Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Electrónica

MAESTRÍA EN INGENIERÍA ELECTRÓNICA,
OPCIÓN INSTRUMENTACIÓN ELECTRÓNICA

TESIS PARA OBTENER EL GRADO DE
MAESTRO EN INGENIERÍA ELECTRÓNICA

IMPLEMENTACIÓN DE LAS PRINCIPALES CAPAS QUE CONSTITUYEN UNA RED NEURONAL CONVOLUCIONAL SECUENCIAL EN LÓGICA RECONFIGURABLE

PRESENTA:

OSCAR KALEB HERNÁNDEZ BADILLO*

ASESORES INTERNOS:

M.C. Nicolás Quiroz Hernández

M.C. Selene Edith Maya Rueda

ASESORES EXTERNOS:

M.I. Leonardo Delgado Toral

M.I. Juan Díaz Téllez

* Becario CONACYT

H. Puebla de Zaragoza, 1 de febrero de 2022

DEDICATORIA

A mi novia Alejandra, que me ha permitido ver la vida a través de sus ojos, me ha apoyado y ha brindado amor infinito. A mi abuelo Francisco Badillo, que desde donde se encuentre logro escuchar un ¡Bravo! lleno de alegría.

AGRADECIMIENTOS

A mis padres, Miriam Badillo Jaramillo y Rolando Hernández Pérez.
A mis hermanos Olivia Hernández Badillo y Jafet Hernández Badillo.

Agradezco al Consejo de Ciencia y Tecnología (CONACyT) y a la Benemérita Universidad Autónoma de Puebla por su apoyo y patrocinio para la realización de este proyecto de tesis.
A mis asesores M.C. Nicolás Quiroz Hernández, M.C. Selene Edith Maya Rueda, M.I. Leonardo Delgado Toral y M.I. Juan Díaz Téllez por asesorar mi trabajo de tesis y compartir su conocimiento y experiencia una vez más, misma que fue de gran ayuda para el desarrollo de este proyecto.

Resumen

Este trabajo presenta la implementación de las principales capas de una red neuronal convolucional, como es la operación de convolución, la función de activación ReLU, el operador Pooling y el perceptrón, además, se implementó una unidad de punto flotante con un formato de 24 bits para realizar las operaciones de suma, resta y multiplicación.

Los módulos fueron implementados en el IDE de desarrollo ISE DESIGN SUITE en su versión 14.6, una vez implementados y conforme se fueron haciendo más robustos, se migraron al IDE de desarrollo VIVADO en su versión 2020.2. Se hizo uso de una PC con un procesador AMD Ryzen 7 de 64 bits con 8 núcleos y con una frecuencia de 2.3 GHz para programar los scripts que arrojan los valores correctos que son comparados con los resultados de los módulos implementados para determinar la calidad de su funcionamiento.

Se implementó un módulo que realiza la suma y resta en formato de 24 bits de punto flotante y un módulo que realiza la multiplicación, es posible realizar divisiones si se representan números con exponente negativo, por ejemplo, la multiplicación de $3 \times 0{,}5$ daría como resultado 1.5; el número 0.5 es igual a 3f0000 en su representación hexadecimal y tiene como exponente -1.

La unidad de punto flotante es utilizada en la operación de convolución y en el perceptrón. Dentro de la operación de convolución se encarga de realizar el producto punto, necesario en cada desplazamiento que realiza el kernel ya que computa el resultado que es enviado a la función de activación ReLU y ésta se encarga de proporcionar el resultado final. Para realizar el desplazamiento del kernel sobre la ventana se implementó una máquina de estados que se encarga de censar la posición del kernel e incrementar la posición de acuerdo al salto que se establece en las características configurables, también, se establecen las dimensiones del kernel. En cuanto al perceptrón, se realiza para realizar el producto punto de las entradas con los pesos sinápticos propios del perceptrón, se establece una función de activación y proporciona a la salida resultados que son -1 y 1.

