticia en el hardware de la FPGA. En consecuencia, se demandaron nuevas herramientas de desarrollo con mayor nivel de abstracción que pudieran atender la creciente complejidad de los sistemas y los exigentes tiempos de mercado (Windh y cols., 2015).
Desde el año 2000 se han comenzado a utilizar HLS (como Catapult C, Bluespec, Autopilot, entre otros), que hacen uso de HLL como C, C++, System C para generar la descripción HDL. El código obtenido es utilizado para configurar la FPGA (Martin y Smith, 2009). Las implementaciones de alto nivel en FPGA, no solo acortan los tiempos de desarrollo y permiten obtener diseños reutilizables sino que expanden el uso de estas tecnologías a programadores de software que no necesitan tener un gran conocimiento sobre el hardware (Nane y cols., 2016). Otro beneficio de las HLS es permitir la optimización en los diseños por medio de la incorporación de sentencias adicionales independientes de la lógica del código, por lo que el programador puede comparar numerosas soluciones sin modificar el algoritmo.
Por todo ello, las HLS han facilitado que las FPGA adquieran mayor relevancia. Una muestra de ello es el aumento en el uso de estos dispositivos como co-procesadores para acelerar secciones de código con extenso paralelismo que demandan alto poder de cómputo, dejando a cargo del CPU de propósito general las porciones seriales de la aplicación (Selvaraj, Daoud, y Zydek, 2013). Es así que grandes empresas y organizaciones han comenzado a utilizar FPGA para optimizar sus procesos:

- Microsoft ha adquirido FPGAs Arria y Stratix de Intel para mejorar el rendimiento y la eficiencia energética de las búsquedas de Bing (*Microsoft Uses Intel FPGAs for Smarter Bing Searches*, 2018).
- La Organización Europea para la Investigación Nuclear (CERN) ha incorporado FPGA en conjunto con CPU y GPU, para procesar los datos del Gran Colisionador de Hadrones (LHC) (*CERN openlab Explores New CPU/FPGA Processing Solutions*, 2017).
- Baidu ha incorporado FPGA de Xilinx para ofrecer servicios de aceleración de aplicaciones en la nube tanto para la industria como para la academia (*Baidu Deploys Xilinx FPGAs in New Public Cloud Acceleration Services*, 2017).
- Facebook ha solicitado profesionales con experiencia en desarrollos en FPGA para trabajar en el campo de realidad aumentada (*FPGA Developer, Augmented Reality*, 2019).

- Amazon se ha asociado con Xilinx para ofrecer un servicio en la nube de desarrollo de sistemas en FPGA que permitan acelerar algoritmos (*Amazon And Xilinx Deliver New FPGA Solutions*, 2017).

- AMD ha comprado Xilinx para aumentar su participación en el sector de computación de altas prestaciones (*AMD to Acquire Xilinx, Creating the Industry's High Performance Computing Leader*, 2020).

Sin embargo, los lenguajes de alto nivel han sido desarrollados para aplicaciones secuenciales. En consecuencia presentan limitaciones al ser utilizados para describir hardware, como por ejemplo: las FPGA no soportan asignación dinámica de memoria o recursión; las operaciones nativas en FPGA sólo utilizan representación en punto fijo, por lo que las operaciones en punto flotante deben ser implementadas por el usuario y dedicar recursos para esa tarea (Che, Li, Sheaffer, Skadron, y Lach, 2008) (Escobar, Chang, y Valderrama, 2016); los HLL tampoco permiten definir tiempos de hardware o especificar la longitud de los bits de las señales (Ren, 2014).
Por todo lo anterior, hay posiciones encontradas entre la utilización de HDL o HLL en FPGA. Si bien la tendencia es utilizar lenguajes de alto nivel que acorten los tiempos de desarrollo, estos lenguajes deben ser extendidos y/o restringidos en sus funcionalidades para adaptarse a los desarrollos hardware. No pasa lo mismo con los HDL que se adaptan muy bien a los desarrollos en FPGA, aunque son más difíciles y complicados de usar.
Otro aspecto importante es el conocimiento que se debe tener de la arquitectura hardware de la FPGA para obtener resultados óptimos aún al utilizar HLL. Como se mencionó antes, las herramientas de HLS fomentan el uso de las FPGA entre programadores de software, pero las prestaciones de los resultados dependerá de las directivas de optimización utilizadas, para lo cual es indispensable conocer el hardware de la FPGA.
Por último, cabe destacar que al utilizar HLL en FPGA, muchas características del diseño son realizadas por las herramientas HLS, como son las señales de control o la interfaz de conexión. Esto supone una disminución en la complejidad, ya que el programador sólo se concentra en la lógica del algoritmo a implementar. El costo por ello es perder detalle en la implementación, volviendo el diseño muy dependiente de la herramienta HLS.
Es indispensable conocer las ventajas y desventajas de cada lenguaje de programación para implementar sistemas en FPGA, así como las sentencias y directivas en cada lenguaje para obtener diseños óptimos.

## 1.3. Objetivos

El objetivo general de esta investigación es comparar los enfoques de desarrollo HDL y HLL en FPGA con respecto a las prestaciones (rendimiento, uso de recursos y esfuerzo de programación) de sus implementaciones para aplicaciones de detección de contornos de imágenes. Los objetivos específicos son:

- Explorar aplicaciones de detección de contornos en imágenes en FPGA que empleen enfoques de desarrollo HDL y HLL.
- Analizar las prestaciones de aplicaciones de detección de contornos en imágenes que hayan sido desarrolladas siguiendo los enfoques HDL y HLL.
- Comparar las prestaciones de los enfoques HDL y HLL para implementaciones de aplicaciones de detección de contornos en imágenes.

## 1.4. Metodología

Esta investigación se planteó como un estudio aprehensivo (Hurtado, 2008) que permitió percibir, en los enfoques de desarrollo HDL y HLL, aspectos o cualidades que determinan una implementación óptima en FPGA de un filtro de detección de contornos en imágenes mediante el algoritmo Sobel. Las variables de estudio seleccionadas para comparar ambos enfoques son el rendimiento, uso de recursos y esfuerzo de programación en ambos diseños. Para cumplir los diferentes objetivos se realizaron las siguientes actividades:

### 1.4.1. Estudio del Estado del Arte sobre el Tema de Investigación

Las actividades de esta etapa se realizaron fundamentalmente durante los primeros meses del plan de trabajo:

- Estudio de antecedentes. Se llevó a cabo un relevamiento de la bibliografía existente en el tema a partir de revistas, publicaciones y tareas de otros grupos de investigación, publicadas por IEEE, ACM y Elsevier.
- Estudio de temas teóricos fundamentales. El objetivo fue profundizar en el conocimiento sobre los aspectos relevantes que afectan el desempeño de implementaciones HDL y HLL en FPGA.

Los resultados de esta etapa se midieron en el incremento y mejora de la calidad del acervo bibliográfico referido al área de estudio y campo de aplicación.

### 1.4.2. Focalización del Objeto de Estudio, Experimentación y Obtención de Resultados

Esta etapa se desarrolló luego de comenzada la etapa anterior. Las actividades desarrolladas fueron:

- Elección de benchmark. Se seleccionó un filtro de imagen convolucional para la detección de contornos Sobel. Las operaciones de convolución son multiplicaciones de datos que se ven favorecidas por la ejecución concurrente de tareas en las FPGA, además de permitir su diseño en bajo y alto nivel.
- Configuración del entorno de experimentación. Se instalaron las herramientas Vivado 2019.1, Vivado HLS 2019.1 y XSDK 2019.1 de la empresa Xilinx para desarrollos con lenguajes HDL y HLL. Las pruebas se realizaron sobre una plataforma de desarrollo ZYBO de la empresa Xilinx, que cuenta con una FPGA xc7z010clg400-1 de la familia Artix y un procesador ARM Cortex-A9 dual-core. Las imágenes de prueba se seleccionaron de repositorios públicos.
- Diseño y desarrollo de soluciones Sobel. Se diseñó y desarrolló una implementación del filtro de detección de contorno Sobel con cada enfoque de programación, utilizando HDL y HLL.
- Experimentación. Se midió el rendimiento, uso de recursos y esfuerzo de programación en ambas implementaciones.

### 1.4.3. Evaluación y Difusión de Resultados

Esta actividad se desarrolló en paralelo con la Actividad 2. En esta etapa se evaluaron y analizaron los datos obtenidos, y se difundieron los resultados de esta investigación en tres publicaciones científicas (ver Sección 1.6).

## 1.5. Contribuciones

Las contribuciones principales de esta tesis son:

- Una descripción del estado del arte de trabajos comparativos entre implementaciones HDL y HLS para FPGA, la cual permite identificar las tendencias de uso para cada enfoque.
- La creación de un repositorio público Git con dos soluciones optimizadas del filtro Sobel implementados en HDL y HLL para una plataforma SoC ZYBO. Al ser un filtro convolucional, resulta sencillo modificar/adaptar el código para implementar otros filtros de imágenes[2].
- Una comparación rigurosa entre enfoques de desarrollo HDL y HLL para FPGA en cuanto a uso de recursos, rendimiento y esfuerzo de programación. Como consecuencia, se identifican fortalezas y debilidades en cada enfoque de desarrollo para su correcta elección.

## 1.6. Publicaciones

Las publicaciones realizadas con los resultados obtenidos en esta tesis son:

- Implementación de Filtro de Detección de Bordes Sobel en SoC usando Síntesis de Alto Nivel.

  R. Millon, F. E. Frati, and E. Rucci, Actas del Congreso Argentino de Sistemas Embebidos (CASE2020), ISBN: 978-987-46297-7-7, págs. 73-75, 2020. http://sedici.unlp.edu.ar/handle/10915/102579.

- Análisis comparativo de implementaciones HLS de filtro Sobel en SoC.

  R. Millon, E. Rucci, and F. E. Frati, Actas del XXVI Congreso Argentino de Ciencias de la Computación (CACIC 2020), ISBN: 978-987-44-1790-9, págs. 639-648, 2020. http://sedici.unlp.edu.ar/handle/10915/114492.

- A comparative study between HLS and HDL on SoC for image processing applications.

  R. Millón, E. Frati, and E. Rucci, Elektron (ISSN 2525-0159), vol. 4, num. 2, págs. 100-106, doi. 10.37537/rev.elektron.4.2.117.2020, 2020.

[2] https://github.com/robertoamt/HDL-HLS-Sobel-filters-

# Marco Teórico

La sección Arquitectura FPGA y su Evolución introduce al lector en la tecnología FPGA y sus características. Comienza con una breve mención de sus orígenes y su evolución a través del tiempo, luego se describe su arquitectura y se detallan sus componentes principales.
La sección denominada Lenguajes HDL y HLL para FPGA, relata los inicios de ambos lenguajes de programación y sus características. Posteriormente, se describen las herramientas HLS y los métodos utilizados por ellos para resolver algunas limitaciones en el uso de los HLL. Se concluye con una mención a las herramientas HLS comerciales y no comerciales más populares.
La sección titulada Métricas para Evaluación de Soluciones HDL y HLL, describe las métricas que se utilizan para evaluar soluciones en FPGA. Se enfatiza en la importancia de disminuir el uso de recursos en FPGA y se brinda un ejemplo de la herramienta Vivado de Xilinx para informar los recursos utilizados. Esta sección finaliza con la definición de productividad de diseño, explicando los parámetros que influyen en ella.
La sección Procesamiento de Imágenes. Detección de Contornos define procesamiento de imágenes y se listan aplicaciones de estas técnicas. Luego, se detalla el algoritmo para la detección de contornos Sobel.
Por último, se incluye un Resumen del capítulo.

## 2.1. Arquitectura FPGA y su Evolución

Las FPGA se crearon en el año 1984 por los co-fundadores de Xilinx, Ross Freman y Bernard Vonderschmitt. Esta tecnología pertenece al grupo de dispositivos lógicos programables (PLD, Programmable Logic Device) junto con las redes lógicas programables y los dispositivos lógicos programables complejos (CPLD, Complex Programmable Logic Device). Dentro de este conjunto, las FPGA tienen la más alta densidad de integración de puertas lógicas (Trimberger, 2015).
Los PLD tienen la capacidad de configurar las conexiones internas de su hardware para generar cualquier circuito digital (Kuon, Tessier, y Rose, 2007). Al igual que los PLD, los ASIC también ofrecen soluciones hardware pero a diferencia de los primeros, los ASIC no son reconfigurables sino que son construidos con un diseño fijo para realizar tareas específicas. Los ASIC ofrecen las mejores prestaciones en cuanto a velocidad de operación, consumo energético y área de silicio utilizado. Sin embargo, sus diseños pueden demandar años de fabricación y costar millones de dólares, además de no permitir modificaciones posteriores sobre el producto final. Esto llevó a las FPGA a ganar terreno sobre los ASIC desde los años 90s (Trimberger, 2015).
La arquitectura interna de las FPGA ha variado a lo largo de los años para

incorporar mayor cantidad de bloques de funciones. Una FPGA esta formada por una matriz de interconexiones programable en la que se disponen distintos bloques de funciones que realizan tareas particulares. La implementación de un sistema digital en FPGA consiste en configurar las tareas que deben realizar los bloques de funciones y en definir las conexiones entre ellos.
Los componentes principales dentro de una FPGA son los Bloques Lógicos Configurables (CLB, Configurable Logic Blocks), Bloques de Entrada/Salida (IOB, Input/Output Blocks), Bloques de RAM (BRAM), y DSP (Digital signal processor). Todos ellos embebidos en la matríz de interconexiones programable, como se observa en la Figura 2.1.

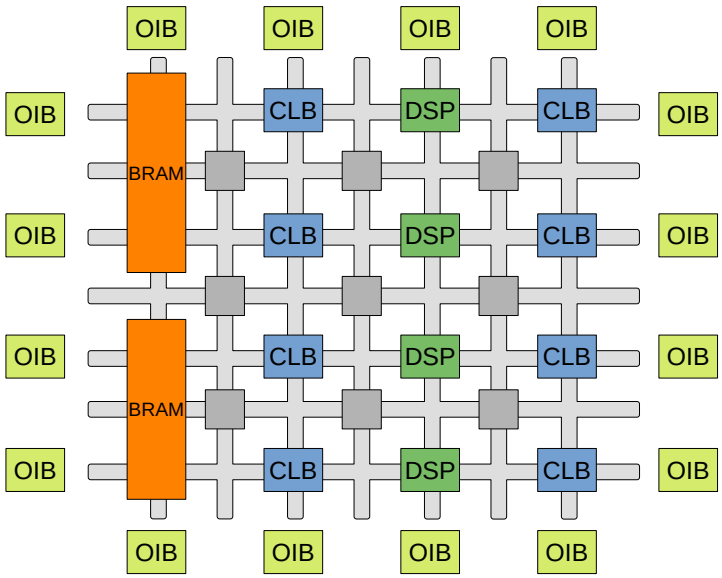


Figura 2.1: Arquitectura interna de una FPGA

Los CLB realizan las funciones lógicas en una FPGA. Están constituidos por Tablas de Búsqueda (LUT, Look Up Table), Flip-Flop (FF) y multiplexores (MUX, multiplexer). En la Figura 2.2 se observa la arquitectura interna de una CLB.
Las LUT están formadas por multiplexores y celdas de memoria (FLASH o SRAM). Son los bloques que permiten generar cualquier función lógica. Al igual que una tabla de verdad, obtienen un valor de salida para cada combinación de entrada. Además, pueden ser utilizadas para almacenar datos por lo que se las denomina memoria distribuida.
Los IOB, observados en la Figura 2.3, permiten transmitir y recibir señales hacia y desde el exterior del FPGA. Estos bloques incorporan muchas funcionalidades, como establecer la impedancia de los terminales, seleccionar los

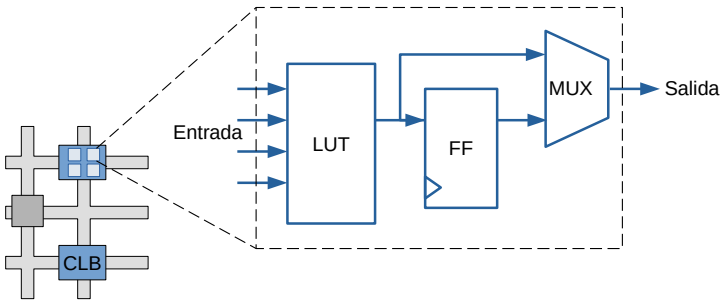


Figura 2.2: Arquitectura interna de una CLB

niveles de tensión, la pendiente y el retardo de la señal, entre otros. Están constituídos por buffers, FF, multiplexores y otros dispositivos.

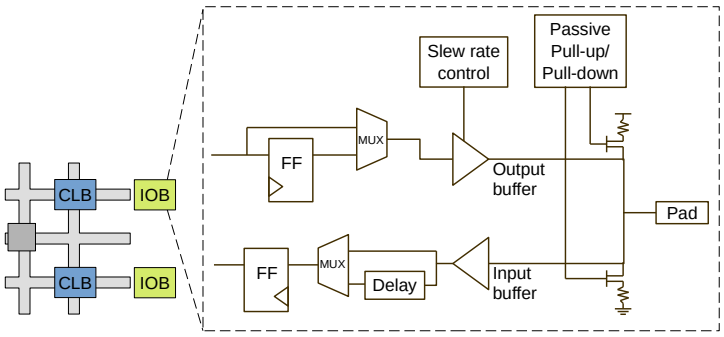


Figura 2.3: Arquitectura interna de una IOB

Los DSP son unidades aritméticas lógicas (ALU, Arithmetic Logic Unit) embebidos en la FPGA. Estos bloques realizan los cálculos aritméticos/lógicos complejos para reservar las LUT a otras secciones del diseño. Un DSP es una cadena de unidades de función y FF. Las unidades de función incluyen sumador/restador, multiplicador y sumador/restador/acumulador como se observa en la Figura 2.4.

Por otro lado, los BRAM son bloques de memoria RAM para almacenar grandes cantidades de datos. Tienen mayor capacidad de almacenamiento respecto a las LUT, pero son más lentas. Dependiendo de la aplicación, el programador puede utilizar los bloques LUT, como almacenamiento o los BRAM. Las LUT son las unidades de memoria más rápidas dentro de las FPGA, aunque también son utilizadas de generar las funciones lógicas.

Como se mencionó antes, todos los bloques que componen la FPGA se encuentran embebidos en una matriz de interconexiones programable. Está compuesta por rutas de diferentes niveles para la conexión de bloques que están a corta y larga distancia (jerarquía de interconexiones). La configuración

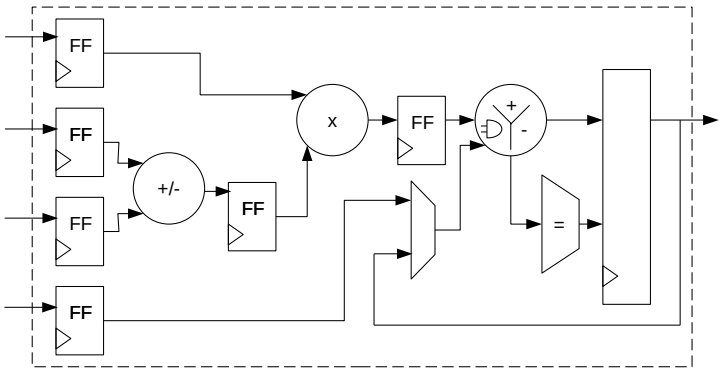


Figura 2.4: Arquitectura interna de un DSP

de rutas se realiza por medio de celdas de memoria, aunque existen otras variantes (Kuon y cols., 2007). En la Figura 2.5 se observan dos tipos de conexión programable. Ambas tienen en común la presencia de transistores y celdas SRAM para habilitar la ruta de conexión.

