UNIVERSIDAD AUTÓNOMA DE SINALOA

FACULTAD DE INFORMÁTICA CULIACÁN
FACULTAD DE CIENCIAS DE LA TIERRA Y EL ESPACIO

POSGRADO EN CIENCIAS DE LA INFORMACIÓN

IMPLEMENTACIÓN EFICIENTE PARA ACELERACIÓN POR HARDWARE EN SISTEMAS FPGA-MICROPROCESADOR

TESIS

QUE COMO REQUISITO PARA OBTENER EL GRADO DE MAESTRO EN CIENCIAS DE LA INFORMACIÓN

PRESENTA:
LIC. MARTIN GALAVIZ BERNAL

DIRECTOR DE TESIS:
DR. JESÚS ROBERTO MILLÁN ALMARAZ

CODIRECTOR DE TESIS:
DR. ARTURO YEE RENDÓN

CULIACÁN, SINALOA. FEBRERO DE 2023

Dedicatoria

A mis padres, hermanos y amigos.

Agradecimientos

A mi familia, por su apoyo incondicional.

Al Dr. Jesús Roberto Millán Almaraz y al Dr. Arturo Yee Rendón por su guía durante el desarrollo de este trabajo.

Al Consejo Nacional de Ciencia y Tecnología (CONACyT) de México por otorgarme una beca para realizar mis estudios de Maestría (CVU 799911).

A la Universidad Autónoma de Sinaloa y al Posgrado en Ciencias de la Información.

A mis profesores por brindarme su conocimiento.

A mis compañeros y amigos por su compañía y risas.

Lista de acrónimos

ARM	*Advanced RISC Machine*
ASIC	*Application Specific Integrated Circuit*
AXI	*Advanced eXtensible Interface*
CISC	*Complex Instruction Set Computing*
CPU	*Central Processing Unit*
DMA	*Direct Memory Access*
FIFO	*First In - First Out*
FPGA	*Field Programmable Gate Array*
GPIO	*General Purpose Input/Output*
GPU	*Graphics Processing Unit*
HDL	*Hardware Description Language*
PL	*Programmable Logic*
PS	*Processing System*
RAM	*Random Access Memory*
RISC	*Reduced Instruction Set Computing*
SoC	*System on a Chip*
VHDL	*Very High Speed Integrated Circuit Hardware Description Language*

Resumen

Actualmente, es cada vez más común implementar algoritmos sobre arquitecturas de cómputo heterogéneas, en donde existe más de una unidad de procesamiento de datos. En esta tesis se pretende integrar arquitecturas de cómputo, en un enfoque híbrido, en donde coexisten un CPU de arquitectura ARM y un FPGA en el mismo circuito integrado. En esta arquitectura, se planea implementar algoritmos con posibilidades de ser procesados en paralelo, tal es el caso de algoritmos de procesamiento de imágenes. Para lograr lo anterior, es necesario modificar los algoritmos secuenciales de procesamiento de imágenes y adaptarlos para que utilicen recursos que posibiliten la paralelización de tareas.

Una de las principales características de los FPGA es la capacidad de reconfigurar su estructura interna, es decir, cambiar el diseño lógico según la necesidades de los usuarios, esta diferencia que existe con la mayoría de circuitos integrados permite desarrollar sistemas digitales especializados en determinadas tareas, por ejemplo, módulos de procesamiento de audio o filtros digitales de series de tiempo, trayendo consigo grandes ventajas en el aprovechamiento de los recursos disponibles y una mejora sustancial en el consumo energético comparado con plataformas de cómputo tradicionales.

Al contar con una arquitectura híbrida, es posible delegar tareas específicas a cada uno de los componentes de la arquitectura, comúnmente las tareas relacionadas con la gestión de periféricos de alto nivel, el manejo de archivos y sincronización entre dispositivos son asignados a la parte del CPU, en cambio, las tareas repetitivas y con la característica de que pueden ser procesadas en paralelo son asignadas al FPGA.

Algunos de los principales aportes de esta investigación son los listados a continuación:

- El desarrollo de una plataforma para el procesamiento de imágenes basado en convoluciones 2D donde intervengan tanto el CPU como el FPGA de una plataforma híbrida.
- El diseño de un sistema digital capaz de hacer un manejo eficiente de datos de entrada (valores de pixeles de imágenes) reduciendo la memoria RAM necesaria para la ejecución del algoritmo principal de procesamiento, en este caso una serie de convoluciones.
- El desarrollo de un programa encargado del intercambio de datos entre ambas arquitecturas, al igual que se encarga de coordinar dichas arquitecturas.

Abstract

Currently, it is increasingly common to implement algorithms on heterogeneous computing architectures, where there is more than one data processing unit. This thesis intends to integrate computing architectures, in a hybrid approach, where an ARM architecture CPU and an FPGA coexist in the same integrated circuit. In this architecture, it is planned to implement algorithms with the possibility of being processed in parallel, such is the case of image processing algorithms. To achieve the above, it is necessary to modify the sequential image processing algorithms a bit and obtain a series of processes that will be executed in parallel.

One of the main characteristics of FPGAs is the ability to reconfigure their internal structure, by changing the logical design according to the needs of users. This difference that exists with most integrated circuits allows the development of digital systems specialized in certain tasks, for example, audio processing modules or time series digital filters, bringing great advantages in the use of available resources and a substantial improvement in energy consumption compared to traditional computing platforms.

By having two architectures coexisting in the same integrated circuit, it is possible to delegate specific tasks to each of the computing architectures, commonly the tasks related to the management of high-level peripherals, the handling of files and synchronization between devices are assigned to the CPU, on the other hand, repetitive tasks and with the characteristic that they can be processed in parallel are assigned to the FPGA.

Some of the main contributions of this research are listed below:

- The development of a platform for image processing based on 2D convolutions where both the CPU and the FPGA of a hybrid platform are involved.
- The design of a digital system capable of efficiently handling input data (image pixel values) by shortening the RAM memory necessary to execute the main processing algorithm, in this case a series of convolutions.
- The development of a program in charge of exchanging data between both architectures, as well as being in charge of coordinating said architectures.

Índice general

Lista de figuras

Lista de tablas

Lista de códigos

1. Introducción

En esta investigación se abordaron temas relacionados con la utilización de un Field Programmable Gate Array (FPGA) híbrido como acelerador de algoritmos. Se vieron algunos de los usos y alcances que pueden tener estos dispositivos y la ventaja que pudieran representar por encima de otras plataformas o arquitecturas de hardware como lo es un microprocesador (CPU) o una tarjeta de video (GPU).

Un FPGA híbrido cuenta con dos dispositivos (un FPGA[1] y un CPU) acoplados en el mismo chip, pero en el caso del ZYNQ SoC 7000 ambos dispositivos están comunicados por varios canales de comunicación basados en *Advanced eXtensible Interface* (AXI[2]), con base en la descripción de Xilinx (empresa fabricante de FPGA) [42] algunos canales son de alto rendimiento para ser usados en transferencias de datos, otros canales son solo para la comunicación directa entre ambos dispositivos. Tener estos canales de comunicaciones dedicados permite un reloj del bus de datos de hasta de 150 MHz y un ancho de hasta 128 bits haciendo muy eficiente la transferencia de datos, y así utilizar el FPGA como un coprocesador para acelerar los algoritmos mediante hardware.

Los FPGA híbridos de bajo costo más conocidos son la serie ZYNQ SOC 7000 de Xilinx basados en los Artix-7 (serie y modelo de un FPGA de Xilinx), aunque también existen unos que son competencia directa de ellos que son los de la empresa Altera (perteneciente a Intel) que ofrecen sus DEO-Nano-SOC basados en Cyclone V (Serie y modelo de un FPGA de la empresa Altera), ambas plataformas cuentan con un CPU de doble núcleo ARM Cortex-A9 en el mismo chip. Ambas marcas de FPGA son equiparables en rendimiento [31] para ciertas cargas de trabajo con o sin un sistema operativo.

Las aplicaciones donde comúnmente se utilizan este tipo de plataformas son aquellas que necesitan hacer un procesamiento en tiempo real medianamente demandante en cuanto a recursos de CPU, por ejemplo, en el procesamiento de video en tiempo real (a más de 30 cuadros por segundo), la codificación y decodificación de video, imágenes y audio, en algoritmos de encriptación y aplicaciones similares. Esto es así debido a que

[1] Es un dispositivo programable que cuenta con un conjunto de recursos lógicos que pueden ser interconectados en el momento de su utilización. Son utilizados principalmente para prototipar sistemas digitales complejos.

[2] Es un protocolo de interfaz perteneciente a la empresa ARM, es usado en comunicaciones dentro de un chip.

estas plataformas son bastante flexibles y escalables. Un buen diseño puede alcanzar un rendimiento bastante alto para el precio de esta tecnología, un ejemplo seria filtrar 80 series de tiempo de forma paralela a una frecuencia superior a los 100 Mhz, esto es posible gracias a que el FPGA híbrido que se estudia cuenta con 80 Procesadores Digitales de Señales (DSP).

Con base en lo anterior, este trabajo de investigación se utilizó la arquitectura híbrida (FPGA-CPU) para acelerar algoritmos de procesamiento de imagen diseñados para funcionar en la parte del FPGA. La CPU se encargó de funciones de sincronización entre dispositivos, almacenamiento y decodificación de imágenes, mientras que el FPGA se utilizó en el procesamiento de imágenes obtenidas de localidades de RAM para posteriormente retornar el resultado a otra localidad. Además, se implementó el algoritmo que se ejecuta en la CPU en lenguaje C, mientras que, se implementó un sistema digital para acelerar el algoritmo mediante el lenguaje de descripción de hardware *Very High Speed Integrated Circuit Hardware Description Language* (VHDL).

1.1. Planteamiento del problema

Actualmente, en todo el mundo se ha incrementando el consumo de contenido multimedia en la Web y en sí de recursos informáticos con una marcada tendencia de subida en servicios de streaming [32]. Se ha visto cómo ha evolucionado la forma en que se tiene que procesar toda la información que se quiere consumir, es decir, antes bastaba con arquitecturas de computación tradicionales, ahora es necesario utilizar tecnologías emergentes que están mucho más optimizadas para realizar cierto tipo de tareas, un ejemplo de ello, el utilizar GPU para renderizado de videojuegos y contenido multimedia. También, se ha visto que con la automatización de procesos es necesario tener hardware especializado para realizar tareas específicas, y es en este caso, donde la computación tradicional a veces no es suficiente. Un ejemplo claro podría ser el de la conducción automática de automóviles de última generación donde no solamente se tienen que utilizar CPU, sino que también, en algunas ocasiones, se utilizan GPU, en otros casos ASIC (circuito integrado de aplicación específica) y hasta FPGA [9].

Unas tecnologías muy interesantes que han salido recientemente son los FPGA híbridos, es decir, arquitecturas compuestas por otras dos (CPU y FPGA) conviviendo en el mismo chip. Desafortunadamente al igual que los FPGA, el número de expertos adoptando estas tecnologías para acelerar algoritmos es limitado, esto puede verse en la poca popularidad que tienen en instituciones de investigación de la región, a pesar de tener algunas características interesantes en cuanto consumo energético y flexibilidad

que tienen comparado con aceleradores como los GPU en algunas aplicaciones como la implementación de redes neuronales [11].

Una de las principales razones a la que se atribuye una mayor dificultad de utilizar FPGAs o FPGAs híbridos en comparación con otras arquitecturas (como los CPU) es el hecho de que es necesario tener nociones de diseño de sistemas digitales en algún lenguaje *Hardware Description Language* (HDL). Tener dichas nociones requieren una formación en electrónica digital, la cual tiene cierta curva de aprendizaje y una metodología muy distinta a la empleada por programadores. Por mencionar algunas otras dificultades, se tienen que crear el sistema digital del acelerador, se tiene que crear el sistema digital responsable de la comunicación entre FPGA y CPU y se tiene que crear un software encargado de coordinar todo el sistema y que además sea capaz de enviar, formatear y recibir datos. Es necesaria una metodología de co-diseño desde el inicio de cada proyecto que involucre FPGAs híbridos, esto si se planea utilizar las bondades de ambas arquitecturas. Aquí se plantea una metodología para crear una plataforma de aceleración haciendo uso de los FPGA híbridos basada en el co-diseño para mostrar los alcances de esta tecnología.

En este trabajo se desarrolló un proyecto que mostró cómo es que los FPGA híbridos tienen claras ventajas sobre otras tecnologías de cómputo tradicional en determinadas tareas, principalmente en aplicaciones que se ejecuten en tiempo real. Una de las principales ventajas que se mostró es el poder de cómputo que se puede tener con este tipo de tecnologías y la eficiencia energética en comparación de otras tecnologías basadas solamente en CPU.

En resumen, en esta investigación se intenta hacer una exploración de nuevas tecnologías de aceleración por hardware para conocer su factibilidad como opción tecnológica para ejecución de algoritmos demandantes de alto poder de computo como lo son el procesamiento de imágenes.

1.2. Objetivos

1.2.1. General

Desarrollar un plataforma de aceleración de algoritmos basada en una arquitectura híbrida FPGA-CPU orientada a procesamiento de imágenes RGB y en escala de grises por hardware.

1.2.2. Específicos

- Desarrollar una arquitectura de hardware para procesamiento de imagenes usando herramientas HDL.
- Poner en marcha la arquitectura desarrollada con un sistema operativo linux a la medida y con los paquetes de software para realizar un proyecto de procesamiento de imágenes.
- Desarrollar una metodología de comunicación hardware software para el sistema FPGA-CPU propuesto basada en el protocolo AXI.
- Evaluar el sistema de procesamiento de hardware desarrollado mediante un análisis comparativo con otras arquitecturas de cómputo.
- Validar el funcionamiento de la arquitectura.

1.3. Hipótesis

Los algoritmos de procesamiento de imágenes pudieran ser implementados de manera eficiente en arquitecturas híbridas FPGA-CPU con las adecuadas configuraciones y un buen diseño de proyectos de aceleración de algoritmos. Se puede obtener un rendimiento equiparable a una plataforma de cómputo tradicional en la aceleración de algoritmos con el uso de la arquitectura ZYNQ de Xilinx.

1.4. Justificación

Una de las razones más importantes de utilizar arquitecturas híbridas FPGA-CPU es que algunos algoritmos pueden ser acelerados y procesados en tiempos reducidos comparados con una arquitectura convencional de cómputo.

En algunas ocasiones las arquitecturas híbridas tienen gran flexibilidad y las posibilidades de escalabilidad son muy altas al tener dos arquitecturas conviviendo en el mismo dispositivo. También es importante tomar en cuenta que el consumo energético de implementar algoritmos en hardware y no en software es menor, tanto que la arquitectura que se plantea en este trabajo de investigación solo consume 10 Watts.

La aceleración de algoritmos usando FPGAs híbridos es muy poco explorada por las diferentes disciplinas que se tienen que dominar y eso hace que se convierta en un área

de oportunidad para mucha investigación.

1.5. Organización de la tesis

Este documento de tesis está organizado de la siguiente forma. En el Capítulo 2 se describe la plataforma híbrida FPGA-CPU y los elementos que la componen, haciendo énfasis en la forma en la que dichos elementos pueden comunicarse, se definen conceptos relacionados con este tema de investigación y, por último, se hace un estudio del estado del arte de los trabajos de procesamiento de imágenes que han sido implementados en la arquitectura que se planea utilizar en este trabajo. En el Capítulo 3 se hace una descripción de la metodología a utilizar para lograr acelerar un algoritmo de procesamiento de imágenes en una arquitectura híbrida de la familia ZYNQ de Xilinx. En el Capítulo 4 se describen los resultados experimentales de la propuesta realizada en la presente tesis. En el Capítulo 5 se presentan las conclusiones de la investigación realizada y se determina si se obtuvo una implementación exitosa de la aceleración de algoritmos de procesamiento de imágenes por hardware, aquí también se plantean algunas ideas de mejora de la implementación realizada al igual que otras opciones que pueden ser tomadas en cuenta. Este trabajo intenta destacar la relevancia que tienen las plataformas híbridas FPGA-CPU en la aceleración de algoritmos y las ventajas que pueden tener de arquitecturas de procesamiento tradicionales.

2. Marco teórico

En este capítulo se describen los conceptos básicos y técnicos relacionados con este trabajo de investigación. Lo mencionado anteriormente es para facilitar la comprensión de este trabajo al lector.

2.1. Arquitecturas de hardware

En esta sección se revisan algunos conceptos relacionados al hardware utilizado en la presente tesis, mencionando las diferencias que existen entre distintas arquitecturas.

2.1.1. FPGA

Un FPGA o arreglo de compuertas programables en campo puede definirse como un conjunto de recursos lógicos digitales los cuales pueden ser reconfigurados por el usuario, es un dispositivo que ofrece la ventaja de una alta velocidad de prototipado de diseños de sistemas digitales [6]. Estos dispositivos ofrecen capacidades de paralelizaje al momento de diseñar hardware dedicado a ejecutar procesos aptos para el paralelizaje.

Con el rápido prototipado ofrecido por los FPGAs es posible crear sistemas digitales de propósito específico sin necesidad de fabricar un Circuito Integrado para Aplicaciones Específicas (ASIC, por sus siglas en inglés), ahorrando mucho dinero y optimizando el tiempo de diseño y prueba [23]. Año tras año, tras la salida de más y mejores herramientas capaces de reprogramar o reconfigurar estos dispositivos, hay más gente involucrada con su uso y son más las aplicaciones que se les da. En la actualidad ya existen hasta servicios en la nube que te rentan estos dispositivos con el propósito de hacer aceleración de algoritmos desde hardware.

Los FPGAs comúnmente son reconfigurados con ayuda de los lenguajes de descripción de hardware y de ellos existen dos lenguajes de descripción de hardware que son los más utilizados para diseñar sistemas digitales (o un simple circuito combinacional), estamos hablando de VHDL y Verilog.

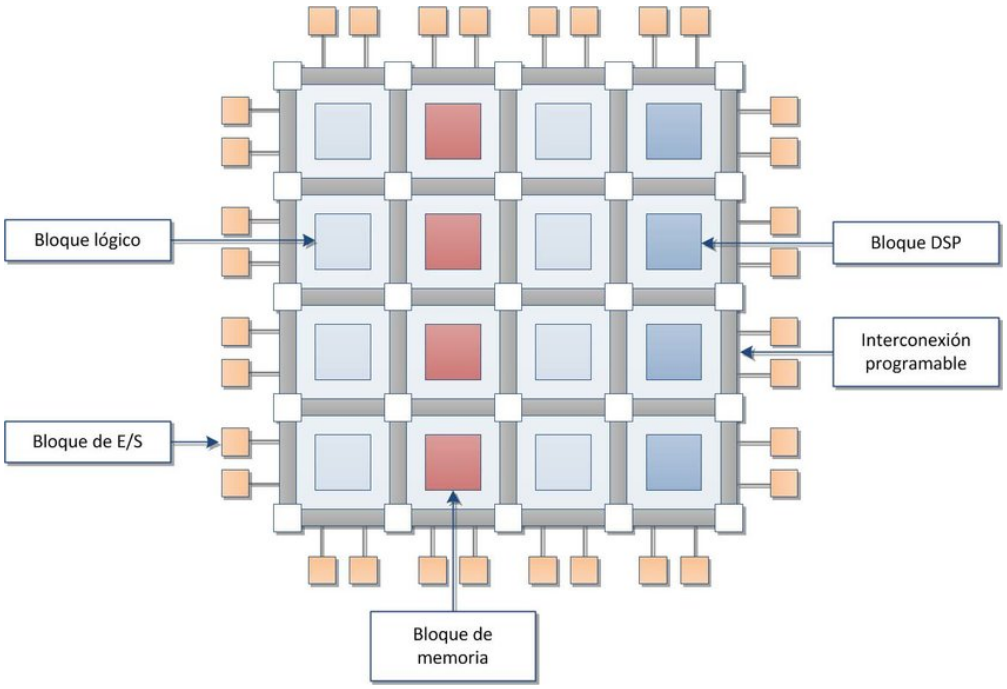


Figura 2-1.: Diagrama de la composición de un FPGA extraído de [39]

Los FPGA están compuestos por una serie de elementos, en primera instancia se tienen los bloques lógicos programables o elementos lógicos, se cuenta con bloques de memoria SRAM integrada en los FPGA, se tienen una serie de puertos de entradas y salidas (E/S), se tiene una malla de interconexión programable y desde hace algunos años cuentan con procesadores digitales de señales (DSP). Estos últimos son el motivo del porque se están utilizando los FPGA en la aceleración de algoritmos. En la Figura **2-1** se puede ver una ilustración de un FPGA.

2.1.2. Microprocesador o CPU

Un CPU es llamado coloquialmente como el cerebro de una computadora, es en este dispositivo donde se ejecutan las instrucciones de cómputo que un usuario desea que su computadora ejecute, estas instrucciones son interpretadas por el CPU a través de un lenguaje de máquinas (binario). En este dispositivo se realizan cálculos lógicos y numéricos. Cada CPU moderno puede tener más de un núcleo de procesamiento. Cada núcleo está constituido por registros, una unidad de control, una unidad aritmético lógica (ALU) y una unidad de cálculo de números reales o con punto decimal (float). Por lo general los CPU modernos trabajan a frecuencias de reloj en el orden de los GHz, es decir, pueden

ejecutar miles de millones de instrucciones cada segundo [2] [17] [29] [37].

Existen diferentes tipos de CPU de acuerdo con su arquitectura.

Arquitectura CISC

La arquitectura de CPU *Complex instruction set computing* (CISC) está caracterizada por tener un conjunto de instrucciones muy amplio donde se permiten operaciones complejas que en muchos casos se ejecutan en más de un ciclo de operación del CPU, es decir, aunque el CPU tenga una frecuencia de reloj determinada no necesariamente se ejecutan las instrucciones completas a la misma frecuencia. Algunos de los objetivos de la arquitectura son la reducción de la cantidad de memoria usada (RAM), la reducción de cargas y descargas de datos del disco duro, busca la compatibilidad entre distintas generaciones de computadoras, entre otros enfoques que llegaban a hacer las instrucciones muy complejas [4].

Por la naturaleza del conjunto de instrucciones es muy difícil lograr un paralelismo de instrucciones entre varios núcleos del CPU ya que no todas las instrucciones se ejecutan en la misma cantidad de ciclos de reloj. Haisamman en 2008 [10] dijo que las principales características de los CPUs CISC son modos de direccionamiento complejos, instrucciones con muchos tamaños, instrucciones que tienen operaciones de memoria a memoria o memoria a registro y una unidad de control[1] microprogramada.

La arquitectura CISC es la más utilizada para el uso en CPUs de computadoras de escritorio de uso domestico y en laptops. Actualmente, algunos de los CPUs más conocidos de esta arquitectura son los fabricados por las empresas Intel y por *Advanced Micro Devices* (AMD).

Arquitectura RISC

La arquitectura de CPU *Reduced Instruction Set Computing* (RISC) está caracterizada por un conjunto de instrucciones reducido en comparación a la arquitectura CISC con el fin de crear CPUs más eficientes energéticamente. Algunas de sus principales características dichas por [10] son las siguientes:

[1] Es uno de los principales componentes de la CPU, existen los de tipo cableada y los de tipo microprogramada. Se encarga de buscar y decodificar instrucciones para que posteriormente dichas instrucciones se ejecuten en la unidad de procesamiento.

1. Instrucciones de tamaño fijo y presentadas en un reducido número de formatos.
2. Solo las instrucciones de carga y almacenamiento acceden a la memoria de datos.
3. Confianza en la optimización del compilador.
4. Instrucciones ejecutadas en un ciclo de máquina.

Algunos ejemplos de CPUs bajo la arquitectura RISC son PowerPC, DEC (del inglés *Digital Equipment Corporation*) Alpha, MIPS (del inglés *Microprocessor without Interlocked Pipeline Stages*), ARM y SPARC (del inglés *Scalable Processor ARChitecture*).

Comparación de arquitectura CISC contra RISC

De las dos anteriores arquitecturas de CPU existe un ejemplo característico de cada una: los CPU x86 siendo los más populares de la arquitectura CISC y los CPU de ARM basados en RISC.

A manera de sintetizar un poco las características de ambas familias de CPU se muestra en la Tabla **2-1** comparativa de estos.

En un artículo publicado por Balid y colaboradores en 2013 [6], se muestran algunas comparaciones entre las arquitecturas x86 y ARM en cuatro implementaciones (cuatro procesadores distintos). El artículo muestra que se ejecutan cuatro *benchmark* o programas de referencia para comparar los cuatro procesadores. Los procesadores utilizados son un Intel Core i7 2700 y un Atom N450 por parte de la arquitectura x86, por parte de ARM se usaron un Cortex-A9 OMAP4430 y un Cortex-A8 OMAP3530. En las pruebas se muestra como los procesadores ARM necesitan de 2.8 a cerca de 30 veces más tiempo para ejecutar los programas de referencia en comparación con el procesador Intel Core i7. También, el consumo energético de los procesadores x86 es de 3 a cerca de 27 veces más alto que el procesador Cortex-A8 OMAP3530 al ejecutar dichos programas de referencia.

2.1.3. FPGA híbrido

Para definir lo que es un FPGA híbrido es necesario definir lo que es un *System on Chip* (SoC). Este es un chip que integra todo lo necesario para su funcionamiento, con esto se refiere a que en un solo circuito integrado se tiene el CPU, el controlador de memoria, interfaces de memoria y otros elementos que podrían colocarse de manera separada. En la Figura **2-2** se muestran dos sistemas similares pero con implementaciones distintas, a la izquierda se ve un *System On Board* (SoB) que contiene una serie de elementos para un sistema de procesamiento y a la derecha se presenta el mismo sistema pero en una

Tabla 2-1.: Comparación entre arquitecturas CISC y RISC.

X86 (CISC)	ARM (RISC)
Tamaño de instrucción variable (de 1 a 16 Bytes)	Tamaño de instrucción fijo (dos o cuatro Bytes)
Mayormente utilizados en computadoras de escritorio	Utilizados principalmente en dispositivos móviles
Consumo alto de energía (por ejemplo 125 W [17])	Bajo consumo de energía (por ejemplo 15 W en una tarjeta Raspberry Pi 4B [40])
Frecuencias de reloj altas (hasta 5 GHz [17])	Frecuencias de reloj bajas con respecto a los x86 actuales (de hasta 3.2 GHz [38]
Ocho registros de 32 bits y seis de 16 bits	16 registros de propósito general
Muy caros de producir	Baratos de producir
Ejecución de instrucciones complejas en pocos ciclos	Ejecución de instrucciones complejas en muchos ciclos
Implementaciones de propósito general	Implementaciones en sistemas embebidos
Tiene una unidad de memoria para implementar instrucciones complejas como operaciones de punto flotante	No tiene unidad de memoria y utiliza un hardware separado para implementar las instrucciones como temporizadores físicos para los retardos

implementación que integra los elementos del SoB en un solo circuito integrado SoC [7].

Los dispositivos SoC FPGA (FPGA-CPU) son dispositivos que en su interior integran una arquitectura de CPU y además un FPGA. Al estar el CPU y el FPGA en el mismo chip, existe una gran integración de las arquitecturas lo que se refleja en el bajo consumo y en el mayor ancho de banda en las comunicaciones de las dos arquitecturas.

Los SoC FPGA cuentan además de un conjunto amplio de periféricos, memoria en el mismo chip, interfaces de comunicación de alta velocidad entre el CPU y el FPGA. Por la singular arquitectura que tienen, estos dispositivos tienen una gran cantidad de aplicaciones con grandes posibilidades de escalabilidad y flexibilidad.

ZYBO

Una de las plataformas más utilizadas del tipo de arquitectura híbrida es ZYBO (*Zynq Board*) de Xilinx debido a su bajo costo y a que está pensada para entusiastas en el área

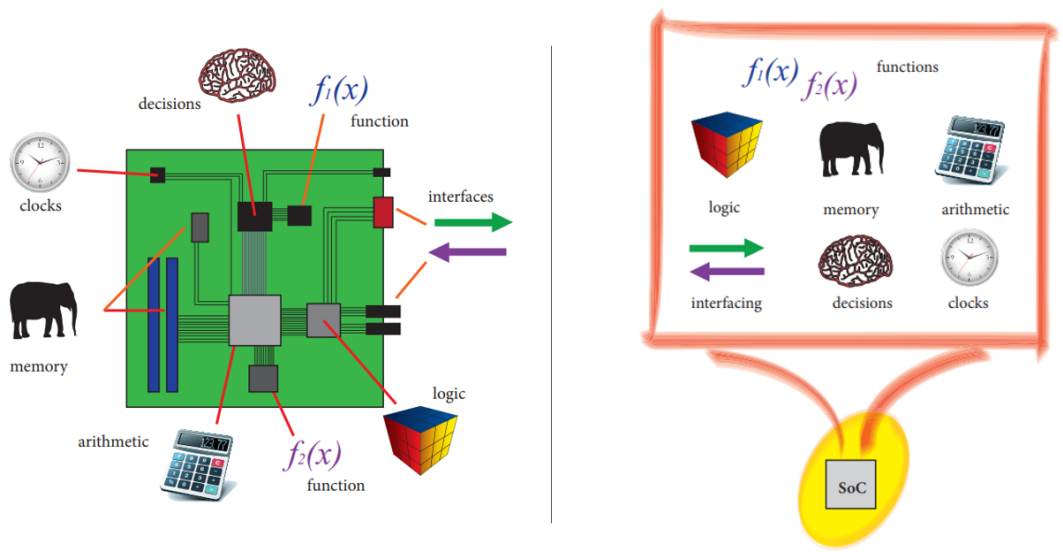


Figura 2-2.: Comparación entre un *System On Board* (izquierda) y un *System On Chip* (derecha), imagen extraída de [7].

de desarrollos de software embebido y diseños de circuitos digitales.

La tarjeta de desarrollo ZYBO es el miembro más pequeño de la familia ZYNQ-7000, la Z-7010 como también se le denomina. Esta familia está basada en la arquitectura Xilinx *All Programmable System-on-Chip* (AP SoC) [8].

El SoC ZYNQ-7010 contenido en la tarjeta ZYBO, combina un CPU de doble núcleo (PS) ARM Cortex-A9 con un FPGA tradicional (PL). El CPU es de grado de aplicación, es decir, es capaz de correr un sistema operativo basado en Linux. A su vez, el FPGA integrado está basado en una arquitectura Xilinx 7-series. Ambos dispositivos se encuentran comunicados a través de interfaces AXI [7].