En general, la operación de convolución en conjunto con la unidad de punto flotante presentan un error igual a 2.452409143921587e-05, debido a la precisión obtenida con el formato de punto flotante de 24 bits, en comparación con el error que presenta el formato de 32 bits con el formato de 64 bits cuyo valor es igual a .003297962345842096e-05.

Los resultados obtenidos con los módulos implementados en comparación con los resultados obtenidos con la PC son óptimos y permiten validar su funcionamiento.

Índice general

Índice de figuras

Índice de Tablas

1.
Introducción

En años recientes, la inteligencia artificial (IA) se ha visto involucrada en una amplia variedad de aplicaciones. La IA hace posible el surgimiento de vehículos autónomos, facilita el proceso educativo, ha mejorado el diagnóstico y tratamiento de enfermedades, además, de ser utilizada en el reconocimiento de patrones y en el procesamiento de señales [1]-[5], entre otros.

En la época donde se vislumbraba el inicio de la inteligencia artificial, personajes como Lady Ada Lovelace y Charles Babbage se vieron involucrados en la invención de "La máquina analítica" (1830 a 1840); conocida como la primera computadora mecánica, misma que estaba asignada únicamente a realizar operaciones mecánicas para automatizar algunos cálculos en la rama del análisis matemático. Sin embargo, los científicos de esa época notaron que la máquina sólo desempeñaba tareas con las que estaban perfectamente familiarizados y por lo tanto sabían la forma en la que debían dar ordenes a la computadora para que ésta pudiera realizarlas. La inteligencia artificial tiene sus primeras apariciones en 1950 [4], después de que un grupo de pioneros se planteó si las computadoras podían aprender por sí solas, una pregunta cuyas respuestas aún se están explorando en la actualidad. Más tarde, Alan Turing, conocido como uno de los padres de la ciencia computacional, planteó un test conocido como "El test de Turing"[6][1].

Aunque la inteligencia artificial tiene menciones antes de 1950, ésta nace formalmente en 1956 [5] en el colegio Dartmouth, cuando John McCarthy junto a Marvin Minsky, Claude Shannon y Nathaniel Rochester reunieron a un grupo de investigadores interesados en la teoría autómata, redes neuronales y el estudio de la inteligencia. Los personajes antes mencionados llevaron a cabo un taller que propuso el estudio de las bases sobre las que se establecen los aspectos del aprendizaje y que algunas características de la inteligencia pueden ser descritas de forma tan precisa que se puede realizar una máquina para simularlas. Después de los esfuerzos realizados para llevar a cabo la investigación, Allen Newell y Herbert Simon de la Universidad Carnegie Tech desarrollaron un programa de razonamiento llamado "El Logista de la Teoría", al que Simon se refería como un programa capaz de pensar de forma no numérica y el cuál fue capaz de realizar demostraciones de teoremas matemáticos.

[1]Es básicamente una conversación entre un ser humano y una máquina diseñada para interactuar verbalmente. Consiste en convencer al ser humano que evalúa el chat de que hay una persona detrás de la conversación. Si se logra convencer al ser humano, entonces la máquina pasa la prueba.

A pesar de que la inteligencia artificial clásica demostró ser adecuada para resolver problemas lógicos bien definidos, como jugar una partida de ajedrez o resolver un cubo de Rubik, resultó ser incapaz de determinar reglas explícitas para resolver problemas complejos, entre los que se encuentran:

- Clasificación de imágenes
- Reconocimiento de voz
- Traducción de lenguaje

Los problemas antes mencionados son tratados por un subconjunto de la inteligencia artificial, llamado aprendizaje de máquina o aprendizaje automático (Figura 1.1), particularmente por los métodos del subcampo denominado aprendizaje profundo (Deep Learning).