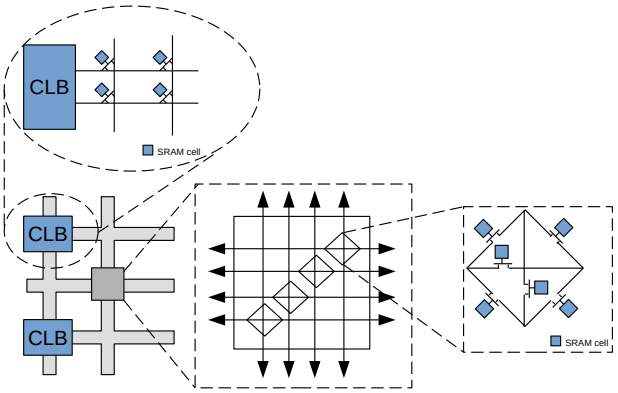


Figura 2.5: Matríz de interconexiones programable

Desde hace años, se ofrecen sistemas de arquitecturas heterogeneas que combinan núcleos de procesamiento x86 o ARM con FPGA denominados Sistemas en Chip (SoC, System On Chip) (Trimberger, 2015). Estas plataformas han ganado mucha aceptación en los desarrollos hardware, concentrando el 63 % de los diseños en FPGA hasta el año 2018 (Foster, 2018). En la Figura 2.6 se observa la arquitectura interna del SoC ZYNQ-7000 de la empresa Xilinx.
El dispositivo presenta dos bloques centrales. Un Sistema de Procesamiento (PS, Processing System) en color celeste donde se encuentra un procesador ARM dual-core con módulos de interfaz SPI, UART, CAN, etc, y otro bloque

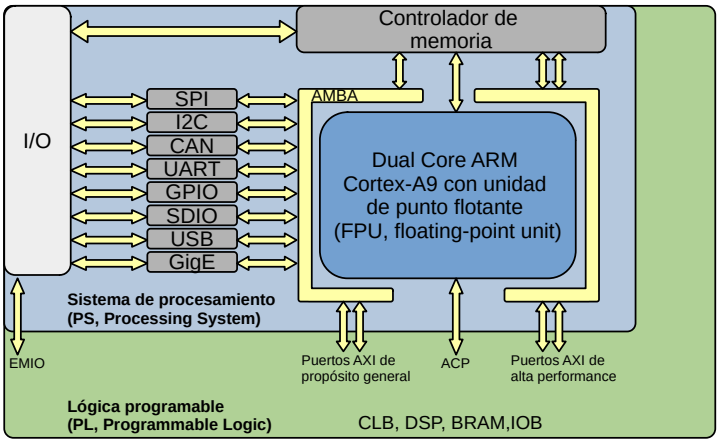


Figura 2.6: Arquitectura interna de un SoC modelo Zybo de la empresa Xilinx

que contiene a la Lógica Programable (PL, Programable Logic) en color verde y corresponde a la FPGA de la familia Artix-7.
La conexión entre el Sistema de Procesamiento y la Lógica Programable se realiza por interfaces extensibles avanzadas (AXI, Advanced eXtensible Interface) representadas con color amarillo en la Figura 2.6. El protocolo AXI forma parte de los canales de interconexión AMBA (Advanced Microcontroller Bus Architecture) desarrollados por la empresa ARM.
En 2003, ARM introdujo la primera versión del protocolo AXI en AMBA3.0 y en 2010 liberó la versión AXI4. Los buses AMBA se han constituído como un estándar abierto para la conexión y el manejo de bloques de funciones en SoC.
Hay tres tipos de interfaces AXI:

- AXI4: Conexión mapeada en memoria de alto rendimiento que permite transmisiones de hasta 256 datos por transacción.
- AXI4-LITE: Comunicación mapeada en memoria de bajo rendimiento. Sólo permite la transmisión de un dato por transacción y es utilizada para la configuración de dispositivos.
- AXI4-STREAM: Conexión punto a punto para transmitir datos ilimitados a alta velocidad.

Independientemente de su tipo, todas las interfaces AXI4 comparten dos características. La primera es que todas las transferencias se realizan en modo maestro-esclavo donde el maestro es quien inicia las peticiones para realizar las tareas y el esclavo responde a esas peticiones. La segunda característica

común a todas las interfaces AXI4 es el uso de tres señales de control denominadas *Ready*, *Valid* y *Last*. Mediante la señal *Ready*, el esclavo le indica al maestro que está preparado para recibir transacciones. Por otro lado, el maestro activa la señal *Valid* al enviar un dato o dirección válido. Entonces, una transferencia es exitosa cuando ambas señales, *Ready* y *Valid*, están activas en alto. La señal *Last* indica el último envío en una transacción.
AXI4-LITE es una conexión simplificada de la conexión AXI4. Utiliza las mismas señales de control aunque con una diferencia significativa en la cantidad de datos que pueden transmitir por transacción. Tienen puertos de lectura y escritura de direcciones de memoria, puertos de lectura y escritura de datos y una conexión de control adicional. En la Figura 2.7 se observan los puertos de direcciones *Write Address (AW)* y *Read Address (AR)*, los puertos de datos *Write Data (W)* y *Read Data (R)* y el puerto de respuesta *Write Response (B)*, además de los puertos de control *Ready*, *Valid* y *Last*.

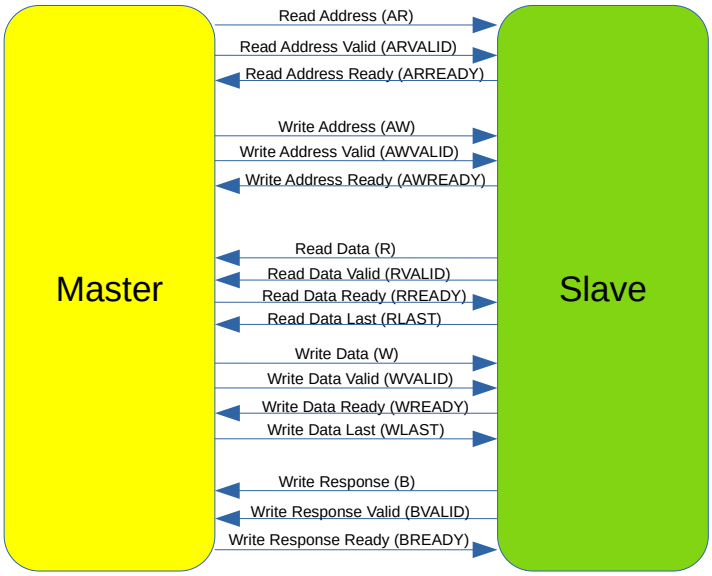


Figura 2.7: Puertos AXI4-LITE

En forma simplificada, para realizar una escritura de datos, el maestro envía la dirección por el puerto AW y el dato por el puerto W. El esclavo escribe el dato en la dirección indicada y envía una señal de respuesta por el puerto B. Del mismo modo, en una lectura de datos, el maestro envía la dirección que quiere leer por el puerto AR y el esclavo responde con el dato leído por el puerto R. En la Figura 2.8 se observa una transacción de escritura completa con AXI4-LITE. Es importante destacar que existen otros puertos de control opcionales que se pueden incorporar.


Figura 2.8: Diagrama temporal de escritura en conexión AXI-LITE

Inicialmente, el maestro coloca la dirección de escritura en el puerto AWADDR y activa la señal AWVALID en espera que el esclavo active la señal AWREADY. Luego que ambas señales AWVALID y AWREADY están activas en alto, se confirma la recepción de la dirección de escritura por parte del esclavo. A continuación, el maestro coloca el dato en WDATA, activa la señal WVALID y la señal WLAST indicando que es el último dato a enviar. Nuevamente, el esclavo activa la señal WREADY en el momento que puede recibir datos y se asegura que la transferencia de dato fue exitosa. Por último, el maestro activa la señal BREADY indicando que esta listo para recibir la confirmación de la escritura de dato. El esclavo envía una señal por el puerto BRESP informando que la tarea se realizó en forma exitosa y activa la señal BVALID. En todas las etapas de la transferencia, el bloque que envía información activa la señal *Valid* correspondiente, y el bloque que recibe la información activa la señal *Ready*.
La interfaz AXI4-STREAM es una conexión unidireccional punto a punto para enviar una gran cantidad de datos. Utiliza menos puertos de conexión respecto a otras interfaces AXI4 como se observa en la Figura 2.9.
El maestro inicia la transferencia de datos a través del puerto *TDATA* y activa en alto el puerto *TVALID*. Al momento en que el esclavo activa el puerto *TREADY* se realiza la tranferencia de los datos. El puerto *TLAST* indica el último dato de la tranferencia. Un puerto adicional denominado *TKEEP* es utilizado por el maestro para marcar los datos que deben ser recibidos por el esclavo. En la Figura 2.10 se observa el diagrama temporal

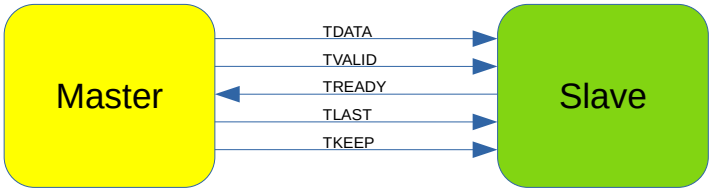


Figura 2.9: Puertos AXI4-STREAM

de una conexión AXI-STREAM.

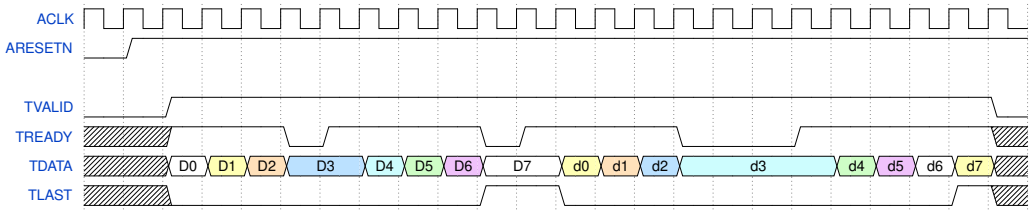


Figura 2.10: Diagrama temporal en conexión AXI-STREAM[1].

En esta conexión, el maestro envía el flujo de datos mientras el esclavo mantiene activa la señal *TREADY*. Al momento que se deshabilita esa señal, el maestro mantiene el dato en el puerto *TDATA* hasta una nueva habilitación de la señal *TREADY*. En la Figura 2.10 se destaca también el puerto *TLAST* para indicar el último dato de la transferencia y la señal *TVALID* que permanece activa en todo momento.

[1]Extraído de https://zipcpu.com/dsp/2020/04/20/axil2axis.html

## 2.2. Lenguajes HDL y HLL para FPGA

Los HDL son lenguajes desarrollados en la década del '80 para describir, simular y sintetizar hardware en FPGA. Hasta ese momento, sólo existían lenguajes de dominio específico y captura esquemática para desarrollar sistemas digitales (Martin y Smith, 2009) (Gurel, 2016). Desde entonces, Verilog y VHDL se han consolidado como los principales HDL para FPGA (Harris y Harris, 2013).
Verilog fue desarrollado por la compañía Gateway Design Automation en 1984 como un lenguaje propietario para simular funcionalidades lógicas. Luego, Cadence adquirió Gateway y en 1990 Verilog se transformó en un estándar abierto bajo el control de Open Verilog International. En la actualidad este lenguaje es conocido como SystemVerilog (IEEE STD 1800-2009) (Harris y Harris, 2013).
Por otro lado, VHDL fue desarrollado en 1981 por el Departamento de Defensa estadounidense para describir y documentar la estructura y comportamiento de un sistema. Luego, se utilizó para simulaciones y finalmente para la síntesis de circuito digitales. En 1987 VHDL se transformó en un estándar IEEE y sufrió numerosas modificaciones desde entonces (Harris y Harris, 2013).
Si bien Verilog y VHDL fueron desarrollados bajo principios similares, su sintaxis es diferente siendo este último más verboso y propenso a errores respecto a Verilog. Independientemente del HDL elegido, la descripción de un sistema digital presenta elementos en común:

- Se debe especificar la interfaz del sistema, constituida por los puertos de entrada y salida, que permitirá la conexión con otros dispositivos.
- Se debe especificar el comportamiento que tendrá el sistema.
- Se deben especificar las señales de tiempo involucradas en el sistema.
- Se debe simular el diseño configurado. Esto se realiza al estimular el sistema con señales de entrada y analizar el comportamiento de las señales de salida.

La complejidad de los HDL demandó nuevas técnicas de programación que permitieran al desarrollador trabajar en niveles mayores de abstracción. Es por ello que desde principios de este siglo, algunas compañías como Synopsis, Cadence y Menthor Graphics comenzaron a ofrecer HLS, herramientas de desarrollo que utilizan de entrada algún HLL como C, C++, SystemC, para obtener como salida el código en HDL (Escobar y cols., 2016). La popularidad del lenguaje C (o cualquiera de sus variantes) expandió el uso de las FPGA.

Por otro lado, la invención de SoC permitió implementar sistemas híbridos entre software/hardware. En general, en estos sistemas la solución se implementa por software en CPU y se recurre a la FPGA para resolver por hardware las secciones de programa que necesitan mayor performance (FPGA usado como co-procesador). Las herramientas HLS permiten diseñar la funcionalidad del sistema con lenguajes de alto nivel tanto para el software embebido como para la lógica del hardware específico. De esta forma, se pueden explorar diferentes alternativas software/hardware con características únicas entre uso de recursos, consumo energético y rendimiento. Esto es posible porque el algoritmo en HLL para FPGA permite el uso de compiladores estándares de C/C++ sin realizar modificaciones al código (Cong y cols., 2011).
Como contrapartida a las ventajas de HLL, el lenguaje C tiene tipos de datos pre-definidos (int, long, char, etc) y no dispone de construcciones nativas para definir tipos de datos con longitud de bits específicas. Además, los HLL fueron creados para ejecutar algoritmos secuenciales que no necesitan especificar jerarquías de sentencias o tiempos de ejecución. En desarrollos hardware, el algoritmo se traslada a los bloques de funciones y a la configuración de las rutas de conexiones entre módulos. La naturaleza en estos sistemas es concurrente, y las jerarquías, tiempos y sincronización de tareas se deben especificar en el código (Ren, 2014).
Otra limitación que presentan los HLL para sintetizar hardware es el manejo de memoria. Una CPU dispone de una memoria para almacenar el programa que se ejecuta en forma secuencial, además de la memoria de datos. En las FPGA, la memoria son numerosos bloques heterogéneos distribuidos por todo el silicio. Debido a ello, las construcciones complejas en HLL como punteros, manejo de memoria dinámica, recursión o polimorfismo no tienen implementación en hardware (Edwards, 2006).
Para superar estas deficiencias, las HLS han incorporado extensiones e impuesto restricciones al lenguaje. Las extensiones se incorporan como librerías y directivas de compilador para especificar concurrencia, tiempos de ejecución, optimizaciones de diseño, entre otros. Como ejemplos se mencionan Handel-C que permite especificar en forma explícita las restricciones temporales, o SystemC que permite especificar eventos según los flancos en la señal de reloj. Por otro lado, la restricción más generalizada entre las HLS es respecto a las construcciones dinámicas. La mayoría de las HLS no permiten manejo de memoria dinámica o no lo aconsejan (Kapre y Bayliss, 2016) (Cong y cols., 2011).
Si bien las HLS han podido solventar muchas deficiencias de HLL, no hay consenso en las extensiones y restricciones que cada una de ellas incorpora. Las implementaciones hardware se tornan muy dependientes de la herramien-

ta de desarrollo y no permite la portabilidad hacia otro hardware (Cong y cols., 2011).
Con las HLS, el algoritmo se reduce a especificar la funcionalidad del diseño, dejando a cargo de las herramientas de alto nivel aspectos muy importantes como la definición de la interfaz y las señales de control. La integración posterior del diseño hacia otros sistemas se torna más compleja.
A pesar de ello, los desarrollos de alto nivel en FPGA están ganando popularidad. Es así que existen herramientas HLS comerciales muy utilizadas como Catapult-C de Calypto (desarrollada por Mentor Graphics), Cynthesizer de Forte Design Systems, Synphony C de Synopsys, Impulse-C de Impulse Accelerated Technologies, C-to-silicon de Cadence, o Vivado HLS (Ex AutoPilot) de la empresa Xilinx, por mencionar algunas. También existen herramientas HLS desarrolladas en el ámbito académico, como LegUp de la Universidad de Toronto, Bambu de Politécnico de Milán, o DWARV de la Universidad de Tecnología Delft, que son muy utilizadas también (Nane y cols., 2016).
Una mención especial merece OpenCL (Open Computing Language) desarrollado por la empresa Apple en colaboración con AMD, IBM, Intel y NVIDIA que luego fue trasladada al Grupo Khronos. OpenCL fue lanzada en 2009 como un estándar abierto de programación paralela para homogeneizar la utilización de CPU multinúcleos, GPU, DSP, FPGA y otros dispositivos de cómputo discretos bajo una misma plataforma. Esto permitió a los programadores portar sus aplicaciones entre distintas tecnologías de hardware (Munshi, 2009). En la actualidad, tanto Intel (Altera) como Xilinx dan soporte a OpenCL para sus placas.

## 2.3. Métricas para Evaluación de Soluciones HDL y HLL

Esta sección se divide en tres subsecciones que son Métricas de Rendimiento, Uso de Recursos y Esfuerzo de Programación y Productividad. La primera subsección describe las métricas utilizadas para medir rendimiento de una aplicación en FPGA. La segunda subsección detalla el uso de recursos informado por la herramienta de desarrollo Vivado de Xilinx. La tercera subsección define productividad de diseño y explica los parámetros involucrados.

### 2.3.1. Métricas de Rendimiento

El rendimiento (*performance*) en una aplicación implementada en FPGA se puede medir por diferentes métricas, entre las que se pueden mencionar latencia (*latency*), frecuencia de reloj, *throughput* y tiempo de ejecución.

**Latencia** es la cantidad de ciclos de reloj que se necesitan para completar una instrucción o para obtener un resultado al completar un conjunto de instrucciones, dependiendo del caso. Se utiliza tanto en procesadores como en FPGA.
Como se observa en la Figura 2.11, un procesador ejecuta una instrucción en un orden específico que incluye:

- Búsqueda de instrucción (IF, Instruction Fetch)
- Decodificación de instrucción (ID, Instruction Decode)
- Ejecución de instrucción (EXE, Execution)
- Operaciones de memoria (MEM, Memory operations)
- Almacenamiento de resultados (WB, Write Back)

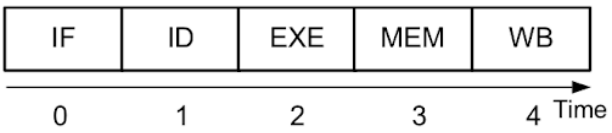


Figura 2.11: Ciclo de instrucción de un procesador[2].

El ciclo de cualquier instrucción en un procesador presenta esa secuencia de pasos debido a la arquitectura fija de los CPU, que utiliza los mismos elementos (ALU, Memoria, Buses, etc). Si se considera que cada etapa del ciclo de instrucción demora 1 ciclo de reloj, una instrucción demoraría 5 ciclos de reloj en ejecutarse (latencia de 5 ciclos de reloj por instrucción).
En una FPGA, no hay una arquitectura fija para ejecutar todas las aplicaciones, sino que el algoritmo configura un circuito específico. Es por ello que, en general, sólo utiliza la etapa de ejecución. A continuación en la Figura 2.12 se observa el ciclo de ejecución de una aplicación en FPGA. La latencia en esta aplicación es 1 ciclo de reloj.
**Frecuencia de reloj** se define como la inversa al mayor tiempo que demora en viajar una señal entre dos registros, y es una restricción de diseño. En la Figura 2.13 se observa el registro fuente que emite la señal, y el registro destino que recibe la señal. Si se considera que cada bloque (A, B, C, D, E) demora 2 ns en realizar su función, el tiempo total para completar un ciclo es 10 ns. En esas condiciones, la frecuencia máxima de operación para esa

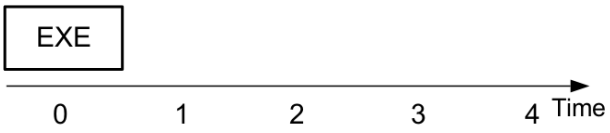


Figura 2.12: Ciclo de ejecución en una FPGA[2].