La Figura **2-3** muestra en forma conceptual como está constituida una arquitectura ZYNQ, y esto es, con un sistema de procesamiento con dos núcleos de CPU, una parte de lógica programable (FPGA) y la interconexión AXI entre ambos componentes.

La tarjeta ZYBO con el SoC ZYNQ-7010 tiene una serie de componentes y módulos con ciertas características que resultan del interés de esta investigación, el la Tabla **2-2** se muestra dicha información.

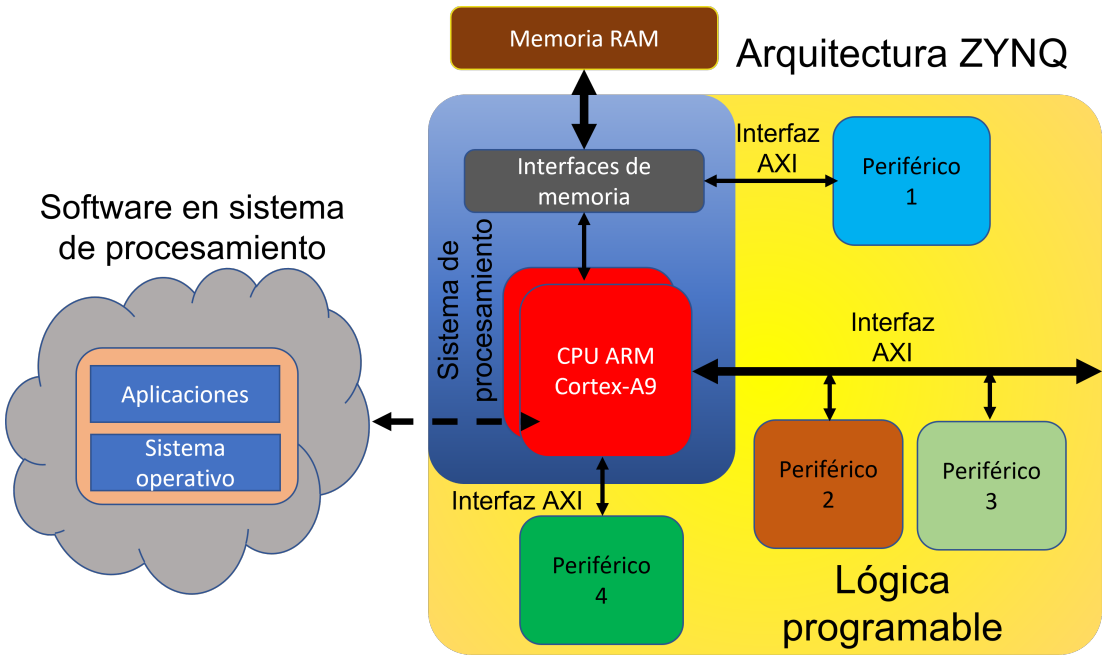


Figura 2-3.: Diagrama simplificado de la arquitectura ZYNQ [7]

AXI

AXI (*Advanced eXtensible Interface*) es un protocolo de comunicación en ráfagas, este protocolo pertenece a un conjunto de especificaciones denominado AMBA (del inglés, *Advanced Microcontroller Bus Architecture*) que es un estándar abierto para la conexión y gestión de bloques funcionales en un SoC [5]. AXI es muy útil a la hora de realizar transacciones de datos en las que estos se transmiten en paralelo en el interior de un circuito integrado. Existen diversas versiones e implementaciones de AXI. A continuación, se describe la interfaz y protocolo necesarios para AXI4 (cuarta versión de AXI).

La comunicación mediante el protocolo AXI se realiza a través de cinco canales independientes. Cada uno de estos canales dispone, además, de tres señales especiales, VALID, READY y LAST, una de cada una por canal, que permiten gestionar las comunicaciones [7].Es importante señalar también el hecho de que, en el protocolo AXI, uno de los extremos actúa como maestro (el dispositivo que coordinará la comunicación) y el otro como esclavo (el que responderá a las peticiones del maestro). Los canales de comunicación mencionados son los siguientes:

- **Canal de escritura**: El canal de escritura se encarga de transportar los datos desde el maestro hasta el esclavo, admitiendo anchos de bus de 8 a 1024 bits de ancho.

Tabla 2-2.: Características de los componentes y módulos de la plataforma ZYBO

Componente / módulo	Característica de interés
CPU Cortex-A9	Doble núcleo de frecuencia de reloj de 650 MHz
Memoria RAM	Tamaño de 512 MB, tipo Double Data Rate 3 (DDR3), 32 bits y ancho de banda de 1050 Mbps
FPGA	Contiene 4400 segmentos lógicos, contiene 240 KB de bloques rápidos de memoria RAM (BRAM), con frecuencias de reloj de hasta 450 MHz y contiene 80 procesadores digitales de señales (DSP)
Ranura para microSD	Con soporte para albergar sistema operativo

- **Canal de lectura**: Este canal transporta información desde el esclavo al maestro, siempre que este solicite una lectura de datos. El ancho de datos puede ser de entre 8 y 1024 bits.

- **Canal de dirección de escritura**: Este canal transporta la información sobre la dirección en la que se van a escribir los datos durante una transacción de escritura de datos y además algunas configuraciones importantes como el tamaño de cada transferencia, la cantidad de transferencias a realizar, entre otras configuraciones.

- **Canal de dirección de lectura**: Este canal transporta la información sobre la dirección de la que se van a leer los datos durante una transacción de lectura de datos y al igual que el canal anterior tiene, este canal cuenta con varias señales de configuración.

- **Canal de respuesta a escritura**: Este canal transporta el reconocimiento (Acknowledgement) del esclavo al maestro una vez se ha completado la escritura.

En la Figura **2-4** se muestra una representación de la arquitectura de escritura de AXI. En esa figura se logran observar los tres canales que corresponden a la arquitectura de la escritura que son el canal de dirección de escritura, el canal de escritura y el canal de respuesta a escritura. Para que el componente maestro transmita datos hacia el componente esclavo es necesario especificar la dirección donde se quieren escribir datos por medio del canal de dirección de escritura, después los datos son enviados por el canal de escritura y por ultimo el esclavo tiene que responder al maestro que la escritura se hizo satisfactoriamente mediante el canal de respuesta a la escritura.

En la Figura **2-5** se puede observar que la arquitectura de lectura de AXI necesita del canal de dirección de lectura y del canal de lectura. Para tener una lectura de datos es necesario que el componente maestro indique la dirección a leer al dispositivo esclavo y

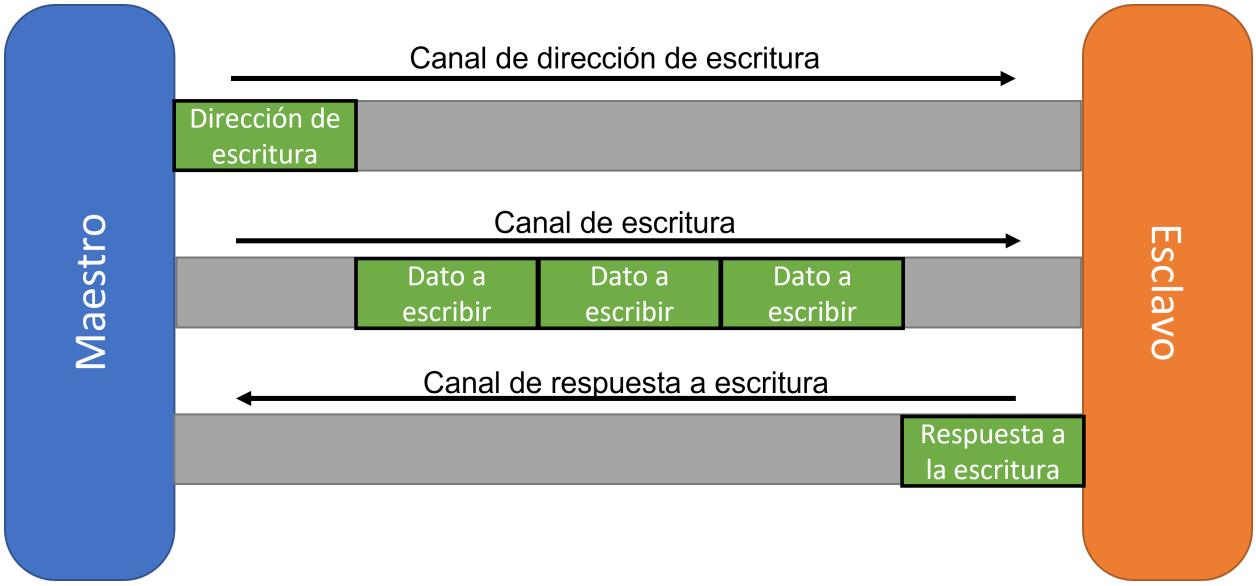


Figura 2-4.: Arquitectura AXI para escritura [7]

por medio del canal de lectura el maestro ya puede empezar a recibir los datos solicitados.

Para describir el funcionamiento del protocolo AXI es necesario que se explique cómo funciona la comunicación de cada canal del protocolo. Esto se hace utilizando las señales VALID, READY y LAST en cada canal con el protocolo denominado ***Handshake***.

El protocolo de comunicación *Handshake* entre maestro y esclavo también es el mismo para todos los canales, y es el siguiente:

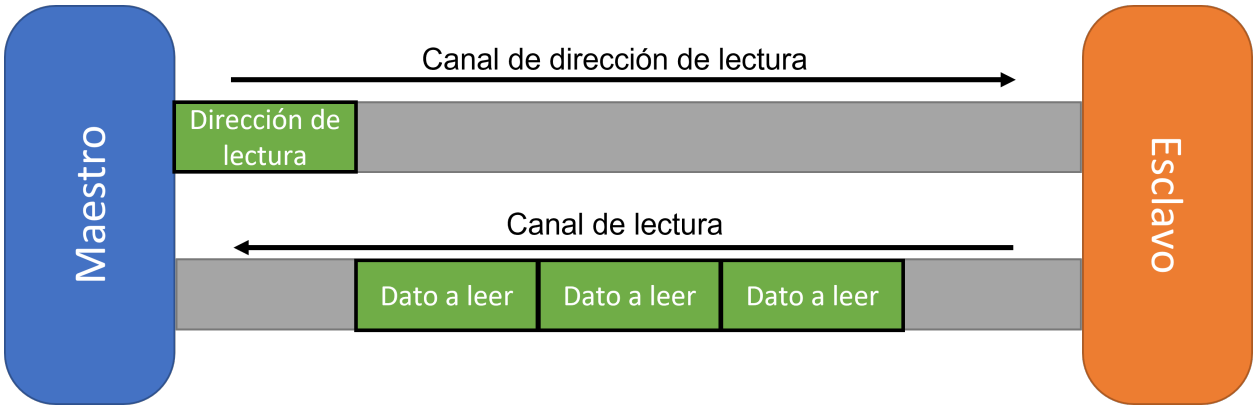


Figura 2-5.: Arquitectura AXI para lectura [7]

1. El extremo que va a enviar los datos cargará los mismos en el bus adecuado, y, una vez hecho esto, pondrá la señal VALID a uno, indicando así al otro extremo que los datos están listos.

2. Por su parte, el extremo que recibe los datos debe poner en valor uno la señal READY para indicar que está listo para realizar la transacción. El envío de datos a través de cualquiera de los canales se produce cuando ambas señales están en un valor lógico uno en un mismo ciclo de reloj. El orden en el que se elevan las señales a un valor lógico uno carece por completo de importancia.

3. La señal LAST se pone en uno únicamente cuando el último bloque de datos esté llegando hasta el receptor. La señal se pondrá en uno a la vez que se ponga en uno el último VALID, de modo que el receptor sabrá, al leer ese paquete de datos, que no van a entrar más.

Existen distintos tipos de interfaces AXI de acuerdo con la naturaleza de la aplicación que se le dará. En la Tabla **2-3** se muestra una comparación entre los tres tipos existentes, que son AXI4 el cual es es una versión completa utilizada principalmente en transferencias de datos hacia y desde la memoria del sistema, AXI-Lite que comúnmente es utilizada para transmisión de señales de control y de estado, y por último AXI-Stream el cual es utilizado para intercambios de datos entre componentes de hardware dentro de la parte del FPGA.

En la arquitectura ZYNQ existen diferentes tipos de interfaces para comunicar el CPU y el FPGA. Cada tipo puede ser utilizada para diferentes propósitos. Dichos tipos de interfaz son *General Purpose* (GP), *High Performance Ports* (HP) y *Accelerator Coherency Port* (ACP) descritos a continuación:

- **GP**. Es un bus de 32 bits con una velocidad de transmisión de hasta poco más de 370 MB/s usando DMA (del inglés *Direct Memory Access*) y alrededor de 126 MB/s sin usar DMA en el intercambio de datos entre CPU-FPGA [30]. Es una interfaz directa, es decir, sin almacenar datos en un buffer.

- **HP**. Son interfaces de alto rendimiento dedicadas a la transmisión de grandes cantidades de datos no mayores a la memoria del sistema, en este caso menos de 512 MB, haciendo uso de memorias de tipo FIFO en los extremos de la interfaz. El ancho del bus de datos puede ser de 32 o 64 bits. Tiene soporte para velocidades de transferencia de alrededor de 440 MB/s usando DMA y alrededor de 167 MB/s sin DMA [30].

- **ACP**. Una única conexión asíncrona de baja latencia entre el FPGA y la SCU (*Snoop Control Unit*[2] por sus siglas en inglés) dentro del CPU, con un ancho de

[2]Este elemento se encarga de asegurar la coherencia en las memorias caché de los núcleos del CPU.

Tabla 2-3.: Comparación de los tipos de interfaz AXI.

Tipos de interfaz	Características
AXI4	▪ Interfaz de datos/dirección mapeada en memoria tradicional. ▪ Soporte de ráfagas de datos.
AXI-Lite	▪ Interfaz de datos/dirección mapeada en memoria tradicional. ▪ Único ciclo de datos (es más adecuado para leer/escribir registros específicos).
AXI-Stream	▪ Solo ráfaga de datos.

bus de 64 bits. De acuerdo con Pavón [35], el puerto ACP tiene como principal característica el asegurar la coherencia de los datos en las transferencias, con el puerto ACP se soluciona el problema de coherencia entre la memoria RAM del sistema y la memoria caché cuando los datos de la RAM son distintos a los de la caché. El FPGA es el maestro.

DMA

Para el intercambio de grandes cantidades de datos entre CPU y FPGA no es muy útil que el CPU se encargue del papel de maestro debido a que se puede interrumpir la transmisión si es que se ocupa de otra tarea. Para liberar al CPU del intercambio de grandes cantidades de datos se hace uso de DMA. Al usar DMA, el CPU solo ordena la transmisión de datos en memoria a la unidad DMA [7].

DMA libera al CPU del proceso del tráfico de datos para poder ocuparse en otras tareas. Para iniciar una transmisión la unidad DMA necesita de la siguiente información:

- La dirección de la fuente de datos que se leerá.

- La dirección de destino a donde se quiere escribir.
- El número de bytes que se desean transferir.

La transferencia de datos puede ser desde la memoria del sistema (RAM de la tarjeta) a un periférico en el FPGA y de un periférico en el FPGA hacia la memoria del sistema.

Reconfiguración en tiempo de ejecución

La reconfiguración en tiempo de ejecución es una técnica para actualizar la configuración de un FPGA en tiempo de ejecución. Esta característica es bastante atractiva en la arquitectura ZYNQ debido a que se pueden reutilizar los recursos lógicos al momento de implementar un sistema digital diferente y, además, desde el sistema operativo ejecutado por parte de CPU es posible reconfigurar la lógica, sin intervención de una computadora externa. Para lograr lo anterior es necesario contar con el archivo de reconfiguración de FPGA, este tiene que ser accesible desde CPU. Este archivo puede ser generado con ayuda del software especializado Vivado[3].

La reconfiguración en tiempo de ejecución ofrece la posibilidad de tener hardware adaptativo a demanda del usuario, solo es necesario tener los archivos de reconfiguración ubicados en la microSD del sistema. Otra cosa importante que debe mencionarse es que el tiempo de reconfiguración es muy corto, siendo tan solo 200 ms unos de los peores casos en una reconfiguración completa del FPGA según un articulo publicado por Kadi y colaboradores [19].

2.1.4. Arquitecturas de procesamiento

Para el procesamiento de señales existen distintas arquitecturas que ayudan a disminuir el tiempo de ejecución de un algoritmo o vuelven más eficiente el uso de los recursos lógicos. A continuación, se describirán algunas de las arquitecturas más conocidas y utilizadas para el procesamiento de señales.

Arquitectura MAC

Una unidad MAC (multiplicador acumulador) es el componente más utilizado en procesadores digitales de señales (DSP) para realizar filtrados, convoluciones, acelerar filtros

[3] Software encargado de compilar, verificar y generar archivos de reconfiguración para FPGAs de Xilinx.

de Respuesta Finita al Impulso (FIR, por sus siglas en inglés) o Transformadas Rápidas de Fourier (FFT, por sus siglas en inglés). Una unidad MAC contiene multiplicadores, sumadores y registros de un determinado ancho de palabra. En esta arquitectura la salida anterior de la unidad MAC se suma con la salida del multiplicador y se acumula [33]. En la Figura **2-6** se muestra la arquitectura de una unidad MAC, donde A y B representan las entradas para la multiplicación, P representa la salida, Z el valor del registro y se contemplan el flujo de datos del sumador y el multiplicador.

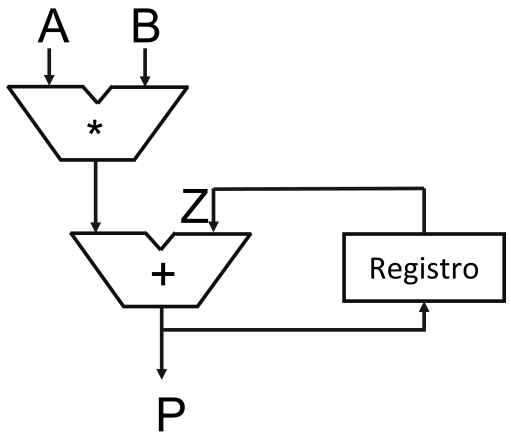


Figura 2-6.: Diagrama de la arquitectura MAC.

En un articulo publicado por Khan y colaboradores [20], describe la importancia que tiene la arquitectura MAC en los DSP, se dice que de la MAC depende el desempeño de los DSP ya que ejecutan su operación principal que es multiplicar-acumular. Es importante mencionar que los DSP surgieron para resolver la demanda del procesamiento de datos a alta velocidad y bajo consumo energético.

Arquitectura Paralela

En un artículo publicado por McBader [28] se describe que la mayoría de los algoritmos de preprocesamiento, como el filtrado, la extracción de bordes y alguna transformación, generalmente requieren una serie de operaciones repetitivas de computación intensiva que a menudo se caracterizan por un paralelismo de grano fino, es decir, tareas sencillas y distribuidas uniformemente en cada unidad de procesamiento. Como tal, a menudo se realizan de manera ineficiente en máquinas secuenciales y se implementan con frecuencia utilizando una matriz paralela de CPUs.

Las arquitecturas de procesamiento paralelas comúnmente son implementadas en GPU y recientemente mediante el uso de FPGA. La cantidad de procesos que pueden para-

lelizarse depende de los recursos del dispositivo, comúnmente el recurso limitante en un FPGA es la cantidad de DSP disponibles en el FPGA.

Arquitectura Pipeline

Una arquitectura pipeline de procesamiento secciona una tarea en diferentes etapas de procesamiento o subtareas con ayuda de registros intermedios. Una de las ventajas de este tipo de arquitecturas es que al mantener cada subtarea simple o con una lógica reducida, es posible aumentar la frecuencia del reloj del sistema digital de procesamiento. Estas arquitecturas son muy comunes en CPUs CISC que tienen instrucciones que requieren varios ciclos de reloj.

En la Figura **2-7** se muestra un ejemplo de la utilización de una arquitectura de procesamiento pipeline. En la figura se observan las diferentes etapas o subtareas en las que se puede dividir una tarea mas compleja, también se observa que entre cada etapa de procesamiento (E) se tienen registros (R) intermedios para un almacenamiento del resultado parcial de una etapa anterior.

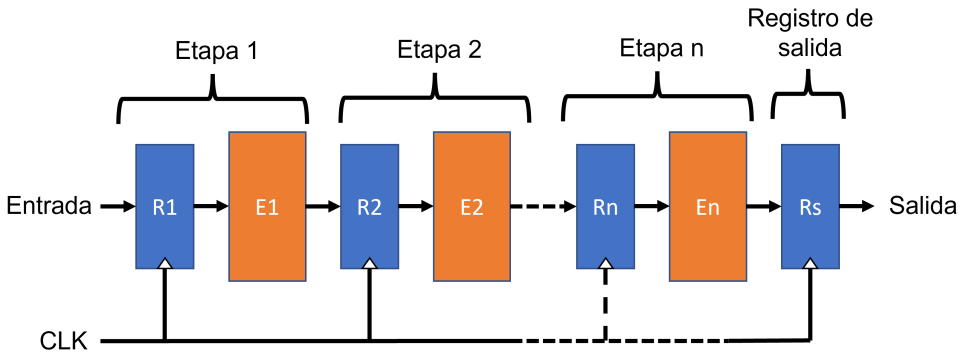


Figura 2-7.: Diagrama de una arquitectura de procesamiento.

2.1.5. Descripción de circuitos digitales

En esta sección se describe la forma en que se crea un sistema digital con la ayuda de lenguajes de descripción de hardware (HDL).

Lenguajes de descripción de hardware

Un lenguaje de descripción de hardware o HDL permite el diseño y simulación de circuitos electrónicos digitales con un nivel de abstracción mayor al de las técnicas basadas en software como el uso de Matlab, LabView o algún entorno de desarrollo basado en un lenguaje de programación como C++ o Python. Estos lenguajes HDL posibilitan la creación de sistemas digitales complejos en FPGAs que serían muy difíciles de conseguir con un conjunto de compuertas lógicas y componentes digitales. Es por lo anterior que los HDL son considerados un salto enorme en su área comparado con la aparición del lenguaje de programación C frente al lenguaje de programación Ensamblador.

En el mercado existe gran variedad de herramientas para la simulación y síntesis de circuitos digitales con el lenguaje VHDL. La gran competencia entre ellas ha ocasionado que las prestaciones que ofrecen sean muy similares

Una descripción VHDL de un circuito de baja complejidad requiere, al menos de estos 3 elementos:

- Bibliotecas (*library*). La potencialidad de un lenguaje depende en gran medida de sus bibliotecas. En la cabecera de cada código fuente o descripción se deben incluir aquellas bibliotecas que se necesiten.
- Entidad (*entity*). Describe la entradas y salidas del diseño, como si se tratase de un empaquetado de un chip.
- Arquitectura (*architecture*). Describe el contenido de ese diseño.

A continuación, en el Código 2.1se muestra un código para crear una compuerta lógica AND utilizando el lenguaje VHDL para dar una idea de la estructura al hacer diseños digitales con ayuda de este lenguaje.

```
1  --Cabecera dónde se colocan las bibliotecas
2  library IEEE;
3  use IEEE.std_logic_1164.all;
4
5  --Entidad que describe cuales son las entradas y salidas
6  entity ejemplo is
7      port (
8          entrada1, entrada2: in std_logic;
9          salida1: out std_logic
10         );
11 end ejemplo;
12
13 --Arquitectura que indica el diseño de la lógica de la descripción
```

```
14 architecture arch of ejemplo is
15 begin
16     salida<=entrada1 and entrada2; --Operación AND
17 end arch;
```

Código 2.1: Descripción de una compuerta AND en VHDL

El Código 2.1 hace uso de la biblioteca estándar básica del IEEE, en el apartado Entidad se declaran las entradas y salidas de la compuerta AND y en la Arquitectura se redirecciona a la salida el resultado de la operación AND (operación lógica) de los dos bits de entrada.

Núcleos de propiedad intelectual

Un núcleo de propiedad intelectual o IP core, por sus siglas en ingles, es una especificación de hardware que puede ser utilizado para configurar un FPGA. Cada diseño digital puede ser un IP core, por ejemplo, una serie de registros paralelos, un coprocesador de audio, un filtro digital entre otras cosas. Todo diseño digital puede ser encapsulado en un IP core para poder ser utilizado en un diseño digital [7].

Los IP cores por lo general están escritos en un lenguaje de descripción de hardware HDL y en ocasiones los IP cores pueden ser modificados por personas ajenas a la creación de estos si es que el código fuente no se protegió contra modificaciones.

Existen muchas formas de obtener IP cores, una de las formas es haciendo diseños propios a partir de un lenguaje de descripción de hardware, también existen repositorios en la Web que contienen un sin fin de IP cores con un sin número de aplicaciones, por último, las empresas como Xilinx ofrecen una serie de IP cores para disponer de ellos en el diseño de sistemas digitales más complejos.

Descripción de circuitos combinacionales

Los circuitos combinacionales son aquellos que no almacenan información. No utilizan *flip-flops*[4] ni registros y, por tanto, no requieren señal de sincronismo (reloj) ni señal de inicio (reset). La salida de estos circuitos está en función únicamente de las entradas actuales del circuito [1].

Es posible describir el funcionamiento de estos circuitos con tan solo una tabla de la verdad y se puede modelar con ecuaciones booleanas. Como ejemplo para este tipo de

[4] Es el elemento básico de almacenamiento en la electrónica.

circuitos encontramos los multiplexores, los convertidores BCD[5] a siete segmentos y un sumador.

Descripción de circuitos secuenciales

Los circuitos basados en lógica secuencial son circuitos que utilizan *flip-flops* para almacenar información. Estos circuitos además cuentan con una señal de sincronización (CLK o reloj) y con una señal de inicio y reset (RST), en el momento que la señal de sincronía cambia de estado (generalmente en el cambio de '0' a '1') se hace una captura de las señales de entrada del circuito secuencial y se evalúan las salidas en función de ellas.

Todos los circuitos síncronos descritos en VHDL suelen utilizar un proceso con la señal de reloj CLK en su lista sensible, más una estructura condicional tipo IF que se activa cuando ocurre una transición en la señal de reloj, para lo que se utiliza el atributo EVENT.

Diseño de máquinas de estado

El diseño de máquinas de estado se puede realizar de una forma muy simple en VHDL, y constituye un claro ejemplo de la capacidad de expresión de los lenguajes de descripción hardware frente a los métodos tradicionales de diseño, como por ejemplo, las máquinas ejecutadas desde software en un CPU que tiene que atender varias tareas se forma secuencial. Una máquina de estados en hardware es atendida en todo momento mientras que para una por software esto es más difícil debido a la cantidad de procesos que ejecuta un CPU aunque éste cuente con más de un núcleo de procesamiento.

Cuando ya se tiene la idea del algoritmo que se quiere implementar en la máquina de estados a diseñar, se procede a construir un diagrama de estados mejor conocido como grafo como el de la Figura **2-8**, este grafo se puede utilizar como guía en la construcción de la máquina de estados.

En la Figura **2-8** se pueden observar cuatro círculos, estos representan los estados o nodos de la máquina de estados, siento el nodo **E0** el estado inicial, es el estado que al aplicar reset (**RST**) la maquina se va a ese estado sin importar en que otro estado se encuentre. Cada estado puede hacer una configuración de señales de salida de acuerdo con las necesidades del diseño. Las flechas de la figura representan las posibles transiciones que existen en la maquina de estados. Las condiciones **C** representa la condición que se debe cumplir para que la transición sobre la que están señaladas se lleve a cabo.

[5] El código BCD es código binario de cuatro bits de ancho de palabra.

Es importante mencionar que la cantidad de condiciones para las transiciones puede ser más de una y las posibles transiciones de un estado a otro pueden ser mas de una.

De ser necesario se pueden crear tablas donde se exprese la configuración de la salida de la máquina de estados en función al estado actual y también se puede crear otra tabla donde se plasmen las condiciones que se tienen que cumplir en cualquier salto posible entre estados [21].

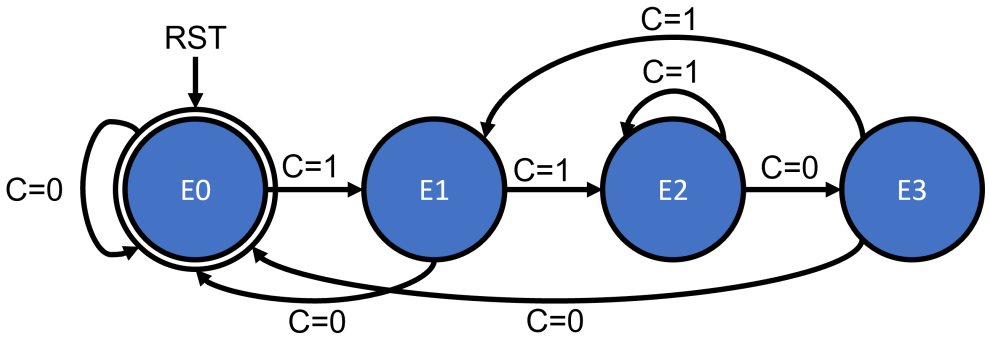


Figura 2-8.: Ejemplo de un diagrama de una máquina de estados finitos.

Diseño digital jerárquico

Es muy común que cuando se hacen diseños en VHDL se tienda a utilizar una metodología donde se divide el sistema a crear en varias partes que hacen una tarea muy específica, se hace uso de la descripción estructural. En general, se crean varios componentes que al final se unen para formar un sistema más complejo, también llamado empaquetado. Cuando esto sucede, los componentes del sistema se comunican entre ellos por medio de señales declaradas en el código de más alta jerarquía [18]. Un componente puede ser un sistema individual especificado por medio de su Entidad y Arquitectura, o bien puede ser insertado en la arquitectura con la palabra reservada ***component***. Los componentes deben ser instanciados en la descripción estructural. La instanciación de componentes es una proposición básica en una arquitectura estructural.