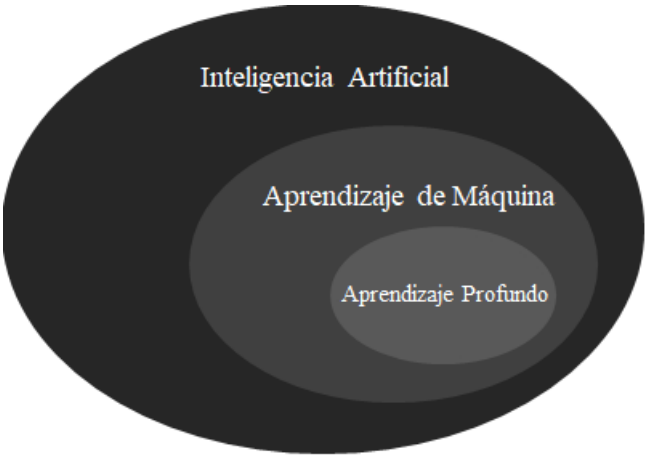


Figura 1.1: Diagrama jerárquico de la inteligencia artificial.

El aprendizaje de máquina ayuda a encontrar automáticamente un conjunto de reglas que permitan convertir un conjunto de datos en representaciones más significativas para una tarea dada. Un sistema de aprendizaje de máquina es entrenado en lugar de programado explícitamente, es decir, que el agente inteligente busca un conjunto de parámetros que determinen las reglas de clasificación de los datos sobre los que será aplicado.

Existen 3 técnicas empleadas para entrenar un agente en el aprendizaje de máquina, las cuales son: aprendizaje reforzado, aprendizaje no supervisado y aprendizaje supervisado.

El aprendizaje reforzado [7] tiene como propósito aprender de una situación y llevar a cabo acciones que maximicen su recompensa, de esta forma, la acción que produzca una recompensa mayor es la mejor acción que puede tomar. Al agente no se le dice qué acciones tomar y debe determinar qué acciones llevan a la mejor recompensa por medio de una dinámica conocida como prueba y error. Existen dos métodos que ayudan a un agente a decidir la acción que debe tomar, puede ser mediante exploración, donde el agente prueba diferentes acciones y descubre si hubo una mejor recompensa, o puede ser mediante la explotación, donde el agente

toma una acción que ha ejecutado previamente y por ende tiene la certeza de que se obtiene una recompensa óptima.

El aprendizaje no supervisado [7] consiste en encontrar estructuras ocultas en colecciones de datos no etiquetados. Es diferente al aprendizaje reforzado, mencionado anteriormente, porque no tiene como objetivo maximizar una señal de recompensa numérica.

El aprendizaje supervisado [8] le presenta a la máquina un numeroso conjunto de datos y un conjunto de etiquetas que le dicen al sistema el tipo de información que contienen los datos que está recibiendo, estos datos y etiquetas sirven como ejemplos relevantes ante una tarea determinada y el sistema se encarga de encontrar una estructura a partir de estos ejemplos, los cuales eventualmente le permiten al sistema generar reglas que automatizan la tarea una vez que ha sido entrenado.

Por otra parte, el Deep Learning [8] mantiene las mismas ramas del Machine Learning con la diferencia de que éste es capaz de extraer características de manera jerárquica que describen algún objeto, tarea, acción o patrón, teniendo cada característica definida en términos de características más simples. Esto es gracias a que emplean redes neuronales en capas, mismas que ayudan a detectar rasgos más significativos a medida que los datos de entrada avanzan a través de éstas.

El Deep Learning reúne conocimiento a partir de la experiencia y evita la necesidad de que un operador humano especifique formalmente el conocimiento que la computadora necesita.

Las redes neuronales convolucionales (CNN) secuenciales forman parte del Deep Learning y del aprendizaje supervisado. Una CNN [3] está conformada por neuronas, las cuales tienen asociados una serie de números considerados como pesos sinápticos, mismos que ayudan a clasificar la información que recibe la neurona una vez que dichos pesos hayan sido calibrados correctamente por medio de una etapa denominada entrenamiento. Cada neurona recibe algunos valores de entrada y realiza un producto punto entre los valores de entrada y los valores actuales de los pesos sinápticos.