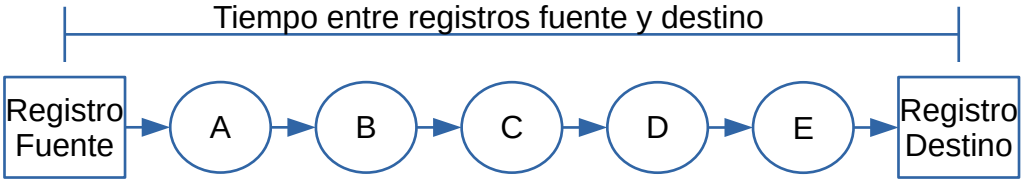


Figura 2.13: Envío de señal entre dos registros en una FPGA sin utilizar *pipelining*

aplicación será 100 MHz (1/10 ns) y no podrá satisfacer los requerimientos de una aplicación que necesite mayor velocidad de ejecución.
Una técnica muy empleada para mejorar el rendimiento en FPGAs es *Pipelining*, la cual consiste en solapar la ejecución de tareas para aumentar el paralelismo en el diseño. Esto se consigue al introducir registros intermedios con el fin de descomponer bloques computacionales grandes en pequeños segmentos. Los registros que se insertan disminuyen el tiempo de viaje de señal entre dos registros, por lo que se aumenta la frecuencia de reloj. Por ejemplo, si ahora se considera que el tiempo de viaje entre dos registros es de 2 ns, la frecuencia de reloj de la aplicación aumenta a 500MHz (1/2 ns). A continuación en la Figura 2.14 se observa el mismo ejemplo pero aplicando la técnica de *pipelining* al incorporar registros intermedios entre los bloques A, B, C, D y E.

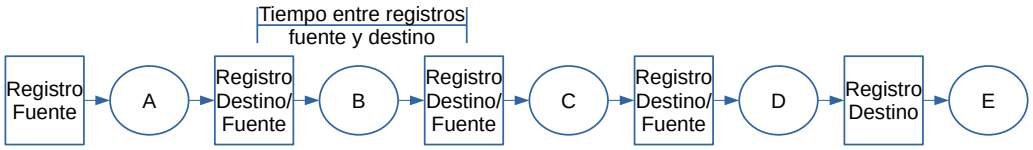


Figura 2.14: Envío de señal entre dos registros en una FPGA al utilizar *pipelining*.

La diferencia en la ejecución de tareas en una función al aplicar la técnica de solapamiento se observa en la Figura 2.15. La función sin esta técnica

[2]Extraído de "Introduction to FPGA Design with Vivado High-Level Synthesis (UG998)", 2019.

(izquierda) permite una nueva lectura de datos (RD) sólo cuando finalizan las tareas de comparación (CMP) y escritura (WD). En cambio, para la misma función con solapamiento (derecha), cada tarea (RD, CMP o WR) puede volver a ejecutarse independientemente del resto de ellas.

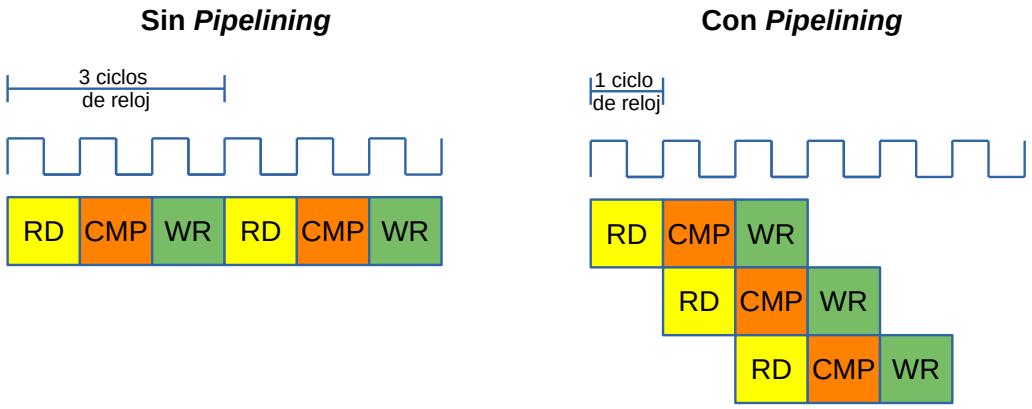


Figura 2.15: Ejecución de tareas. Izq: Función sin *pipelining*. Der: Función con *pipelining*.

La técnica de solapamiento permite aumentar la frecuencia de operación pero también incrementa la latencia, por lo que es un compromiso entre esos dos parámetros que se debe evaluar al implementar un sistema.
***Throughput*** se define como la cantidad de ciclos de reloj que necesita la lógica de procesamiento para aceptar una nueva entrada de datos y resulta favorecida por el solapamiento de tareas. Como se observó en el Figura 2.13, el circuito sin solapamiento puede recibir una nueva entrada de datos cada 10 ns. Esto se debe a que se necesitan resolver todas las regiones A, B, C, D, y E antes de recibir nuevos datos. Al aplicar solapamiento, las regiones están separadas por registros, por lo que se pueden ejecutar todas ellas en simultáneo.
**Tiempo de ejecución** es el tiempo que demora un sistema en realizar todas las tareas y depende de la latencia del sistema y de la frecuencia de operación.

### 2.3.2. Uso de Recursos

La complejidad de un sistema digital que se implementa en FPGA tiene una relación directa con la cantidad de recursos utilizados. Esto constituye una restricción cuando el hardware no dispone de los recursos suficientes para implementar un sistema de gran complejidad. El programador de FPGA busca optimizar su aplicación al demandar el menor consumo de recursos

posible que cumpla con los requerimientos del sistema y, al mismo tiempo, permita la incorporación futura de otros módulos al sistema.
La herramienta Vivado de Xilinx informa el uso de recursos de la aplicación en porcentaje, respecto al total de recursos disponibles. La Figura 2.16 muestra los recursos informados por Vivado.

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| Expression | - | - | 0 | 17 |
| FIFO | - | - | - | - |
| Instance | - | 1 | 17920 | 17152 |
| Memory | - | - | - | - |
| Multiplexer | - | - | - | - |
| Register | - | - | 7 | - |
| Total | 0 | 1 | 17927 | 17169 |
| Available | 650 | 600 | 202800 | 101400 |
| Utilization (%) | 0 | ~0 | 8 | 16 |

Figura 2.16: Uso de recursos en una FPGA desarrollada en Vivado[3].

### 2.3.3. Esfuerzo de Programación y Productividad

Las herramientas HLS son creadas para disminuir la brecha entre la creciente complejidad de los sistemas y la productividad de los programadores para desarrollarlos. La productividad es la relación que existe entre la calidad de los resultados obtenidos y el esfuerzo empleado para tal fin. Es un parámetro difícil de medir por la subjetividad de sus variables.
Entre los indicadores más sencillos para medir esfuerzo de programación y productividad, se encuentran la cantidad de líneas de código fuente (SLOC, source lines of code) (Bhatt, Tarey, y Patel, 2012) y el tiempo de desarrollo. Esta última incluye el tiempo empleado para realizar el diseño y las pruebas de validación hasta obtener un sistema funcional. La ventaja principal de ambas métricas es la simpleza de cálculo, aunque no reflejan la complejidad

[3]Extraído de "Vivado Design Suite User Guide: Getting Started (UG910)", 2018.

del sistema debido a que dependen de la experiencia del programador con el lenguaje y las herramientas de desarrollo.
Una de las propuestas más completas para evaluar la productividad de diseño se explica en (Pelcat y cols., 2016). El autor utiliza métricas para evaluar la eficiencia del diseño y la calidad de la implementación obtenida.
Para determinar la eficiencia del diseño, el autor utiliza el costo de ingeniería no-recurrente (NRE, non-recurring engineering) de diseño y de verificación. El NRE es el costo de producir nuevo código y está relacionado con la experiencia del programador. Se expresa en unidades de tiempo. Para comparar dos aplicaciones en HDL y HLL el autor calcula la ganancia en NRE como:

$$G_{NRE} = \frac{tHDL_{design} + tHDL_{verific}}{tHLS_{design} + tHLS_{verific}} \tag{2.1}$$

Las variables que están en el numerador de la fracción corresponden a los tiempos de diseño y verificación de la aplicación con lenguaje HDL. En el denominador de la fracción se encuentran las mismas variables correspondientes al diseño con el uso de HLS.
La $G_{NRE}$ sólo brinda información de los tiempos de desarrollo empleados pero no informa sobre la calidad de la aplicación. Para ello, se utiliza otra métrica que expresa la calidad de los resultados ($QoR$, quality of results) en función de la pérdida de calidad en el diseño ($L_Q$ , quality loss).
$L_Q$ se expresa como una suma ponderada del periodo de operación, los recursos utilizados y la latencia del sistema. Se normaliza cada variable de acuerdo a las limitaciones del diseño. La fórmula para el cálculo de $L_Q$ se muestra a continuación.

$$L_Q = \frac{a_1.lutHLS_n + a_2.regHLS_n + a_3.ramHLS_n + a_4.dspHLS_n + a_5.latHLS + a_6.prdHLS}{a_1.lutHDL_n + a_2.regHDL_n + a_3.ramHDL_n + a_4.dspHDL_n + a_5.latHDL + a_6.prdHDL} \tag{2.2}$$

Los primeros 4 términos del numerador corresponden a los valores de recursos utilizados en el diseño HLS (LUT, Registros, Ram y DSP). Cada valor se normaliza respecto al total de recursos que dispone la placa FPGA. Los últimos 2 términos corresponden a las restricciones del diseño como latencia y período de operación máximos permitidos (por ej. el período máximo de operación para asegurar una tasa de transferencia). Los valores del denominador son homólogos y corresponden a la implementación en HDL. Los coeficientes $a_i$ se utilizan para favorecer alguna característica en particular.
La productividad de diseño ($P_D$, desing productivity) tiene en cuenta tanto la pérdida de calidad de la aplicación como la ganancia NRE. Se define como:

$$P_D = \frac{G_{NRE}}{L_Q} \tag{2.3}$$

Un diseño en HLS se considera exitoso si su PD es mayor a 1, por el contrario un valor menor a 1 expone deficiencias de HLS respecto a HDL.
Por último, existen otros métodos para estimar esfuerzos y costos de desarrollo software que son ampliamente estudiados en Ingeniería de Software pero que no se aplican en el ámbito de las FPGAs. Entre los métodos más utilizados se puede mencionar CoCoMo (Constructive Cost Model), complejidad ciclomática, análisis de punto de función, entre otros (Jones, 2007).

## 2.4. Procesamiento de Imágenes. Detección de Contornos

Una imagen digital se define como un conjunto de puntos o píxeles dispuestos en una matriz de dos dimensiones ($x$ e $y$). El procesamiento digital de imágenes consiste en la aplicación de técnicas y algoritmos computacionales complejos sobre una imagen para extraer información de ella o mejorar sus características (Pavan Kumar, 2019).
Entre estos algoritmos, se pueden mencionar los relacionados a detección de bordes o contornos, ampliamente utilizados como etapa previa a otros algoritmos de procesamiento (Nausheen, Seal, Khanna, y Halder, 2018). Las técnicas de detección de bordes identifican zonas en la imagen donde los píxeles presentan cambios abruptos en intensidad o niveles de grises, permitiendo segmentar una imagen en regiones discontinuas. Como resultado, se reduce significativamente la cantidad de datos en la imagen sin alterar sus propiedades estructurales (Rashmi, Kumar, y Saxena, 2013).
Al momento de detectar contornos, no existe un único método. Los operadores Laplacianos (o de segundo orden) permiten detectar bordes en cualquier orientación pero tienen una alta tasa de detección de falsos positivos y son muy sensibles al ruido. Por otro lado, se encuentran los algoritmos basados en gradientes (o de primer orden), que se pueden considerar menos sensibles al ruido respecto a los operadores Laplacianos. Sin embargo, no detectan bordes diagonales y generan contornos muy gruesos. Dentro de los algoritmos basados en gradiente, se destaca el operador Sobel por tener mejor inmunidad al ruido respecto a otras alternativas como los operadores Roberts-Cross y Prewitt (Rashmi y cols., 2013).

### 2.4.1. Filtro Sobel

El filtro Sobel es un método de detección de contornos de primer orden o basado en gradientes que se aplica a imágenes en escala de grises (Nausheen

y cols., 2018) para detectar sus componentes de alta frecuencia. Un componente de alta frecuencia se genera por cambios abruptos de intensidad en píxeles contiguos de la imagen, los cuales se corresponden con un borde. En la Figura 2.17 se observa la imagen *kodim21*[4] a la que se han detectado sus bordes por medio de un filtro Sobel.

Figura 2.17: Filtro Sobel aplicado a imagen Kodim21. Izq: Imagen original. Der: Imagen resultante.

El vector gradiente detecta esos cambios por medio de las derivadas parciales en los ejes horizontal y vertical. La Ecuación 2.4 define al vector gradiente por sus componentes horizontal $G_x$ y vertical $G_y$ (Chaple y Daruwala, 2014).

$$\nabla f = \begin{pmatrix} G_x \\ G_y \end{pmatrix} \tag{2.4}$$

El módulo o magnitud del vector gradiente se denomina fuerza de borde y expresa la tasa de cambio por unidad de distancia. De este modo, una mayor variación de intensidad en píxeles vecinos genera un aumento en la magnitud del gradiente que se computa como un borde.
La magnitud del gradiente se define por la Ecuación 2.5.

$$|\nabla f| = \sqrt{{G_x}^2 + {G_y}^2} \tag{2.5}$$

Las implementaciones del filtro Sobel emplean dos máscaras de convolución $M_x$ y $M_y$ para obtener las componentes horizontal y vertical del vector gradiente, respectivamente. Las máscaras de convolución se muestran en la Figura 2.18.
Desde un punto de vista matemático, las componentes del gradiente se obtienen de la multiplicación de cada máscara con la imagen. El proceso se realiza

[4]Extraído de https://github.com/lemire/kodakimagecollection

$$M_x = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad M_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Figura 2.18: Máscaras de convolución Sobel

de izquierda a derecha y de arriba hacia abajo hasta recorrer la imagen completa. En la Figura 2.19 se puede observar el proceso de convolución entre una imagen y una máscara del filtro Sobel.

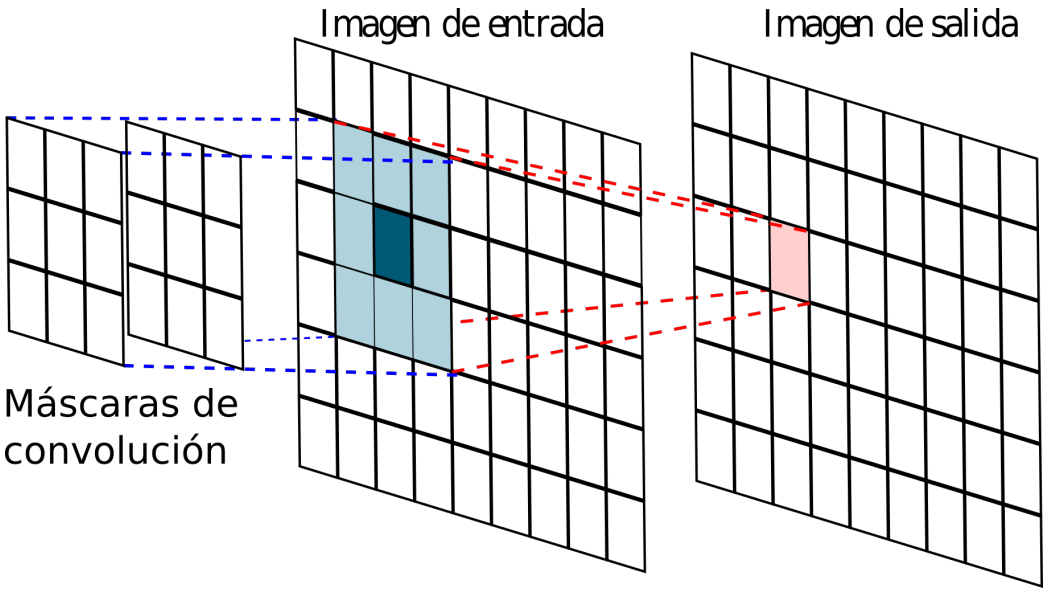


Figura 2.19: Proceso de convolución Sobel

### 2.4.2. Variantes de Sobel

Existen distintas alternativas del filtro Sobel que tienen por objetivo variar el costo computacional y/o la calidad de imagen obtenida. Un método popular para disminuir la carga computacional consiste en aproximar la magnitud del gradiente por la suma de los valores absolutos de sus componentes, evitando así recurrir a operaciones en punto flotante. Mediante esta alternativa, varios trabajos han logrado reducir el tiempo de procesamiento (Chaple y Daruwala, 2014; Nosrat y S. Kavian, 2012; Sanduja y Patial, 2012). En la Ecuación 2.6 se presenta la fórmula de aproximación para el cálculo de la magnitud del

vector gradiente.

$$|\nabla f| \approx |G_x| + |G_y| \tag{2.6}$$

Por otra parte, también existen implementaciones arbitrarias de Sobel, como la usada por el editor de imágenes GIMP [5] o la librería de visión por computadora OpenCV [6]. GIMP computa la magnitud del gradiente empleando la fórmula convencional (expresada en la Ecuación 2.5) pero además dividiendo su resultado por un número real fijo (5.66). Por su parte, OpenCV emplea un enfoque similar al de GIMP, sólo que usando la formula de aproximación (expresada en la Ecuación 2.6) y dividiendo por un número entero fijo (2).

## 2.5. Resumen

En este capítulo se describieron los inicios de la tecnología FPGA, los elementos principales que la componen y su evolución a lo largo de los años hasta alcanzar a las arquitecturas heterogeneas de la actualidad denominadas SoC. Estas plataformas que combinan CPUs y FPGA gozan de considerable aceptación actualmente.
Luego se describieron los lenguajes tradicionales HDL para desarrollar hardware en FPGA, destacados por la posibilidad de definir tipos de datos con longitudes de bits particulares y de especificar los tiempos de ejecución en las tareas pero con la desventaja de que son difíciles, verbosos y propenso a errores. También se incluyeron enfoques de desarrollo hardware de alto nivel para FPGA con el uso de HLL que permiten disminuir la complejidad y el tiempo de diseño pero con limitaciones como la imposibilidad de definir jerarquías de sentencias o datos con longitudes de bits específicas.
A continuación, se presentaron métricas para evaluar un sistema en FPGA que incluyen indicadores de rendimiento, uso de recursos y esfuerzo de programación. Por último, se explicaron temas de bases útiles para el entendimiento de este trabajo como procesamiento de imágenes y técnicas de detección de contornos (en particular, en el operador Sobel).

[5] GNU Image Manipulation Program (GIMP). https://www.gimp.org/
[6] Open Source Computer Vision Library (OpenCV). https://https://opencv.org/

# Propuesta

Este capítulo inicia en la sección Sistema Sobel de Procesamiento de Imágenes en FPGA donde se describen todos los módulos que componen el sistema de procesamiento de imagen. Contiene tres subsecciones que explican el funcionamiento de cada bloque del núcleo de procesamiento. Luego, las secciones Núcleo de Procesamiento de Imagen HLS y Núcleo de Procesamiento de Imagen HDL detallan el diseño, funcionamiento e implementación de los bloques que forman el núcleo de procesamiento en los lenguajes HLL y HDL. Finalmente, se presenta un Resumen del capítulo.

## 3.1. Sistema Sobel de Procesamiento de Imágenes en FPGA

El sistema de procesamiento propuesto en este trabajo fue diseñado para una plataforma heterogenea SoC y detecta los bordes de imágenes a color en formato BMP. Para operar, el sistema lee las imágenes desde una memoria microSD y aplica dos técnicas de procesamiento. La primer técnica consiste en la conversión de cada píxel a color por otro píxel en escala de grises, y la segunda técnica detecta el contorno de las imágenes por medio del algoritmo Sobel. Las imágenes resultantes se almacenan en memoria microSD.
El sistema utiliza bloques de funciones que pertenecen al Sistema de Procesamiento y a la Lógica Programable. La estructura del sistema propuesto se observa en la Figura 3.1.