Con lo anterior, ya se sabe que existen los componentes del diseño y el código que empaqueta a dichos componentes. Cada componente puede ser agregado al empaquetado como un *Component* siendo instanciado con esa palabra reservada y agregando el contenido del *entity* del componente a agregar. En el Código 2.2 se presenta un ejemplo de la instanciación de un componente en VHDL de un generador de señales. El código solo nos sirve para mostrar la estructura de una instanciación de una descripción de hardware

como un componente.

```
component sclk_gen  --Nombre del componente igual al nombre de la entidad
--de su descripción
  port (
      clk, rst:in std_logic; --Puertos de entrada
      bt, clr: in std_logic; --Puertos de entrada
      y:out std_logic        --Puerto de salida
  );
end component;  --Se cierra la entidad del componente
```

Código 2.2: Ejemplo de instancia de un componente en VHDL.

Un componente instanciado podrá ser usado en el empaquetado que lo instanció hasta que todas las señales de entrada y salida de este sean conectadas a señales o puertos de entrada/salida del empaquetado.

A la tarea de conectar las señales de los componentes con señales o puertos del empaquetado se le conoce como mapeo de los puertos y existen dos formas de hacerlo, por medio de posiciones y utilizando los nombres de los puertos.

Mapeo de puertos por posiciones

En la asociación posicional, las señales en el mapa de puertos deben listarse en el mismo orden en el cual se declararon los puertos en la entidad del componente. Para que la asociación sea posible, las señales deben ser del mismo tipo.

Cuando se utiliza este tipo de mapeo las señales que se conectaran al componente solo se tienen que listar en el orden que se estableció en la entidad del componente. Un ejemplo de este método se ve en el Código 2.3. En dicho código se esta usando de ejemplo un mapeo de puerto de un componente llamado CompEjemplo el cual tiene 5 puertos. Los nombres de los puertos colocados corresponden con nombres de señales o puertos del empaquetado que esta usando el componente CompEjemplo.

```
c1: CompEjemplo port map(clk,rst,opc_r,byte_rx,dato);
```

Código 2.3: Ejemplo de mapeo de puertos por posición.

c1: Nombre de referencia del componente en el empaquetado.

CompEjemplo: Es el nombre del componente instanciado (tiene que ser el mismo que el de su respectiva entidad o entity).

port map(): “Función” que permite mapear las señales.

El contenido de los paréntesis del port map() es el listado de señales que se conectaran al componente CompEjemplo.

Mapeo de puertos por nombre

A diferencia del anterior método de mapeo, el mapeo de puertos por nombre no necesita de un orden de las señales a listar, aquí se debe escribir el nombre de la señal en el componente y su respectiva asignación del empaquetado separado con los caracteres $=>$. Este método ayuda a reducir la probabilidad de cometer errores en el momento en que se hace el mapeo. En el Código 2.4 se muestra un mapeo de puertos con este método de un componente de ejemplo llamado sclk_gen. En el código se observa como se hace el mapeo por nombre.

```
1 c1: sclk_gen
2   port map(
3       clk=>clk,
4       rst=>srst,
5       bt=>tb,
6       clr=>clr_tb,
7       y=>ssclk
8   );
```

Código 2.4: Ejemplo de mapeo de puertos por nombre.

Como se observa en el Código 2.4, cada puerto es mapeado de manera explicita haciendo uso de los nombres que se le dan a los puertos en el componente y en el empaquetado que instancia dicho componente.

2.2. Software especializado

En el desarrollo de esta investigación es necesario hacer uso de herramientas especializadas en la creación de sistemas digitales para una plataforma específica (Xilinx). Además, se hizo uso de unas herramientas para la creación de un sistema operativo basado en Linux que se ejecutaría en la tarjeta ZYBO.

2.2.1. Vivado

Vivado es un entorno de diseño integrado (IDE) que permite desarrollar proyectos de diseño de sistemas digitales en, principalmente, FPGA y SoC de la empresa Xilinx. Este IDE proporciona al usuario una interfaz grafica (GUI) con una serie de herramientas para el desarrollo de proyectos [43].

En la Figura **2-9** se muestra la ventana principal de Vivado cuando se tiene un proyecto abierto y debajo una descripción de cada parte.

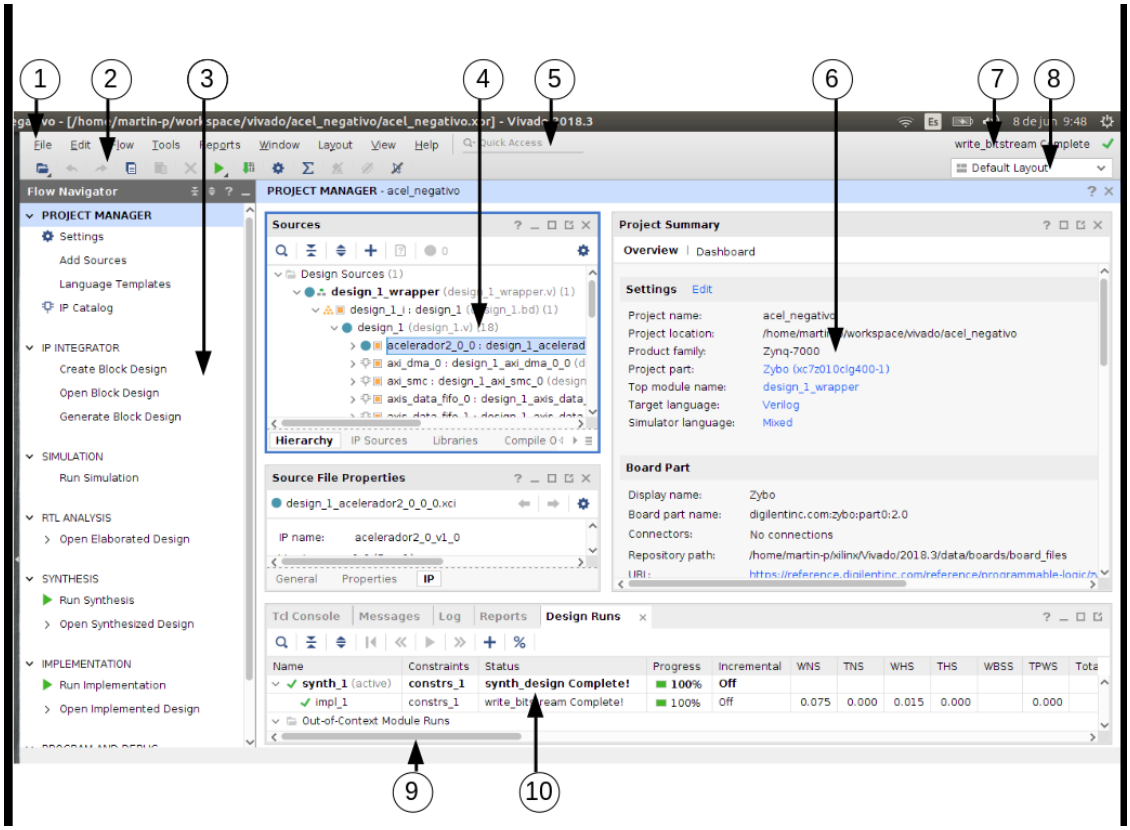


Figura 2-9.: Entorno de visualización IDE de Vivado [43].

1. Barra de menú
2. Barra de herramientas principal
3. Navegador de flujo
4. Área de ventanas de datos
5. Campo de búsqueda de acceso rápido al comando de menú
6. Espacio de trabajo
7. Barra de estado del proyecto
8. Selector de diseño
9. Barra de estado
10. Área de ventanas de resultados

2.2.2. Petalinux Tools

Son una serie de herramientas que ofrecen la posibilidad de crear, personalizar e implementar sistemas operativos basados en Linux destinados a ejecutarse en sistemas de procesamiento de Xilinx.

Petalinux incluye herramientas para personalizar el gestor de arranque, el kernel de Linux, el sistema de archivos, las bibliotecas y los parámetros del sistema. A su vez, en plataformas SoC FPGA, las herramientas de Petalinux permiten sincronizar el software ejecutado en el CPU con el diseño personalizado de hardware en el FPGA [45].

De acuerdo con la guía de referencia de Petalinux [44], el flujo de trabajo que debe seguirse es el que se presenta en la tabla **2-4**.

Tabla 2-4.: Flujo de diseño con Petalinux [44].

Tarea del flujo de diseño	Herramienta o flujo de diseño
Creación de plataforma de hardware (solo para hardware personalizado)	Vivado
Crear Proyecto PetaLinux	petalinux-create -t project
Inicialice el proyecto PetaLinux (solo para hardware personalizado)	petalinux-config –get-hw-description
Configurar opciones de nivel de sistema	petalinux-config
Crear componentes de usuario	petalinux-create -t COMPONENT
Configurar el kernel de Linux	petalinux-config -c kernel
Configurar el sistema de archivos raíz	petalinux-config -c rootfs
Construya el sistema	petalinux-build
Empaquetar para implementar el sistema	petalinux-package
Arranque el sistema para la prueba	petalinux-boot

2.3. Imagen digital

Una imagen digital puede definirse como una función bidimensional de intensidad de luz $f(x, y)$, donde x e y son las coordenadas espaciales del punto a representar. El rango de intensidad comúnmente es de 0 a 255 correspondiente a una longitud finita de palabra de 8 bits sin signo, aunque este rango es superior cuando aumentan la cantidad de bits que representan dicha intensidad.

Las imágenes digitales pueden clasificarse de manera general en dos tipos: imágenes de mapas de bits y las imágenes vectoriales. En esta investigación solo se hace uso de mapas de bits.

2.3.1. Imagen de mapa de bits o bitmap

Un mapa de bits es una estructura de datos que representa una rejilla o matriz rectangular de pixeles o puntos de color [41]. Este tipo de imágenes son obtenidas por medio de escáner o cámaras digitales. La representación de esta es a través de un conjunto de pixeles.

Un píxel es la mínima unidad de representación de las imágenes digitales. En cuanto mayor sea la cantidad de pixeles en una imagen digital, mejor sera la calidad de esta

y pueden representarse con mayor detalles las formas contenidas en dicha imagen, al mismo tiempo entre mayor sea la cantidad de pixeles mayor será el espacio ocupado por la imagen en disco.

2.3.2. Imágenes RGB

El modelo RGB tiene separados tres canales de intensidad, estos tres canales corresponden a la intensidad de los tres colores primarios rojo, verde y azul que componen cada pixel en la imagen [41]. Los pixeles pasan a ser una tupla de tres subpixeles y por ende la imagen pasa a ser una función del tipo $f(x, y, c)$, donde c es la coordenada del canal.

Para explicar mejor lo que es una imagen RGB, se tiene en la Figura **2-10** una serie de imágenes. La imagen de la parte superior izquierda, con la etiqueta RGB, representa una imagen RGB, es decir, cuenta con una representación donde intervienen los tres canales anteriormente mencionados (rojo, verde y azul). En las otras tres imágenes solo se ve la intensidad de color de cada uno de los canales de la imagen RGB. En la imagen con la etiqueta R se muestra la representación del canal rojo de la imagen RGB. De la misma manera, lo anterior ocurre para las imágenes etiquetadas con la letra G y B que muestran solo la representación del canal verde y azul respectivamente.

2.3.3. Formato de imágenes digitales PAM

El formato de imágenes digitales se refieren a las características asignadas a los archivos gráficos. Estas características pueden dar información acerca del tipo de imagen, el tamaño, la resolución y el tipo de compresión si es que se utiliza y bajo que método.

Existe una gran cantidad de formatos para las imágenes digitales, en este caso el formato de imagen que se utiliza en esta investigación es denominado **Portable Arbitrary Map** o simplemente **PAM**. Este es un formato de imagen digital que cuenta con dos apartados bien definidos:

- Encabezado
- Contenido de la imagen

En el encabezado se especifican el ancho y largo de la imagen, la cantidad de canales de la misma, el máximo valor de intensidad representable (comúnmente 255), el tipo de tupla y un final de encabezado. El archivo de la imagen PAM puede ser visualizado con

Figura 2-10.: Separación de canales de una imagen RGB.

un editor de textos y con ello se puede observar fácilmente el contenido del encabezado.

El formato PAM incluye un tipo de tupla que contempla la transparencia en imágenes RGB, este tipo de tupla es denominada RGB_ALPHA, al incluir un canal de transparencia la imagen resultante tiene en total 4 canales y esto es de especial interés para esta investigación debido a que estas tuplas pueden ser representadas con 32 bits, es decir, 4 bytes. Este tamaño de tupla toma importancia al momento de la transmisión de datos en el sistema utilizado.

En el Código 2.5 se muestra un ejemplo de un encabezado de una imagen digital de formato PAM, se observa cómo se especifica el ancho y alto de la imagen (en pixeles), se muestra del tamaño de las tupas (4 bytes) y el tipo de estas, es decir, RGB_ALPHA. También se muestra que al inicio está la palabra **P7**, esta indica que es una imagen

PAM y esto se hace debido a que el formato PAM pertenece a una familia de formatos y cada uno tiene su palabra para especificar su formato. Por último, se ve como se coloca ENDHDR, esto es para especificar el final del encabezado.

```
P7
WIDTH 227
HEIGHT 149
DEPTH 4
MAXVAL 255
TUPLTYPE RGB_ALPHA
ENDHDR
```

Código 2.5: Ejemplo de encabezado de una imagen de formato PAM.

En el apartado de contenido se encuentran los datos del contenido de cada pixel (tupla) en formato binario. La cantidad de tuplas necesarias para la representación de la imagen debe ser al menos la cantidad representada por la multiplicación del alto por el ancho de la imagen especificados en el encabezado.

Por último, es importante que se mencione que este formato no cuenta con ningún tipo de compresión y para la lectura y escritura de estas imágenes no es necesario un proceso de codificación y decodificación. Esta característica y el hecho de tener tuplas de 4 bytes fueron determinantes en la elección del formato PAM en esta investigación.

2.4. Funciones matemáticas para el procesamiento de imágenes

Algunas de las funciones que demandan un alto rendimiento computacional en la actualidad son, por mencionar unos casos, la visión artificial, el procesamiento de imágenes y el aprendizaje profundo. Estas tareas en la actualidad estas siendo delegadas a aceleradores por hardware por razones de eficiencia energética y de rendimiento [14]. Estos aceleradores muchas veces tienden a ser implementados con la ayuda de FPGA's.

2.4.1. Función negación

Esta función básicamente consiste en invertir los valores de intensidad de una señal de entrada, en este caso, una imagen de entrada [27]. La inversión de valores de intensidad se hace comúnmente a imágenes en escalas de grises, es decir, imágenes de un solo canal, pero también es aplicado a imágenes RGB. Este operador es particularmente útil para mejorar los detalles grises en regiones negras (si se habla de imágenes en escala

de grises).

La inversión o negación se hace tomando en cuenta el valor máximo de intensidad representable, comúnmente en imágenes el valor máximo de intensidad es **255**. En la Ecuación 2.1 se muestra la función de la negación para un canal de una imagen de dimensión $N * M$.

$$ImagenNegativa(i,j) = 255 - ImagenEntrada(i,j) \tag{2.1}$$

Donde $0 \leq i \leq M-1$ y $0 \leq j \leq N-1$.

Un ejemplo de una negación puede observarse en la Figura **2-11**. A la izquierda se muestra la imagen de entrada y a la derecha la negación de la entrada. En imágenes biomédicas esta técnica es particularmente muy utilizada.

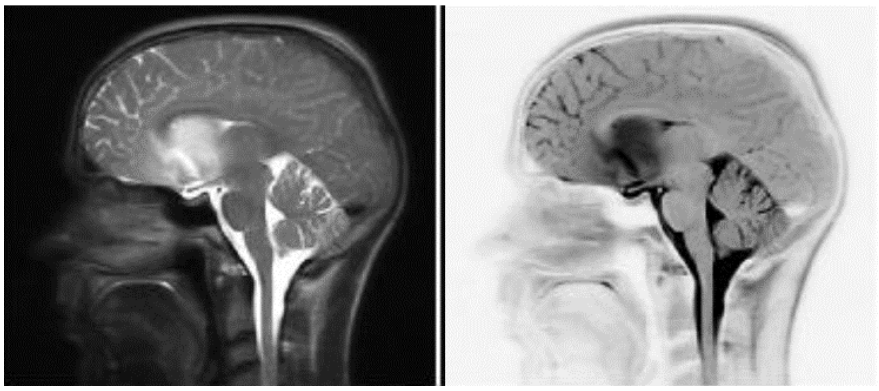


Figura 2-11.: Ejemplo de aplicación de la función de negación en imágenes a escala de grises [27].

La operación de la negación también puede aplicarse a imágenes RGB. En la Figura **2-12** se muestra la aplicación de la función de la negación en una imagen RGB.

2.4.2. Función de convolución 2D

De acuerdo con palabras de Hecht, V. y colaboradores [16], la convolución 2D es uno de los algoritmos de procesamiento de imágenes más utilizado, principalmente en aplicaciones en tiempo real como lo son el análisis de imágenes médicas, la detección de bordes y el filtrado digital.

Figura 2-12.: Ejemplo de aplicación de la función de negación en imágenes RGB.

La convolución 2D esta definida por la Ecuación 2.2.

$$y(n,m) = \sum_{l=0}^{L-1} \sum_{k=0}^{K-1} h(k,l) * x(n-k+K_0, m-l+L_0) \tag{2.2}$$

Donde $y(n,m)$ es el valor de intensidad del pixel ubicado en (n,m) para la imagen de salida,h representa una matriz de coeficientes o kernel de convolución, $h(k,l)$ es el valor del coeficiente del kernel de convolución en la posición (k,l), $x(n,m)$ es la intensidad del pixel de la imagen de entrada en la posición (n,m), K y L señalan el tamaño del kernel de convolución y K_0 y L_0 son las coordenadas del centro del kernel.

2.5. Trabajos relacionados

2.5.1. Arquitecturas de aceleración de algoritmos

En un estudio comparativo hecho por HajiRassouliha y colaboradores [15], donde se pusieron en contraste tres tecnologías para la aceleración de algoritmos relacionados con la visión artificial y el procesamiento de imágenes, compararon DSPs, FPGAs y GPUs. En este estudio se vio que en algoritmos simples los DSPs cumplen con lo necesario para lograr una implementación, con bajo consumo energético y resultan ideales para implementaciones portables. En el artículo se mostraron algunos resultados de dos trabajos de investigación donde se implementaron algoritmos de visión estereoscópica, en uno de ellos se podían procesar imágenes estereoscópicas de 640x480 pixeles a ocho cuadros por segundo con un DSP de modelo OMAP3530 y en otro trabajo implementado con un DSP DM642 se procesaron imágenes de 356x292 pixeles a 8.92 cuadros por segundo en la ejecución de algoritmos de visión estereoscópica.

HajiRassouliha señala que los GPUs resultaron bastante competentes para la aceleración de algoritmos complejos y simples, son bastante buenos en algoritmos altamente paralelizables y el tiempo de desarrollo e implementación es corto. Algunos de los resultados más mencionados de este articulo fueron de visión estereoscópica, en uno de los resultados mostrados se menciona el procesamiento de imágenes de 384x288 pixeles a cinco cuadros por segundo en una implementación en una GPU GTX 480 del 2010, otro muestra que se pudieron procesar imágenes de 450x375 pixeles cada 100 ms haciendo uso de una GPU GTX 570 del 2010 en la implementación de un algoritmo denominado Suma de Diferencias Absolutas (SAD por sus siglas en inglés) y, por último, en una implementación de visión estereoscópica para la detección de objetos en tiempo real en una GPU GTX 680 del 2012 alcanzan a procesar 114 imágenes por segundo de 1024x768 pixeles.

En el caso de los FPGAs HajiRassouliha y colaboradores resaltan las posibilidades de acelerar algoritmos complejos y simples con una gran flexibilidad a la hora de modificar dichos algoritmos, tienen un bajo consumo energético y por lo tanto son aptos para aplicaciones portables. Algunos trabajos mencionados en el artículo son la rectificación de imágenes con visión estereoscópica donde se muestra que con un FPGA Virtex-6 de Xilinx se pueden procesar hasta 367 imágenes por segundo de tamaño 640x480 pixeles, también, en la implementación de un algoritmo de detección de rostro usando un FPGA Virtex-5 de Xilinx se pueden procesar 307 imágenes cada segundo de tamaño 640x480 pixeles. En comparaciones hechas para medir los tiempos de cómputo entre los GPUs y los FPGAs algunas veces los GPUs resultan más veloces y a veces no, todo depende del algoritmo acelerado y de los parámetros de este. Por ejemplo, en la implementación

de filtrado 2D, visión estereoscópica, y la implementación del algoritmo de clusterización K-means una GPU GTX 280 resulta mejor que un FPGA Virtex-4 de Xilinx solo con tamaños de filtros menores a 11. También, al comparar un GPU NVidia GTX 295 contra en FPGA Altera Stratix-3 en la implementación de convoluciones 2D se tiene que en la GPU se pueden procesar 120 imágenes por segundo de 1280x720 pixeles con un kernel de convolución de tamaño 25 (matriz de convolución de 25x25) mientras que el FPGA solo puede procesar 80 imágenes por segundo con imágenes de 1280x720 pixeles y con un kernel de tamaño 25.

En el estudio de HajiRassouliha y colaboradores se muestran los alcances de las tres diferentes tecnologías estudiadas en el artículo que son DSPs, GPUs y FPGAs y al final se puede concluir que se recomienda el uso de FPGAs en el caso de la aceleración de algoritmos que requieran velocidades de procesamiento altas, de algoritmos complicados en aplicaciones de bajo consumo y en algoritmos personalizados.

Por otro lado, Feng y colaboradores [11], hacen un estudio comparativo de tres tareas muy importantes en la visión por computadora, estas tareas son la clasificación de imágenes, la detección de objetos y la segmentación de imágenes, cada tarea usando técnicas de aprendizaje profundo como VGG-19, VGG-16 y ResNet-152 por mencionar algunas. Los algoritmos son acelerados tanto en GPUs como en FPGAs y se comparan en términos de Giga Operaciones Por Segundo (GOP/s). Los autores reportan una diferencia grande de consumo energético entre ambas tecnologías. Se obtiene que los FPGAs realizan muchas más operaciones que los GPUs con la misma energía, por ejemplo, en la implementación de VGG-19 una GPU GTX TITAN X se logran obtener 6.82 GOP/J (Giga operaciones entre cada Joule) mientras que usando FPGAs Stratix-V GSD8 se obtienen 38.13 GOP/J. En el mismo artículo, Feng y colaboradores muestran que el rendimiento de los algoritmos implementados tienen mejor desempeño en un GPU que en un FPGA, reportan que el desempeño de un GPU GTX TITAN X en una implementación de VGG-19 es de 1704 GOP/s mientras que dicha implementación en un FPGA XC7 VX690T de Xilinx se obtienen 1220 GOP/s.

En un artículo publicado en 2019, Melo y colaboradores [30] se pone en evidencia los alcances del intercambio de información entre los dos componentes principales de una arquitectura híbrida FPGA-CPU, en específico, una arquitectura ZYNQ 7000 de una tarjeta ZYBO. En este estudio se logra observar que la velocidad del intercambio de información entre dispositivos se acerca a los 500 MB/s en el mejor caso (usado DMA desde el CPU) y con cerca de 440 MB/s (usando AXI DMA). Cabe destacar que los resultados que obtuvieron fueron resultados de experimentos sencillos que buscaban medir el alcance de las interfaces de comunicación y por ello no se usó un sistema operativo al

hacer los experimentos de transferencias de datos[6]. Este trabajo muestra una idea de la velocidad de transferencia de datos que se puede tener y el tipo de aplicaciones que pueden ejecutarse en estas plataformas.

El anterior artículo perfila la tecnología de FPGA híbridos para la realización de aceleración de algoritmos teniendo las ventajas de los FPGAs, mencionadas anteriormente, y contando con los beneficios de tener un CPU incrustado en el mismo chip comunicados a una alta velocidad.

2.5.2. Algoritmos de visión artificial y procesamiento de imágenes

Aunque las arquitecturas híbridas FPGA-CPU no tengan una gran popularidad, más que nada por el trabajo que conlleva realizar proyectos con estas, existe una serie de aplicaciones en procesamiento de imágenes y visión artificial en las que estas se han estado utilizando en los últimos años.

Algunos trabajos que ejemplifican el tipo de aplicaciones para los que están hechas estas plataformas es el de Maheshwari [26], en donde se dice que debido al alto costo computacional que requiere la detección de bordes de imágenes y video es necesario optar por este tipo de arquitecturas, también plantea que debido a la evolución de los entornos de diseño digital es posible crear estos algoritmos de detección de bordes en un lenguaje convencional (en este caso C++) y generar un coprocesador por hardware con la ayuda de algunas herramientas de alto nivel. En su articulo, Maheshwari muestra como logra crear un detector de bordes basado en el algoritmo *Canny edge detection*, logrando procesar 60 imágenes por segundo de un tamaño de 1920x1080 pixeles con las técnicas mencionadas de desarrollo mediante C++.

Goel y colaboradores en 2019 [13] diseñaron un algoritmo para realizar un filtrado de la mediana en imágenes con ruido impulsivo tomando en cuenta las capacidades de paralelismo que ofrecen las plataformas híbridas como lo son la arquitectura ZYNQ. En su trabajo se hacía uso de la comparación de cierta cantidad de B bits más significativos de cada pixel para que, a través de un método de histogramas, se aproximara la obtención de la mediana de una determinada ventana de la imagen. En su algoritmo de manera paralela se determinaba si existía dicho ruido en la ventana analizada y se decidía si el pixel pivote se sustituía con la aproximación de la mediana o se quedaba el pixel original. En este trabajo se obtuvo un rendimiento que permitió filtrar 282 imágenes de tamaño

[6] Un CPU no necesariamente necesita un sistema operativo para funcionar y como ejemplo se tienen casi todos los microcontroladores de uso comercial, estos tienen un CPU dentro de ellos y no usan sistema operativo

Full HD (1920x1080 pixeles) cada segundo. Se abordó la posibilidad de paralelizar comparadores y se explotó lo más posible la lógica combinacional para conseguir filtrar las imágenes.

Gao y colaboradores demostraron en 2018 [12] que en el procesamiento de imágenes en tiempo real la plataforma ZYNQ tiene ventajas por encima de plataformas de cómputo tradicionales como el hecho de contar con un FPGA incrustado en el mismo chip que el CPU, haciendo posible una capacidad mayor de paralelizar tareas en el FPGA y teniendo un ancho de banda de cerca de 500 MB/s por canal en la comunicación FPGA-CPU. En este trabajo se intentaba localizar marcas en imágenes con propósitos de poner a prueba la plataforma. Se usaron métodos de filtrado digital de imágenes con ayuda de un filtro gaussiano, se realizaron operaciones de segmentación, acumulación de coordenadas de interés y el cálculo del promedio de dichas coordenadas para obtener el punto medio de una marca colocada en la imagen. Se demostró que al usar de manera simultánea tanto el CPU como el FPGA se obtienen velocidades de procesamiento 15.6 veces más altas que al usar solo CPU. El enfoque que se usó fue el diseñar los núcleos de procesamiento con ayuda de Vivado HLS en un lenguaje de alto nivel como lo es C++.

En un trabajo relacionado con la fusión de imágenes publicado por Qu y colaboradores [36], se hace uso de métodos de convolución con un kernel laplaciano de 5x5 en una etapa de descomposición de imágenes, se hacen otras operaciones como el cálculo del promedio de las capas resultantes de la descomposición de imágenes (dos imágenes a fusionar) y nuevamente se aplica otra convolución para una reconstrucción de una imagen de salida ya fusionada. En este trabajo los procesos de convolución y operaciones sobre pixeles se hace en la parte del FPGA de la arquitectura ZYNQ teniendo una fusión satisfactoria con un tiempo total de procesamiento de 28.85 ms por imagen fusionada obtenida de tamaño 256x256 pixeles. Los autores defienden que usar estas arquitecturas brindan un grado de flexibilidad y reconfiguración envidiable por otras arquitecturas, además ponen en evidencia que gracias a la velocidad de intercambio de datos entre ambos componentes de la arquitectura y a la flexibilidad de los FPGA se obtiene un poder de cómputo necesario para hacer procesamiento en tiempo real de imágenes. El enfoque que se utilizó en este trabajo es la modularidad y se utilizó una arquitectura pipeline-paralela en el diseño digital del algoritmo.

Por su parte, Kowalczyk y colaboradores en 2017 en un artículo publicado [22] muestran cómo es posible hacer la detección de movimiento de objetos con una cámara móvil. La detección de los objetos la hicieron tomando en cuenta los valores del color de los objetos y con un algoritmo a base de comparadores lograban estimar la posición del objeto deseado. Una vez se tenía la posición del objeto deseado se hacía un seguimiento con el control de la dirección donde apuntaba la cámara al objeto deseado para que nunca

saliera de cuadro. Este algoritmo fue implementado con una cámara de resolución 1280 x 720 a 60 cuadros por segundo usando la plataforma ZYNQ y se hizo uso de la parte del FPGA para la detección del objeto mientras que el control de posición de la cámara se implementó en el CPU con un controlador Proporcional-Integral-Derivativo (PID por sus siglas en inglés).

En 2020, Li y colaboradores [25] utilizaron una plataforma ZYNQ para reconocer en tiempo real líneas de cultivo a partir del procesamiento de imágenes RGB. Algunos de los métodos utilizados fueron la aplicación del índice de vegetación ExG (*Excess Green index*) para resaltar las áreas con plantas, a partir de ahí se pasaba a un método de segmentación para por último utilizar un filtro de la mediana con un kernel de 5x5 para la eliminación de ruido en las imágenes resultantes. Después por medio de otros métodos de cálculo de densidad estimaban la orientación de las líneas de cultivo. En esta investigación se contrastó que la obtención de los resultados deseados se hace hasta 49 veces más rápido con la arquitectura ZYNQ comparado con una computadora convencional haciendo uso de un CPU Intel i3-2130.

2.5.3. Diversidad de algoritmos acelerados en FPGAs híbridos

Existe una gran variedad de algoritmos que se han acelerado mediante FPGAs híbridos, estos algoritmos no necesariamente tienen que ser orientados al procesamiento de imágenes. Los autores de los siguientes trabajos dejan en claro las posibilidades de los algoritmos que pueden implementar en plataformas ZYNQ.