Para construir una arquitectura de CNN se usan 3 tipos principales de capas [3], como son:

- Capa convolucional, integrada por:
  - Operación de convolución
  - Función de activación[2]
- Capa Pooling
- Capa totalmente conectada

La capa final de una CNN es una capa totalmente conectada que hace uso de múltiples perceptrones interconectados[3].

[2]En este trabajo de tesis se implementará únicamente la función de activación Rectified Linear Unit (ReLU); cuyo funcionamiento será explicado más a detalle en el capítulo 3.

[3]Frank Rosenblatt [9] fue el creador del perceptrón [10] es la red neuronal más básica que existe en el aprendizaje supervisado y es capaz de clasificar elementos que sean linealmente separables.

La motivación de sistemas de redes neuronales artificiales [11] es adaptar el cálculo altamente paralelo inspirado en sistemas de aprendizaje biológicos, construidos por redes complejas de neuronas interconectadas; sin embargo, la mayoría de redes neuronales artificiales de sistemas embebidos corren en máquinas secuenciales que emulan procesos distribuidos. De tal manera, que la base de este trabajo consiste en la implementación de las capas de una Red Neuronal Convolucional Secuencial en un sistema reconfigurable, para explotar la capacidad de concurrencia ofrecida por los mismos. Particularmente se usa el FPGA Spartan 6, este tipo de sistemas tienen la capacidad de ser reconfigurados[4] cada vez que la modificación de alguna característica dentro del sistema sea necesaria [12], [13]. Uno de los principales retos que lleva la implementación de una Red Neuronal Convolucional en un sistema reconfigurable es la implementación de una Unidad de Punto Flotante [14], la cual se encargará de realizar las operaciones que conlleva cada capa de la red. Se propone la utilización de un formato de punto flotante de 24 bits basado en el formato bfloat16 de Tensorflow [15], con el cual, se pretende reducir el error que tiene el bfloat16 al agregar 8 bits en la parte fraccionaria del número para ampliar la precisión.

[4]Esta característica está relacionada con la capacidad de agregar componentes o realizar modificaciones de hardware al diseño.

1.1. Objetivos

A continuación, se presentan los objetivos propuestos para guiar el desarrollo del proyecto.

1.1.1. Objetivo General

- Diseñar e implementar los elementos que forman parte de la arquitectura de una red neuronal convolucional en un sistema reconfigurable (FPGA).

1.1.2. Objetivos Particulares

1. Implementar y validar un módulo de punto flotante (suma, resta y multiplicación).
2. Implementar y validar la operación de convolución sobre arreglos de bidimensionales en VHDL.
3. Implementar la función de activación ReLU.
4. Implementar y validar el operador POOLING.
5. Implementar y validar el perceptrón.

1.2. Justificación

El sistema implementado beneficiaría a los sistemas basados en Edge Computing [5][16]. Permitiendo realizar análisis como clasificaciones, detecciones y regresiones en el lugar donde se adquieren los datos. A diferencia de enfoques basados en procesadores que ejecutan las operaciones de forma secuencial, nuestro trabajo sería más eficiente al evaluar las capas de forma concurrente y en comparación con una GPU, permitiría implementaciones de bajo costo [2], [13], [17]-[23]. En resumen, la implementación de este proyecto permitirá desarrollar sistemas empotrados que hagan uso de una CNN para resolver diferentes tipos de problemas, como son, reconocimiento de patrones, procesamiento de señales, ingeniería biomédica y reconocimiento de objetivos militares, entre otros [2].
El contenido de esta tesis se organiza de la siguiente manera, en el capítulo 2 se presentan los trabajos más relacionados a nuestra propuesta, haciendo una comparación entre ellos. Los conceptos teóricos necesarios para el desarrollo de nuestra propuesta son presentados en el capítulo 3. En el capítulo 4 se presenta la metodología que se llevó a cabo para desarrollar este proyecto, en el capítulo 5 se presentan los resultados obtenidos, en el capítulo 6 se presentan las conclusiones y el trabajo futuro.