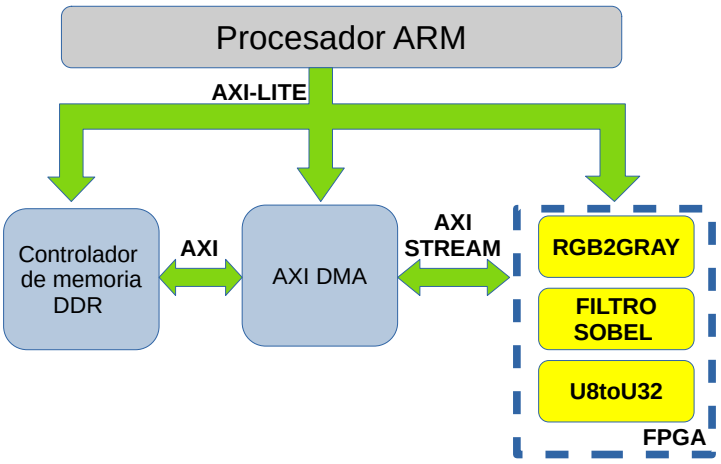


Figura 3.1: Sistema de procesamiento de imagen

El *procesador ARM*, el *AXI DMA* y el *controlador de memoria DDR* pertenecen al Sistema de Procesamiento del SoC. Estos bloques no realizan

procesamiento sobre las imágenes sino que llevan a cabo tareas complementarias para el funcionamiento del sistema. Entre ellas, se destaca la función del procesador para configurar los bloques de todo el sistema y ejecutar la aplicación de prueba.
Por otro lado, el núcleo de procesamiento (en color amarillo) está formado por tres bloques sintetizados en la Lógica Programable del SoC. Estos bloques son *RGB2GRAY*, *FILTRO SOBEL* y *U8toU32*, y tienen por objetivo acelerar por hardware la detección de contornos en imágenes.
Los primeros dos bloques del núcleo de procesamiento realizan operaciones sobre los píxeles de la imagen al convertirla de color a escala de grises y luego detectar los contornos en la imagen resultante. El último bloque no realiza técnica de procesamiento sino que permite recuperar el tamaño original de la imagen.
En la Figura 3.1 también se observan, en color verde, las interfaces AXI4 que comunican todos los bloques del sistema. Entre ellas, las interfaces AXI4-STREAM se utilizan para transmitir los pixeles de las imágenes entre los distintos bloques de funciones y las interfaces AXI4-LITE para configurar los módulos de los dispositivos.
Las tareas que realizan los módulos del sistema de procesamiento de imagen se observan en la Figura 3.2.

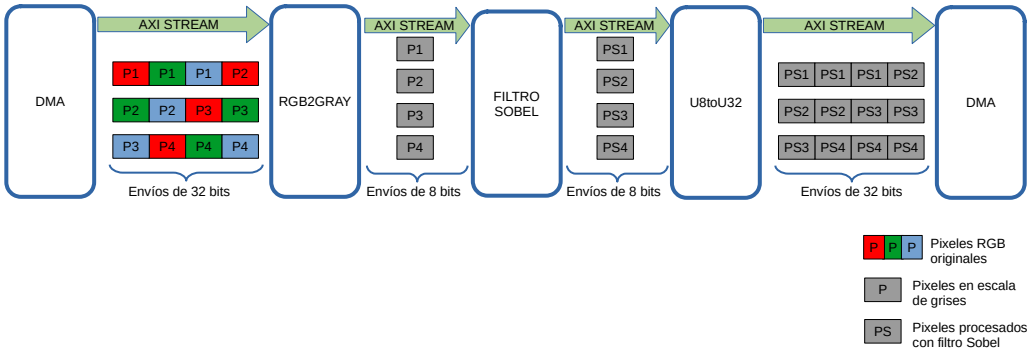


Figura 3.2: Funcionamiento del sistema

El bloque DMA accede a la memoria microSD para la lectura de la imagen y envía el flujo de píxeles por medio de una interfaz AXI4-STREAM hacia el primer bloque del núcleo de procesamiento RGB2GRAY. Allí, se aplica la primera técnica de procesamiento de imagen donde los pixeles a color RGB se convierten a escala de grises al promediar el valor de sus tres canales. Esto reduce el tamaño de la imagen, dado que cada píxel ocupa 8 bits en lugar de los 24 bits requeridos originalmente.
Los píxeles resultantes se envían por interfaz AXI4-STREAM desde RGB2GRAY

a FILTRO SOBEL, el segundo bloque del núcleo de procesamiento. En FILTRO SOBEL, se realizan los cálculos definidos por el operador Sobel para obtener los contornos de la imagen. Los nuevos píxeles son enviados por medio de una interfaz AXI4-STREAM hacia el último bloque del núcleo de procesamiento denominado U8toU32. En este bloque se recupera el tamaño de imagen original al expandir cada píxel de 8 bits a su equivalente en 24 bits. Finalmente, los pixeles expandidos son enviados desde U8toU32 hacia el DMA por medio de una interfaz AXI4-STREAM para ser almacenados en memoria. Cabe destacar que el bloque DMA recibe y envía palabras de 32 bits por lo que los módulos del núcleo de procesamiento que se comunican con el DMA (RGB2GRAY y U8toU32) tienen interfaces compatibles con esa cantidad de bits. A su vez, dentro del núcleo de procesamiento las comunicaciones se realizan en bytes (8 bits) para simplificar el diseño.
Para llevar a cabo el estudio comparativo se realizaron dos implementaciones, en HDL y HLL, de cada bloque que compone el núcleo de procesamiento (RGB2GRAY, FILTRO SOBEL y U8toU32). En primer lugar, se realizaron pruebas sobre el sistema de procesamiento de imagen con el núcleo de procesamiento en HLL. Luego, los bloques en HLL fueron reemplazados por sus versiones HDL para su testeo.

### 3.1.1. Bloque RGB2GRAY

El bloque RGB2GRAY convierte píxeles a color de tres canales (RGB) por sus equivalentes en escala de grises, utilizando el método basado en la media aritmética (promedio de los tres canales RGB). Una vez completado el proceso de conversión, envía los píxeles resultantes al segundo bloque del núcleo de procesamiento, el FILTRO SOBEL.
En la Figura 3.3 se observa el funcionamiento del bloque RGB2GRAY, tanto en HDL como HLL. Cada ciclo de operación de RGB2GRAY consiste en la recepción de tres palabras consecutivas de 32 bits desde el DMA, seguido del cálculo de la media aritmética en cada píxel (RGB) y luego el envío consecutivo de cuatro píxeles de 8 bits hacia FILTRO SOBEL. Respecto a las tres palabras recibidas, la primera corresponde al pixel_1 (P1) más el canal R del pixel_2 (P2), la segunda contiene los canales GB del pixel_2 (P2) sumado a los canales RG del pixel_3 (P3), y la tercera recepción comprende al canal B del pixel_3 (P3) más el pixel_4 (P4).

### 3.1.2. Bloque FILTRO SOBEL

El FILTRO SOBEL es el bloque responsable de obtener los bordes de las imágenes. Recibe los píxeles en escala de grises desde RGB2GRAY y obtiene

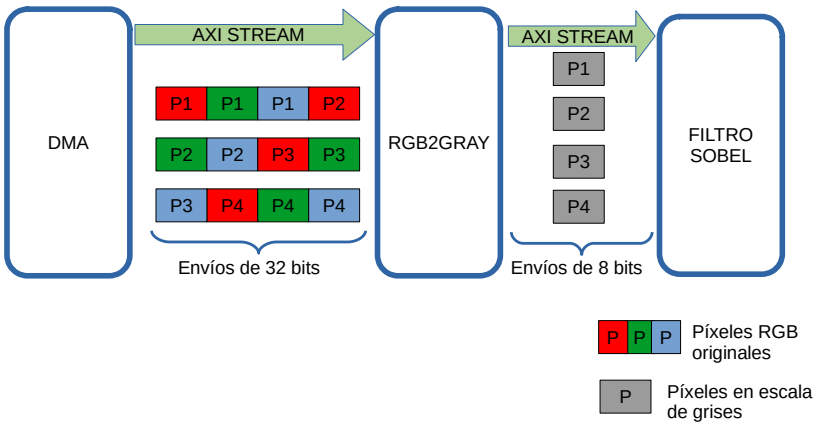


Figura 3.3: Funcionamiento del bloque RGB2GRAY

otros píxeles con los bordes de la imagen que son enviados a U8toU32. Este proceso se observa en la Figura 3.4.

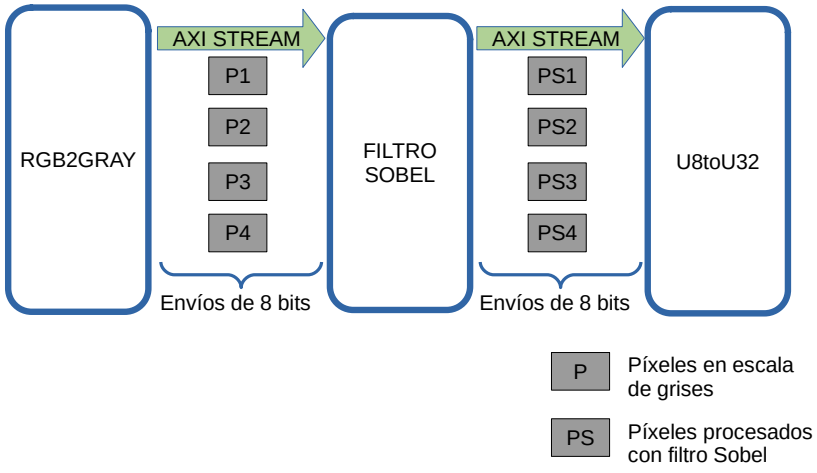


Figura 3.4: Funcionamiento del bloque FILTRO SOBEL

En este bloque no se realizan recepciones o envíos consecutivos. Cada píxel en escala de grises (P) que ingresa a FILTRO SOBEL genera otro píxel (PS) con los contornos de la imagen.
Para operar, el bloque utiliza una estructura de memoria denominada *ventana deslizante* que selecciona los píxeles a convolucionar con las máscaras Sobel horizontal y vertical. La ventana deslizante tiene una dimensión de 3×3 que se corresponde con las máscaras del filtro. Asimismo, las implementaciones Sobel en FPGA suelen utilizar una estructura de memoria adicional

denominada *buffers de línea* con el objetivo de reducir el consumo de memoria (Vallina, Kohn, y Joshi, 2012). Los buffers de línea almacenan filas completas de la imagen y permiten mantener el contexto de píxeles necesario para que el filtro pueda operar, evitando guardar la imagen completa. Dependiendo del diseño del filtro, se pueden utilizar más o menos buffers de línea, aunque en todos los casos resulta en una cantidad menor al tamaño de imagen. Las implementaciones Sobel desarrolladas en esta tesis utilizan dos buffers de línea para la versión HDL y tres para la versión HLL. Respecto al algoritmo de cómputo Sobel, ambas implementaciones utilizan la fórmula de aproximación 2.6 presentada en el Capítulo 2, Sección Procesamiento de Imágenes. Detección de Contornos.
Para comprender el ciclo de operación del FILTRO SOBEL, en la Figura 3.5 se observa una implementación con tres buffers de línea aunque esa cantidad puede variar, como se mencionó anteriormente. En la Figura 3.5 se encuentra de izquiera a derecha la imagen original en escala de grises, los tres buffers de línea (en color amarillo), y la imagen de contornos resultante. Sobre los buffers de línea se puede notar (en color rojo) la ventana deslizante de dimensiones 3×3, y el nuevo píxel (en color verde).

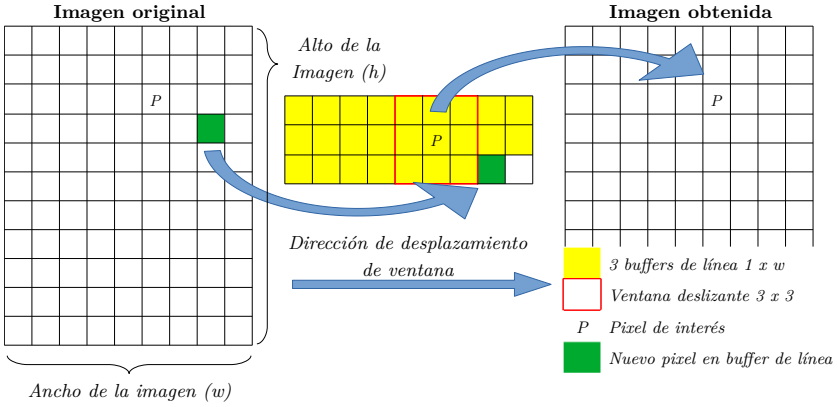


Figura 3.5: Implementación del bloque FILTRO SOBEL

El funcionamiento de FILTRO SOBEL se puede resumir en las siguientes acciones: en primer lugar, el bloque queda a la espera de un nuevo píxel, el cual es almacenado en el buffer de línea inferior. A continuación, los datos en buffer de línea son transferidos a la ventana deslizante para realizar la convolución con las máscaras Sobel. El valor resultante de esa operación es enviado para conformar la imagen de contornos. Este proceso continúa, desplazando la ventana deslizante de izquierda a derecha con cada nuevo píxel recibido hasta procesar toda la imagen.

### 3.1.3. Bloque U8toU32

El bloque U8toU32 es responsable de recuperar el tamaño original de la imagen. Para ello, recibe en forma consecutiva cuatro píxeles de 8 bits desde el FILTRO SOBEL, luego los expande a 24 bits y finalmente realiza tres envíos consecutivos de palabras de 32 bits hacia DMA. Todo este proceso se observa en la Figura 3.6. Este bloque no realiza procesamiento sobre los píxeles de la imagen por lo que es el más sencillo de implementar.

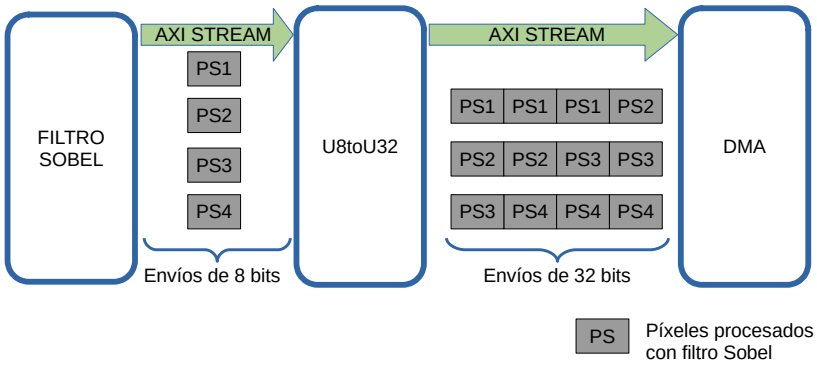


Figura 3.6: Funcionamiento del bloque U8toU32

## 3.2. Núcleo de Procesamiento de Imagen HLS

Cada bloque del núcleo de procesamiento en HLS se diseñó como una función en lenguaje C. En estas implementaciones, todos los detalles de bajo nivel quedaron a cargo de la herramienta HLS. En particular, las interfaces AXI4-STREAM y AXI4-LITE se implementaron por medio de directivas *INTERFACE*, el solapamiento de tareas se logró con la directiva *PIPELINE* y el acceso concurrente a los datos en memoria con *ARRAY PARTITION*. Estas directivas no son nativas del lenguaje C sino que son extensiones interpretadas por las herramientas HLS para optimizar las descripciones hardware. A su vez, se utilizaron los tipos de datos definidos *int* y *char* del lenguaje C.

### 3.2.1. RGB2GRAY HLS

Las operaciones del bloque RGB2GRAY implementado en alto nivel comienzan con la recepción consecutiva de tres palabras de 32 bits desde el DMA, que corresponden a cuatro píxeles RGB de la imagen original. En cada recepción, la palabra de 32 bits se almacena en una variable de tipo entero sin signo

denominada *data_rcv*. Luego, el contenido de *data_rcv* se divide en cuatro bytes que se guardan en un arreglo de tipo caracter sin signo denominado *data_rcv_u8*. Este arreglo se llena al almacenar los 12 bytes correspondientes a las tres recepciones. El bloque continúa con el cálculo del promedio de los tres canales para cada píxel y el envío de los cuatro valores resultantes hacia FILTRO SOBEL. En la Figura 3.7 se observa el pseudocódigo del bloque RGB2GRAY en HLS.

```
Para row=0 hasta Filas_Imagen
   Para col=0 hasta (Columnas_Imagen*3/4)
      data_rcv ← dato nuevo  //Recibo palabra de 32 bits
      rem ← col%3  //Obtengo resto de la división
      Para i=0 hasta 3
        data_rcv_u8 [rem*4+i]← (data_rcv >>8*i) & 0xFF //Divido la palabra en bytes
      Fin para
      Si rem = 2
         Para i=0 hasta 3
           //Obtengo el promedio de 3 bytes consecutivos
           data_sum ← ( data_rcv_u8[i*3] +  data_rcv_u8[i*3+1] +  data_rcv_u8[i*3+2])/3
           envío data_sum //Envío el promedio obtenido
         Fin para
       Fin si
   Fin para
Fin para
```

Figura 3.7: Pseudocódigo RGB2GRAY en HLS

Las operaciones de RGB2GRAY de alto nivel se realizan dentro de dos bucles anidados. El bucle externo itera en función de las filas de la imagen y el bucle interno respecto a la cantidad de columnas. En la Figura 3.7 también se observa la división de *data_rcv* en bytes, el almacenamiento de cada byte en el arreglo *data_rcv_u8*, el cálculo del promedio de tres bytes consecutivos almacenados en el arreglo y el envío de los promedios obtenidos.

### 3.2.2. FILTRO SOBEL HLS

El diseño del FILTRO SOBEL de alto nivel propuesto en este trabajo utiliza tres buffers de línea con el mismo ancho de la imagen dispuestos uno sobre otro y una ventana deslizante de dimensiones de 3×3 como se observa en la Figura 3.8. La imagen original se encuentra a la izquierda, los buffers de línea (en color amarillo) junto con la ventana deslizante (en color rojo) en el centro y la imagen generada a la derecha. Además, se incluyen las operaciones realizadas por el bloque.

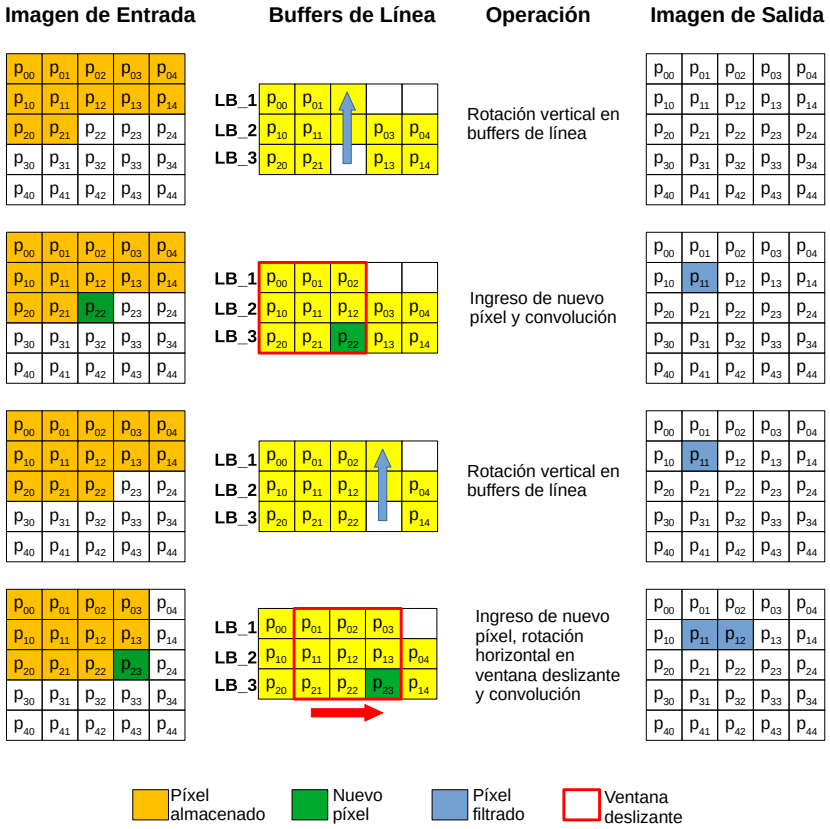


Figura 3.8: Funcionamiento del filtro Sobel en alto nivel

Cada píxel que ingresa al FILTRO SOBEL se almacena en el buffer de línea inferior. Para ello, antes de su almacenamiento se genera una rotación vertical en los buffers de línea. Esta acción produce el descarte del contenido en el buffer de línea superior, la transferencia de valores entre ellos y el posterior almacenamiento del nuevo píxel en el buffer de línea inferior. El nuevo píxel se almacena en la misma columna respecto a la imagen original. Asimismo, la ventana deslizante comienza a operar luego de recibido el primer píxel, desplazandose en forma lateral sobre los buffers de línea con cada nuevo dato hasta completar el ancho de la imagen.
De acuerdo a la funcionalidad del bloque y las características del lenguaje utilizado, los tres buffers de línea se codificaron como un arreglo bidimensional de $3 \times AnchoImagen$ elementos, y la ventana deslizante como otro arreglo de dimensiones 3×3. En la Figura 3.9 se observa el pseudocódigo para la implementación del filtro Sobel en alto nivel.
El bloque presenta dos bucles anidados en función de las dimensiones de la imagen y una espera hasta recibir un píxel nuevo. En ese momento se produ-

```
Para row=0 hasta Filas_Imagen
  Para col=0 hasta Columnas_Imagen
    Espero dato nuevo  //Espera hasta recibir nuevo píxel
    Para k=0 hasta 1
     line_buffer[k][col] ← line_buffer[k+1][col] //Rotación vertical LB
    Fin para
     line_buffer[2][col] ← dato_nuevo //Almaceno el nuevo valor en el LB inferior
      Para x=0 hasta 1
        Para y=0 hasta 1
          window[x][y] ←  window[x][y+1]  //Desplazo la ventana deslizante a la derecha
        Fin para
       window[x][2]  ← line_buffer[x][col]  //Almaceno la columna de LB en la ventana deslizante
      Fin para
       data_out ← convolución (window)  //Realizo la convolución
       envío data_out //Envío el pixel resultante de la convolución
  Fin para
Fin para
```

Figura 3.9: Pseudocódigo Filtro Sobel en HLS

ce la rotación vertical en los buffers de línea y el almacenamiento posterior del nuevo píxel en el buffer de línea inferior. También se produce el desplazamiento lateral en la ventana deslizante antes de realizar la convolución para enviar el resultado a U8toU32.
Respecto a las directivas de compilador, además de utilizar las directivas *INTERFACE* para sintetizar las interfaces AXI4, se emplearon dos directivas de rendimiento: la directiva *PIPELINE* para el solapamiento y ejecución concurrente de tareas, y *ARRAY_PARTITION* para la asignación de múltiples bloques de memoria a los arreglos, favoreciendo el acceso concurrente a los datos.