Kyrtsakas y Muscedere [24] implementaron en esta arquitectura un codificador y decodificador de imágenes en formato JPG en la parte del FPGA de la arquitectura ZYNQ como un coprocesador designando esta tarea al hardware dedicado que se diseñó, liberando al CPU de esta tarea. A su vez, se hicieron comparaciones de tiempos de decodificación de la arquitectura diseñada en hardware y un método por software en el CPU de la ZYNQ utilizando una biblioteca denominada libjpeg-turbo y se obtuvo un tiempo de decodificación de hasta 4.25 veces mejor con el módulo hecho con hardware que el método por software. Esta es una muestra de la posibilidad de carga de trabajo que se le puede delegar a un diseño en hardware dedicado en el FPGA como si fuera un simple periférico más.

Panait en 2019 [34] demostró como es posible la aceleración de funciones hash haciendo uso de esta arquitectura, a su vez demostró el rendimiento que se puede alcanzar al utilizar DMA desde el FPGA para acceder a regiones de memoria y hacer un tráfico de datos en masa más eficiente que pasarlos a través del CPU, la implementación realizada

se enfocó en la minería de la criptomoneda Bitcoin alcanzando hasta 50 M hashes por segundo. Lo anterior es posible gracias a que en estas arquitecturas se tienen diferentes canales AXI, uno de estos tipos de canales esta optimizado para el acceso a memoria desde el FPGA.

Por su parte, Aimar y colaboradores [3] en 2019 publicaron un trabajo donde muestran como crean un acelerador de redes neuronales convolucionales (CNN) basado en hardware al que llamaron NullHop, el acelerador se diseña tratando de optimizar la máxima cantidad de recursos reduciendo la precisión de los pesos de las neuronas, también hacen uso de la función de activación Unidad Lineal Rectificada (ReLU por sus siglas en inglés). El acelerador está pensado para implementar distintas redes neuronales tales como VGG-16, VGG-19, Giga1Net, RoshamboNet y Face Detector CNN. En una implementación de este acelerador en una plataforma ZYNQ se obtuvieron hasta 17.196 GOP/s en el mejor caso en la red neuronal VGG-19 y solo 0.61 GOP/s en la red neuronal Face Detector. Con el acelerador en la plataforma ZYNQ es posible obtener hasta 28.8 GOP/s/W de eficiencia energética, en contraste Aimar menciona que un GPU comúnmente alcanza 10 GOP/s/W.

3. Metodología

En este capítulo se presenta la metodología utilizada para la elaboración de este trabajo de investigación, se describirá cada paso realizado y el por qué se realizó. También, se menciona sobre el uso de los materiales necesarios, así como también el software necesario en esta investigación. Por la naturaleza de la investigación, se desglosa la metodología en varias subsecciones, cada una dedicada a un elemento que compone esta investigación.

De manera general y como se muestra en el diagrama de la Figura **3-1**, la metodología seguida en esta investigación consiste en diferentes etapas, en la figura se muestra que se tiene iniciar diseñando el sistema digital que se quiere implementar en la parte del FPGA y para ello se hace uso del lenguaje VHDL, una vez que se tienen los archivos VHDL que describen el sistema digital deseado, se hace uso de la herramienta Vivado para la verificación del diseño y para la generación de los archivos de reconfiguración específicos para la plataforma que se desea usar (en nuestro caso la tarjeta ZYBO). Los archivos de reconfiguración son posteriormente utilizados por la herramientas de Petalinux para la creación de un sistema operativo a la medida de la plataforma usada, tomando en cuenta el sistema digital que se quiere implementar en la parte del FPGA. Por último, es necesario desarrollar aplicaciones para lograr la comunicación y configuración entre el CPU y el FPGA.

Aunque el diagrama de la Figura **3-1** muestra un poco la metodología general, esta cambia un poco después de la primer puesta en marcha de un diseño funcional de un hardware base con un sistema operativo funcional. Esto se debe a que es posible utilizar las reconfiguraciones del FPGA con ayuda del sistema operativo desde el CPU, este procedimiento se mencionó en secciones anteriores y cuando sea requerido en secciones posteriores se explicará más a detalle.

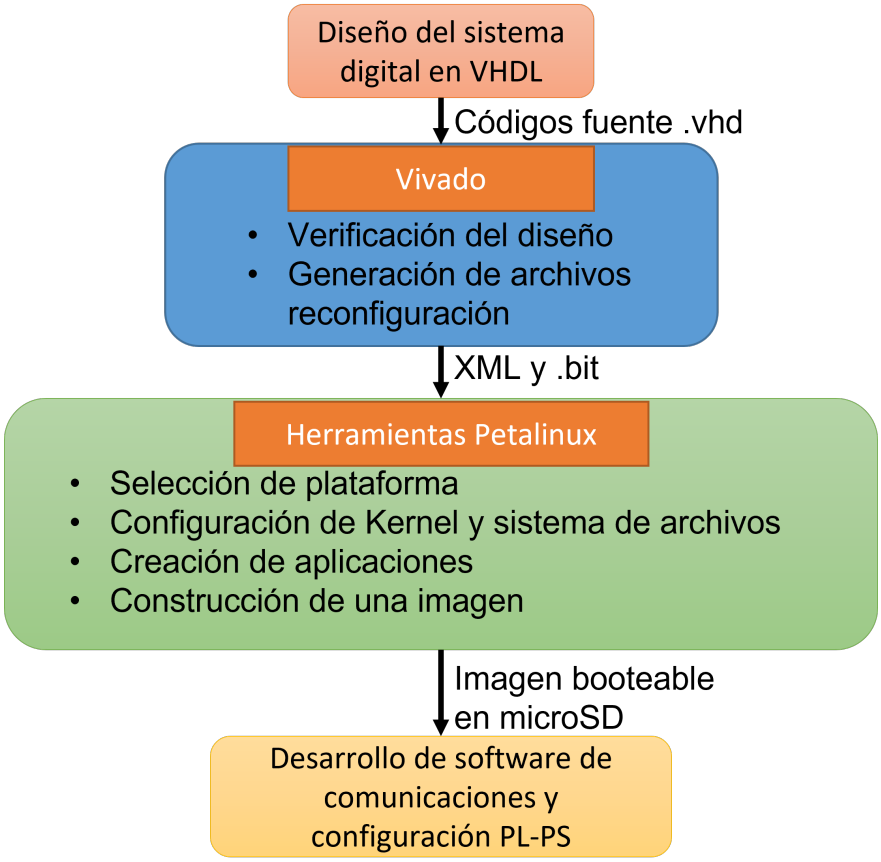


Figura 3-1.: Metodología general.

3.1. Diseño de hardware base

En este trabajo se desarrolló una plataforma lo más general posible en la aceleración de algoritmos de procesamiento de imágenes, es por ello por lo que de forma general siempre se estará trabajando sobre la misma base en el diseño del hardware de aceleración. En la Figura **3-2** se muestra la composición del diseño general base sobre el que se trabaja, en la figura se pueden identificar tres componentes principales los cuales son la CPU, el FPGA y la memoria RAM del sistema, se puede ver que siempre se hace uso de la memoria RAM para compartir información entre CPU y FPGA, esta configuración es la que resultó mas adecuada para el tipo de proyecto que se realiza. Dentro del FPGA hay ciertos bloques en color azul encargados principalmente en la administración y control de las salidas y entradas de datos haciendo uso de los buses AXI GP y AXI HP. Los buses conectados al bloque DMA son HP y serán utilizados para la transmisión masiva de datos (en nuestro caso de imágenes), los buses conectados a la región verde de la memoria RAM son utilizados para configuración y control de DMA y para intercambiar datos entre

el CPU y el FPGA en forma de registros de ubicación fija con ayuda de los registros AXI-lite. Por último, el bloque Acelerador es el encargado de procesar información recibida y es el bloque que se puede modificar para que lleve a cabo la tarea que se quiera realizar.

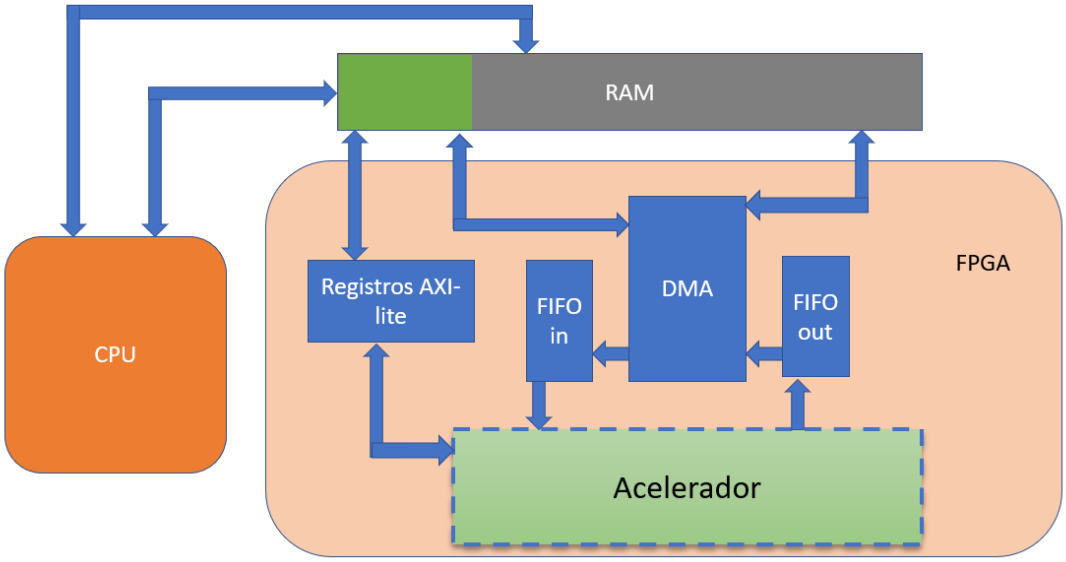


Figura 3-2.: Hardware base de esta investigación.

El trabajar sobre un diseño base es bastante útil debido a que solo tiene que crearse un sistema operativo con ayuda de las herramientas de Petalinux. El crear un sistema operativo a la medida hace posible insertar los controladores necesarios para el aprovechamiento de los recursos de la plataforma y la comunicación con el FPGA, de modo que si el hardware base cambia agregando o quitando algún bus de comunicaciones o cambiando la ubicación de los registros de control de DMA el sistema operativo no podrá comunicarse con el hardware. Entonces, dejando estático todo lo relacionado con las comunicaciones entre CPU y FPGA, es posible modificar el hardware restante sin tener que crear un nuevo sistema operativo haciendo más ágil la tarea de modificar y probar diferentes configuraciones de hardware en la aceleración de algoritmos.

Como se puede observar en la Figura **3-2** de la implementación base propuesta, desde el FPGA del ZYBO se puede acceder a la memoria RAM de la plataforma con ayuda de un bloque DMA, esto facilita la transferencia de datos entre ambas partes del SoC (CPU y FPGA). Además, se cuenta con memorias FIFO de entrada y salida para la transferencia de los datos hacia y desde la memoria RAM, estas ayudan con la sincronización de los datos en las transferencias de datos.

Por su parte, se cuenta con una serie de registros AXI Lite dedicados para comunicar CPU y FPGA para ayudar con el control de procesos dentro del FPGA y ayudar en la sincronización de los datos con banderas de estado.

Cuando se quiere hacer una nueva implementación de hardware a partir del diseño base la parte que cambia es el bloque del acelerador, éste es el que realiza la operación deseada como una convolución 2D y una negación a los datos de entrada para dar un resultado dirigido a la FIFO de salida. Es importante mencionar que este bloque puede o no tener una gestión de memorias dentro de él, esto dependerá del tipo de dato que se trabaje y el tipo de operación a realizar. Por ejemplo, en una operación de convolución 2D con imágenes será necesario almacenar temporalmente los valores de los pixeles de la matriz a la que se le aplicará la convolución 2D hasta concretar la operación.

Una vez descritos los componentes necesarios en el diseño del hardware base, lo que falta es la implementación. Para materializar el diseño base es necesario hacer uso de la herramienta Vivado y utilizar algunos de los muchos IP cores que vienen incorporados en este software.

Para el diseño base basta con utilizar la forma de diseño por bloques con la que cuenta Vivado, básicamente este diseño se hace instanciando bloques y uniendo las conexiones de forma correcta. En algunas ocasiones, Vivado hace sugerencias de los bloques complementarios que se necesitan en nuestro diseño y también hace sugerencias de las conexiones que deben hacerse entre los bloques instanciados.

En este diseño se utilizan los bloques llamados ZYNQ7 Processing System, AXI4-Stream Data FIFO, AXI Direct Memory Access, algunos bloques sugeridos por Vivado y un bloque que represente el algoritmo a implementar llamado Acelerador. Estos bloques serán descritos a continuación y en la Figura **3-3** se observa la interconexión de los bloques que se describirán.

El bloque ZYNQ7 Processing System es un bloque que se encarga de instanciar el CPU del SoC, con este bloque es posible hacer configuraciones del CPU, habilitar señales de control, habilitar buses de comunicación, modificar la señal de reloj generada para el FPGA entre muchas configuraciones útiles. Dos configuraciones obligatorias que se hacen en este bloque son la habilitación de por lo menos un canal de comunicaciones AXI HPO para poder hacer un intercambio de ráfagas de datos con ayuda de DMA. La otra configuración es la habilitación de una señal de reloj de por lo menos 100 MHz, esta señal será el reloj maestro de toda la lógica generada en el FPGA.

El bloque AXI4-Stream Data FIFO es el encargado de crear las memorias FIFO utilizadas

en nuestro diseño. Estos bloques pueden ser configurados para tener cierto tamaño de longitud de palabra y número de localidades de memoria disponibles, es posible configurar aspectos relacionados con los buses de entrada y salida los cuales son AXI Stream. Estos bloques son los responsables de recibir los datos adquiridos por DMA y ponerlos a disposición de ser utilizados por el co-procesador de hardware. De la misma manera, cuando se quieren regresar los datos procesados se usa otra FIFO y ésta se encarga de administrar las señales necesarias con las que se comunica con DMA para que el proceso de transferencia de datos se realice de forma eficiente.

El bloque AXI Direct Memory Access es el que instancia DMA, el bloque permite configurar la forma en que se leen y escriben datos en la memoria del sistema. Este bloque es muy importante debido a que tiene la capacidad de leer y escribir datos sin necesidad de la interferencia del la CPU cuando se está ejecutando alguna de las dos acciones. El CPU interactúa con este bloque a través de registros localizados en una dirección fija de memoria, dicha interacción es puramente para procesos de configuración y visualización del estado de las transferencias de datos.

Los bloques complementarios son agregados por parte de Vivado y son utilizados para control de reset, hacer multiplexado de buses y algunos otros procesos.

El Acelerador es el bloque que realiza la operación deseada. Este es el único bloque que se crea con ayuda de VHDL y con ayuda de Vivado se puede ver como un bloque en el entorno de desarrollo por bloques. Este bloque en el diseño base es una simple operación de suma, es decir, al dato de entrada se le suma un valor constante.

Cuando el diseño está completo, se prosigue a compilarlo en busca de algún error de implementación para después generar un archivo binario encargado de configurar físicamente el hardware del sistema que estamos usando (Zybo). Una vez que se verifica que se pudo generar dicho archivo binario, se prosigue a exportar el diseño con ayuda de Vivado.

Todos los bloques anteriormente descritos conforman el diseño base que será el punto de partida de esta investigación y, como se mencionó anteriormente,la interconexión de cada bloque se muestra en la Figura **3-3**.

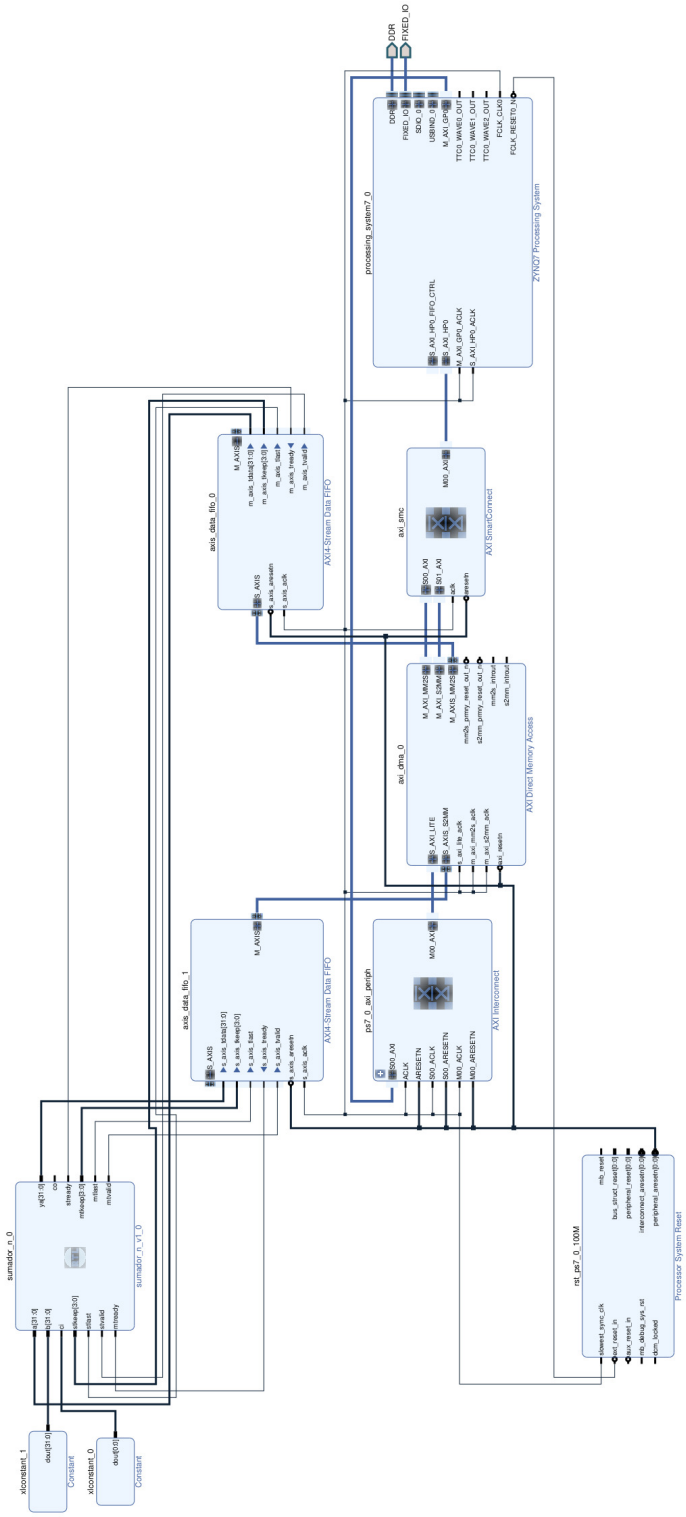


Figura 3-3.: Diseño base en conformado por los bloques ZYNQ7 Processing System, AXI4-Stream Data FIFO, AXI Direct Memory Access entre otros, mostrando la interconexión de dichos elementos, representado en el campo de diseño a bloques de Vivado.

3.2. Creación del sistema operativo base

Una vez que se tiene el hardware base sobre el que se trabajará, es necesario que se construya, compile y empaquete un sistema operativo para que administre los recursos del sistema y además se tengan los controladores de hardware adecuados al diseño especifico que se creó, algunos de los controladores son utilizados en la comunicación entre el CPU y el FPGA y estos son agregados en esta etapa de la metodología. Para lograr lo anterior, se hace uso de las herramientas de Petalinux.

Es necesario que la versión de Petalinux sea la misma que la versión de Vivado donde se creó el hardware base. Petalinux está disponible para sistemas operativos Linux y en esta investigación se está utilizando Ubuntu 16.04.

Petalinux puede trabajar a partir del hardware creado desde Vivado o sobre plantillas de hardware desarrolladas por alguien más. En esta investigación se trabaja con diseños de hardware creados desde Vivado, más específicamente sobre el hardware base descrito en la sección anterior el cual está orientado al procesamiento de imágenes con las funciones de convolución 2D y negación descritas en secciones subsecuentes.

Para obtener una descripción puntual del proceso de la creación, compilación y empaquetado de un sistema operativo funcional a partir del hardware base descrito en la sección anterior véase el Anexo A.

También, para ver el proceso de *booteo* del sistema operativo generado en una microSD que será el disco de arranque de la plataforma utilizada (ZYBO) puede visitar el Anexo B.

3.3. Diseño personalizado de acelerador genérico

En este apartado se describe la forma en que se desarrolló el diseño en hardware de un acelerador de algoritmos de procesamientos de imágenes genérico y una serie de componentes que ayudan en el tráfico, acomodo y el almacenamiento temporal de datos, todo para implementar varios algoritmos de procesamiento de imágenes. Además, se muestra como está constituido dicho acelerador y a su vez se explica cada componente interno del mismo. Todo esto tomando en cuenta que se parte del diseño base de la primer sección de este capítulo.

El diseño general del acelerador es ilustrado en la Figura **3-4**, en dicha figura se mues-

tran los componentes del acelerador y la interconexión que se tiene con otros bloques del exterior, pertenecientes al diseño base. Es importante que se señale que este diagrama no es muy detallado para la rápida comprensión del lector. En secciones posteriores se describirá más a detalle cada componente.

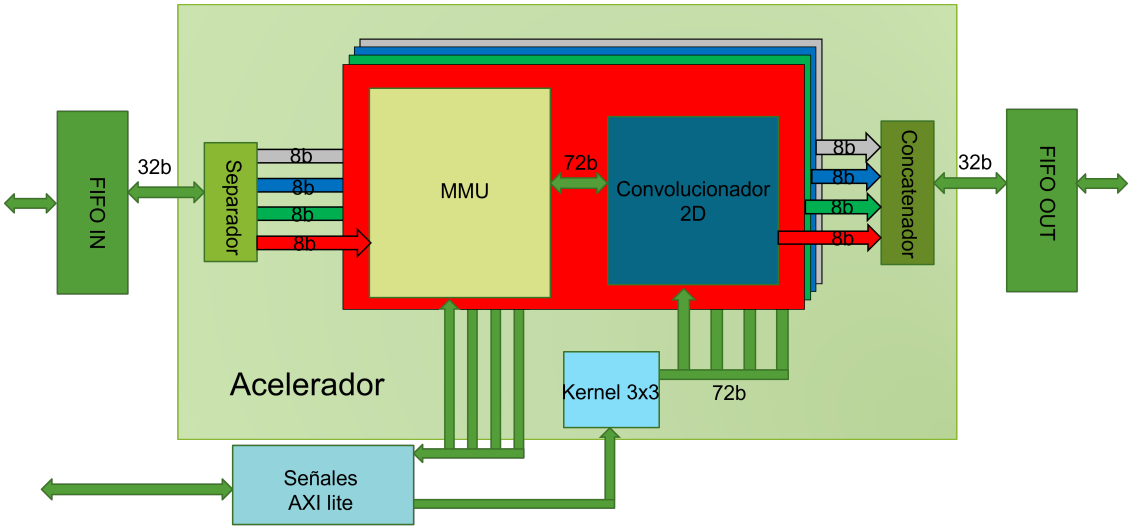


Figura 3-4.: Diagrama general del acelerador

La Figura **3-4** muestra un diagrama que contiene un bloque llamado Acelerador el cual esta constituido por los bloques Separador, MMU, Operador, Kernel 3x3 y Concatenador, todo el bloque Acelerador es el encargado de generar imágenes resultantes de las imágenes de entrada. El bloque Separador se encarga de separar las tuplas de entrada en 4 señales de datos (una señal por canal), el bloque Kernel 3x3 es un banco de registros que contiene los coeficientes del kernel de convolución, el bloque Concatenador esta encargado de unir los datos resultantes de cada canal en tuplas de cuatro datos que representarán los pixeles de salida, por último, el bloque MMU se encarga de la gestión de memoria al nivel del Acelerador y rotación de registros para facilitar que el bloque Operador aplique la función deseada (convolución 2D o negación). Los dos últimos bloques se repiten para la cantidad de canales con los que se trabajará, en este caso cuatro canales, los cuatro canales son trabajados de forma simultanea en paralelo.

En el diagrama de la Figura **3-4** muestra que fuera del Acelerador se encuentran los bloques de un par de memorias FIFO, una de entrada y otra de salida con buses de datos de 32 bits, por dichos buses se transmiten las tuplas de datos correspondientes a los pixeles de entrada provenientes de la imagen de entrada y las tuplas de datos resultantes de los pixeles procesados por el bloque Acelerador. Las memorias sirven como interfaz entre el Acelerador y el bloque DMA del diseño base.

En la Figura **3-4** puede observarse que existen cuatro salidas de datos de ocho bits proveniente de los cuatro bloques Operador hacia el bloque Concatenador al igual que cuatro entradas de ocho bits hacia los cuatro bloques MMU desde el bloque Separador. Dicho lo anterior, es posible tener cuatro núcleos de aceleración separados, cada uno encargando de procesar un canal color de la imagen de entrada de forma paralela.

El bloque llamado Señales AXI lite en la Figura **3-4** tienen una interacción directa con el bloque Acelerador para llevar a cabo la tarea de cambiar los coeficientes del bloque Kernel 3x3, controlar la inicialización de la lectura de datos en los bloques MMU y además servir para informar al CPU acerca del estado del bloque Acelerador. Este bloque es generado con ayuda de Vivado.

A continuación, se describe cada bloque que compone el acelerador.

3.3.1. Bloque Separador

Este bloque tiene una función de separar las tuplas de entrada de 32 bits en cuatro datos separados de ocho bits o un byte. Esta separación se hace debido a que cada canal de las imágenes se procesa de forma separada. Este diseño se pensó teniendo en cuenta que cada pixel puede representar cuatro canales, en nuestro caso el canal rojo, verde, azul y alfa. Y cada pixel es representado por una tupla de 32 bits. El tener cuatro canales en las imágenes facilita el transporte de información entre el FPGA y la memoria del sistema, debido a que el ancho de palabra del bus de datos de DMA es de 32 bits. Por último, no es necesaria la integración de las tuplas de los pixeles de la imagen de entrada debido a que las imágenes PAM están organizadas en tuplas de 4 bytes.

En la Figura **3-5** se muestra la separación de canales. En este bloque la función de separación se realiza de forma paralela y combinacional, sin necesidad de algún proceso de multiplexado. Los canales de datos son de ocho bits y son nombrados arbitrariamente como R, G, B y A. Los canales se ordenan de acuerdo a la figura, el primer canal es el alfa, después el azul, después el verde y por ultimo el rojo, cada canal ocupa ocho bits.

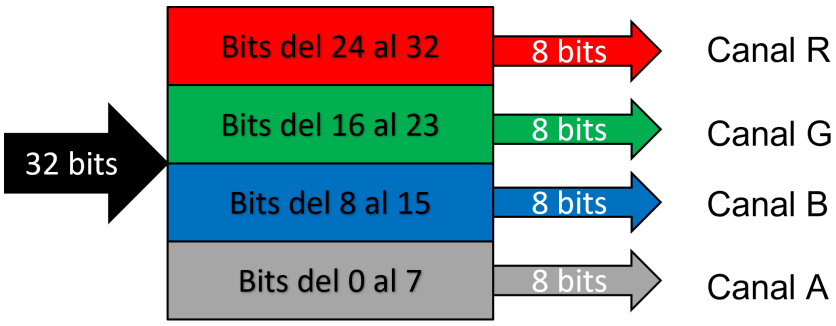


Figura 3-5.: Diagrama del separador de canales.

3.3.2. Bloque Concatenador

Este bloque, al igual que el anterior, desempeña una función muy básica, este bloque se encarga de concatenar los diferentes canales de ocho bits en una tupla de salida de 32 bits listo para ser transmitido a la memoria FIFO de salida. La tarea de este bloque es opuesta al bloque Separador. La tarea de este bloque se realiza de forma paralela y combinacional.

En la Figura **3-6** se ilustra cómo está constituido de forma abstracta este bloque tomando en cuenta los cuatro canales a unir. En dicha figura se muestra que en los primeros ocho bits se almacena el resultado del canal alfa, en los siguientes el azul, después el verde y por último el rojo en los ocho bits restantes de la tupla de 32 bits.

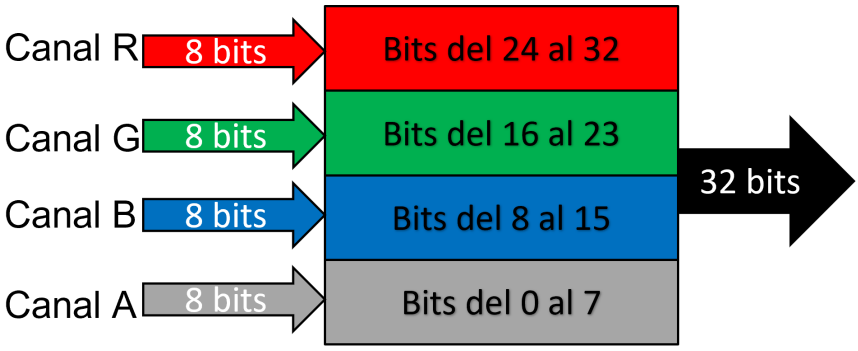


Figura 3-6.: Diagrama del concatenador de canales.

3.3.3. Bloque Kernel 3x3

Este bloque se encarga de separar y acomodar los coeficientes recibidos a través de registros AXI lite para crear un kernel de convolución listo para ser utilizado por parte del

bloque Operador. El bloque recibe tres señales AXI Lite de 32 bits, las cuales contienen nueve valores válidos de ocho bits que serán los coeficientes del kernel.

En la Figura **3-7** se ilustra el acomodo de los datos recibidos por parte de registros AXI Lite para tener una salida de 72 bits en la cual están contenidos nueve coeficientes de ocho bits. De igual forma, en la Figura **3-7** puede verse de forma matricial el acomodo de los datos.