[5]En el Edge Computing, el procesamiento pesado es llevado a cabo *in-situ*. Esta técnica consiste en colocar un dispositivo en el lugar donde se adquieren los datos para procesarlos y después enviar esta información al siguiente sistema; esto permite liberar ancho de banda al enviar datos procesados y no datos crudos.

1.3. Descripción del trabajo

Se propone la implementación de las principales capas de una red neuronal convolucional secuencial en un FPGA (figura 1.2). Se implementa una comunicación serial con una velocidad de 19,200 baudios para recibir una imagen en formato de punto flotante desde la PC, donde se hace uso de una interfaz gráfica implementada bajo el lenguaje de programación Python. El motivo de que las imágenes se envíen en formato de punto flotante se debe a que las imágenes procesadas por una red neuronal convolucional están normalizadas entre 0 y 1, de tal manera que la normalización se realiza desde la PC para tenerlas en el formato requerido.

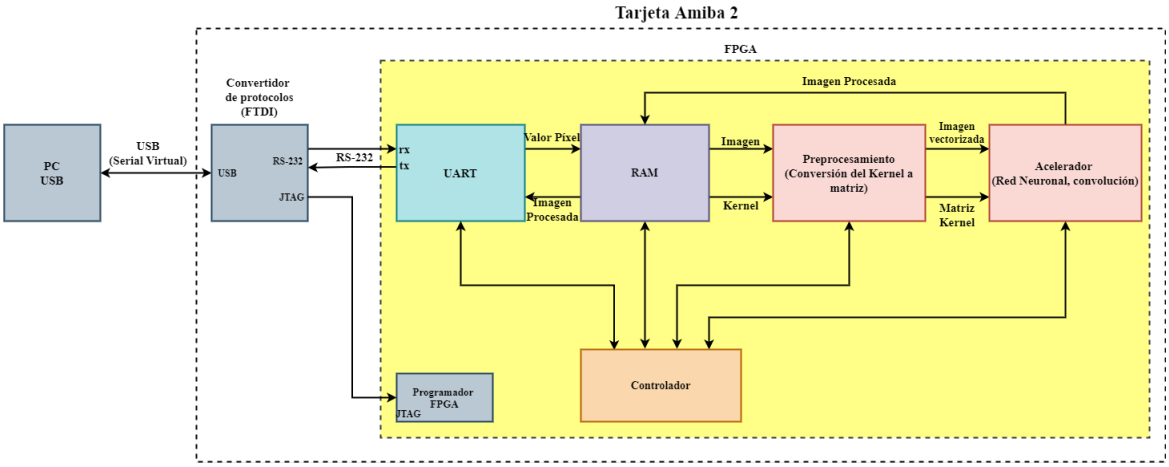


Figura 1.2: Diagrama de bloques de una Red Neuronal Convolucional Secuencial implementada en un FPGA.

La imagen se almacena en memoria RAM hasta que se completa su recepción, después se envía a una etapa de preprocesamiento que permite realizar una adecuación de la imagen y el kernel. La siguiente etapa se considera "Acelerador", donde se explotan las características de un FPGA para poder acelerar el procesamiento en una Red Neuronal Convolucional Secuencial.
La Red Neuronal Convolucional se compone de 3 capas principales, como son:

- Capa Convolucional
  - Operación de Convolución
  - Función de activación ReLU
- Capa Pooling
- Capa Totalmente conectada
  - Perceptrón Multicapa
  - Función de activación SOFTMAX

Se implementa la Capa Convolucional, el operador Pooling y el Perceptrón. Se hace uso de la tarjeta de desarrollo AMIBA2 para implementar el sistema.