### 3.2.3. U8toU32 HLS

El bloque U8toU32 es el último de los módulos que componen el núcleo de procesamiento y su función es asegurar que la imagen generada por el núcleo de procesamiento tenga las mismas dimensiones que la imagen original.
El ciclo de operación de U8toU32 comienza con la recepción de cuatro píxeles consecutivos desde FILTRO SOBEL. Cada píxel, de 8 bits, se almacena en tres lugares consecutivos de un arreglo de tipo caracter denominado *data_rcv* con el fin de expandir el tamaño de imagen y compensar la reducción obtenida en el bloque RGB2GRAY. Luego de almacenar los cuatro píxeles en 12 lugares del arreglo *data_rcv*, el bloque realiza tres envíos consecutivos de palabras de 32 bits hacia el DMA para guardar la nueva imagen en memoria.
En la Figura 3.10 se observa el pseudocódigo del bloque U8toU32 en HLS. Del mismo modo que RGB2GRAY y FILTRO SOBEL, el funcionamiento

de U8toU32 se realiza dentro de dos bucles anidados en función de las dimensiones de la imagen. Se destaca la asignación de un puntero de enteros a la misma dirección del arreglo de caracteres *data_rcv* para simplificar los envíos de palabras de 32 bits.

```
//Asigno a un puntero entero la misma dirección del arreglo de caracteres
*intPtr ← (int*)data_rcv
Para row=0 hasta Filas_Imagen
   Para col=0 hasta Columnas_Imagen
     data_in← dato nuevo //Recibo pixel
     rem ← col%4 //Obtengo resto de la división
     //Almaceno el nuevo pixel en tres lugares de memoria consecutivos
     data_rcv[3*rem] ← data_in
     data_rcv[3*rem + 1] ← data_in
     data_rcv[3*rem + 2] ← data_in
     Si rem = 3
       Para i=0 hasta 2
         data_out ← *(intPtr + 1)
         envío data_out //Envío palabra de 32 bits
       Fin para
     Fin si
   Fin para
Fin para
```

Figura 3.10: Pseudocódigo U8toU32 en HLS

## 3.3. Núcleo de Procesamiento de Imagen HDL

Los bloques del núcleo de procesamiento en bajo nivel se describieron en VHDL. Cada implementación comenzó con una especificación de su funcionalidad para luego diseñar el hardware capaz de realizar esas tareas. En estas implementaciones, todos los detalles debieron ser diseñados en bajo nivel como la descripción de una Máquina de Estados Finita (MEF) responsable de sincronizar todas las tareas del bloque o la cantidad de bits en cada señal.
La mayor complejidad en los diseños de bajo nivel se evidenció al sintetizar las interfaces de comunicación AXI y el solapamiento de tareas. Para la descripción de las interfaces AXI4-STREAM y AXI4-LITE fue necesario definir en la entidad de cada bloque los puertos de entrada y salida involucrados, además de otras señales de control. Por otra parte, el solapamiento de tareas se logró con la incorporación de registros intermedios entre operaciones. Todo ello demandó una gran cantidad de pruebas de validación y reescritura de código hasta obtener un diseño funcional y eficiente, a diferencia de las versiones en HLL que se lograron con el uso de directivas.

### 3.3.1. RGB2GRAY HDL

La versión RGB2GRAY en HDL realiza las mismas tareas que la versión de alto nivel. Este bloque recibe tres palabras de 32 bits consecutivas desde el DMA para luego obtener los píxeles en escala de grises y finalmente realizar cuatro envíos consecutivos a FILTRO SOBEL.
La sincronización de tareas de RGB2GRAY se logró con la MEF que se observa en la Figura 3.11. La misma comienza en un estado denominado *START* donde se inicializan las señales. Luego, pasa al estado *RCV* donde espera hasta recibir las tres palabras que contienen los cuatro píxeles. Concluidas las recepciones, el bloque pasa al estado *AVG* donde se calcula el promedio de los tres canales de cada píxel y, finalmente, se envían los resultados obtenidos en el estado *SEND*. Con el fin de asegurar el correcto envío de datos se incluyen dos estados adicionales denominados *WAIT_READY* y *MISSED*. En el primero de ellos, el bloque queda en espera hasta que FILTRO SOBEL pueda recibir un nuevo píxel, mientras que el bloque pasa a *MISSED* cuando se lo envía pero FILTRO SOBEL no lo recibe y el píxel se pierde.

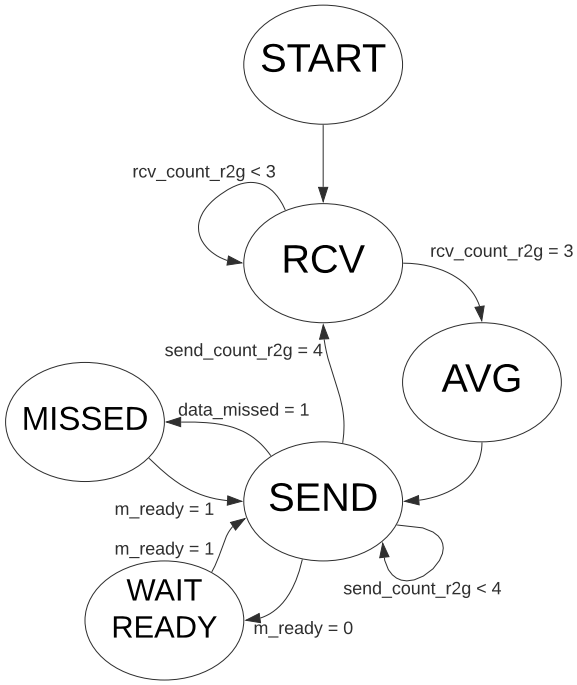


Figura 3.11: MEF bloque RGB2GRAY

### 3.3.2. FILTRO SOBEL HDL

El FILTRO SOBEL de bajo nivel tiene un funcionamiento similar respecto a la implementación de alto nivel. El mismo consiste en almacenar filas completas de la imagen en buffers de línea para luego transferir los píxeles a la ventana deslizante y realizar la convolución con las máscara Sobel. Sin embargo, la implementación en VHDL demandó sólo dos buffers de línea, a diferencia de la versión en HLL que necesitó tres. En esta implementación, cada buffer de línea se sintetizó como un bloque de memoria RAM con un ancho de palabra de 8 bits y una capacidad de almacenamiento de 1920 datos. A diferencia de otras propuestas que sintetizan los buffers de línea con registros de desplazamiento, esta arquitectura obtiene dos beneficios: por un lado aumenta la velocidad del sistema al permitir la lectura y escritura concurrente de datos (memoria RAM de doble puerto) y por otro lado se evita el uso excesivo de memoria distribuida en el FPGA (CLB y registros).
A su vez, la ventana deslizante se implementó con tres registros de desplazamiento de tres flip-flop cada uno. Nuevamente, una MEF es responsable de coordinar todas las tareas del bloque.

La descripción hardware de los buffers de línea y la ventana deslizante se observa en la Figura 3.12. Los bloques de memoria RAM que representan los buffers de línea se denominan LB_1 y LB_2 y se muestran en colores rojo y azul. Además, se observa la ventana deslizante conformada por los nueve flip-flop y uno adicional en color verde para almacenar el nuevo pixel. Los puertos de lectura de datos (Q) de LB_1 y LB_2, y el puerto de salida del flip-flop de nuevo píxel están conectados a la ventana deslizante y la proveen de píxeles. Con esta configuración, cada vez que se recibe un dato, se produce un desplazamiento y una nueva carga de píxeles en ventana deslizante. A su vez, los buffers de línea se conectan en cascada al unir el puerto de salida (Q) de LB_2 con el puerto de escritura (D) de LB_1. Esto permite que el píxel leído desde LB_2 luego sea almacenado en LB_1, logrando una rotación vertical semejante a la implementación de alto nivel. También se destaca la carga de píxeles en LB_2 desde el flip-flop de nuevo píxel.

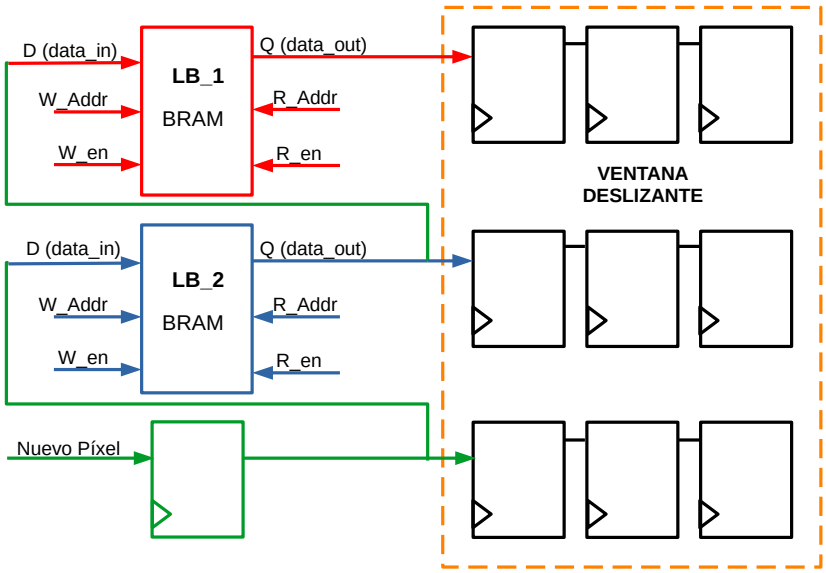


Figura 3.12: Síntesis de la implementación Sobel

Todas las operaciones de este bloque se realizan en cuatro pulsos de reloj y se observan en la Figura 3.13. En el primer ciclo de reloj, ingresa un nuevo píxel a FILTRO SOBEL que se almacena en un flip-flop. Al mismo tiempo, se ejecuta una operación de lectura en LB_2 y LB_1 donde los píxeles leídos se posicionan en sus puertos (Q). En el pulso de reloj siguiente se realiza una carga de datos en la ventana deslizante y se ejecuta una operación de escritura sobre los buffers de línea. Es así que la ventana deslizante recibe nuevos datos conformados por el último píxel ingresado al bloque más los datos leídos en

LB_1 y LB_2. A su vez, cada buffer de línea guarda el valor disponible en su puerto de escritura (D), es decir, LB_2 almacena el nuevo píxel y LB_1 hace lo mismo con el valor leído previamente en LB_2 (rotación vertical en buffers de línea). En el tercer ciclo de reloj se realiza la convolución entre las máscaras Sobel y los píxeles almacenados en la ventana deslizante para, finalmente, enviar el resultado obtenido en el pulso de reloj siguiente.

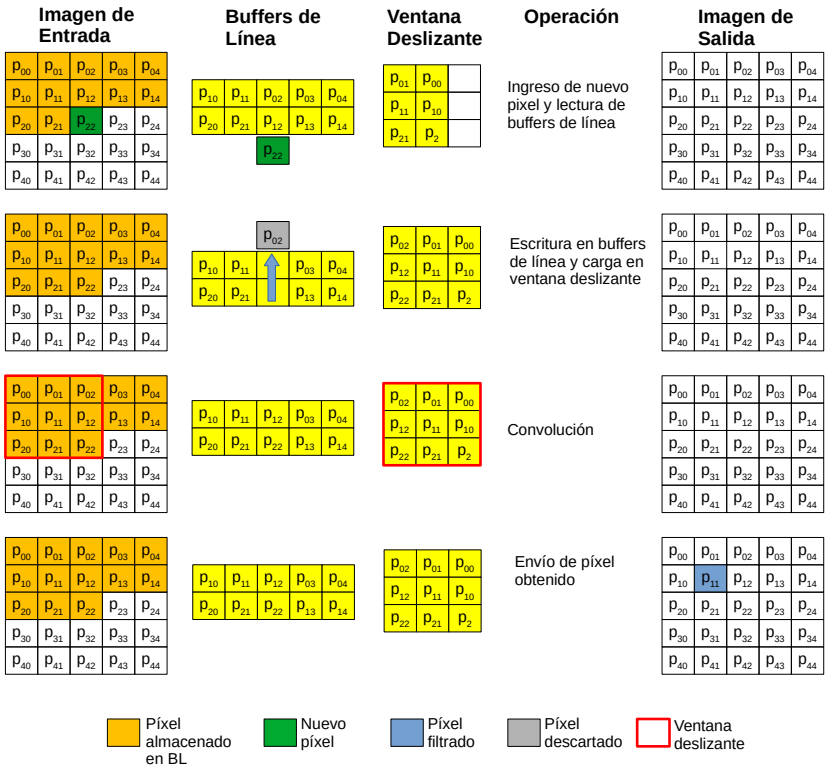


Figura 3.13: Acciones de filtro Sobel HDL

La implementación del FILTRO SOBEL en HDL sólo es comparable con su versión en alto nivel al asegurar la ejecución concurrente de sus tareas. Esto se conoce como *pipelining* y se logra al incorporar registros intermedios al diseño. En la Figura 3.14 se observan las tareas ejecutadas en forma solapada por el FILTRO SOBEL. En el primer ciclo de reloj se realiza la primera tarea que comprende el ingreso del nuevo pixel y la lectura de los buffers de línea, mientras las demás tareas permanecen inactivas. Luego, en el segundo ciclo de reloj, se realizan dos tareas en forma concurrente: el ingreso de nuevo pixel y una nueva lectura de los buffers del línea (primera tarea), al mismo tiempo que se realiza la segunda tarea comprendida por una escritura en los buffers de línea y la carga de píxeles en ventana deslizante. Esta situación continúa

incorporando más tareas en cada ciclo de reloj hasta ejecutar todas ellas en el cuarto pulso de reloj.

| Tarea | Ciclo de Reloj | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| Ingreso de nuevo pixel y lectura de los buffers de línea | Pixel 1 | Pixel 2 | Pixel 3 | Pixel 4 | Pixel 5 |
| Escritura en los buffers de línea y carga de la ventana deslizante | | Pixel 1 | Pixel 2 | Pixel 3 | Pixel 4 |
| Proceso de convolución | | | Pixel 1 | Pixel 2 | Pixel 3 |
| Envío de datos | | | | Pixel 1 | Pixel 2 |

Figura 3.14: *Pipelining* en filtro Sobel

Las cuatro tareas de FILTRO SOBEL descriptas son sincronizadas y controladas por la MEF que se observa en la Figura 3.15. La MEF comienza en el estado *START* responsable de inicializar algunas señales y luego pasa al estado *NORMAL* donde se realizan las cuatro tareas. Nuevamente, dos estados adicionales denominados *MISSED* y *WAIT_READY* aseguran la correcta transmisión de datos a través de las interfaces AXI4-STREAM hacia el bloque *U8toU32*.

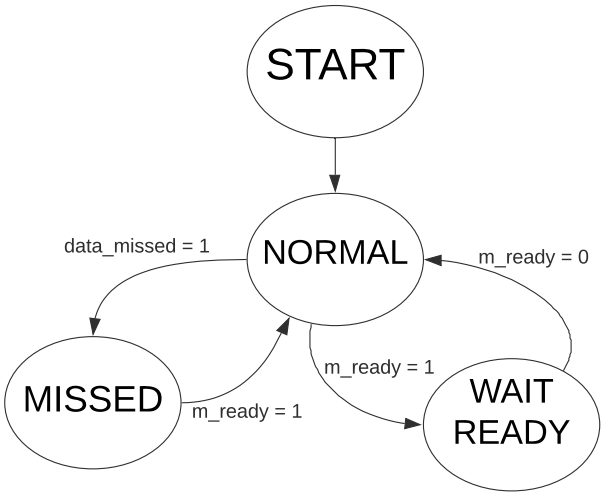


Figura 3.15: MEF bloque Sobel

De todos los estados que tiene la MEF, destaca *NORMAL* porque es el responsable de lograr que todas las operaciones se realicen en forma concurrente.

La descripción en VHDL de este estado se observa en la Figura 3.16.

```
--Tarea 1
--Ingreso de nuevo píxel
--y lectura de los buffers de línea
data_in <= S_AXIS_TDATA; --Recepción del dato
en_st0 <= s_valid;
en_st1 <= en_st0; --habilitación de tareas
en_st2 <= en_st1;
m_valid <= '0'; --dato no valido a enviar

--Tarea 2
--Escritura en los buffers de línea
--y carga de la ventana deslizante
if en_st0 = '1' then
    w_0 <= data_in & w_0(0 to w_0'high-1);
    w_1 <= line_buffer_2_q0 & w_1(0 to w_1'high-1);
    w_2 <= line_buffer_1_q0 & w_2(0 to w_2'high-1);
end if;

--Tarea 3
--Proceso de convolución
if en_st1 = '1' then
    data_calc <= sobel_kernel(w_2,w_1,w_0);
end if;

--Tarea 4
--Envío de datos
if en_st2='1' then
    data_out <= data_calc;
    if m_ready = '1' then
        m_valid <= '1';
    else
        sobel_fsm <= WAIT_READY;
    end if;
end if;
```

Figura 3.16: Sección de código de FILTRO SOBEL en VHDL

De acuerdo al diseño propuesto, la habilitación de cada tarea se realiza por medio de las señales *en_st0*, *en_st1* y *en_st2*, conectadas en cascada. Con este esquema de conexiones, el valor de *en_st0* es transferido a *en_st1* en cada pulso de reloj. Lo mismo sucede con *en_st2* que recibe el valor de *en_st1*.
El ciclo de operación de este bloque comienza al recibir un nuevo píxel que se almacena en la señal *data_in*, al mismo tiempo que la señal *en_st0* recibe el valor '1' de *s_valid*. Adicionalmente, *s_valid* es utilizado para habilitar la lectura de los buffers de línea. En el pulso de reloj siguiente se ejecuta la tarea 2, comprendida por la carga de píxeles en ventana deslizante *(w_0, w_1 y w_2)* por medio de la señal *en_st0*, y la escritura en los buffers de línea. Los valores que se almacenan en ventana deslizante son *data_in* que contiene el píxel nuevo, y los píxeles previamente leídos de los buffers de línea *Line_buffer_2_q0* y *Line_buffer_1_q0*. En el tercer ciclo de reloj, la señal *en_st1* habilita la convolución entre los píxeles almacenados en la ventana

deslizante y las máscaras Sobel. Finalmente, el resultado de la convolución es enviado en el cuarto ciclo de reloj por medio de la señal *en_st2*.