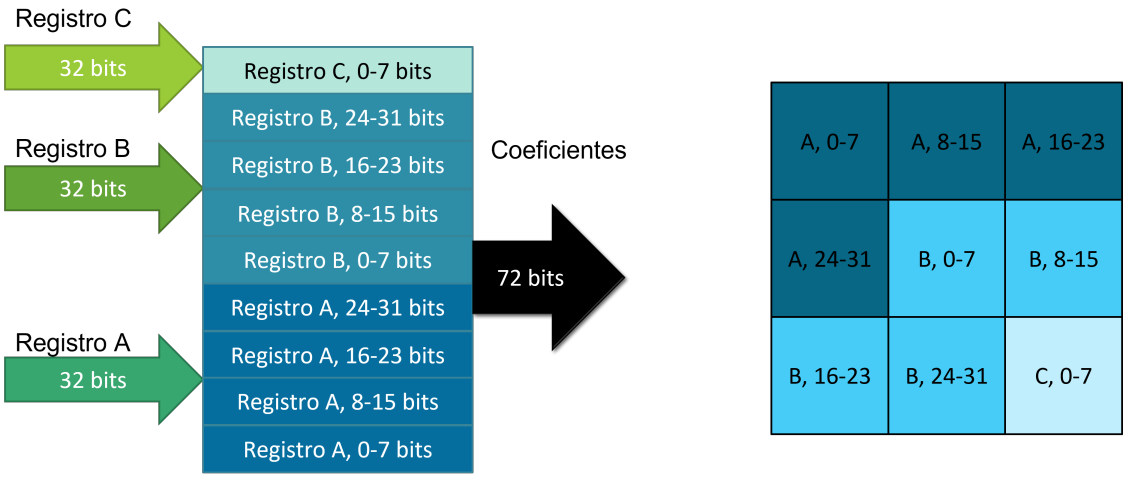


Figura 3-7.: Representaciones del kernel 3x3 a nivel de registros (izquierda) y a nivel de matriz (derecha).

El bloque del kernel 3x3 también puede implementarse solamente con lógica combinacional tomando los coeficientes de ocho bits de los registros de 32 bits. Lo anterior es una tarea que se realiza usando VHDL sin hacer ningún corrimiento de bits, solo se hace un mapeo de los registros de 32 bits en los diferentes registros de ocho bits de los coeficientes del kernel. VHDL puede trabajar a nivel de bits facilitando esta tarea. Algo similar aplica a la implementación de los dos bloques anteriores.

3.3.4. Bloque MMU

Este bloque, a diferencia de los tres anteriores, es mucho más complejo y está compuesto por otros bloques mas sencillos. En general el MMU es una unidad de manejo de memoria diseñada en esta investigación. Es el diseño de hardware más complejo de esta investigación, incluso más que el bloque que corresponde al operador.

Algunas de las funciones de este diseño son el de recibir los datos de entrada, acomodar adecuadamente los datos en bloques de BRAM (incluidos en el MMU), sincronizar la lectura y escritura de datos en las BRAMs y, además, acomodar los datos de salida en un vector de 72 bits listos para ser tomados por el bloque Operador.

En la Figura **3-8** se ilustran los componentes del bloque MMU y la interacción que tienen sus componentes y algunos componentes externos. Se observan tres componentes principales, estos son el bloque MMU WR, RAMS y MMU RD descritos en secciones subsecuentes.

En la Figura **3-8** solo se representa uno de los cuatro bloques MMU utilizados en esta investigación, los otros tres son idénticos a excepción de la señal de entrada llamada Dato entrada que corresponde al dato de uno de los cuatro canales de la imagen PAM (R, G, B y A). En el diagrama de la figura solo se representa uno de los cuatro bloques MMU para simplificar la visualización.

En la Figura **3-8** también se observa que se tienen una serie de señales con el nombre **V** y **R**, estas señales tienen la función de comunicar y coordinar ciertos bloques para funcionar como una arquitectura de procesamiento Pipeline. Un ejemplo de un uso de estas señales seria el de indicar desde el bloque MMU WR al bloque MMU RD que ya hay datos escritos en las memorias RAM para que comience a leerlos. Las otras señales serán explicadas en las en las siguientes secciones.

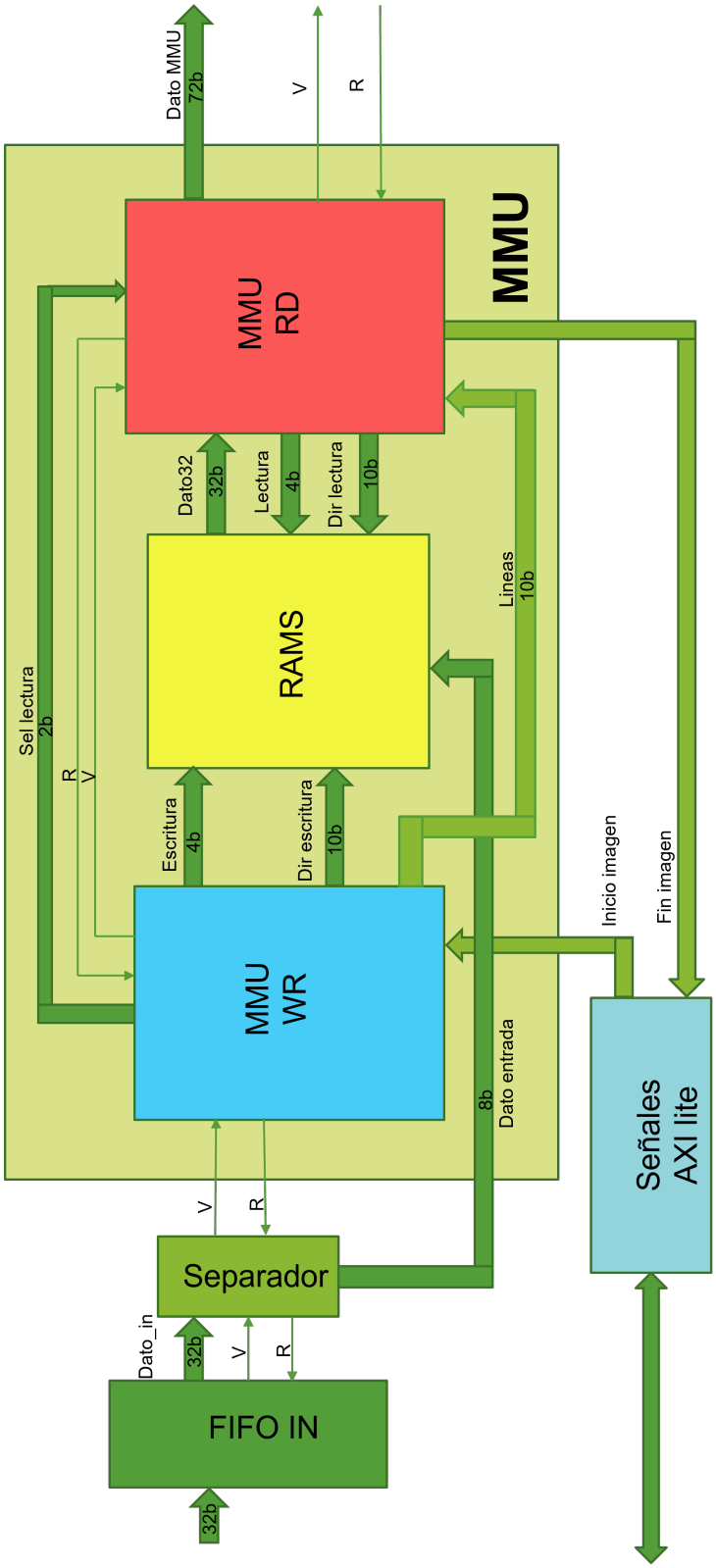


Figura 3-8.: Composición del MMU.

Bloques de RAM

En la Figura **3-8** se observa que se tiene un bloque llamado RAMS, este bloque esta compuesto a su vez por cuatro bloques de RAM del tipo Dual Port RAM, es decir, cada bloque de RAM tiene un puerto de escritura y uno de lectura. En la Figura **3-9** puede verse como está conformado un bloque de RAM.

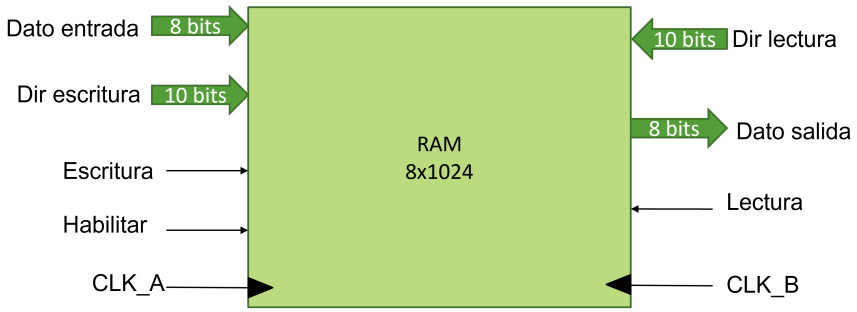


Figura 3-9.: Dual Port RAM.

En la Figura **3-9** se ilustran los dos puertos mencionados de un bloque de RAM de tamaño 8 x 1024, es decir, en esta RAM caben 1024 registros de ocho bits. En el puerto de escritura se necesita un bus de datos de ocho bits de longitud de palabra (**Dato entrada**), un bus de direcciones de escritura de 10 bits (**Dir escritura**), una señal de reloj (**CLK_A**) y las respectivas señales de habilitación (**Escritura** y **Habilitar**). En el puerto de lectura se tiene un vector de datos de salida de ocho bits (**Dato salida**), un vector de entrada para la dirección de lectura de 10 bits (**Dir lectura**), una señal de reloj (**CLK_B**) y una señal de habilitación a la lectura (**Lectura**).

Como se mencionó, el bloque de la Figura **3-9** es una de las cuatro RAM contenidas en el bloque RAMS. Los cuatro bloques tienen el mismo **Dato entrada**, el mismo vector **Dir escritura**, el mismo vector **Dir Lectura** y las mismas señales **CLK_A** y **CLK_B**. Lo que tiene por separado cada bloque son las señales **Lectura** y **Escritura** y **Dato salida**. La señal **Habilitar** siempre permanece habilitada.

En la Figura **3-10** se puede ver la composición mencionada anteriormente. El bloque Concat se encarga de concatenar las cuatro señales **Dato salida** en un vector de 32 bits denominado **Dato32**.

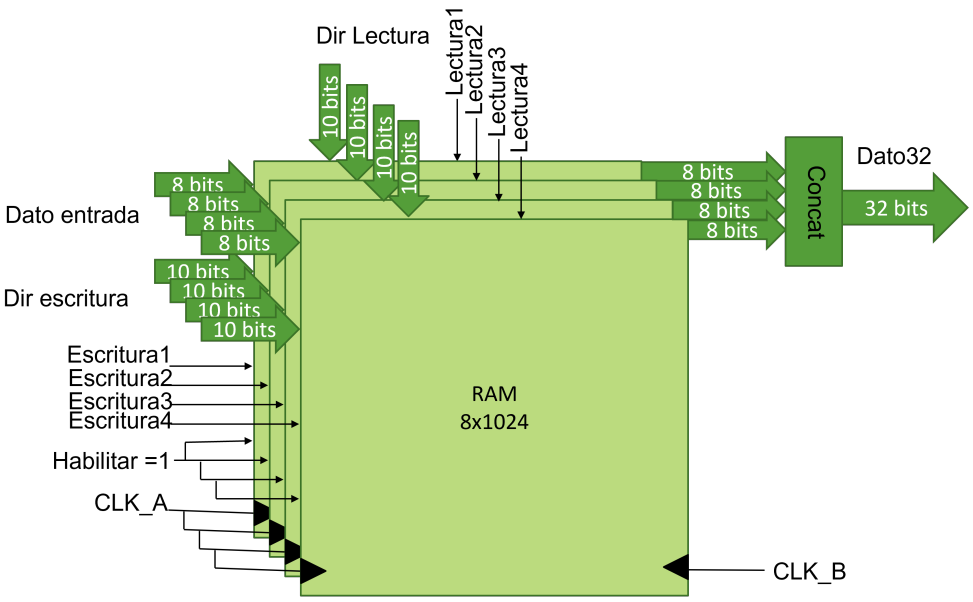


Figura 3-10.: Bloque compuesto por 4 BRAMS.

Los bloques de RAM o BRAM en FPGAs son utilizados para almacenar datos de forma temporal, y en nuestro caso, el diseño del bloque compuesto RAMS tiene una utilidad fundamental. En este bloque será posible escribir en un bloque de RAM y leer en los tres restantes de forma simultánea.

El procedimiento anterior, es útil debido a que las imágenes se planean leer a través de DMA una línea horizontal a la vez, cada línea estará almacenada temporalmente en alguno de los cuatro bloques de RAM y al momento de leer los datos contenidos en los bloques de RAM se hará la lectura de tres líneas a la vez, esto para poder formar una máscara de 3x3 elementos para un proceso de convolución 2D que se ejecutará en un bloque externo al MMU.

La lógica está pensada para que mientras un bloque de RAM se está escribiendo, los otros tres se estén leyendo. En una subsección más adelante, se describe como es el orden de escritura y lectura de los bloques de RAM y, por ende, se entenderá como se hace la total manipulación del bloque RAMS del MMU.

Bloque MMU WR

Este bloque es el encargado de escribir los datos recibidos desde el exterior del MMU en los bloques de RAM. Éste bloque se encarga de controlar la recepción de datos, de hacer una rotación de registros para siempre estar escribiendo en algún bloque de RAM que no

se esté utilizando por el bloque MMU RD y, además, el bloque MMU WR se encarga de avisar al bloque MMU RD que línea horizontal de la imagen se está leyendo. La señal de inicio de este bloque es **Inicio imagen** proveniente del bloque Señales AXI lite mostrado en la Figura **3-8**

En la Figura **3-11** se observa la composición del bloque MMU WR. Cuenta con una máquina de estados finitos llamada FSM_MMU_WR para el control de todo el proceso, tiene un par de contadores, uno dedicado a contar la posición horizontal de la imagen (Cont_x) y otro para la vertical (Cont_y), también cuenta con un pequeño módulo llamado Selector_w.

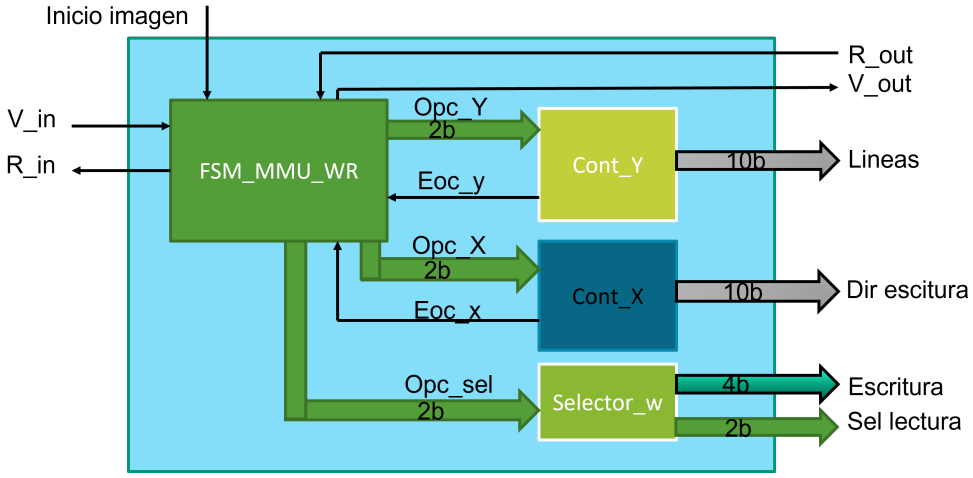


Figura 3-11.: Bloque MMU WR.

El bloque Selector_w es el encargado de hacer la rotación para definir que bloque de RAM se escribe y cuáles se leen con las señales **Escritura** (cada bit del vector puede habilitar una de las cuatro RAM) y **Sel lectura** respectivamente. En todo momento, mientras se está escribiendo en un bloque de RAM, se debe estar leyendo en los tres restantes. Esto para optimizar el tiempo de lectura y procesamiento de la imagen de entrada. El orden de rotación y acomodo de los registros leídos en las RAM a cargo de **Sel lectura** es el que se muestra en la Figura **3-12**.

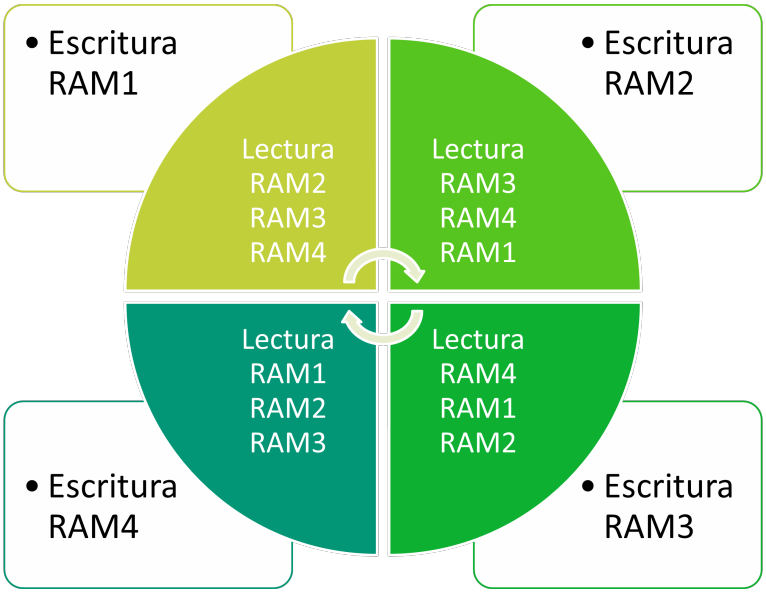


Figura 3-12.: Orden de selección de registros por parte del Selector_w.

La figura **3-12** intenta explicar que el acomodo de registros de las RAM en la lectura se hace de tal forma que el registro superior corresponde a la línea leída más antigua de las tres, el segundo registro corresponde a la penúltima línea leída y el tercer registro corresponde a la línea más recientemente leída. Todo esto mientras se escribe una nueva línea.

Bloque MMU RD

Este bloque MMU RD tiene como funciones la lectura de los bloques de RAM, la función de avisar cuando se termine de procesar la imagen completa con la señal **Fin imagen**, tiene la habilidad de sincronizarse con el bloque MMU WR para que cuando uno termine su función el otro lo espere, además, este bloque entra en operación solo si el bloque Operador (fuera del MMU) está listo para procesar.

El bloque MMU RD coloca los datos leídos de tal forma que crea máscaras de 3x3 de registros de ocho bits de acuerdo con la señal proveniente del bloque MMU WR llamada **Sel lectura**. La disposición de los datos se hace de tal forma que se hace un barrido de la imagen de entrada con la máscara mencionada.

Como se puede ver, en la Figura **3-13**, este bloque está compuesto por otros bloques más pequeños. Cuenta con una FSM llamada FSM_MMU_RD que controla todas las tareas del bloque MMU RD, tiene un contador que es utilizado para desplazarse en horizontal en las imágenes (Cont), contiene un bloque Hold_y que sirve para habilitar una señal

cuando se está en la última línea de la imagen, se tiene un bloque llamado Selector_rd el cual es controlado por el bloque Selector_w del bloque MMU WR, por último, se cuenta con un banco de registros (Reg_bank) que es el encargado de almacenar la máscara de 3x3 anteriormente mencionada.

En la Figura **3-13** el bloque Selector_rd toma tres bytes de la señal **Dato32** (lectura de las memorias RAM) y los acomoda de acuerdo a la explicación mencionada en la Figura **3-12**. Posteriormente estos tres datos son leídos por el bloque Reg_bank para formar la mascara de 3x3 elementos que saldrá por la señal **Dato MMU**. Tienen que hacerse al menos tres lecturas de registros al inicio de cada linea para poder realizar la convolución 2D en el bloque Operador.

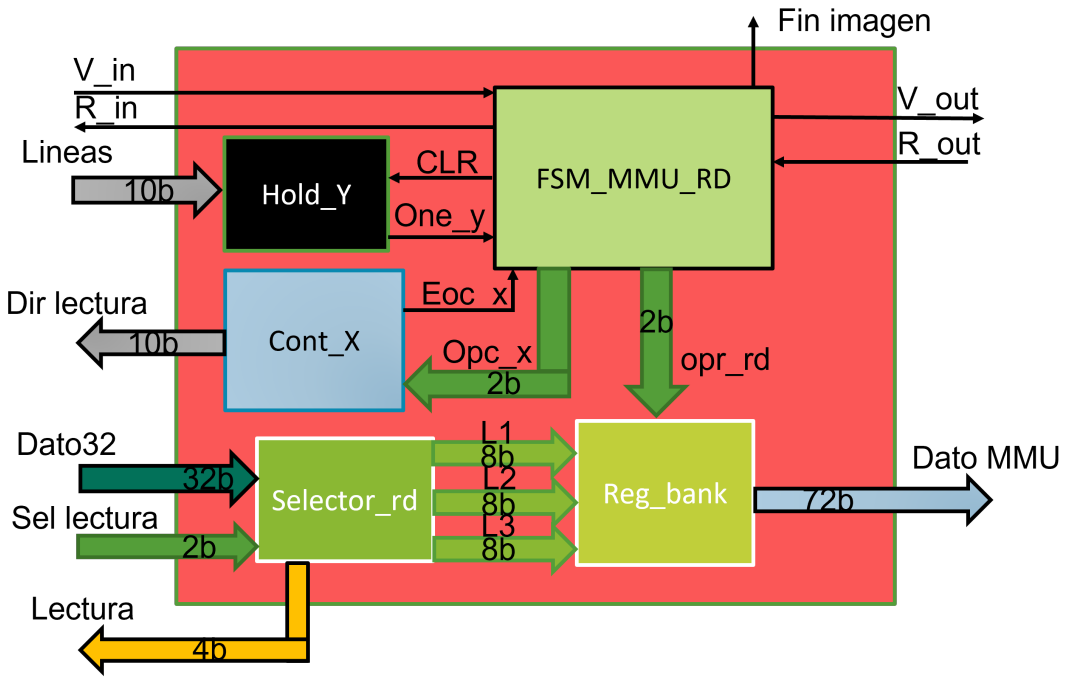


Figura 3-13.: Bloque MMU RD.

3.3.5. Bloque operador

Este bloque es el encargado de realizar la operación que queramos implementar en este acelerador. En esta investigación se usaron dos casos de estudio diferentes, la operación de la negación y la operación de la convolución 2D.

A continuación, se describen las implementación de cada uno de las operaciones realizadas.

Operación de negación

Esta operación no requiere de un acomodo en forma de máscara de los datos de entrada, tampoco necesita de los coeficientes del kernel de convolución. Este bloque toma el dato central de ocho bits de la máscara **Dato MMU** (ver Figura **3-13**) y le aplica la negación. En la Figura **3-14** se muestra la operación de la negación descrita en forma de diagrama.

Como se puede ver en la Figura **3-14**, **d5** representa el byte central de la señal **Dato MMU**. A este byte es al que se le aplica la negación para obtener la señal **salida**.

La negación es una operación muy simple y fue utilizada para probar el sistema integrado del acelerador con el diseño base.

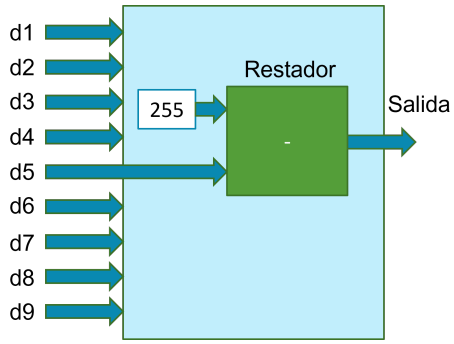


Figura 3-14.: Diagrama de la negación.

Operación de convolución 2D

Esta operación hace uso de toda la máscara saliente del MMU (**Dato MMU**) y también hace uso del kernel de convolución del bloque Kernel 3x3. Esta operación está ilustrada en la Figura **3-15**. Las entradas d y k se refieren a los datos contenidos en la señal **Dato MMU** y a los coeficientes del kernel de convolución. Ambos tipos de datos son vectores de ocho bits. La señal **salida** es el resultado de la convolución 2D y también es un vector de ocho bits.

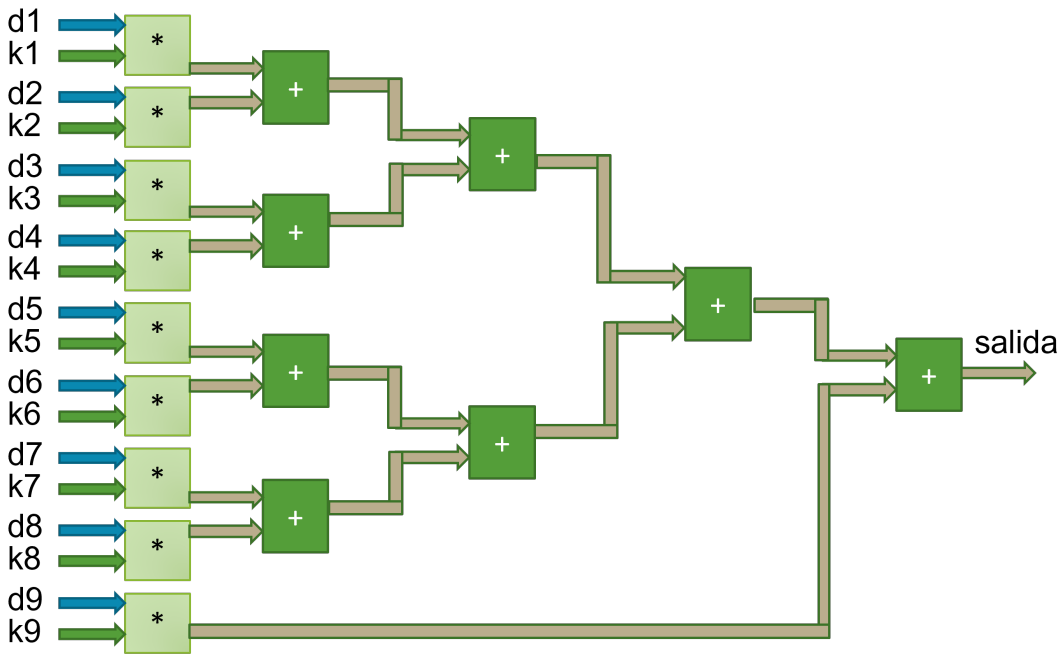


Figura 3-15.: Diagrama de la convolución.

En esta operación es necesario expandir los vectores para crear números reales de punto fijo de 18 bits para poder hacer las multiplicaciones con el ancho de palabra requerido para el uso de los multiplicadores dedicados de los DSP de la parte del FPGA, en el resultado de la última suma es necesario reducir el vector resultante a ocho bits. Los multiplicadores utilizados en esta operación son combinacionales y admiten un ancho de palabra de hasta 18 bits.

3.4. Software de interfaz para la negación por hardware

En esta sección se describe el diseño de software para el control de la operación de la negación usando el hardware descrito en secciones anteriores. De manera general, el programa está escrito en el lenguaje de programación C y obedece el diagrama de flujo de la Figura **3-16**.

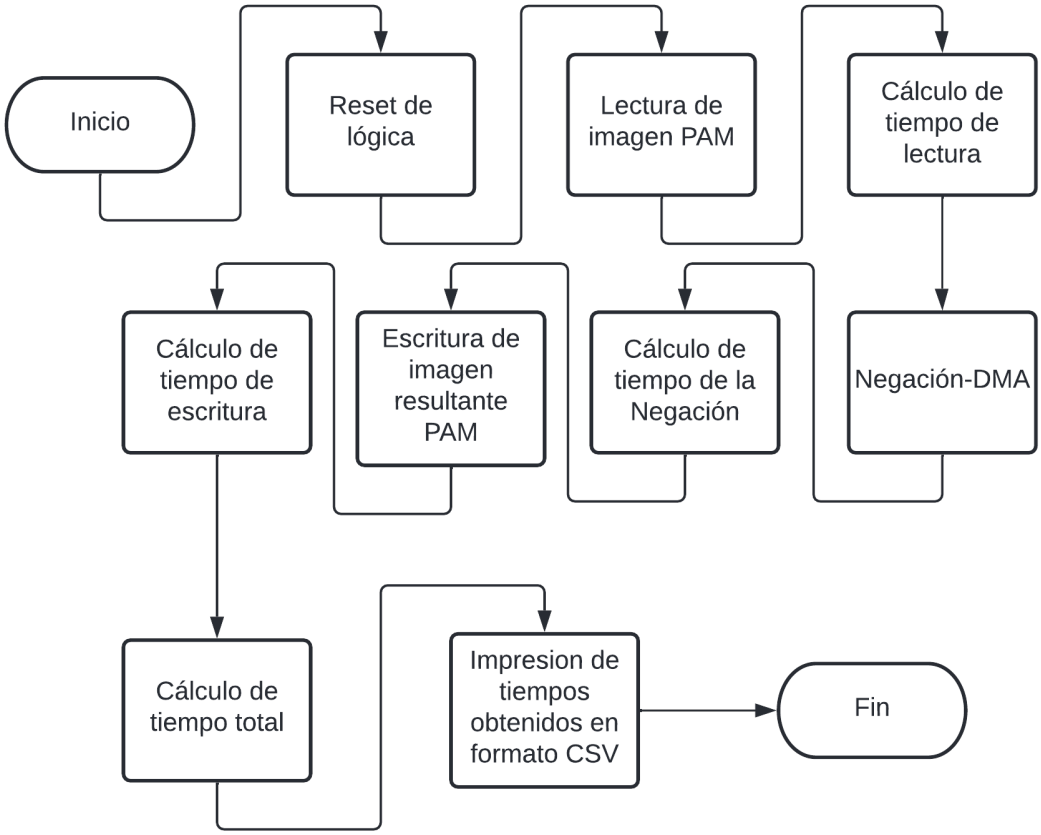


Figura 3-16.: Diagrama de flujo del software de interfaz de la negación por hardware.

En este programa se utiliza la asignación de memoria a los arreglos que contendrán los datos de la imagen leída (**ArrayR**) y la imagen resultante (**ArrayW**) desde antes de la ejecución de la función de lectura de imagen. Esto se hace de esta forma porque al arreglo se le asigna una dirección fija, misma que se utilizará al momento de comunicarle a DMA la ubicación de dicho arreglo. Además, el tamaño de las imágenes siempre es el mismo, 640 x480 pixeles. En caso de querer procesar imágenes de otros tamaños se tiene que modificar el tamaño de los bloques de RAM y modificar algunos parámetros como el limite de los contadores usados en los bloques MMU WR y MMU RD, es decir, es necesario modificar el hardware.

Las lecturas de tiempo se hacen siempre con ayuda de la biblioteca time.h y la función clock(). Todos los tiempos medidos se calculan en segundos para mantener una escala estándar en todas las mediciones.

La función de lectura de la imagen hace una revisión del encabezado del formato PAM

para evitar leer imágenes corruptas o de un tamaño no valido. Los datos de los pixeles son almacenados en formato binario en el arreglo **ArrayR** anteriormente mencionado y este arreglo tiene sus casillas con un tamaño de 32 bits, el tamaño necesario para evitar correcciones de tipo de dato.

El proceso de la negación se hace mediante hardware, por ello lo único que se hace en este programa es inicializar y configurar DMA para que tome los datos de la imagen y retorne los datos procesados en el arreglo **ArrayW**.

3.5. Software de interfaz para la convolución 2D por hardware

El programa para llevar a cabo la convolución 2D por hardware es bastante similar al programa realizado para la negación por hardware. Tan solo viendo el diagrama de flujo de la Figura **3-17** de este programa, en el bloque después del inicio, se puede ver que el único cambio introducido es la escritura de los coeficientes del kernel de convolución y esto se hace en la dirección correspondiente a los registros AXI lite para que los cambios sean percibidos en el FPGA.

Las direcciones de los registros de control de DMA, de los registros de las banderas de DMA, y de los registros de entrada y salida de AXI lite son asignadas por Vivado durante la creación del hardware base.

El algoritmo de la convolución 2D es llevado a cabo de la misma forma que la negación, solo se habilita DMA y el hardware hace todo el proceso. Al terminar la convolución 2D se tiene un arreglo en memoria del sistema que contiene la imagen procesada.

Los coeficientes utilizados para la convolución fueron 1/9 representado con un byte, es decir 00011100_2 asumiendo que cada valor esta después del punto decimal, y se asignó ese valor debido a que se quiso probar la convolución 2D con el filtro de la media. Usando una imagen con ruido Gaussiano se podría saber si la convolución funciona como debería.

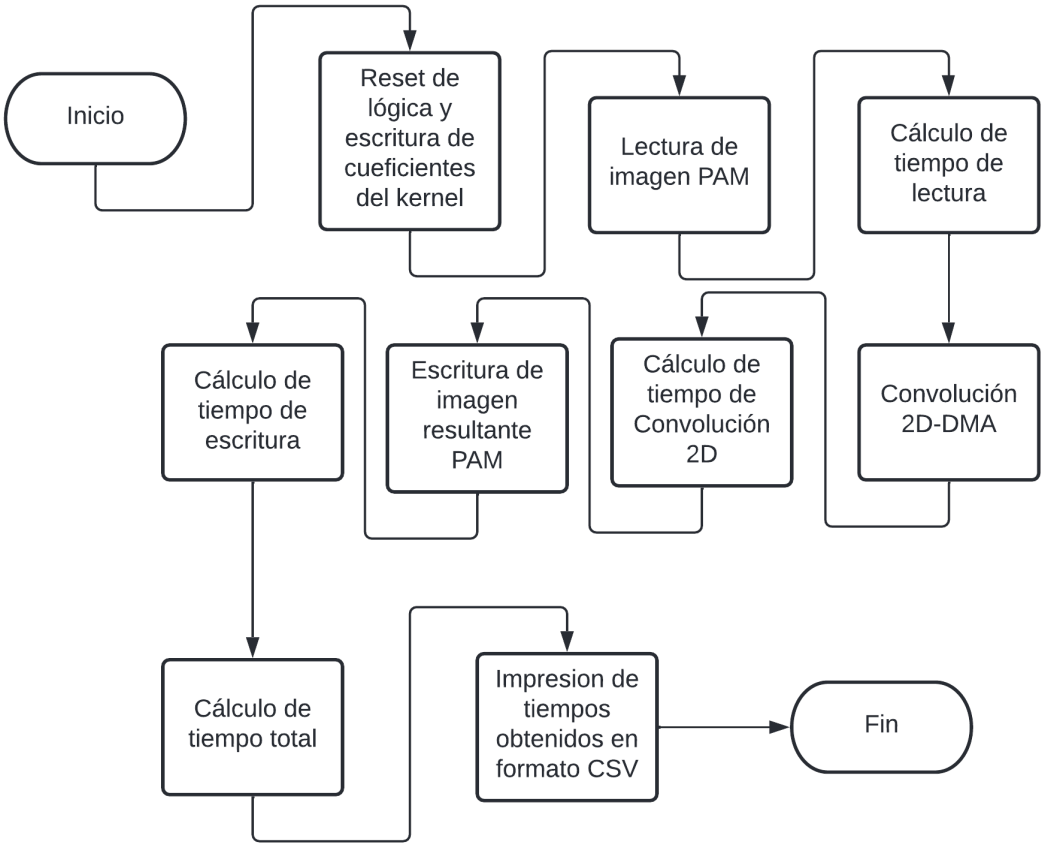


Figura 3-17.: Diagrama de flujo del software de interfaz de la convolución 2D por hardware.

3.6. Negación solo por software

En este apartado se describirá el software que se programó para realizar la función de negación de una imagen RGB de formato PAM tan solo utilizando el CPU. El diagrama de la Figura **3-18** muestra a grandes rasgos el funcionamiento de dicho programa.

En el diagrama se puede resaltar que el proceso de la negación, una vez que la imagen se ha leído y se tiene cargada en un arreglo, se hace con ayuda de un recorrido de cada pixel de la imagen, es decir, un recorrido matricial considerando la imagen como una matriz, este recorrido se hace de forma secuencial en una forma como se muestra en la Figura **3-19**, es decir, un pixel a la vez iniciando en la parte superior izquierda y recorriendo de izquierda a derecha, de arriba hacia abajo. En cada posición de pixel se calcula la operación de negación.

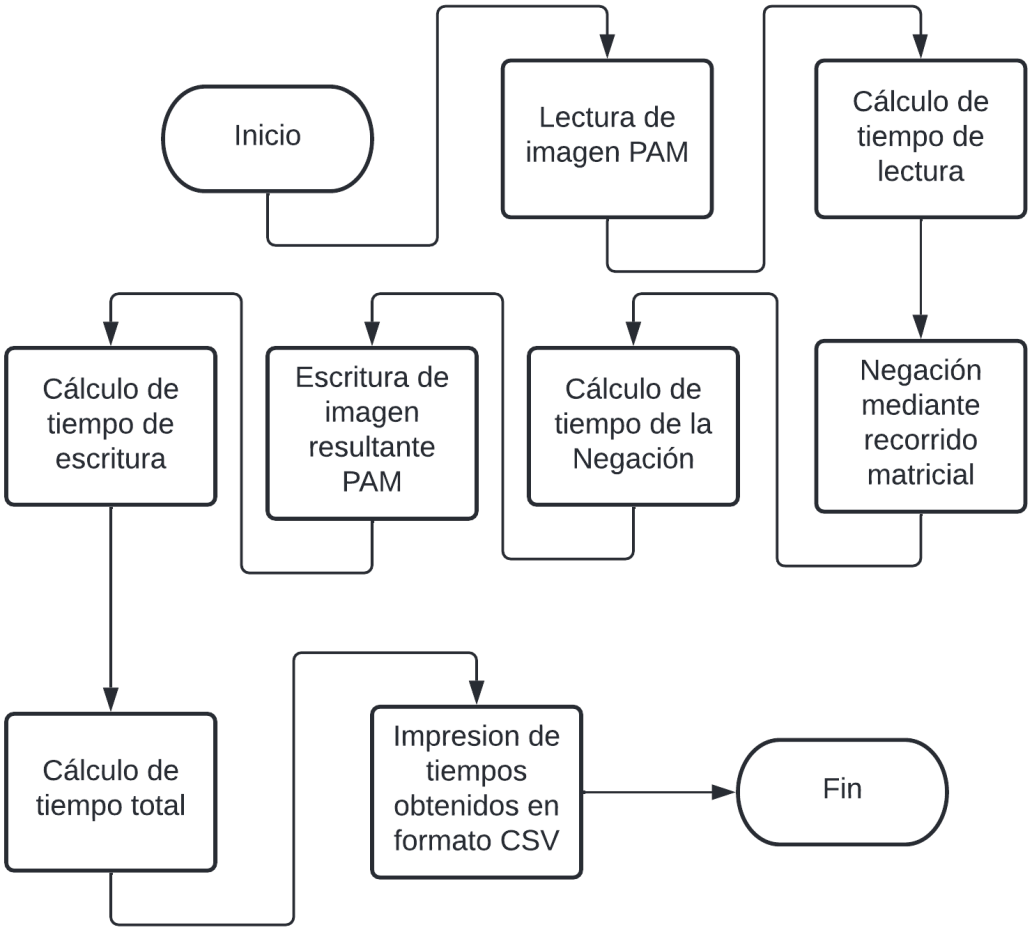


Figura 3-18.: Diagrama de flujo del software de la negación solo por software.

En la Figura **3-19** el recorrido es como se mencionó en el anterior párrafo. Los desplazamientos se dan de izquierda a derecha como lo señalan las flechas negras y al finalizar una linea en la imagen se prosigue a saltar al inicio de otra nueva linea como lo indican las flechas azules.

Una de las diferencias que tiene este software comparado con el de la ejecución de la negación por hardware es que en este software hace uso de memoria dinámica para almacenar los arreglos que contendrán la imagen de entrada y la imagen de salida. En el software de interfaz para la negación requería que la memoria de los arreglos que contienen la imagen de entrada y salida fuera estática con una dirección fija para señalar a DMA donde están los arreglos que contendrán las imágenes.

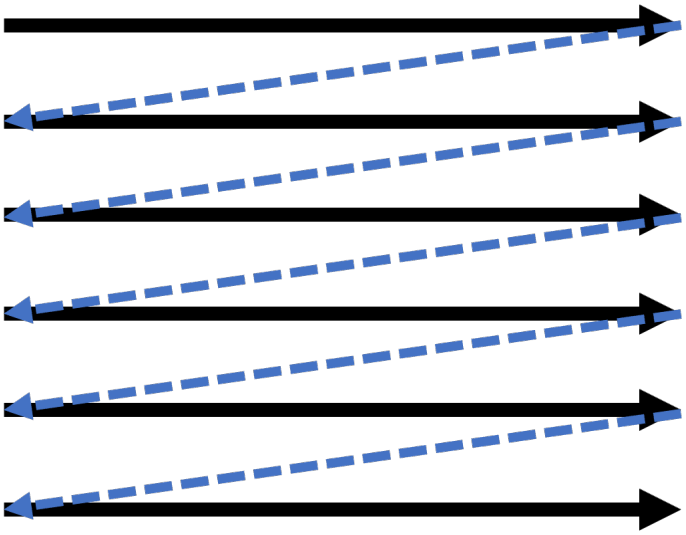


Figura 3-19.: Diagrama que representa el recorrido matricial que se le hace a cada imagen para aplicar la operación negación.

3.7. Convolución 2D solo por software.

El programa creado para realizar la convolución únicamente por software está descrito en la Figura **3-20**. En dicho programa puede destacarse que el kernel de convolución se va desplazando con un recorrido matricial por toda la imagen (como en la negación en la Figura **3-19**) y se va generando la imagen resultante aplicando la convolución 2D en cada desplazamiento de un pixel.

La convolución en cada máscara de la imagen se hace con ayuda de un algoritmo de multiplicación- acumulación señalado en el diagrama como algoritmo MAC y es mostrado en la Ecuación 3.1, este algoritmo es parecido al de la arquitectura de procesamiento MAC pero hecho de manera secuencial en un CPU. La función es aplicada a cada canal RGB de la imagen (no es necesario hacerlo para el canal Alfa).

$$a = a + (m_n * k_n) \tag{3.1}$$

Donde a tiene un valor inicial 0, m_n y k_n son las variables que toman los valores de la máscara de convolución y el kernel de convolución respectivamente. Las variables se multiplican para después sumar el resultado con lo acumulado en a. La operación se realiza en un ciclo que se ejecuta nueve veces, es decir, el valor de n es un entero que cumple con $1 \leq n \leq 9$. Esto es así en para cubrir los nueve coeficientes del kernel de

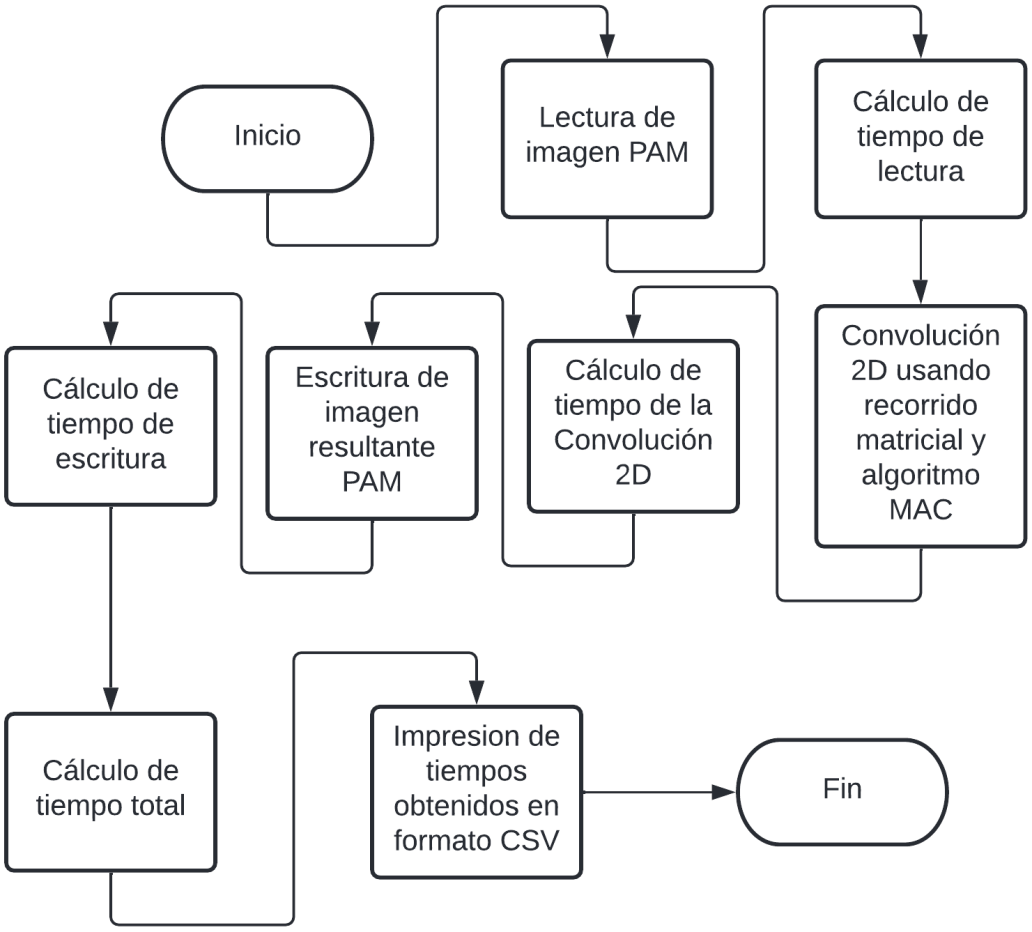


Figura 3-20.: Diagrama de flujo del software de la convolución 2D solo por software.

convolución 2D de 3x3. El valor de a resultante después la finalización del ciclo será el valor de la convolución del pixel donde se encuentra la máscara de convolución.

4. Resultados experimentales

En este capítulo se describen los resultados obtenidos de la implementación de la metodología propuesta en este trabajo de investigación. Principalmente se harán comparaciones de tiempo de procesamiento de dos algoritmos básicos de procesamiento digital de imágenes que se implementan de tres formas diferentes.

En primera instancia, se hará un estudio comparativo de la implementación de la función de la negación para después pasar a discutir los resultados de la implementación de la función de la convolución 2D con un caso de estudio de un filtro de la media. Todos los resultados experimentales de la implementación de los algoritmos mencionados son obtenidos de dos plataformas de procesamiento, la tarjeta ZYBO y una laptop personal.

En la Tabla **4-1** se muestran algunas características de interés de la tarjeta ZYBO y de la laptop utilizada. Dichas características sirven para tener más puntos de comparación al momento de implementar cada método, principalmente los basados solamente en software.

Tabla 4-1.: Comparación entre el ZYBO y la laptop.

Característica	ZYBO	Laptop
Frecuencia de reloj del CPU	650 Mhz	1.8 GHz
Tamaño de memoria caché L1	64 kB	320 kB
Tamaño de memoria caché L2	512 kB	2 MB
Tamaño de memoria RAM	512 MB	12 GB
Tipo de memoria RAM	DDR3	DDR3
Sistema operativo	Creado con las herramientas de Petalinux	Ubuntu 16.04
Tipo de almacenamiento	microSD clase 10	SSD Sata 2.5 pulgadas
Velocidad teórica de lectura del almacenamiento	100 MB/s	500 MB/s
Velocidad teórica de escritura del almacenamiento	85 MB/s	350 MB/s

4.1. Resultados de la operación negación

Los resultados que se presentan a continuación son los obtenidos tras ejecutar la operación de la negación de imágenes, es decir, invertir los valores de los colores RGB de una imagen de entrada aplicando una operación de resta.

En total, se están utilizando tres métodos para realizar esta operación con imágenes:

- El método SW_ZYBO.
- El método SW_LAP.
- Y el método HW_ZYBO.

El método SW_ZYBO consiste en utilizar el CPU de la tarjeta ZYBO con ayuda de un programa hecho en el lenguaje C ejecutado sobre Petalinux. Las imágenes de entrada para este programa y los dos posteriores son de formato PAM de cuatro canales con una profundidad de ocho bits por canal (el canal Alfa se descarta).

El método SW_LAP es la ejecución del mismo programa utilizado en el método SW_ZYBO pero utilizando una laptop con sistema operativo Ubuntu 16.04 y con un CPU de arquitectura AMD64 x86.

El método HW_ZYBO consiste en usar tanto el CPU como el FPGA de la tarjeta ZYBO para realizar la negación de imágenes RGB. El CPU se encarga, mediante un programa escrito en lenguaje C, leer y escribir las imágenes de entrada y salida respectivamente. El CPU indica al FPGA cuando la imagen de entrada se encuentra cargada en la memoria RAM para que posteriormente el FPGA, ayudado por el bloque DMA, lea de la dirección de memoria donde se encuentra la imagen para proseguir a realizar la operación de la negación en el FPGA. El FPGA, ayudado por DMA, retorna la imagen resultante de la negación en una dirección de la memoria RAM previamente indicada por el CPU.

En total se realizaron 30 ejecuciones de cada método con arranques en caliente, es decir, no se vacían las memorias caché entre cada ejecución. La Figura **4-1** muestra los tiempos promedios para la ejecución total del programa, la lectura de la imagen, la negación y la escritura de la imagen resultante. Las barras de error corresponden a la desviación estándar de cada medida.

Como se muestra en la Figura **4-1**, el método SW_LAP es el más rápido con un tiempo total de 0.01051 segundos (s), pero también es el que tiende a oscilar más en sus tiempos de respuesta, es decir tiene una mayor desviación estándar. Por otro lado, el

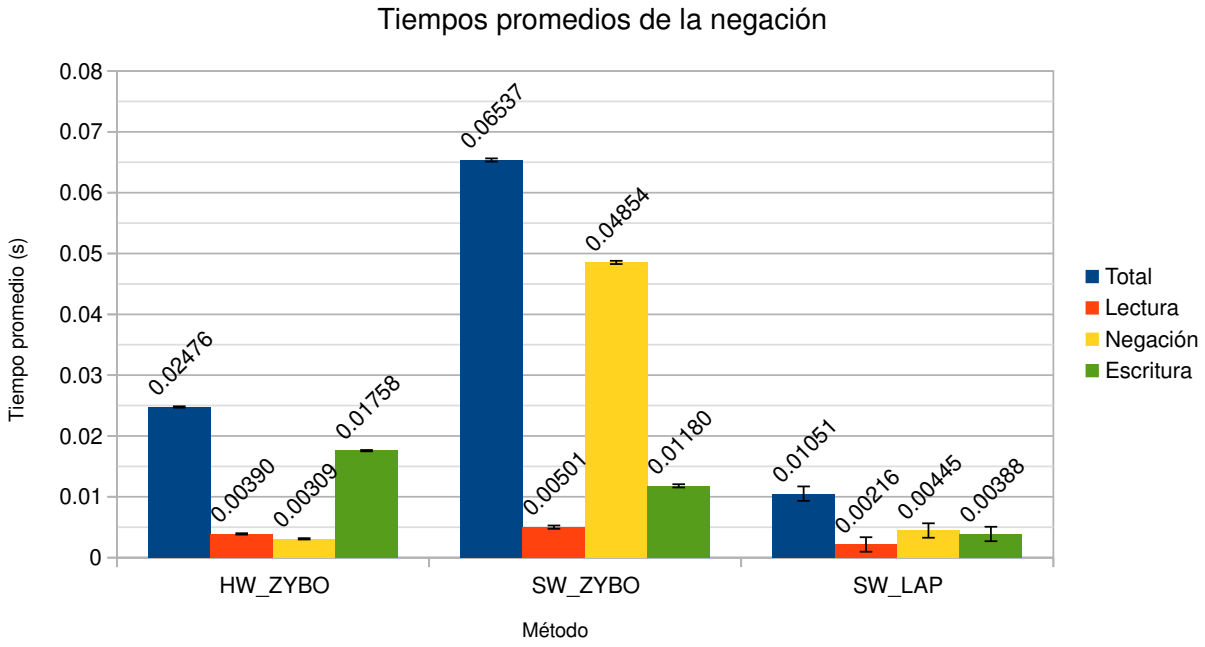


Figura 4-1.: Tiempos promedios al implementar los tres métodos para realizar la negación.

método HW_ZYBO es el que obtiene tiempos de respuesta estables (con menos desviación), mientras que su tiempo total de ejecución fue el doble que el método SW_LAP con 0.02476s. El menor tiempo para realizar la operación de negación (representada por el color amarillo) fue obtenida por el método HW_ZYBO con 0.00309s.

El método SW_ZYBO fue el que demoró más tiempo en total con 0.06537 s, pero tiene una desviación estándar menor al método SW_LAP, esto se puede deber a que el sistema operativo creado con Petalinux no tiene tantos procesos ejecutándose a diferencia que un sistema operativo pensado para una PC.

El hecho de que la tarea de la negación sea más rápida en el método HW_ZYBO que en los otros dos puede atribuirse a que esta tarea no se ve interrumpida en hardware, mientras que si se usa por software el CPU puede interrumpir la tarea para ejecutar otros procesos que estén en espera de utilizar el CPU.

A manera de ilustrar mejor las oscilaciones de tiempo entre cada experimento por cada método, se muestra en la Figura **4-2** las desviaciones estándar de cada proceso.

Particularmente, resaltamos que en el método de HW_ZYBO las oscilaciones en el tiem-

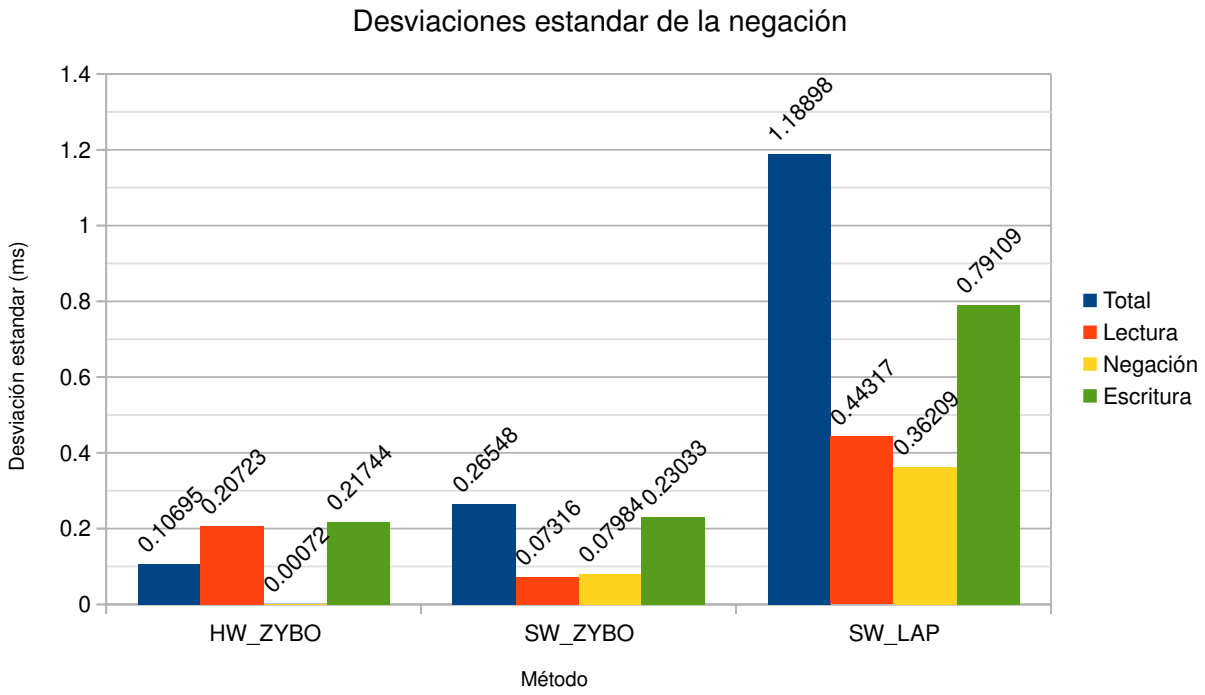


Figura 4-2.: Desviación estándar del tiempo necesario para calcular la negación.

po de procesamiento de la operación de negación durante las 30 ejecuciones son muy estables con 0.00072s de desviación estándar.

Para explicar claramente los tiempos que tarda cada proceso por cada método, se muestran en la Figura **4-3** los tiempos acumulados de cada tarea.

Es fácil notar que la tarea de la lectura de las imágenes es relativamente rápida comparado con las otras dos tareas y donde más se marca esta diferencia es utilizando solo software en la ZYBO debido que con este método la lectura tarda 0.00501s, mientras que la operación de la negación y la escritura de la imagen tardan 0.04854s y 0.01180s respectivamente. La escritura es algo tardada en relación con la lectura y negación del método HW_ZYBO, mientras que la lectura y la negación tardan 0.00390s y 0.00309s respectivamente, la escritura tarda 0.01758s en promedio. Todo lo anterior puede ser atribuido a los medios de almacenamiento empleados en cada uno de los métodos.

Para marcar más los tiempos relativos de cada tarea con respecto al tiempo total de cada método se muestra en la Figura **4-4** que porcentaje de tiempo tarda cada tarea con respecto al tiempo total de la ejecución de cada método.

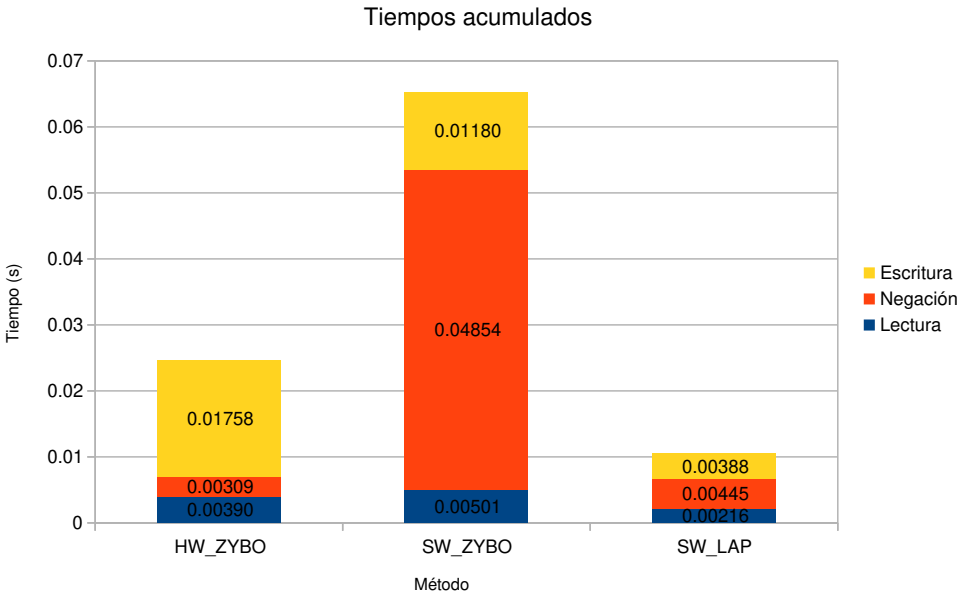


Figura 4-3.: Tiempos acumulados de la negación.