### 3.3.3. U8toU32 HDL

El bloque U8toU32 de bajo nivel ejecuta las mismas tareas que la versión de alto nivel. Comienza con una espera hasta recibir cuatro píxeles consecutivos desde FILTRO SOBEL. Luego, expande el tamaño de cada píxel y finalmente, envía tres palabras consecutivas de 32 bits hacia el DMA. La sincronización de tareas en este bloque se realiza con la MEF de la Figura 3.17. Comienza inicializando señales en el estado *START* para luego ir al estado *RCV* donde espera hasta completar las cuatro recepciones de 8 bits. En cada recepción, se almacena el nuevo píxel en tres lugares de memoria para restablecer el tamaño de imagen original. Finalizadas las recepciones, pasa al estado *SEND* para realizar los envíos correspondientes. Si el ciclo de operación se ejecuta sin problemas, el bloque sólo utiliza estos tres estados. Sin embargo, *WAIT_READY* y *MISSED* son estados que utiliza el bloque en caso que surjan problemas de comunicación con el DMA.

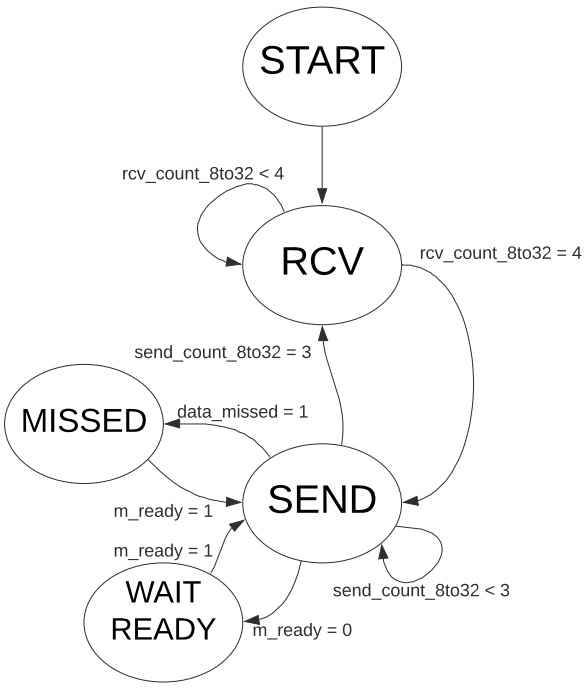


Figura 3.17: MEF bloque U8toU32

## 3.4. Resumen

En este capítulo se presentó el sistema de procesamiento de imagen para detección de contornos diseñado para una plataforma SoC. Este sistema incluye bloques de funciones que pertenecen al sistema de procesamiento y a la lógica programable del SoC. Los bloques del sistema de procesamiento son el procesador ARM, el DMA y el controlador de memoria DDR. De ellos, el más importante es el procesador que es responsable de configurar todos los bloques del sistema y ejecutar la aplicación de prueba. Por su parte, el DMA en conjunto con el controlador de memoria DDR, son responsables de leer y almacenar las imágenes en memoria.
Por otro lado, los bloques descriptos en la lógica programable del SoC son RGB2GRAY, FILTRO SOBEL y U8toU32. Estos bloques fueron desarrollados en dos lenguajes de programación: C para la versión de alto nivel y VHDL para la descripción en bajo nivel. Los tres bloques forman el núcleo de procesamiento del sistema y tienen por función detectar los contornos en las imágenes:

- El bloque RGB2GRAY recibe los píxeles a color desde el DMA y obtiene otros píxeles en escala de grises al promediar el valor de sus tres canales. En consecuencia, se reduce el tamaño de la imagen.
- El FILTRO SOBEL es el segundo bloque del núcleo de procesamiento y es responsable de detectar los contornos en imágenes por medio del algoritmo Sobel.
- El bloque U8toU32 recibe los píxeles desde FILTRO SOBEL, los expande para recuperar el tamaño original de la imagen y los envía al DMA para su almacenamiento.

Respecto a las implementaciones del núcleo de procesamiento, se destaca que las versiones de alto nivel se obtuvieron al codificar en lenguaje C la funcionalidad de los bloques, sin incorporar detalles de bajo nivel. En particular, las interfaces de comunicación AXI en todos los bloques al igual que el solapamiento y la concurrencia de tareas en FILTRO SOBEL se realizaron por medio de las directivas de compilador que sólo son interpretadas por la herramienta Vivado HLS de Xilinx.
Un proceso de desarrollo diferente fue seguido al describir los bloques del núcleo de procesamiento en VHDL. Para ello se comenzó conociendo la funcionalidad que debía tener cada bloque para luego diseñar el hardware correspondiente. En las descripciones de bajo nivel fue necesario conocer en detalle los protocolos AXI (AXI4-STREAM y AXI4-LITE) para implementar las

interfaces de cada bloque. El solapamiento de tareas se logró por medio de la inclusión de registros intermedios entre las distintas etapas del FILTRO SOBEL. Esta tarea no fue trivial y demandó una gran cantidad de pruebas de verificación y reescritura de código hasta obtener los resultados esperados.

# Resultados

Este capítulo comienza en la sección Comparación de Enfoques HDL/HLS con una breve descripción del problema, los experimentos a realizar y los resultados esperados. Luego se describe el Entorno Experimental de esta investigación que comprende la plataforma hardware y las herramientos de desarrollo del fabricante Xilinx. El capítulo continúa en la sección Experimentos Realizados donde se detalla todo el proceso experimental para obtener los resultados de esta investigación. Posteriormente, se incluyen tres secciones donde se comparan ambas implementaciones Sobel respecto al Uso de Recursos, Rendimiento y Costo de Programación. La sección Trabajos Relacionados compara los resultados de esta investigación con otros trabajos similares. Finalmente, se presenta un Resumen del capítulo.

## 4.1. Comparación de Enfoques HDL/HLS

Se implementaron dos versiones de un filtro Sobel para realizar el estudio comparativo entre enfoques HDL y HLL para FPGA. Las implementaciones se analizaron en función de tres métricas: costo de programación, uso de recursos y rendimiento. El costo de programación se midió en función del tiempo de desarrollo (en horas) y la cantidad de líneas de código fuente de cada implementación. Por otro lado, los recursos de una implementación estan asociados con su complejidad y pueden representar una restricción de diseño. Finalmente el rendimiento se midió por tres parámetros que son latencia, *throughput* y tiempos de ejecución.
Las pruebas a cada versión Sobel se realizan sobre una plataforma SoC ZYBO usando cuatro imágenes de distintos tamaños, extraídas de repositorios públicos.

## 4.2. Entorno Experimental

Se sintetizaron dos versiones del filtro Sobel con herramientas de Xilinx para llevar a cabo este estudio, una versión de alto nivel en lenguaje C y otra versión de bajo nivel en VHDL. Los bloques del núcleo de procesamiento RGB2GRAY, FILTRO SOBEL y U8toU32 de alto nivel se sintetizaron con la herramienta Vivado HLS 2019.1. Cada bloque demandó, inicialmente su diseño y validación en lenguaje C, y posteriormente su exportación como un núcleo de propiedad intelectual (IPCore). En estos bloques, Vivado HLS generó los controladores (Drivers) para su utilización en la aplicación de prueba.
Luego, se describieron los mismos bloques en lenguaje VHDL dentro de la

herramienta Vivado 2019.1. Estos bloques también demandaron pruebas de validación aunque no necesitaron ser exportados como núcleos de propiedad intelectual. Para estas implementaciones, Vivado no generó los controladores por lo que debieron ser desarrollados e incorporados a la aplicación de prueba. Concluido el desarrollo de las dos implementaciones Sobel, se diseñó el sistema de procesamiento de imagen (descripto en el Capítulo 3) en un entorno gráfico dentro de la herramienta Vivado 2019.1. Finalmente, se realizaron pruebas reales al sistema mediante una aplicación en lenguaje C dentro de XSDK 2019.1. Ambas versiones Sobel utilizaron la misma aplicación de prueba con leves modificaciones (sólo fue necesario cambiar los controladores del núcleo de procesamiento).
El hardware elegido para llevar a cabo las pruebas reales fue la plataforma de desarrollo ZYBO. Esta placa integra un SoC ZYNQ-7000 de Xilinx formado por un procesador ARM cortex-A9 dual-core y una FPGA XC7Z010-1-CLG400C. En la Figura 4.1 se observa la placa de desarrollo ZYBO[1].

## 4.3. Experimentos Realizados

Las pruebas reales en hardware se llevaron a cabo con cuatro imágenes en formato BMP descargadas de repositorios públicos: *Mandrill* de 512×512[2], *Kodim23* de 768×512[3], *Owl* de 1920×566[4], y *Lightbulbs* de 1920×1080[5]. Todas las imágenes se almacenaron en una memoria microSD de 4 Gb que luego se conectó a la placa ZYBO.
La aplicación de prueba consistió en leer una imagen almacenada en memoria microSD, enviar los píxeles al núcleo de procesamiento por medio del DMA, esperar la recepción de los píxeles procesados y almacenar la nueva imagen en memoria microSD. Dentro de la aplicación de prueba se registraron dos tiempos con la función *XTime_ GetTime* de la librería *xtime_ l.h*. El tiempo de operación del núcleo de procesamiento denominado *Tiempo Sobel*, que sólo contempló las tareas realizadas por los bloques *RGB2GRAY, FILTRO SOBEL y U8toU32*; y el *Tiempo Total* del sistema que incluyó, además, la configuración inicial de los bloques y los accesos a memoria para la lectura y escritura de imágenes.

[1] Extraído de https://reference.digilentinc.com/reference/programmable-logic/zybo/start
[2] http://sipi.usc.edu/database/database.php
[3] http://www.cs.albany.edu/~xypan/research/img/Kodak/kodim23.png
[4] https://pixabay.com/es/photos/lechuza-granero-ave-animales-1710659/
[5] https://pixabay.com/es/illustrations/bombillas-de-luz-para-vidrio-5488573/

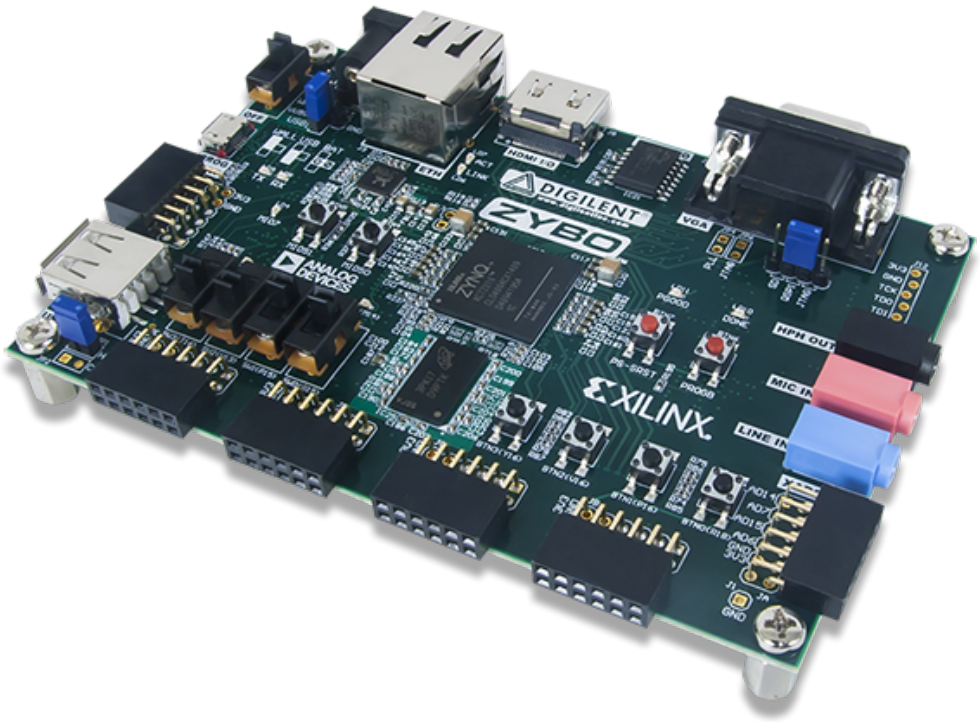


Figura 4.1: Placa de desarrollo Zybo

Se realizaron 10 pruebas de filtrado con cada imagen para las dos versiones Sobel. Para ello, se cargó en la placa ZYBO los bits de configuración del sistema de alto nivel y se ejecutó 10 veces la aplicación de prueba con cada imagen para calcular el promedio de los tiempos Sobel y Total. Concluidas las pruebas con el filtro Sobel de alto nivel, el mismo fue reemplazado por la implementación de bajo nivel antes de repetir todo el proceso.
Las imágenes de prueba así como las obtenidas luego del filtrado se observan en las Figuras 4.2 a 4.5. Se midió la similitud de las imágenes generadas por ambas implementaciones por medio de su distancia Hamming, logrando una semejanza de 100 %.

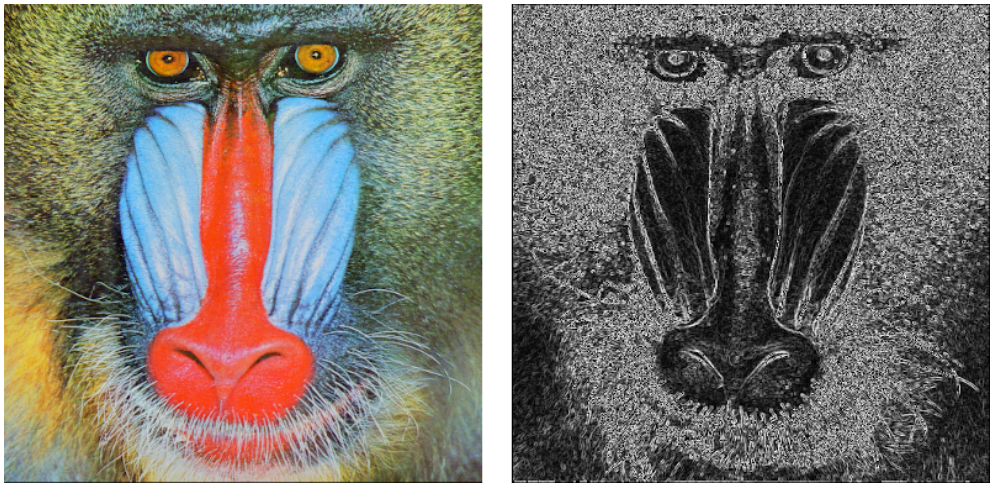

Figura 4.2: Imagen de prueba *Mandrill*.

Figura 4.3: Imagen de prueba *Kodim23*.

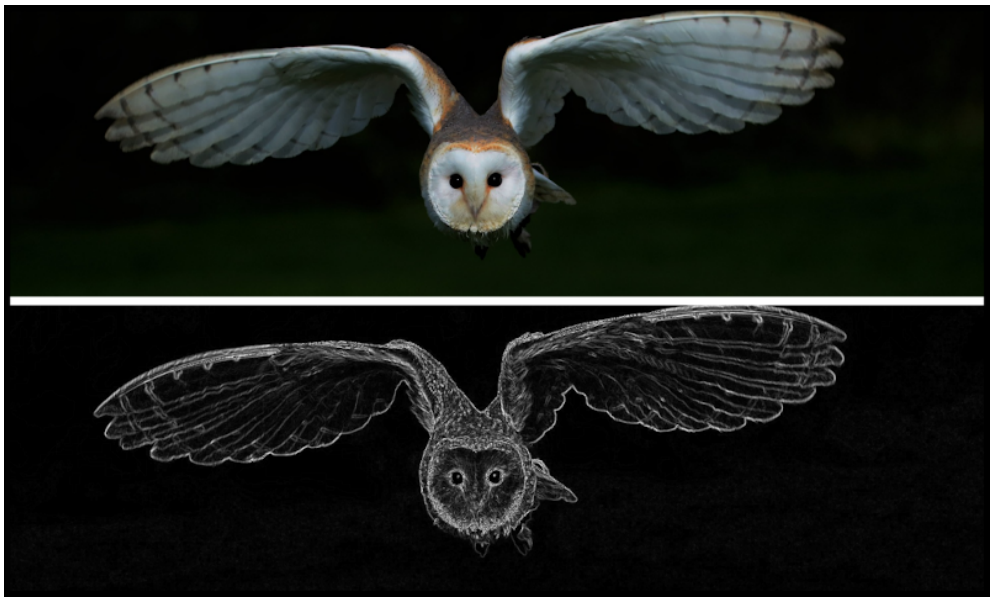

Figura 4.4: Imagen de prueba *Owl*.

## 4.4. Uso de Recursos

Los recursos de cada implementación Sobel fueron asignados por la herramienta Vivado 2019.1. En la Tabla 4.1 se presentan los recursos utilizados por ambas implementaciones Sobel. Los valores son informados en porcentaje respecto al total de recursos en el SoC de la placa ZYBO. Las columnas denominadas *S.LUTs*, *S.Registers*, *F7 Muxes*, *BRAM* y *DSPs* hacen referencia a las LUTs, registros, multiplexores, bloques de RAM y DPS.

Tabla 4.1: Uso de recursos

| **Versión** | **Uso de recursos** | | | | |
|---|---|---|---|---|---|
| | **S.Lut** | **S.Reg** | **F7 Muxes** | **BRAM** | **DSPs** |
| **HDL** | 0.8 % | 0.5 % | 0 % | 1.6 % | 0 % |
| **HLS** | 4.4 % | 2.3 % | $< 0.1$ % | 1.6 % | 0 % |

Ambas implementaciones tuvieron un consumo similar en multiplexores y bloques de RAM, además de no utilizar DSP. Las mayores diferencias se observaron en las LUTs y los registros. La implementación HLS tuvo un consumo mayor de 5.5× en LUTs y 4.6× en registros respecto a la versión HDL. Sin embargo, en ningún caso representó una restricción de diseño.

Figura 4.5: Imagen de prueba *Lightbulbs*.

## 4.5. Rendimiento

Las dos implementaciones Sobel se desarrollaron para trabajar a una frecuencia de operación de 100 MHz (o un período de reloj de 10 ns). En el caso de la implementación de alto nivel, la herramienta Vivado HLS generó un reporte de rendimiento en función del tiempo de viaje de señal entre registros y la latencia de cada bloque, como se observa en la Figura 4.6. En la sección *Timing (ns)* del reporte se indica el tiempo de señal entre registros máximo permitido (*Target*), el tiempo estimado (*Estimated*) y un margen de seguridad (*Uncertainty*). De acuerdo a los valores informados en la Figura 4.6, el bloque FILTRO SOBEL superó las restricciones temporales al tener un tiempo estimado de 8.74 ns respecto al máximo permitido de 10 ns. En la sección *Latency (clock cycles)*, el reporte de rendimiento informó las latencias máxima y mínima, y los intervalos iniciales máximo y mínimo del bloque.

**Performance Estimates**

**Timing (ns)**

**Summary**

| Clock | Target | Estimated | Uncertainty |
|---|---|---|---|
| ap_clk | 10.00 | 8.740 | 1.25 |

**Latency (clock cycles)**

**Summary**

| Latency | | Interval | | |
|---|---|---|---|---|
| min | max | min | max | Type |
| 2076611 | 2076612 | 2076601 | 2076601 | loop rewind(delay=0 initiation interval(s)) |

Figura 4.6: Reporte de rendimiento del bloque FILTRO SOBEL en HLS.