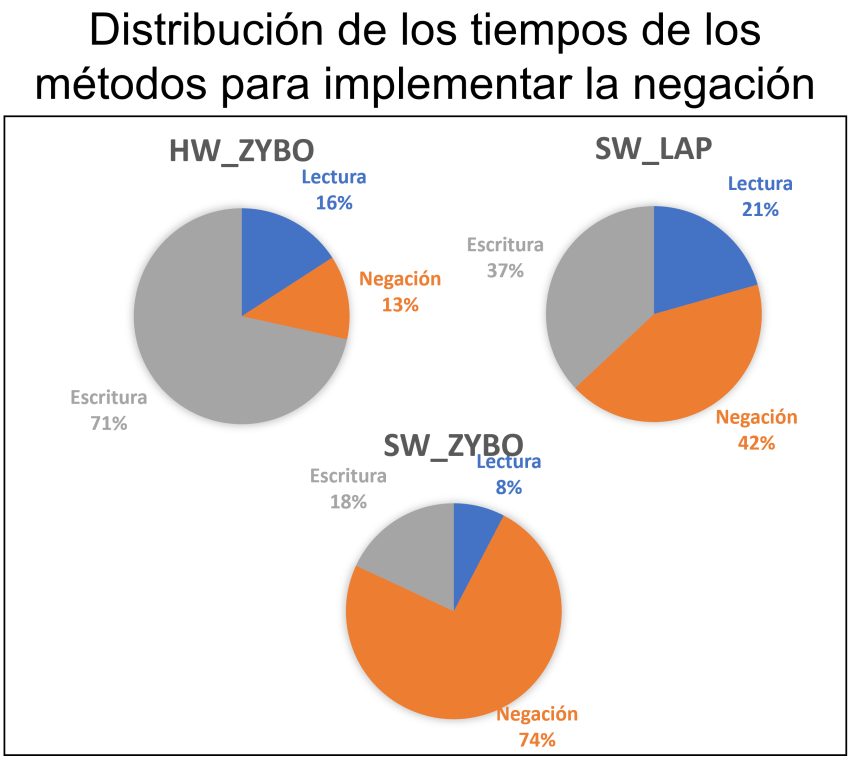
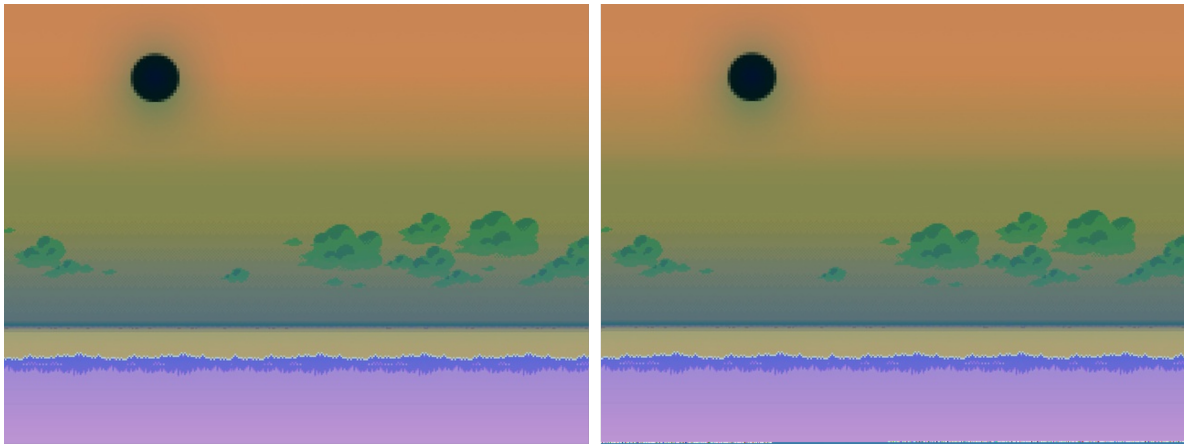


Figura 4-4.: Tiempos acumulados relativos de los métodos para lograr la negación.

El método donde relativamente se ocupa más tiempo para realizar la negación es SW_ZYBO con un 74.27 % del tiempo total, mientras que el más rápido relativamente es usando HW_ZYBO el cual utiliza el 12.57 % del tiempo total que tarda toda la ejecución. Desafortunadamente el proceso de la escritura de la imagen en el método HW_ZYBO es de casi el 73 % del tiempo total y eso que se ha optimizado lo más posible. Esto último puede estar sucediendo por la unidad de almacenamiento utilizada (microSD) lo cual representa un cuello de botella.

Para probar la negación en los distintos métodos se utilizó una imagen de entrada como la que se muestra en la Figura **4-5**, dando como resultado la imagen de la izquierda de la Figura **4-6** por parte de los métodos implementados solo mediante software. A la derecha de la Figura **4-6** se tiene la imagen resultante de la operación de la negación implementada en hardware.

Figura 4-5.: Imagen de prueba para aplicar la negación.



(a) Imagen resultante usando solo software.

(b) Imagen resultante usando el FPGA como acelerador.

Figura 4-6.: Comparación de las imágenes resultantes de los métodos para obtener la negación

La imagen resultante de la implementación de método HW_ZYBO es diferente a la obtenida por software solo en la parte inferior de la imagen, esto se debe a que con este método no se generaron pixeles en los extremos de la imagen resultante pero los espacios de los pixeles de los extremos fueron ocupados por ruido al final de la imagen.

4.2. Resultados de la operación de la convolución 2D

Los resultados que se mostrarán a continuación son los correspondientes a la operación de la convolución 2D en el caso de estudio de la aplicación de un filtro de la media con un kernel de convolución de 3x3. Cabe mencionar que la convolución 2D se implementó a cada uno de los tres canales de la imagen de entrada.

Se implementaron tres métodos, dos de ellos implementados puramente en software y uno mediante software y hardware (FPGA), los métodos fueron nombrados como se muestra a continuación:

- C2D_SW_ZYBO: Este método implementó la convolución 2D mediante software ejecutado en el CPU ARM de la tarjeta ZYBO.
- C2D_SW_LAP: Este método también se implementó con el mismo software que el caso anterior. La diferencia es que este se ejecutó en una laptop.

- C2D_HW_ZYBO: Este se implementó mediante hardware y software en la tarjeta ZYBO.

Se ejecutó cada una de las tres implementaciones un total de 30 veces con arranques en caliente con una imagen de tamaño de 640x480 pixeles RGB. En la Figura **4-7** se muestran los tiempos promedio de cada implementación ejecutada separada en tiempo total, lectura de la imagen, la convolución y la escritura de la imagen resultante.

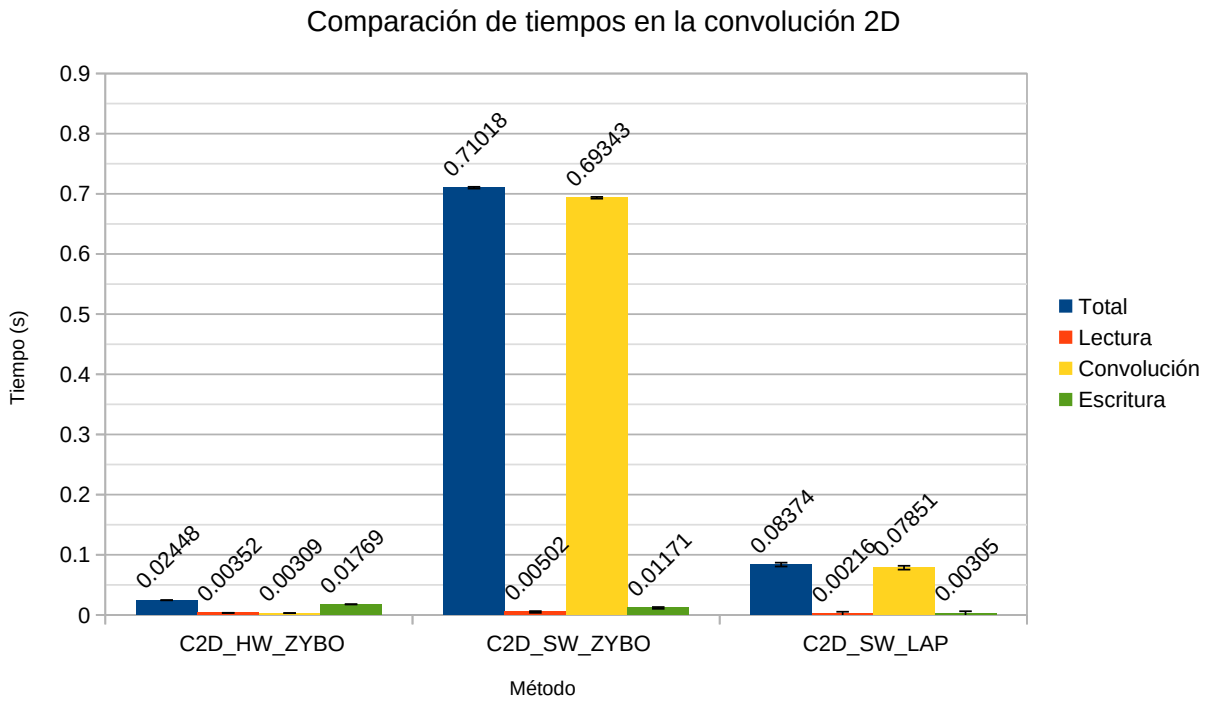


Figura 4-7.: Tiempos promedios de las tres implementaciones para la convolución 2D.

Como se puede ver, el tiempo total que demora la implementación por el método C2D_SW_ZYBO es de 0.71018s el cual es muy grande, tanto que hasta impide notar los detalles de los tiempos de los otros dos métodos que son más del interés de esta investigación. Es por ello que en la Figura **4-8** se muestran solo las dos implementaciones más rápidas.

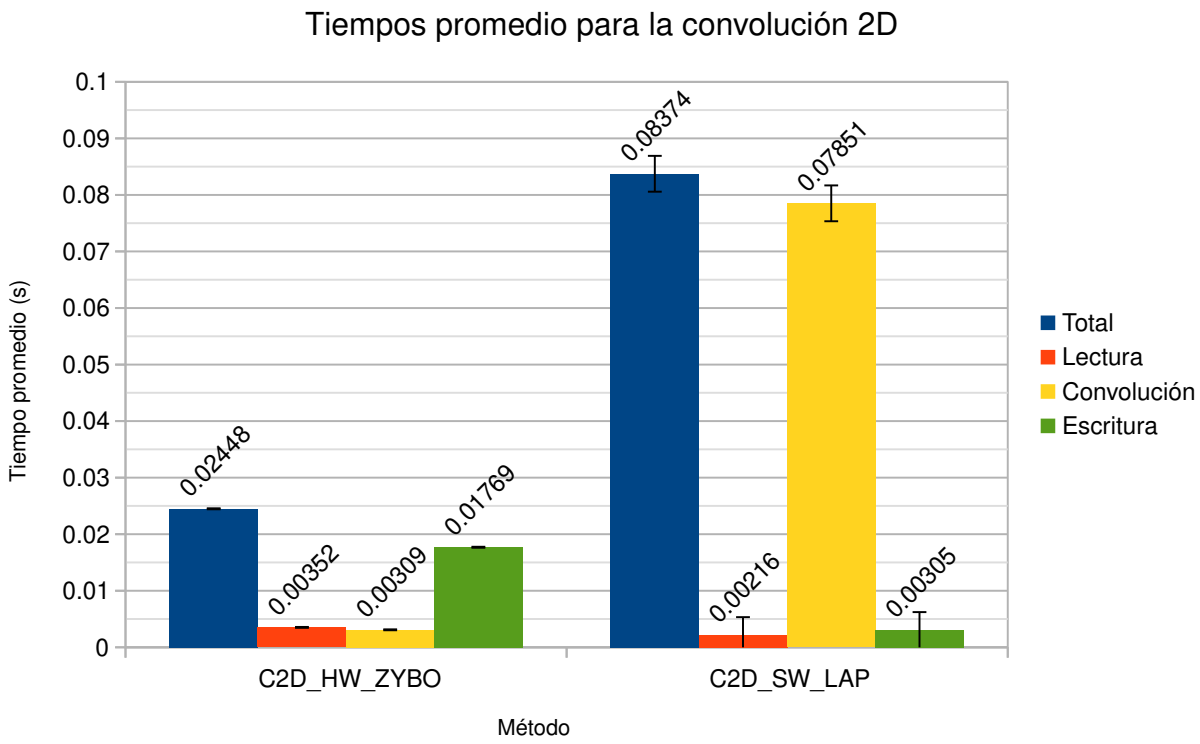


Figura 4-8.: Tiempos promedios para la convolución 2D en dos métodos.

Es posible notar en la Figura **4-8** que la ejecución del método C2D_HW_ZYBO ejecutado en 0.02448s supera más de tres veces la velocidad de procesamiento comparando con la implementación por el método C2D_SW_LAP que tarda 0.08374s en total. Y si se presta atención, la operación de la convolución es poco mas de 25 veces más rápido en hardware que en software, 0.00309s contra 0.07851s respectivamente. Lo anterior se puede atribuir a la posibilidad de paralelizar muchas operaciones cuando se utiliza el FPGA mientras que en un CPU solo se ejecuta una operación en cada núcleo a la vez. Ambos métodos pueden optimizarse, en el FPGA se puede disponer de más recursos para aumentar la paralelización y utilizar bloques de memoria RAM de cuatro canales, en el caso de las implementaciones por software es posible paralelizar la operación de la convolución y utilizar todos los núcleos del CPU.

Si se quisiera utilizar la arquitectura híbrida en un sistema de visión artificial, donde no es del todo necesario almacenar las imágenes resultantes, la Figura **4-9** muestra cómo es que tomaría aun más ventaja la arquitectura híbrida sobre una arquitectura basada solo en CPU.

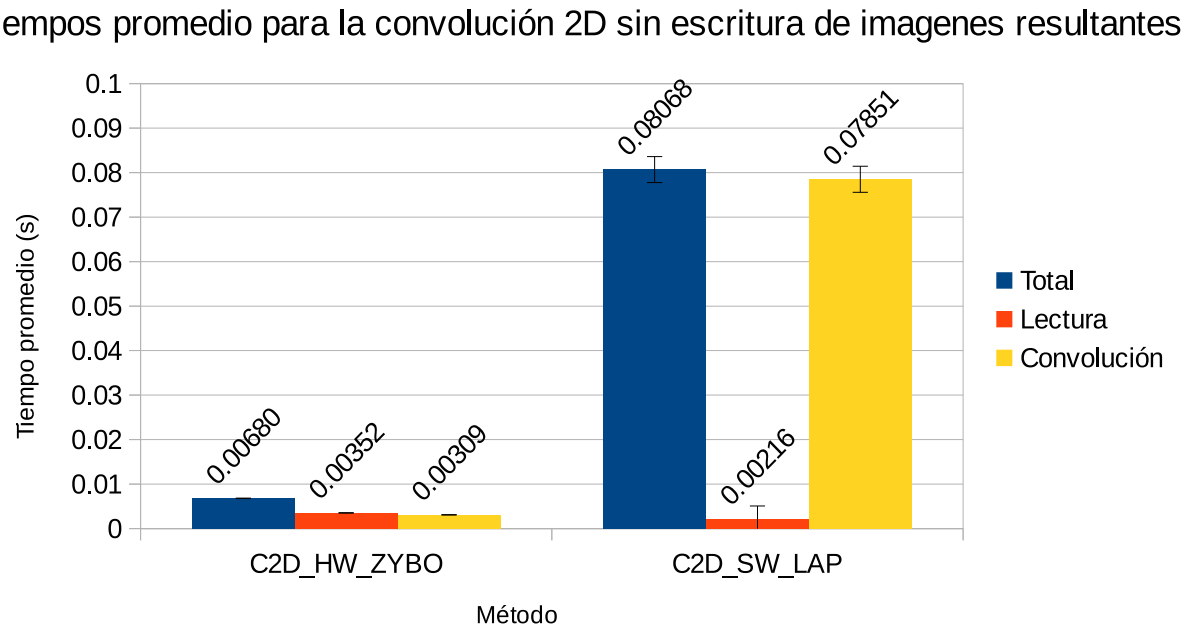


Figura 4-9.: Tiempos promedios para la convolución 2D en dos métodos sin la escritura.

Se puede ver en la Figura **4-9** que el tiempo total utilizando el método C2D_HW_ZYBO es de 0.0067969s, mientras que el tiempo total con el método C2D_SW_LAP es de 0.080681767s, casi 12 veces mas lento. Es aquí donde se puede aprovechar la arquitectura híbrida.

En la Figura **4-10** es posible ver las desviaciones estándar de la implementación de cada método.

Al igual que con la operación de la negación, la implementación por el método C2D_HW_ZYBO resulta ser la que tiene menos desviación estándar en total y la que tiene mayor es la implementada por el método C2D_SW_LAP, 0.0845 milisegundos (ms) contra 3.16695ms respectivamente. Esto se debe a que el CPU tiene que atender varios procesos (principalmente relacionados al sistema operativo), mientras que el FPGA ejecuta el proceso de forma continua sin interrupciones.

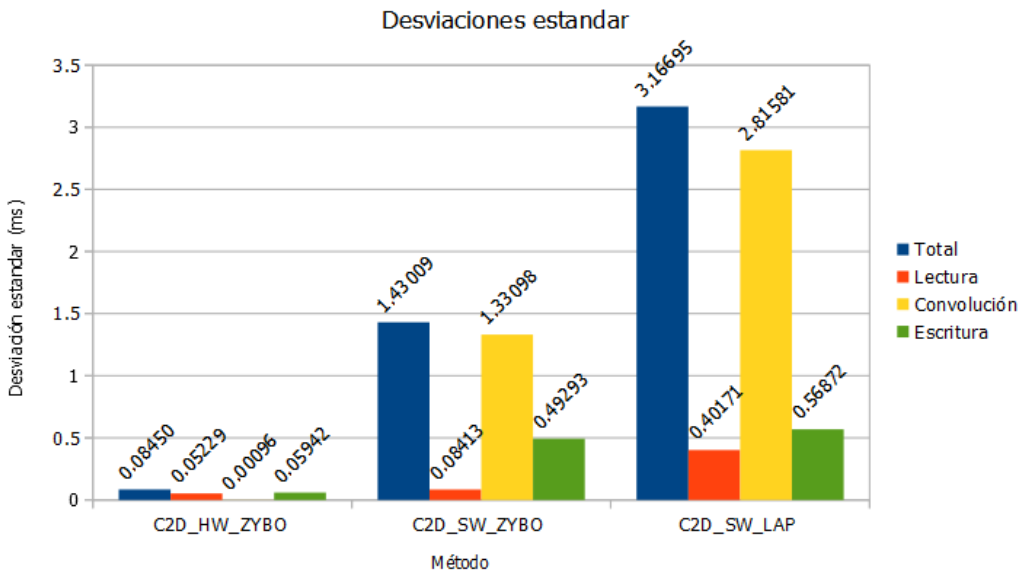


Figura 4-10.: Desviación estándar del tiempo necesario para calcular la convolución 2D.

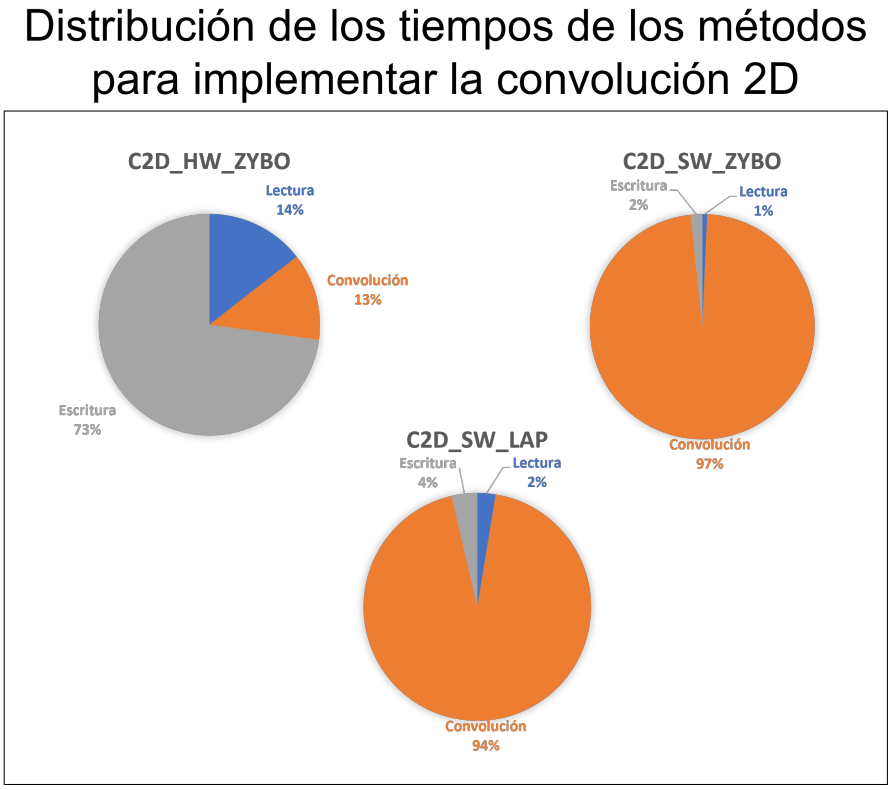


Figura 4-11.: Tiempos acumulados de los métodos para realizar la convolución 2D.

También, como puede verse en la Figura **4-10**, el método C2D_SW_ZYBO tiene una desviación estándar total 1.43009ms, menor que la del método C2D_SW_LAP. Esto es debido a que el sistema operativo utilizado en el ZYBO es más sencillo que el utilizado en la laptop con la que se comparó.

En la Figura **4-11** se muestra cómo se distribuye el tiempo de cada proceso en el tiempo total de la ejecución del algoritmo de la convolución de cada método.

Se puede ver que en la implementaciones por software el tiempo que se usa en el proceso de la convolución es de más del 90 % mientras que el la implementación por hardware solo se utiliza cerca del 13 % del tiempo total de la implementación en el proceso de la convolución. En esta parte es donde se pueden apreciar las ventajas de utilizar un FPGA para realizar operaciones en paralelo.

La imagen utilizada para las pruebas fue una que contuviera ruido de tipo gaussiano y además fuera RGB, esto para poder notar la efectividad del caso de estudio (filtro de la media). En la Figura **4-12** se muestra la imagen de entrada para la convolución 2D.

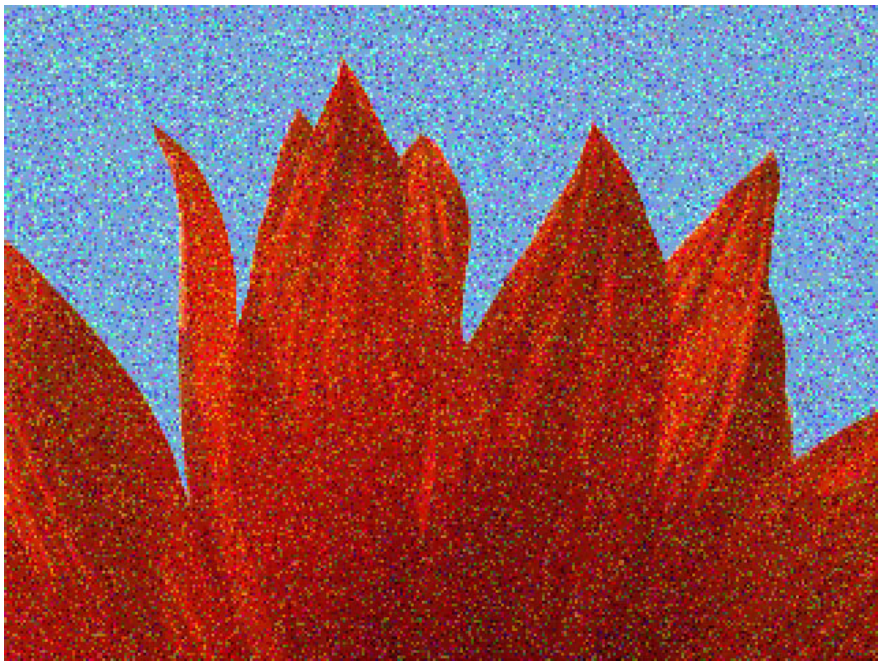
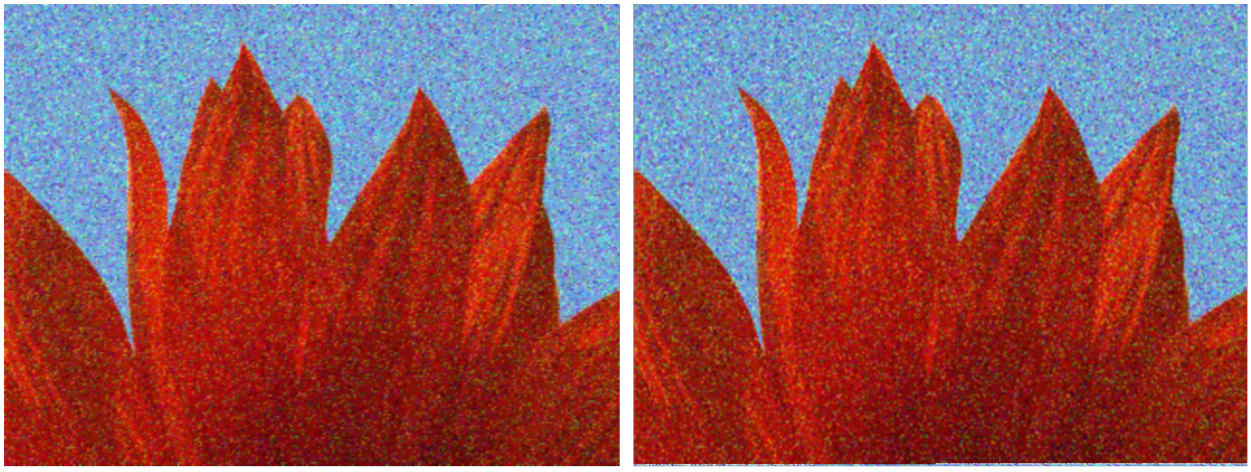


Figura 4-12.: Imagen de entrada para aplicar la convolución 2D.

En la Figura **4-13** se muestra como son las imágenes resultantes del proceso de convolución por medio se software (izquierda) y por medio de hardware (derecha).



(a) Imagen resultante usando solo software.

(b) Imagen resultante usando el FPGA como acelerador.

Figura 4-13.: Comparación de las imágenes resultantes de los métodos para obtener la convolución 2D.

En la Figura **4-13** se puede notar que si se aplicó de manera satisfactoria el filtrado digital con el filtro de la media y los resultados son los esperados para este proceso, tanto en la implementación por software como por hardware se ve como es que el ruido gaussiano se suaviza. También, se cree que la imagen resultante podría verse mejor aumentando el tamaño del kernel de convolución.

Al igual que lo ocurrido con las imágenes resultantes de la implementación de la negación HW_ZYBO, en la imagen resultante de la convolución 2D por hardware, usando el método C2D HW_ZYBO, se tiene una imagen resultante con unos pixeles de ruido al final de la imagen, esto se debe a que por hardware no se procesan los pixeles de los extremos y los espacios de esos pixeles se trasladan al final de la imagen en forma de ruido.

Por ultimo, es importante mencionar que aunque los resultados de los métodos por hardware y software son ligeramente distintos no les quita el hecho de que son correctos. En la implementación por hardware no se intentaron obtener resultados exactamente iguales a los obtenidos por software, la idea es lograr ejecutar el algoritmo de la negación y la convolución 2D. Incluso las implementaciones por software pueden ser distintas en diferentes implementaciones tan solo cambiando el tipo de dato que de los coeficientes del kernel de convolución, haciendo redondeos distintos de valores en cálculos de diferentes implementaciones o trabajando distinto los valores valores de los pixeles de los extremos de la imagen de entrada.

5. Conclusiones y discusión

El principal objetivo de este trabajo de investigación fue el crear una plataforma de aceleración de algoritmos para el procesamiento de imágenes RGB haciendo uso de una arquitectura híbrida FPGA-CPU. Con base en los resultados experimentales obtenidos se pudo corroborar que la arquitectura híbrida permitió acelerar por hardware los algoritmos de procesamiento de imágenes, debido a que el FPGA se encargó de las operaciones aritméticas que demandaban más tiempo, logrando obtener una velocidad de procesamiento superior al obtenido usando solo CPU. Con esto se puede decir que se logró completar de forma más que satisfactoria el objetivo principal, logrando hacer ver las ventajas de la arquitectura híbrida como el gran poder de cómputo requerido para llevar cabo las aceleraciones realizadas.

El principal caso de estudio de esta investigación fue la aceleración de convoluciones 2D en un experimento de filtrado digital con un filtro de la media. Dicho algoritmo logro generar resultados similares a los obtenidos mediante implementaciones por software, pero con la diferencia que mediante hardware se obtuvieron tiempos de ejecución tres veces menores que la implementación en una arquitectura de CPU tradicional usando solo un núcleo del mismo. Lo anterior es en cuanto al tiempo total de la ejecución de un experimento, pero si se toma en cuenta solo el tiempo en que se demora la arquitectura en ejecutar el algoritmo de la convolución 2D se obtiene que la arquitectura donde solo se usa software en una laptop tarda más de 25 veces lo que demora la arquitectura híbrida.

En base en los resultados experimentales, se determinó que la arquitectura híbrida es lenta en la lectura de la imagen de entrada y, aun más, en la escritura de la imagen de salida, hasta alrededor del 73 % del tiempo total de ejecución. Este cuello de botella fácilmente puede ser atribuido al medio de almacenamiento que utiliza la plataforma híbrida que es una microSD y, aunque sea clase 10, no tiene comparación con discos de estado sólido instalados en la plataforma con la que se comparó (laptop personal).

De acuerdo con los resultados, el tiempo que se tarda la arquitectura híbrida en escribir la imagen resultante es casi 6 veces mayor que la escritura de la imagen resultante en un SSD en la arquitectura basada en CPU. Lo anterior nos indica que si cambiamos el medio de almacenamiento en la plataforma híbrida en teoría se pueden obtener mejores resultados en los algoritmos implementados en esta investigación, sin embargo, la plata-

forma ZYBO no tiene forma física dedicada para otro tipo de disco.

Otro punto por comentar es que la aceleración de los algoritmos implementados se hizo con un consumo menor a los 10 Watts, que es el consumo máximo suministrado a la plataforma ZYBO donde se hicieron todos los experimentos. El consumo es tan reducido que ésta arquitectura resulta ideal para una gran cantidad de proyectos donde sea necesaria una capacidad de cómputo alta y un consumo energético bajo como lo serian aplicaciones de seguridad e inspección de productos en tiempo real.

5.1. Trabajo a futuro

Dado a que los resultados obtenidos con esta plataforma son satisfactorios de acuerdo a las comparaciones hechas, se plantea a futuro implementar una serie de algoritmos como lo son la ejecución de redes neuronales pequeñas, algunos algoritmos para el procesamiento de series de tiempo y podría ser el mejorar y generalizar una plataforma para el procesar imágenes reduciendo los cuellos de botella ocasionados por la lectura y escritura en disco.

Como se mencionó en algún momento, el ZYBO trae implementado uno de los SoC más básicos de la familia ZYNQ, con esto se abre la posibilidad de utilizar plataformas modernas y con más recursos como por ejemplo las tarjetas ZYNQ UltraScale+, esto para poder tener libertad de implementar algoritmos que necesiten más recursos lógicos y con elevado consumo de memoria.

A. Anexo: Compilación y empaquetado de sistema operativo

A continuación, se describe el proceso para la compilación y empaquetado del sistema operativo usando las herramientas de Petalinux que se arrancará en el ZYBO.

1. Se exporta el hardware diseñado en Vivado incluyendo el archivo binario de reconfiguración, también llamado en Vivado como bitstream.

2. Se ejecuta el Shell Script llamado “settings.sh”, éste está ubicado en la carpeta de instalación de Petalinux. Este paso se encarga de instanciar las herramientas necesarias para la creación del sistema operativo.
   - . /Dir_petalinux/settings.sh

3. Se crea un nuevo proyecto especificando el nombre y la plataforma a la que se dirige, en este caso ZYNQ.
   - petalinux-create –type project –template zynq –name HardwareBase
   - El nombre del Proyecto es “HardwareBase” en este caso.

4. Nos dirigimos al directorio del proyecto que se acaba de crear.
   - cd HardwareBase

5. Cuando estamos en la carpeta del proyecto que se acaba de generar, se prosigue a importar el hardware anteriormente generado con Vivado. Desde este punto ya se podrá personalizar el sistema operativo de acuerdo con nuestras necesidades. Es decir, se puede editar tanto el sistema de archivos del sistema operativo como el kernel de Linux.
   - petalinux-config –get-hw-description /Dir_proyecto_hardware/ Project_HW/ Project_HW.sdk/

   Al terminar de ejecutar el comando anterior se desplegará una ventana de configuración igual a la de la Figura **A-1**

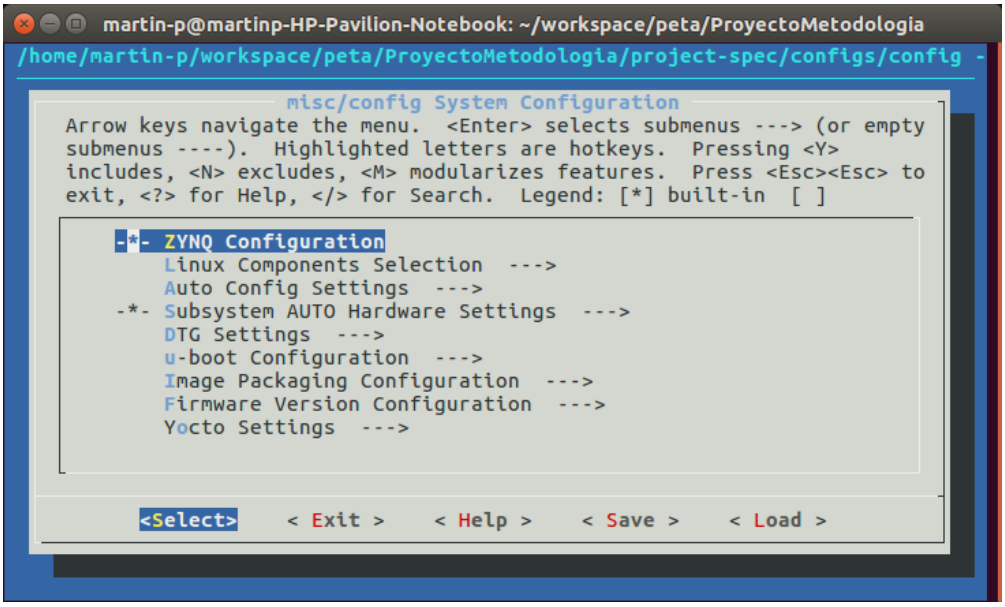


Figura A-1.: Ventana inicial de configuración de Petalinux.

6. En el menú desplegado es necesario configurar que el sistema de archivos se escriba sobre la microSD de la tarjeta, para ello se hace lo que se muestra en las Figuras **A-2** y **A-3**.

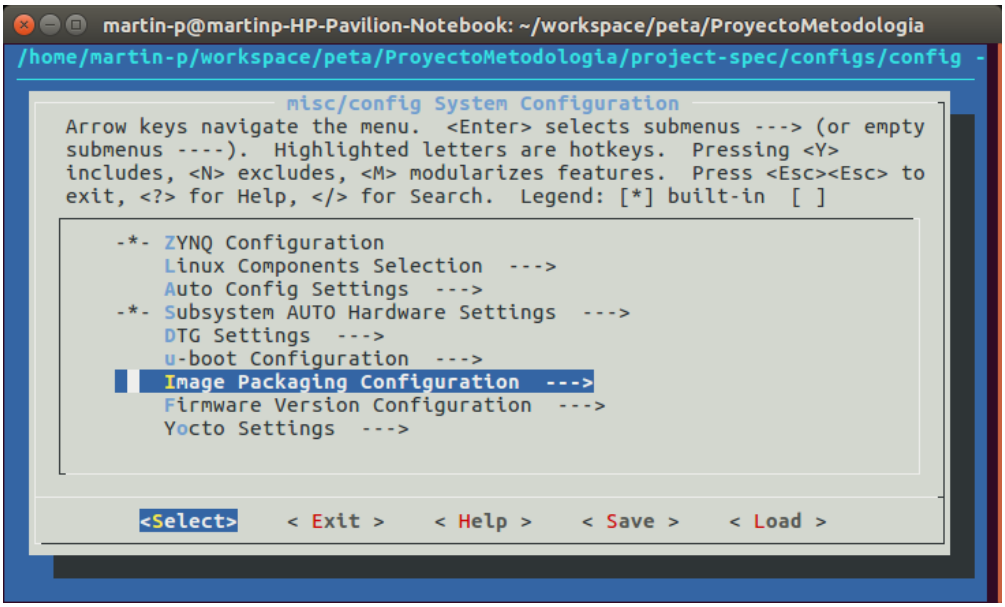


Figura A-2.: Ventana de selección de opciones de empaquetado.

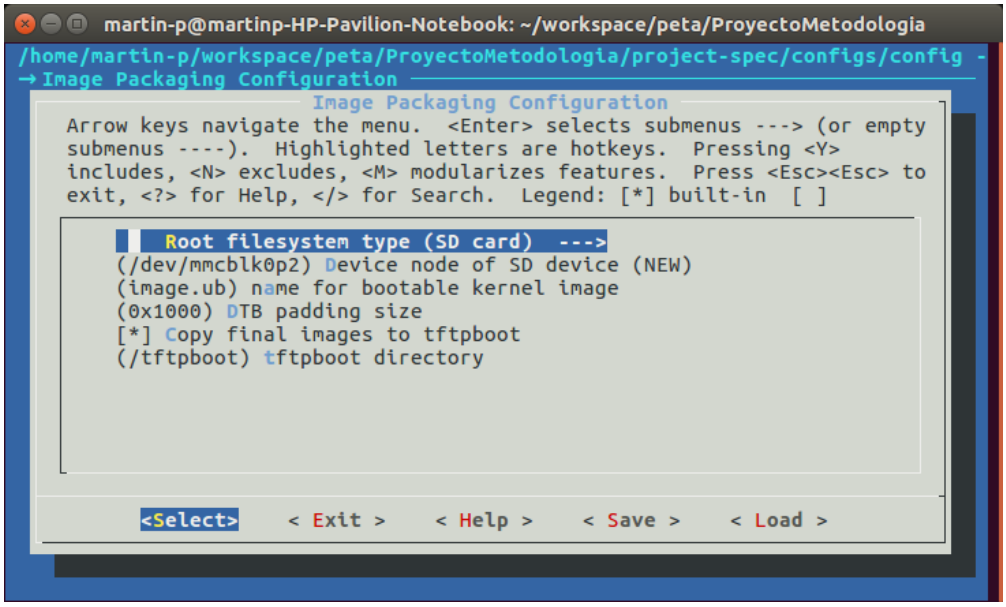


Figura A-3.: Selección del sistema de archivos sobre la microSD.

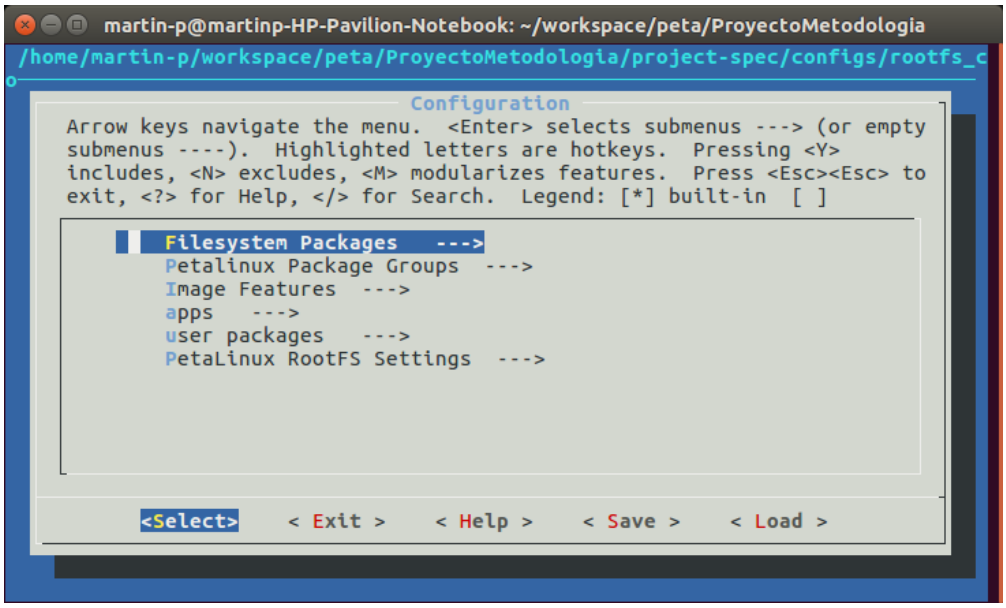


Figura A-4.: Configuración del sistema de archivos.

7. Una vez realizado lo anterior, nos salimos de las ventanas de configuración guardando todos los cambios.

8. Estando de nuevo en el terminal se accede a otra ventana de configuración, esta ventana da la posibilidad de editar el sistema de archivos, se añade todo lo que se

necesita que contenga el sistema operativo (bibliotecas, programas y compiladores).

- petalinux-config -c rootfs

Se desplegará una ventana como la que se muestra en la Figura **A-4**, desde esta ventana será posible añadir todo lo que se quiera en el sistema de archivos.

9. Para poder incluir algún programa extra como, por ejemplo, el editor de texto “nano”, es necesario modificar un archivo de texto llamado “petalinux-image-full.bbappend” agregando la siguiente línea.

   - IMAGE_INSTALL_append = “ nano”

   Es importante dejar el espacio después de las primeras comillas. La Figura **A-5** ilustra este procedimiento.

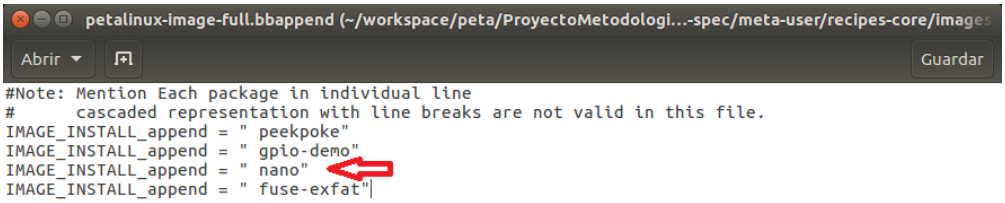


Figura A-5.: Modificación de archivo de configuración para incluir ”nano”.

10. Una vez hecho lo anterior, se necesita repetir el paso 6 para especificar que se quiere añadir la aplicación “nano”.

    - petalinux-config -c rootfs.
    - se añade nano desde el menú.
    - Se sale del menú guardando los cambios.

11. Es posible modificar las configuraciones del kernel agregando ciertos controladores de nuestro interés, en este caso drivers para tener acceso al puerto USB con el que cuenta la plataforma usada.

    - petalinux-config -c kernel.
    - Se seleccionan los cambios a incluir (Drivers USB en este caso).
    - Se sale del menú guardando los cambios.

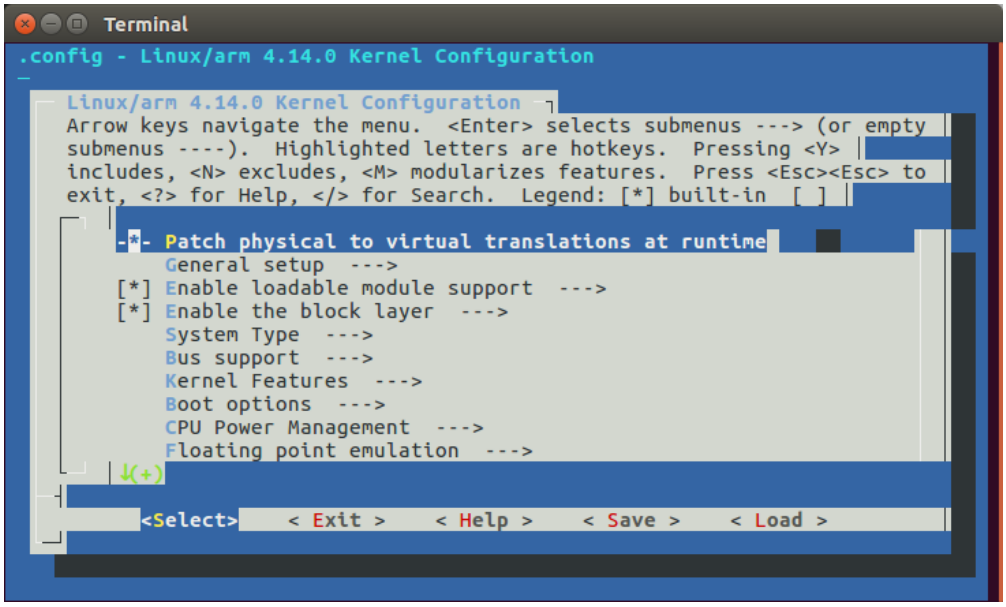


Figura A-6.: Ventana de configuración del kernel.

En la Figura **A-6** se puede visualizar la ventana de configuración del kernel que se despliega al querer modificar dicho kernel.

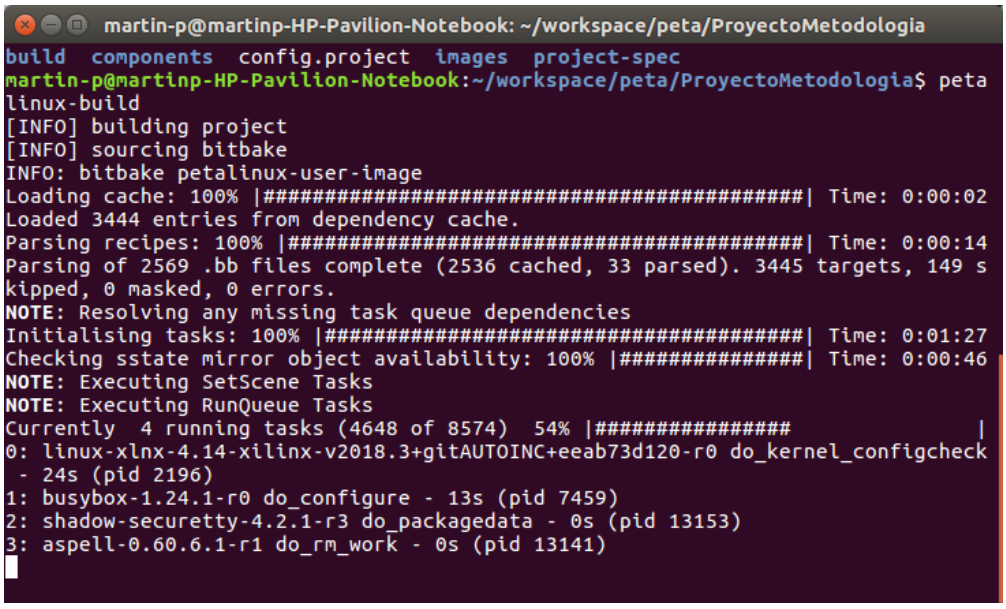


Figura A-7.: Ejecución de la compilación del sistema operativo.

12. Una vez se tengan todas las configuraciones hechas, se compila el sistema con la siguiente instrucción.

- petalinux-build

Este proceso por lo general es tardado y mientras dicho proceso se completa se puede ver la terminal como se muestra en la Figura **A-7**.

13. Cuando este proceso termina es necesario empaquetar el sistema operativo para poder arrancarlo desde la microSD. Se utiliza el siguiente comando.

    - petalinux-package –boot –fsbl images/linux/zynq_fsbl.elf –fpga images/linux/system.bit –u-boot

    Terminando el proceso ya se puede decir que se tiene el sistema operativo listo para ser copiado a una microSD.

B. Anexo: Creación de microSD de arranque

En esta sección se desplegarán una serie de instrucciones necesarias para la creación de una microSD de arranque para el sistema operativo construido con la herramientas de Petalinux para la plataforma Zybo.

1. Se formatea la microSD que se utilice.
   - Una partición de por lo menos 500 MB (puede ser mayor) en formato fat32. Esta partición se podría llamar “BOOT”.
   - Una segunda partición en formato ext4 del tamaño restante con respecto a la microSD utilizada. A esta partición se le llamará “rootfs”
   - Se recomienda usar una microSD de alta velocidad de transferencia de datos (clase 10 al menos).
   - El formateo puede hacerse con la herramienta “Gparted” incluida en Ubuntu 16.04.
2. Para instalar el sistema operativo en la microSD es necesario hacer los siguientes pasos.
   - Borrar lo que esté contenido en las particiones.
     - sudo rm -rf /media/usuario/rootfs/
     - sudo rm -rf /media/usuario/BOOT/
   - Copiar algunos de los archivos generados en el Anexo A a la microSD.
     - sudo cp images/linux/rootfs.cpio /media/usuario/rootfs/
     - sudo cp images/linux/BOOT.BIN /media/usuario/BOOT/
     - sudo cp images/linux/image.ub /media/usuario/BOOT/
   - Extraer el sistema de archivos en la partición “rootfs”.
     - sudo tar xvf images/linux/rootfs.tar.gz -C /media/usuario/rootfs/

3. Con todo lo anterior ya es posible extraer la microSD. Para poder arrancar el sistema operativo es necesario configurar físicamente la tarjeta ZYBO como se muestra en la Figura **B-1**, poniendo el puente azul en los pines que indican SD.

Figura B-1.: Configuración de arranque en ZYBO.

Concluidos los pasos anteriores, se procede con encender el ZYBO y que éste arranque el sistema operativo desde la microSD. Hay que tener en cuenta que el sistema operativo inicialmente arrancará con una emulación de SSH a través del puerto UART que tiene la tarjeta. Dicho lo anterior, es necesario conectar el ZYBO a una computadora mediante un cable USB-microUSB con una comunicación serial configurada a 115200 bps.

Para realizar la conexión pudiera utilizarse el software Putty y conectarse haciendo las configuraciones mencionadas. Para acceder al sistema operativo es necesario acceder con el usuario “root” y por defecto la contraseña es ”root”. En la Figura **B-2** se muestra la terminal del sistema operativo base.

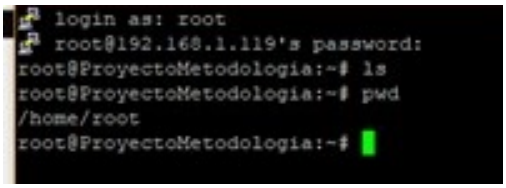


Figura B-2.: Terminal del sistema operativo base accedido desde Putty.

Una vez ubicado en la terminal del sistema operativo, pudiera consultarse la IP del ZYBO (si es que este se conecta a la red local) para después acceder al sistema desde SSH.

Bibliografía

[1] H. Z. Abidin, M. Kassim, K. A. Othman, y M. Samad, "Incorporating vhdl in teaching combinational logic circuit," en *2010 2nd International Congress on Engineering Education*. IEEE, 2010, pp. 225–228. Disponible en: https://doi.org/10.1109/ICEED.2010.5940796

[2] Advanced Micro Devices Inc., "Procesadores para computadoras de escritorio AMD Ryzen™," 2022. Disponible en: https://www.amd.com/es/processors/ryzen

[3] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I. A. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S. C. Liu, y T. Delbruck, "NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 3, pp. 644–656, 2019. Disponible en: https://doi.org/10.1109/TNNLS.2018.2852335

[4] S. O. Aletan, "An overview of risc architecture," en *Proceedings of the 1992 ACM/-SIGAPP Symposium on Applied computing: technological challenges of the 1990's*, 1992, pp. 11–20.

[5] Arm Limited, "AMBA," 2022. Disponible en: https://developer.arm.com/Architectures/AMBA

[6] W. Balid y M. Abdulwahed, "A novel fpga educational paradigm using the next generation programming languages case of an embedded fpga system course," *IEEE Global Engineering Education Conference, EDUCON*, pp. 23–31, 2013. Disponible en: https://doi.org/10.1109/EduCon.2013.6530082

[7] L. Crockett, R. Elliot, M. Enderwitz, y R. Stewart, *The Zynq Book: Embedded Processing Withe ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC*. Strathclyde Academic Media, 2014. Disponible en: https://is.muni.cz/el/1433/jaro2015/PV191/um/The_Zynq_Book_ebook.pdf

[8] Digilent, *ZYBO™ FPGA Board Reference Manual*, Digilent. Disponible en: https://digilent.com/reference/_media/reference/programmable-logic/zybo/zybo_rm.pdf

[9] S. Du, T. Huang, J. Hou, S. Song, y Y. Song, “Fpga based acceleration of game theory algorithm in edge computing for autonomous driving,” *Journal of Systems Architecture*, vol. 93, pp. 33–39, 2 2019. Disponible en: https://doi.org/10.1016/j.sysarc.2018.12.009

[10] H. El-aawar, “Cisc vs . risc hardware and programming complexity measures of addressing modes,” pp. 43–48, 2006. Disponible en: https://doi.org/10.1109/MEMSTECH.2006.288660

[11] X. Feng, Y. Jiang, X. Yang, M. Du, y X. Li, “Computer vision algorithms and hardware implementations: A survey,” *Integration*, vol. 69, no. July, pp. 309–320, 2019. Disponible en: https://doi.org/10.1016/j.vlsi.2019.07.005

[12] H. Gao, H. Dai, y Y. Zeng, “High-speed image processing and data transmission based on vivado hls and axi4-stream interface *,” *2018 IEEE International Conference on Information and Automation (ICIA)*, vol. 201510, pp. 575–579, 2018. Disponible en: https://doi.org/10.1109/ICInfA.2018.8812379

[13] A. Goel, M. O. Ahmad, y M. N. Swamy, “Design of a 2d median filter with a high throughput fpga implementation,” *Midwest Symposium on Circuits and Systems*, vol. 2019-Augus, pp. 1073–1076, 2019. Disponible en: https://doi.org/10.1109/MWSCAS.2019.8885009

[14] S. Govindarajan, K. Chitnis, M. Mody, G. Shurtz, S. Shivalingappa, y T. Kim, “Flexible and Efficient sharing of High Performance Hardware Accelerators in a Safe , Secure , Virtualized System,” *IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*, vol. 2020, 2020. Disponible en: https://doi.org/10.1109/ICCE-Asia49877.2020.9277328

[15] A. HajiRassouliha, A. J. Taberner, M. P. Nash, y P. M. Nielsen, “Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms,” *Signal Processing: Image Communication*, vol. 68, no. July, pp. 101–119, 2018. Disponible en: https://doi.org/10.1016/j.image.2018.07.007

[16] V. Hecht, K. Ronner, y P. Pirsch, “An Advanced Programmable 2D-Convolution Chip for Real Time Image Processing,” *IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 1991, pp. 1897–1900, 1991. Disponible en: https://doi.org/10.1109/ISCAS.1991.176778

[17] Intel Corporation, “Procesadores Intel® Core™ i7 de 12 Generación,” 2022. Disponible en: https://ark.intel.com/content/www/es/es/ark/products/series/217837/12th-generation-intel-core-i7-processors.html

[18] J. A. Jara y D. G. Maxinez, *VHDL El Arte de Programar Sistemas Digitales (Spanish Edition)*. Cecsa, 2003.

[19] M. A. Kadi, P. Rudolph, D. Göhringer, y M. Hübner, "Dynamic and partial reconfiguration of Zynq 7000 under Linux," *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, vol. 2013, 2013. Disponible en: https://doi.org/10.1109/ReConFig.2013.6732279

[20] P. A. Khan y R. S. Mishra, "Comparative analysis of different algorithm for design of high-speed Multiplier Accumulator unit (MAC)," *Indian Journal of Science and Technology*, vol. 9, no. 8, 2016. Disponible en: https://doi.org/10.17485/ijst/2016/v9i8/83614

[21] U. Kodwani, S. Rajurkar, y S. G. Mundada, "Realization of sequential circuit using finite state machine," en *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, 2017, pp. 472–476. Disponible en: https://doi.org/10.1109/ICCONS.2017.8250767

[22] M. Kowalczyk, T. Kryjak, y M. Gorgon, "Object tracking with the use of a moving camera implemented in heterogeneous zynq soc-a demo," *Conference on Design and Architectures for Signal and Image Processing, DASIP*, vol. 2017-Septe, pp. 1–2, 2017. Disponible en: https://doi.org/10.1109/DASIP.2017.8122123

[23] I. Kuon, R. Tessier, y J. Rose, *FPGA architecture: Survey and challenges*, 2007, vol. 2. Disponible en: https://doi.org/10.1561/1000000005

[24] G. Kyrtsakas y R. Muscedere, "An fpga implementation of a custom jpeg image decoder soc module," *Canadian Conference on Electrical and Computer Engineering*, pp. 30–33, 2017. Disponible en: https://doi.org/10.1109/CCECE.2017.7946826

[25] S. Li, Z. Zhang, F. Du, y Y. He, "A new automatic real-time crop row recognition based on soc-fpga," *IEEE Access*, vol. 8, pp. 37 440–37 452, 2020. Disponible en: https://doi.org/10.1109/ACCESS.2020.2973756

[26] B. C. Maheshwari, J. Burns, M. Blott, y G. Gambardella, "Implementation of a scalable real time canny edge detector on programmable soc," *2017 International Conference on Electrical and Computing Technologies and Applications, ICECTA 2017*, vol. 2018-Janua, pp. 1–5, 2017. Disponible en: https://doi.org/10.1109/ICECTA.2017.8251977

[27] S. H. Malik, "Comparative Study of Digital Image Enhancement Approaches," 2014, pp. 3–7. Disponible en: https://doi.org/10.1109/ICCCI.2014.6921749

[28] S. McBader y P. Lee, “An fpga implementation of a flexible, parallel image processing architecture suitable for embedded vision systems,” en *Proceedings International Parallel and Distributed Processing Symposium*, 2003, pp. 5 pp.–. Disponible en: https://doi.org/10.1109/IPDPS.2003.1213415

[29] MediaTek, “MediaTek Dimensity 1200,” 2022. Disponible en: https://www.mediatek.com/products/smartphones-2/mediatek-dimensity-1200

[30] R. A. Melo, B. Valinoti, M. B. Amador, L. G. Garcia, A. Cicuttin, y M. L. Crespo, “Study of the data exchange between programmable logic and processor system of zynq-7000 devices,” *2019 10th Southern Conference on Programmable Logic, SPL 2019 - Proceedings*, pp. 3–8, 2019. Disponible en: https://doi.org/10.1109/SPL.2019.8714328

[31] R. F. Molanes, L. Costas, J. J. Rodriguez-Andina, y J. Farina, “Comparative analysis of processor-fpga communication performance in low-cost fpsocs,” *IEEE Transactions on Industrial Informatics*, vol. 3203, pp. 1–1, 2020. Disponible en: https://doi.org/10.1109/tii.2020.3015833

[32] J. Morley, K. Widdicks, y M. Hazas, “Digitalisation, energy and data demand: The impact of internet traffic on overall and peak electricity consumption,” *Energy Research and Social Science*, vol. 38, pp. 128–137, 4 2018. Disponible en: https://doi.org/10.1016/j.erss.2018.01.018

[33] C. P. Narendra y K. M. R. Kumar, “Low power compressor based mac architecture for dsp applications,” en *2015 IEEE International Conference on Signal Processing, Informatics, Communication and Energy Systems (SPICES)*, 2015, pp. 1–5. Disponible en: https://doi.org/10.1109/SPICES.2015.7091393

[34] O. Panait, L. Dumitriu, y I. Susnea, “Hardware and software architecture for accelerating hash functions based on soc,” *Proceedings - 2019 22nd International Conference on Control Systems and Computer Science, CSCS 2019*, pp. 136–139, 2019. Disponible en: https://doi.org/10.1109/CSCS.2019.00031

[35] J. Pavón, “Movimiento eficiente de datos para un SoC basado en ZYNQ,” Tesis de Maestria, Universidad de Alcalá Escuela Politécnica Superior, 2015. Disponible en: https://ebuah.uah.es/dspace/handle/10017/24163?locale-attribute=es

[36] X. Qu, S. Zhang, H. Huo, Y. Gu, y Y. Sun, “Design for a reconfigurable image fusion system base on all programmable system on chip,” *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2015*, pp. 1903–1907, 2016. Disponible en: https://doi.org/10.1109/FSKD.2015.7382238

[37] Qualcomm Technologies Inc., “Snapdragon 690 5G Mobile Platform,” 2022. Disponible en: https://www.qualcomm.com/products/application/smartphones/snapdragon-6-series-mobile-platforms/snapdragon-690-5g-mobile-platform

[38] ——, “Snapdragon 8+ Gen 1 Mobile Platform,” 2022. Disponible en: https://www.qualcomm.com/products/application/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-8-plus-gen-1-mobile-platform

[39] E. Rucci, “Evaluación de rendimiento y eficiencia energética de sistemas heterogéneos para bioinformática,” Tesis de Doctorado, Universidad Nacional de La Plata, 2016. Disponible en: https://doi.org/10.35537/10915/66533

[40] The Raspberry Pi Foundation, “Raspberry Pi 4 Tech Specs,” 2022. Disponible en: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

[41] M. A. T. Viveros, “Introducción a la creación de imágenes digitales para multimedia interactivo,” 2017. Disponible en: https://programas.cuaed.unam.mx/repositorio/moodle/pluginfile.php/1161/mod_resource/content/1/contenido/index.html

[42] Xilinx, *LogiCORE IP AXI DMA v7.1*, Xilinx. Disponible en: https://www.xilinx.com/content/dam/xilinx/support/documents/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

[43] ——, *Vivado Design Suite User Guide*, Xilinx. Disponible en: https://docs.xilinx.com/v/u/2017.2-English/ug910-vivado-getting-started

[44] ——, *PetaLinux Tools Documentation*, Xilinx, 2018. Disponible en: https://docs.xilinx.com/v/u/2018.2-English/ug1144-petalinux-tools-reference-guide

[45] ——. (2022) Petalinux tools. Disponible en: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html#tools