Para el caso de los desarrollos en HDL, se obtuvo el reporte temporal del sistema completo de procesamiento de imagen observado en la Figura 4.7. En ella, el parámetro *Requirenment* es el valor temporal establecido de 10 ns y *Path Delay* es el mayor retardo temporal entre registros en el diseño con un valor de 8.877 ns, confirmando que el sistema supera la restricción de tiempo. Por otro lado, la latencia no fue informada por la herramienta Vivado pero se pudo estimar dentro de la aplicación de prueba en función del tiempo de ejecución.
Los rendimientos de ambas implementaciones Sobel se compararon en función de *throughput* y latencia. La primera métrica se determinó con ayuda del analizador lógico integrado (ILA) en la placa ZYBO al medir la cantidad de ciclos de reloj transcurridos entre el primer píxel ingresado al núcleo de

```
Report Design Analysis

Table of Contents
-----------------
1. Setup Path Characteristics 1-1

1. Setup Path Characteristics 1-1
---------------------------------

+-------------------------+-----------------------------------------------------------------------------------------------+
|      Characteristics    |                                          Path #1                                              |
+-------------------------+-----------------------------------------------------------------------------------------------+
| Requirement             |                                                                                        10.000 |
| Path Delay              |                                                                                         8.877 |
| Logic Delay             | 1.874(22%)                                                                                    |
| Net Delay               | 7.003(78%)                                                                                    |
| Clock Skew              |                                                                                        -0.052 |
| Slack                   |                                                                                         0.748 |
| Clock Relationship      | Safely Timed                                                                                  |
| Logic Levels            |                                                                                             8 |
| Routes                  |                                                                                             9 |
| Logical Path            | FDRE LUT5 LUT3 LUT4 LUT6 LUT4 LUT3 LUT3 LUT3 FDRE                                             |
| Start Point Clock       | clk_fpga_0                                                                                    |
| End Point Clock         | clk_fpga_0                                                                                    |
| DSP Block               | None                                                                                          |
| BRAM                    | None                                                                                          |
| IO Crossings            |                                                                                             0 |
| Config Crossings        |                                                                                             3 |
| SLR Crossings           |                                                                                             0 |
| PBlocks                 |                                                                                             0 |
| High Fanout             |                                                                                            15 |
| Dont Touch              |                                                                                             0 |
| Mark Debug              |                                                                                             0 |
| Start Point Pin Primitive | FDRE/C                                                                                      |
| End Point Pin Primitive | FDRE/CE                                                                                       |
| Start Point Pin         | m_valid_i_reg/C                                                                               |
| End Point Pin           | WORD_LANE[1].USE_ALWAYS_PACKER.BYTE_LANE[1].USE_RTL_DATA.USE_REGISTER.M_AXI_WDATA_I_reg[44]/CE |
+-------------------------+-----------------------------------------------------------------------------------------------+
```

Figura 4.7: Reporte temporal del sistema completo con núcleo de procesamiento HDL.

procesamiento y el primer valor devuelto por éste, que es la cantidad de ciclos de reloj necesarios para completar el pipeline de tareas. En las Figuras 4.8 y 4.9 se observan los diagramas temporales de las dos implementaciones, la línea vertical de color rojo marca el primer dato recibido por el núcleo de procesamiento, la línea vertical amarilla hace los mismo con el primer valor enviado hacia el DMA y los pulsos de reloj se observan en la línea horizontal superior. En la Figura 4.8 se evidencia que el filtro Sobel en HDL recibe el primer valor en el segundo pulso de reloj y envía su resultado en el décimosexto, demorando 14 ciclos de reloj. Del mismo modo, en la Figura 4.9 se observa que el primer dato recibido por la implementación de alto nivel ocurre también en el segundo pulso de reloj pero demora 65 ciclos de reloj en realizar el primer envío.

Los tiempos de ejecución de cada filtro se presentan en la Tabla 4.2. Las columnas *T. Sobel* y *T. Total* muestran los tiempos de ejecución Sobel y Total del sistema, y la columna *Acel* indica el cociente entre los tiempos de ejecución Sobel para HDL y HLS. Con este último parámetro se pudo enfatizar la diferencia de rendimientos entre ambas versiones.

Los tiempos de ejecución Sobel en HDL fueron menores para todas las imágenes de prueba. Esto demostró que la versión de bajo nivel tuvo mejor rendimiento, independientemente del tamaño de imagen. La mayor diferencia se

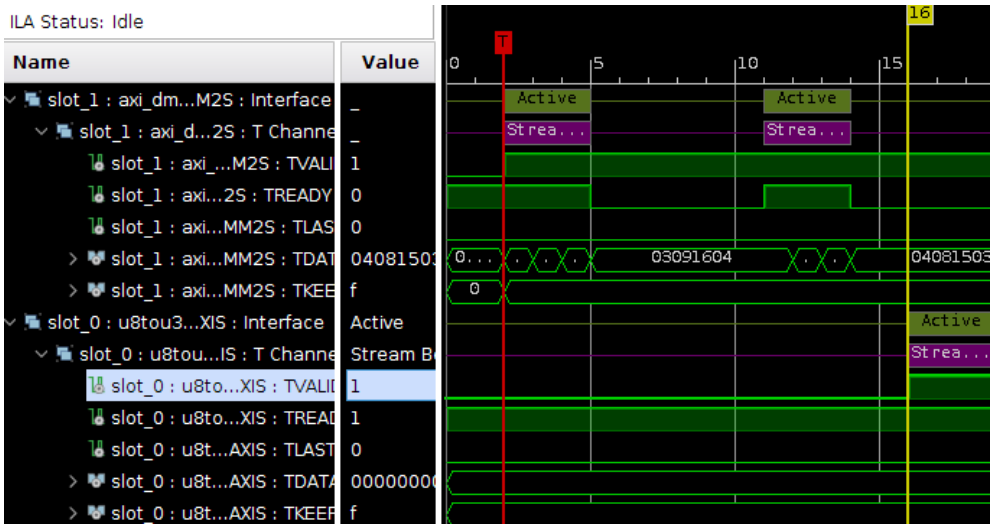


Figura 4.8: *Throughput* en implementación HDL

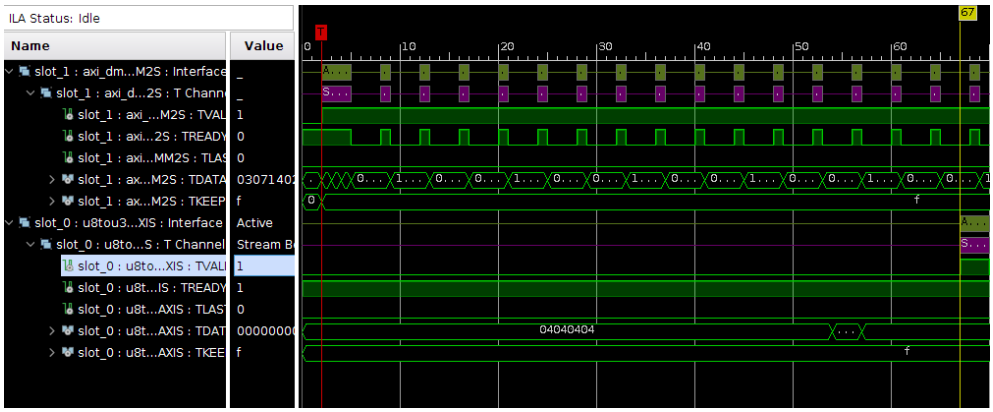


Figura 4.9: *Throughput* en implementación HLS

Tabla 4.2: Tiempos de ejecución

| **Imagen** | **T. Sobel (ms)** | | | **T. Total (ms)** | |
|---|---|---|---|---|---|
| | **HDL** | **HLS** | **Acel.** | **HDL** | **HLS** |
| **Mandrill** (512×512) | 5.9 | 39.3 | 6.6× | 816 | 837 |
| **Kodim23** (768×512) | 8.8 | 39.3 | 4.4× | 1078 | 1026 |
| **Owl** (1920×566) | 24.4 | 43.4 | 1.7× | 2280.4 | 2329.5 |
| **Lightbulbs** (1920×1080) | 58 | 82.9 | 1.4× | 3985.9 | 3954.5 |

evidenció al procesar la imagen más pequeña (*Mandrill*) logrando una aceleración de 6.6×. Esa diferencia disminuyó al aumentar el tamaño de imagen, obteniendo una aceleración de 1.4× en la imagen más grande (*Lightbulbs*). También se constató que los tiempos de ejecución en la implementación de bajo nivel tuvieron un comportamiento cercano al lineal en función del tamaño de la imagen; en cambio la versión de alto nivel presentó un retardo mínimo independientemente del tamaño de la imagen. Al diseñar los bloques de alto nivel para procesar imágenes de hasta 1920×1080, la lógica adicional que emplea Vivado HLS condiciona el tiempo mínimo de ejecución en las imágenes de menor tamaño. Sin embargo, al superar el retardo inicial, los tiempos de ejecución crecen al aumentar el tamaño de la imagen (aunque no en forma proporcional).

## 4.6. Costo de Programación

En este trabajo el costo de programación se estimó en función de dos parámetros: la cantidad de líneas de código fuente (SLOC) y el tiempo de desarrollo para alcanzar una versión funcional y completa del sistema. Como estos dos indicadores no están exentos de subjetividad, se completa su análisis con una comparación cualitativa del esfuerzo requerido en cada enfoque. Ambas partes son complementarias y proveen un análisis más comprensivo en la cuestión.

En la Tabla 4.3 se observa el número de archivos, la cantidad de SLOC y el tiempo de desarrollo en horas para cada implementación. La medición de SLOC se realizó con la herramienta cloc [6].

Tabla 4.3: Costo de Programación

| **Versión** | **Costo de programación** | | |
|---|---|---|---|
| | **# Archivos** | **SLOC** | T. **Desarrollo (horas)** |
| **HDL** | 4 | 493 | 384 |
| **HLS** | 2 | 90 | 121 |

La versión HLS se desarrolló en primer lugar, demandó 121 horas de programación y 90 SLOC (en dos archivos) para obtener una versión funcional del sistema. Luego se desarrolló la versión HDL, con una ventaja adicional por el conocimiento adquirido en el desarrollo de la versión de alto nivel. A pesar de ello, el filtro Sobel en HDL demandó 384 horas y 493 SLOC dispuestos en cuatro archivos. Esto representó un incremento de 5.5× SLOC y 3.2× horas en la versión HDL respecto a la alternativa HLS.
Luego de implementar y someter a prueba cada versión del filtro Sobel, se evidenció que las fortalezas en un enfoque de programación constituían, a su vez, debilidades en el otro enfoque. Comenzando con los desarrollos en alto nivel, estos gozaron de dos beneficios conjuntos. Por un lado, se obtuvieron diseños simples y legibles por las caracteríticas propias de los HLL mientras que, por otro lado, las herramientas HLS tomaron gran relevancia al resolver todas las particularidades de bajo nivel. Es así que cada bloque en alto nivel se diseñó como una función en lenguaje C sin involucrarse en la arquitectura hardware. El algoritmo de cada bloque resultó tan versátil que fue utilizado, casi sin modificaciones, para desarrollar versiones software de los mismos. Con respecto a las HLS, estas herramientas fueron responsables de implementar las interfaces de comunicación AXI4 y la ejecución concurrente de tareas a través de simples directivas de compilador. Es más, estas herramientas permitieron incluso generar diversas implementaciones de un mismo bloque de función sin modificar el código fuente hasta lograr el mejor diseño, sólo variando las directivas de compilador. Otra ventaja de las HLS es la generación automática de los drivers para manipular las implementaciones de alto nivel en la aplicación de prueba, así como los reportes donde se detalla el uso de recursos y las métricas de rendimiento de cada bloque sintetizado. Esta situación disminuye aún más la complejidad del desarrollo.

[6] https://github.com/AlDanial/cloc

Sin embargo, algunas de las ventajas de este enfoque de desarrollo contribuyeron en sus debilidades. Si bien las directivas de compilador simplificaron los diseños, estos se tornaron muy dependientes de la herramienta de síntesis Vivado HLS, obligando a modificar el código en caso de portarlo a otra HLS. Otra debilidad de este enfoque es la restricción de las optimizaciones sólo a través de directivas, impidiendo que el programador personalice algunas secciones del código en bajo nivel. Esto llevó a que la herramienta implemente esas optimizaciones en forma automática y no se obtuvieran resultados tan eficientes como las versiones de bajo nivel. Por todo ello, los diseños en HLL resultaron más fáciles de diseñar y testear, con menos líneas de código aunque emplearon mayor cantidad de recursos y los rendimientos obtenidos fueron menores respecto a las versiones en HDL.
En sentido opuesto, los desarrollos de bajo nivel demandaron una etapa de diseño extra respecto a las versiones de alto nivel. En primer lugar, fue indispensable conocer en detalle la funcionalidad de cada bloque para luego describir el hardware responsable de realizar cada tarea. Esta etapa también demandó conocer el hardware de la plataforma de desarrollo para determinar la viabilidad del diseño. Sumado a ello, la herramienta de desarrollo Vivado no brindó directivas u otro método que permitiera simplificar el diseño de los bloques como sí sucedió con Vivado HLS. Esto se evidenció al describir las interfaces de conexión AXI4 en bajo nivel y una MEF responsable de comandar y sincronizar la ejecución de tareas concurrentes.
Más allá de las debilidades mencionadas, se destacan algunas virtudes del enfoque de desarrollo en HDL. Si bien no existen directivas de compilador que simplifiquen el diseño del bloque, la necesidad de describirlo en bajo nivel lo independiza de la herramienta de desarrollo y permite su portabilidad a otras propuestas. Esto trae aparejada otra ventaja que es la posibilidad de optimizar el diseño a un grado mayor respecto al desarrollo en alto nivel al evitar los procesos automáticos de descripción de hardware propios de las HLS. Es así que los bloques en HDL se diseñaron con una cantidad precisa de registros para lograr el solapamiento de tareas, la descripción de cada señal se realizó con la cantidad exacta de bits e incluso las interfaces de comunicación AXI4 se optimizaron para la funcionalidad requerida. En consecuencia, los bloques en HDL demandaron un esfuerzo de desarrollo superior al requerir mayores tiempos de diseño y pruebas, y más líneas de código. Sin embargo, los bloques resultantes fueron más eficientes en el uso de recursos y obtuvieron mejores rendimientos respecto a las versiones en alto nivel.

## 4.7. Trabajos Relacionados

Existen otros estudios comparativos entre lenguajes de programación HDL y HLL para FPGA que utilizan módulos de procesamiento de imagen como caso de estudio (Hiraiwa y Amano, 2013)(Gurel, 2016)(Zwagerman, 2015). Este trabajo se diferencia del resto en que no se utilizan herramientas adicionales, librerías de imagen/video u otro factor que pueda favorecer alguna de las dos implementaciones.
El sistema de procesamiento de imagen se desarrolló con las herramientas Vivado, Vivado HLS y XDSK de Xilinx, utilizando solo lenguajes de bajo y alto nivel. No se usaron programas adicionales como Core Generator o MATLAB que pudieran simplificar el diseño de los bloques. Se destaca la descripción en HDL de las interfaces AXI4 así como el diseño del solapamiento de tareas que tornó más complejo el diseño de bajo nivel pero que permitió obtener mayores rendimientos y mejor uso de recursos.
Para una comparación justa, cada módulo del núcleo de procesamiento se describió en ambos lenguajes de programación y se comprobó que tengan la misma funcionalidad. Para ello, se diseño un único sistema de procesamiento y se modificó solamente su núcleo.
Las pruebas reales en placa ZYBO se realizaron con una aplicación creada en XSDK utilizada para ambas versiones del filtro. Se usaron los drivers generados en forma automática por la herramienta Vivado HLS para el funcionamiento de los módulos en HLL, no así con los módulos en HDL donde los drivers tuvieron que ser desarrollados. El sistema operativo autónomo de Xilinx fue elegido con el objetivo de mantener el diseño genérico aunque también se puede embeber un sistema operativo multi-propósito (como Linux) u orientado a tiempo real (como FreeRTOS).
En (Hiraiwa y Amano, 2013) se implementó una arquitectura de procesamiento de video en tiempo real en lenguajes HDL y HLL con cuatro filtros: gausiano, detector de contornos, erosión y dilatación. El autor no brindó detalles de diseño e implementación de los bloques ni de la comunicación entre ellos. Además, utilizó la herramienta Core Generator para desarrollar las interfaces AXI4 de los módulos en HDL. Al no describir las interfaces AXI4 en bajo nivel, se favoreció la implementación HDL demandando menos tiempo de desarrollo pero, al mismo tiempo, no se obtuvo el rendimiento y uso de recursos esperado. Aún así, los resultados obtenidos por el autor presentaron la misma tendencia a los resultados obtenidos en la presente investigación: la versión de alto nivel consumió entre 3× y 4× mayor cantidad de recursos pero el tiempo de desarrollo fue menor, siendo de 15 días aproximadamente para la versión HDL y de sólo 3 días para la versión HLL.

En forma similar, en (Gurel, 2016) se compararon varios filtros de imagen desarrollados en HDL y HLL. Los filtros elegidos para el caso de estudio fueron: Sobel, suavizado, dilatación, erosión e histograma. En este trabajo se brindaron directivas de optimización para ambos enfoques de programación aunque no se incluyeron imágenes de prueba. Tampoco se mencionó el uso de librerías, sistema operativo o metodología para realizar las pruebas. Respecto al diseño de los filtros, no se brindó detalles de la implementación de lo buses de comunicación. Es posible que las pruebas se hayan realizado en forma teórica sobre los filtros, sin incluirlos en un sistema de procesamiento de imagen. Los resultados obtenidos favorecen a las implementaciones HDL, aunque algunas variantes HLL fueron superiores.
Un tercer trabajo comparativo entre lenguajes para FPGA que utilizó una aplicación de procesamiento digital se describe en (Zwagerman, 2015). Se eligió como caso de estudio un filtro de desenfoque sobre una plataforma SoC ZC702. En este trabajo el autor partió de una aplicación de prueba del fabricante Xilinx y sólo modificó una pequeña parte del sistema. Del mismo modo que en (Hiraiwa y Amano, 2013), no se describieron las interfaces AXI4 en HDL. En este caso se utilizaron núcleos IP provistos por el fabricante. Nuevamente, al no describir las interfaces AXI4 en bajo nivel, el tiempo de desarrollo HDL disminuyó con un impacto negativo en el uso de recursos y rendimiento. Respecto a los resultados obtenidos, el autor incluyó el uso de recursos, la frecuencia de operación y el tiempo de desarrollo de ambas implementaciones pero no informó el tiempo de ejecución en cada una de ellas. Los resultados muestran la misma tendencia que los obtenidos en esta tesis.

## 4.8. Resumen

En este capítulo se describieron las herramientas de desarrollo de Xilinx y la plataforma hardware ZYBO utilizadas para el desarrollo experimental de este trabajo. Luego, se presentaron las imágenes de prueba originales y resultantes luego del filtrado. A continuación, se analizaron los resultados obtenidos respecto a uso de recursos, rendimiento y costo de programación para ambas versiones. En ellos se evidenció que la versión HLS fue menos eficiente en uso de recursos al emplear más de 5.5× en LUTs y 4.6× en registros respecto a la versión HDL aunque no representó una restricción de diseño en ningún caso.
El rendimiento se evaluó en función de la *throughput* y tiempo de procesamiento para cada imagen. Nuevamente, la versión HDL fue superior en todos los aspectos al demorar 14 ciclos de reloj en generar el primer resultado contra

los 65 ciclos de reloj de la implementación HLL. Respecto a la latencia y los tiempos de ejecución, la versión HDL fue más rápida para todos los tamaños de imágenes seleccionados, aunque el cociente de mejora fue decreciendo a medida que la imagen aumentaba. Aún así, fue capaz de lograr una mejora de hasta 1.4× en la imagen más grande (1920×1080).
En relación al costo de programación, se eligieron como parámetros la cantidad de archivos, SLOC y el tiempos de desarrollo requerido por cada implementación Sobel. Se evidenció una mayor complejidad en la variante HDL respecto a la HLL al demandar 5.5× mayor cantidad de SLOC y 3.2× más tiempo de desarrollo. Incluso la experiencia adquirida al realizar la variante HLL en primer lugar no fue suficiente para favorecer a la variante HDL.
Para realizar una evaluación más amplia del costo de programación, además de comparar SLOC y tiempo de ejecución, en esta tesis se incluyó una comparación cualitativa del esfuerzo requerido en cada enfoque. La implementación en HDL requirió más tiempo de desarrollo y SLOC respecto a su contraparte en HLS. Esto se debe a que el enfoque en HLS permite al programador enfocarse en la funcionalidad del sistema sin necesidad de definir los recursos hardware u otros mecanísmos de bajo nivel. Por ejemplo, los puertos de comunicación y el solapamiento de tareas fueron implementados usando directivas de compilador en el enfoque HLS. En sentido contrario, el programador debe describirlos manualmente a nivel RTL en el enfoque HDL.
Por último, se presentó una discusión entre esta investigación y algunos trabajos similares. Ninguno de los trabajos analizados implementa ambas versiones de los filtros sin utilizar software adicional o núcleos IP. Tampoco incluyen en sus investigaciones la metodología empleada para realizar las pruebas en hardware o los tiempos de ejecución obtenidos. Aun así, los resultados informados en esos estudios presentan una tendencia similar a los resultados obtenidos en esta tesis.

# Conclusiones y Trabajos Futuros

En este capítulo se exponen las Conclusiones de esta tesis y luego se plantean posibles líneas de Trabajos Futuros.

## 5.1. Conclusiones

Desde sus orígenes, las FPGA se han destacado por su gran eficiencia energética, alto rendimiento y versatilidad al reconfigurar su arquitectura interna. A pesar de ello, las FPGA no tuvieron la aceptación esperada, principalmente por su elevado costo de programación. Los lenguajes HDL como VHDL y Verilog se caracterizan por ser muy verbosos, fuertemente tipados y propensos a errores, además de demandar un gran conocimiento de la arquitectura de las FPGA. Esto llevó a un aumento en los tiempos de desarrollo conforme crecía la complejidad de los sistemas, impidiendo responder a los tiempos del mercado.
Todo ello condujo a nuevos enfoques de desarrollo de alto nivel para FPGA que disminuyeran la complejidad y el tiempo de diseño. Es así que en las últimas décadas, se incorporaron herramientas HLS que utilizan algún HLL para obtener la descripción HDL del sistema. Estas herramientas, en conjunto con las tecnologías híbridas que incluyen procesadores y FPGA en el mismo dispositivo, acercaron los enfoques de desarrollo software y hardware.
Con las HLS, el programador sólo se concentra en la funcionalidad del sistema, dejando a cargo de la herramienta la asignación de recursos, sincronización de tareas y otras cuestiones de bajo nivel. Sin embargo, los HLL fueron creados para aplicaciones destinadas a procesadores y tienen limitaciones al describir hardware como no permitir definir tiempos de hardware o longitudes de bits, entre otros.
Aún así, desde hace años se evidencia un aumento en el uso de HLL respecto a HDL con el fin de disminuir el tiempo de desarrollo, aunque no hay una tendencia clara en cuanto al uso de recursos y el rendimiento alcanzado por cada lenguaje, estando fuertemente influenciado por las características del problema a resolver, las herramientas utilizadas y las especificaciones de diseño. En este contexto, resulta imprescindible conocer fortalezas y debilidades de cada enfoque de desarrollo para FPGA que permita su correcta elección en función de la aplicación a realizar. Es por ello que el objetivo general de esta investigación es *comparar los enfoques de desarrollo HDL y HLL en FPGA respecto a las prestaciones (rendimiento, uso de recursos y esfuerzo de programación) de sus implementaciones para aplicaciones de detección de contornos de imágenes.* A continuación, se detallan los objetivos específicos propuestos y el cumplimiento de los mismos:

- *Explorar aplicaciones de detección de contornos en imágenes en FPGA que empleen enfoques de desarrollo HDL y HLL*

  En el Capítulo 1 se describió el estado del arte de trabajos comparativos en distintos ámbitos entre enfoques de desarrollo HDL y HLL para FPGA. La descripción incluyó los aportes, las limitaciones y los resultados de cada trabajo seleccionado. En general, las implementaciones de bajo nivel alcanzaron rendimientos mayores y utilizaron menos recursos respecto a las versiones de alto nivel, aunque se encontraron excepciones.

  La elección del filtro Sobel como caso de estudio para esta investigación radica en que es un filtro convolucional, por lo que resulta sencillo modificar/adaptar el código para implementar otros filtros de imágenes. Esta cuestión favorece a la generalización de los resultados obtenidos.

- *Analizar las prestaciones de aplicaciones de detección de contornos en imágenes que hayan sido desarrolladas siguiendo los enfoques HDL y HLL*

  El Capítulo 2 representa el marco teórico de esta tesis, sentando las bases para la propuesta posterior. En ese capítulo, se estudiaron diferentes cuestiones relacionadas a FPGAs: su arquitectura y evolución a través del tiempo, los lenguajes de programación disponibles y algunas métricas para evaluación de prestaciones en esta clase de sistema. Adicionalmente, también se estudió el procesamiento digital de imágenes y, en particular, la detección de contornos a través del operador Sobel.

  Luego, se describieron las fortalezas y limitaciones de los lenguajes HDL y HLL utilizados en los desarrollos en FPGA. Se continuó con una descripción de las métricas de prestaciones seleccionadas para la comparación de las implementaciones Sobel.

  En el Capítulo 3, se presentó una descripción detallada del diseño y arquitectura de las dos propuestas de sistemas de procesamiento de imágenes para la detección de contornos Sobel utilizando HDL y HLL. Ambas propuestas comparten el núcleo de procesamiento, el cual está formado por tres bloques: RGB2GRAY, FILTRO SOBEL y U8toU32.

  Para llevar a cabo el estudio comparativo se realizaron dos implementaciones, en HDL y HLL, de cada uno de los bloques que componen el núcleo de procesamiento. Los bloques fueron sintetizados en la Lógica Programable del SoC, siendo su objetivo acelerar por hardware la detección de contornos en imágenes. En primer lugar, se realizaron pruebas sobre el sistema de procesamiento de imagen con el núcleo de

procesamiento en HLL. Luego, los bloques en HLL fueron reemplazados por sus versiones HDL para su testeo.

Las pruebas a cada versión Sobel se realizaron sobre una plataforma SoC ZYBO usando cuatro imágenes de distintos tamaños, extraídas de repositorios públicos. Se midió el rendimiento, uso de recuros y esfuerzo de programación de cada variante, analizando estas métricas tanto en forma individual como inter-relacionada.

A diferencia de otras propuestas, este trabajo se destaca del resto por no requerir de herramientas adicionales, librerías de imagen/video u otro factor que pueda favorecer alguna de las dos implementaciones. También resulta importante mencionar que el filtro Sobel en HDL desarrollado en esta investigación utiliza un enfoque diferente respecto a otros trabajos de la comunidad científica. Los buffers de línea se sintetizaron como bloques de memoria RAM, a diferencia de otras propuestas que utilizan registros de desplazamiento(Nosrat y S. Kavian, 2012)(Mehra y Verma, 2012), lo que reduce el consumo de memoria en la FPGA y aumenta la velocidad del sistema..

- *Comparar las prestaciones de los enfoques HDL y HLL para implementaciones de aplicaciones de detección de contornos en imágenes*

  En el Capítulo 4 se compararon los resultados obtenidos en las dos implementaciones Sobel respecto al uso de recursos, rendimiento y costo de programación. De acuerdo a los resultados obtenidos, los desarrollos en alto nivel emplearon mayor cantidad de recursos hardware y obtuvieron menor rendimiento respecto a la implementación HDL. Esto sucedió por el proceso automático de la herramienta Vivado HLS para obtener la descripción HDL del sistema a partir de algoritmo en alto nivel, generando más señales de control y una MEF más grande. Esta situación generó un impacto negativo en el tiempo de respuesta y la cantidad de recursos utilizados.

  Por otro lado, la versión Sobel en HDL demandó más tiempo de desarrollo y mayor cantidad de SLOC respecto a la versión del filtro en HLS, aunque se observaron mejoras en cuanto a uso de recursos y rendimiento del filtro en bajo nivel. Respecto al uso de recursos, la plataforma hardware ZYBO fue capaz de sintetizar ambas versiones del filtro sin emplear una gran cantidad de los mismos. En cuanto al rendimiento, se observó una diferencia en tiempo de ejecución significativa para imágenes pequeñas a favor del filtro en HDL, pero esa diferencia disminuyó al aumentar el tamaño de imagen.

Finalmente, se analizaron las diferencias entre las implementaciones Sobel de este trabajo respecto a otras propuestas existentes. Aunque los resultados encontrados presentan tendencias similares a investigaciones previas, este estudio se destaca del resto por varias cuestiones. En primer lugar, se implementaron sistemas Sobel completos y funcionales para ambos enfoques de programación FPGA, los cuales se encuentran disponibles en un repositorio web público para beneficio de la comunidad. En segundo lugar, no se requerió de herramientas adicionales, librerías de imagen/video u otro factor que pudiera favorecer alguna de las dos implementaciones. En tercer lugar, se documentó minuciosamente el diseño de las propuestas, el proceso de comparación y los resultados obtenidos. La combinación de estos factores permitió alcanzar una comparación rigurosa de enfoques de desarrollo para FPGA de mayor calidad.

Del cumplimiento de los objetivos específicos, se considera que se ha alcanzado el objetivo general estipulado al inicio de esta investigación. Luego del análisis de los resultados obtenidos se concluyó que el esfuerzo de programación requerido por HDL fue significativamente mayor respecto al de HLS y sólo se obtuvo una leve mejora en relación al uso de recursos y rendimiento. En contextos similares a los de este estudio, HDL sólo resultaría conveniente en los diseños donde el uso de recursos y/o el tiempo de respuesta fueran parámetros de diseño sumamente críticos. De otra forma, HLS es una mejor opción que permite reducir significativamente el esfuerzo de programación y el tiempo de desarrollo.

## 5.2. Trabajos Futuros

Las líneas de trabajo futuro que se desprenden de esta investigación son:

- Modificar las implementaciones Sobel HDL y HLL para permitir el procesamiento de video en tiempo real. Los resultados obtenidos con los filtros modificados permitirían complementar y extender las conclusiones de esta investigación.
- Extender el estudio comparativo entre enfoques de programación para FPGA utilizando como caso de estudio algoritmos no convolucionales. Como se mencionó en el Capítulo 1, las FPGA tienen gran aceptación en el ámbito de procesamiento de imágenes por la naturaleza convolucional de las aplicaciones. Utilizar otros casos de estudio pueden

establecer fuertes vínculos entre el tipo de aplicación y el enfoque de programación en FPGA.

- Explorar otros lenguajes de desarrollo para FPGA (como puede ser OpenCL, oneAPI o Python), ya que contribuiría a enriquecer el estudio comparativo realizado.

# Referencias


Aledo, D., Schafer, B., y Moreno, F. (2019, 03). Vhdl vs. systemc: Design of highly parameterizable artificial neural networks. *IEICE Transactions on Information and Systems*, *E102.D*, 512-521. doi: 10.1587/transinf.2018EDP7142

*Amazon and xilinx deliver new FPGA solutions.* (2017). Descargado 2019-11-01, de https://www.forbes.com/sites/moorinsights/2017/09/27/amazon-and-xilinx-deliver-new-fpga-solutions/#5e40eb0b2370

*Amd to acquire xilinx, creating the industry's high performance computing leader.* (2020). Descargado de https://www.amd.com/en/press-releases/2020-10-27-amd-to-acquire-xilinx-creating-the-industry-s-high-performance-computing

*Baidu deploys xilinx FPGAs in new public cloud acceleration services.* (2017). Descargado 2019-10-11, de https://www.xilinx.com/news/press/2017/baidu-deploys-xilinx-fpgas-in-new-public-cloud-acceleration-services.html

Bhatt, K., Tarey, V., y Patel, P. (2012). Analysis of source lines of code(SLOC) metric. *International Journal of Emerging Technology and Advanced Engineering*, *2*.

*CERN openlab explores new CPU/FPGA processing solutions.* (2017). Descargado 2019-10-11, de https://www.hpcwire.com/2017/04/14/xeon-fpga-processor-tested-at-cern/

Chaple, G., y Daruwala, R. D. (2014, abril). Design of Sobel operator based image edge detection algorithm on FPGA. En *2014 International Conference on Communication and Signal Processing* (pp. 788–792). Melmaruvathur, India: IEEE. Descargado 2019-12-13, de http://ieeexplore.ieee.org/document/6949951/ doi: 10.1109/ICCSP.2014.6949951

Che, S., Li, J., Sheaffer, J. W., Skadron, K., y Lach, J. (2008). Accelerating compute-intensive applications with gpus and fpgas. En *2008 symposium on application specific processors* (p. 101-107). doi: 10.1109/SASP.2008.4570793

Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., y Zhang, Z. (2011, 05). High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *30*, 473 - 491. doi: 10.1109/TCAD.2011.2110592

Edwards, S. A. (2006). The challenges of synthesizing hardware from c-like languages. *IEEE Design Test of Computers*, *23*(5), 375-386. doi: 10.1109/MDT.2006.134

Escobar, F. A., Chang, X., y Valderrama, C. (2016). Suitability analysis of fpgas for heterogeneous platforms in hpc. *IEEE Transactions on Parallel and Distributed Systems*, *27*(2), 600-612. doi: 10.1109/TPDS.2015.2407896

Foster, H. D. (2018). 2018 fpga functional verification trends. En *2018 19th international workshop on microprocessor and soc test and verification (mtv).* doi: 10.1109/MTV.2018.00018

*FPGA developer, augmented reality.* (2019). Descargado 2019-11-01, de https://www.facebook.com/careers/v2/jobs/283243269009556

Gurel, M. (2016). *A comparative study between RTL and HLS for image processing applications with FPGAs* (Tesis de Master). Descargado de https://escholarship.org/uc/item/9vx1s37b

Harris, S., y Harris, D. (2013). *Digital design and computer architecture* (2.ª ed.). Morgan Kaufmann. Descargado de https://www.elsevier.com/books/digital-design-and-computer-architecture/harris/978-0-12-394424-5

Hiraiwa, J., y Amano, H. (2013). An fpga implementation of reconfigurable real-time vision architecture. En *2013 27th international conference on advanced information networking and applications workshops* (p. 150-155). doi: 10.1109/WAINA.2013.131

Hurtado, J. (2008). *Cómo formular objetivos de investigación* (segunda ed.). Quirón.

Jones, T. C. (2007). *Estimating software costs* (2.ª ed.). USA: McGraw-Hill, Inc.

Kapre, N., y Bayliss, S. (2016). Survey of domain-specific languages for fpga computing. En *2016 26th international conference on field programmable logic and applications (fpl)* (p. 1-12). doi: 10.1109/FPL.2016.7577380

Kuon, I., Tessier, R., y Rose, J. (2007, 01). Fpga architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, *2*, 135-253. doi: 10.1561/1000000005

Marc-André, T. (2018). Two fpga case studies comparing high level synthesis and manual hdl for hep applications. *arXiv: Instrumentation and Detectors*. Descargado de https://www.researchgate.net/publication/326057215_Two_FPGA_Case_Studies_Comparing_High_Level_Synthesis_and_Manual_HDL_for_HEP_applications

Martin, G., y Smith, G. (2009). High-level synthesis: Past, present, and future. *IEEE Design Test of Computers*, *26*(4), 18-25. doi: 10.1109/MDT.2009.83

Mehra, R., y Verma, R. (2012). Area efficient FPGA implementation of sobel edge detector for image processing applications. , *56*(16), 7–11. Descargado 2021-03-29, de http://research.ijcaonline.org/volume56/number16/pxc3883086.pdf doi: 10.5120/8973-3086

*Microsoft uses intel FPGAs for smarter bing searches.* (2018). Descargado 2019-10-11, de https://www.eweek.com/cloud/microsoft-uses-intel-fpgas-for-smarter-bing-searches

Munshi, A. (2009). The opencl specification. En *2009 ieee hot chips 21 symposium (hcs)* (p. 1-314). doi: 10.1109/HOTCHIPS.2009.7478342

Nane, R., Sima, V., Pilato, C., Choi, J., Fort, B., Canis, A., ... Bertels, K. (2016). A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *35*(10), 1591-1604. doi: 10.1109/TCAD.2015.2513673

Nausheen, N., Seal, A., Khanna, P., y Halder, S. (2018, febrero). A fpga based implementation of sobel edge detection. *Microprocess. Microsyst.*, *56*(C), 84–91. Descargado de https://doi.org/10.1016/j.micpro.2017.10.011 doi: 10.1016/j.micpro.2017.10.011

Nosrat, A., y S. Kavian, Y. (2012). Hardware description of multi-directional fast sobel edge detection processor by VHDL for implementing on FPGA. *International Journal of Computer Applications*, *47*, 1-7. doi: 10.5120/7533-9872

Pavan Kumar, M. (2019). *Hardware Acceleration of Edge Detection Using HLS.* Descargado 2019-11-01, de https://bit.ly/3i88uBW (Undergraduate thesis, California State University)

Pelcat, M., Bourrasset, C., Maggiani, L., y Berry, F. (2016). Design productivity of a high level synthesis compiler versus hdl. En *2016 international conference on embedded computer systems: Architectures, modeling and simulation (samos)* (p. 140-147). doi: 10.1109/SAMOS.2016.7818341

Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., ... Burger, D. (2015). A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, *35*(3), 10-22. doi: 10.1109/MM.2015.42

Rashmi, Kumar, M., y Saxena, R. (2013, junio). Algorithm and Technique on Various Edge Detection : A Survey. *Signal & Image Processing: An International Journal (SIPIJ)*, *4*(3), 65-75. Descargado de https://doi.org/10.5121/sipij.2013.4306 doi: 10.5121/sipij.2013.4306

Ren, H. (2014). A brief introduction on contemporary high-level synthesis. En *2014 ieee international conference on ic design technology* (p. 1-4). doi: 10.1109/ICICDT.2014.6838614

Sanduja, V., y Patial, R. (2012). Sobel edge detection using parallel architecture based on FPGA. *International Journal of Applied Information Systems*. Descargado de https://www.ijais.org/archives/volume3/number4/220-0515

Selvaraj, H., Daoud, L., y Zydek, D. (2013, 09). A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing. En (Vol. 240). doi: 10.1007/978-3-319-01857-7_47

Stamoulias, I., Kachris, C., y Soudris, D. (2017). Hardware accelerators for financial applications in hdl and high level synthesis. En *2017 international conference on embedded computer systems: Architectures, modeling, and simulation (samos)* (p. 278-285). doi: 10.1109/SAMOS.2017.8344641

Stanciu, A., y Gerigan, C. (2017). Comparison between implementations efficiency of hls and hdl using operations over galois fields. En *2017 ieee 23rd international symposium for design and technology in electronic packaging (siitme)* (p. 171-174). doi: 10.1109/SIITME.2017.8259883

Tessier, R., Pocek, K., y DeHon, A. (2015). Reconfigurable computing architectures. *Proceedings of the IEEE*, *103*(3), 332–354. doi: 10.1109/JPROC.2014.2386883

Trimberger, S. M. (2015). Three ages of fpgas: A retrospective on the first thirty years of fpga technology. *Proceedings of the IEEE*, *103*(3), 318-331. doi: 10.1109/JPROC.2015.2392104

Vallina, F. M., Kohn, C., y Joshi, P. (2012). *Zynq all programmable SoC sobel filter implementation using the vivado HLS tool.* Descargado de https://bit.ly/3h6egD1

Windh, S., Ma, X., Halstead, R. J., Budhkar, P., Luna, Z., Hussaini, O., y Najjar, W. A. (2015). High-level language tools for reconfigurable computing. *Proceedings of the IEEE*, *103*(3), 390–408. Descargado

2019-08-27, de http://ieeexplore.ieee.org/document/7086410/ doi: 10.1109/JPROC.2015.2399275

Wu, Q., Ha, Y., Kumar, A., Luo, S., Li, A., y Mohamed, S. (2014). A heterogeneous platform with gpu and fpga for power efficient high performance computing. En *2014 international symposium on integrated circuits (isic)* (p. 220-223). doi: 10.1109/ISICIR.2014.7029447

Yang, H., Zhang, J., Sun, J., y Yu, L. (2014, 10). Review of advanced fpga architectures and technologies. *Journal of Electronics*, *31*, 371-393. doi: 10.1007/s11767-014-4090-x

Zwagerman, M. (2015). *High level synthesis, a use case comparison with hardware description language* (Tesis de Master). Descargado de https://scholarworks.gvsu.edu/theses/755