2.
Estado del arte

En el presente capítulo se muestra una investigación de los FPGA que se tienen actualmente en el mercado y que se destacan por tener recursos lógicos y características similares a los FPGA que se usan en los trabajos del estado del arte, también, se muestran los trabajos relacionados a la técnica que involucra la implementación de las principales capas de una Red Neuronal Convolucional Secuencial en FPGA; mismos que se clasificarán en las diferentes áreas de estudio concernientes a este trabajo de tesis, como son, unidad de punto flotante (FPU), convolución, operador pooling, función de activación y MLP; todas las anteriores implementadas en el lenguaje de descripción de hardware VHDL.

2.1. FPGA y Tarjetas de Desarrollo

Esta sección contiene la información concerniente a los FPGA que se adecuan a las características requeridas en este proyecto de investigación. Dado que las operaciones que serán implementadas consumen una gran cantidad de recursos lógicos, como es la unidad de punto flotante (FPU) y la convolución, la investigación se enfocó a un sector de FPGA que cuentan con una cantidad de recursos lógicos que se encuentren dentro de un rango establecido por los FPGA usados en trabajos relacionados [2], [13], [17]-[23], de esta manera, es posible desarrollar los bloques necesarios en este proyecto asegurando que la cantidad de recursos lógicos no tenga un papel limitante.

2.1.1. Comparación y selección del FPGA

La tabla 2.1 muestra los FPGA de 6 empresas diferentes, entre las que se encuentran Xilinx e Intel. Se muestran 5 familias de FPGA de la empresa Xilinx, 5 familias de FPGA de la empresa Intel y por último 4 familias de FPGA de 4 empresas distintas que no dominan el mercado. Los FPGA mostrados fueron seleccionados de acuerdo a los FPGA usados en los trabajos que forman parte del estado del arte, como son Zynq [13], Spartan 3 [20], [23], Spartan 6 [2]. Partiendo de las características que tienen estos FPGA, se buscaron FPGA con características similares. En el caso de la familia Spartan 6 se tienen 3,849 celdas lógicas en su modelo más básico y 147,443 en su modelo más robusto. La familia que tiene más celdas lógicas son

los FPGA de Intel, con la familia Agilex Serie F con 392,000 en su modelo más básico y 2,692,760 en su modelo más robusto, mismo que tiene 259 Mb de memoria.
Dentro de este análisis, también se tienen familias de empresas que no forman parte de las empresas líderes en el mercado de FPGA, tal es el caso de empresas como Microsemi, Efinix y Quicklogic. Siendo la última quien tiene la familia de FPGA más básica, llamada PolarPro 3, con 1,019 celdas lógicas, sin embargo, tiene 73 Mb en memoria, por lo que fue contemplada en la investigación. La familia Trion, de la empresa Efinix tiene un máximo de 192,000 celdas lógicas y 14.1 Gb de memoria.

Tabla 2.1: Comparación de los FPGA existentes en el mercado.

Empresa	Familia	Celdas lógicas	Memoria (kb)	DSP Slices	Transectores	Pines I/O
Xilinx	Spartan 6	3,840 - 147,443	216 - 4,824	8 - 180	máx. 8	132 - 540
Xilinx	Spartan-7	6,000 - 102,400	180 - 4,320	10 - 160	-	100-400
Xilinx	Virtex-7	582,720 - 876,160	28,620 - 50,760	1,260 - 2,520	12.5 - 13.1 Gbps	850 - 300
Xilinx	Zynq-7000	28,000 - 444,000	2100 - 26500	80 - 2,020	máx. 16	100 - 400
Xilinx	Artix-7	12,800 - 215,360	720 - 13,140	40 - 740	6.6 Gbps	150 - 500
Intel	Agilex Serie F	392,000 - 2,692,760	38 - 259000	-	-	360 - 768
Intel	Arria 10	160,000 - 1,150,000	9 - 53000	-	12 - 72 de 17.4Gbps	192 - 768
Intel	Cyclone V GX	25,000 - 301,000	1,760 - 12,200	25 - 342	máx. 12 de 3.125	128 - 560
Intel	Cyclone IV GX	14,000 - 150,000	540 - 6,480	-	0 - 8 de 3.125 Gbps	72 - 475
Intel	Max 10	2,000 - 50,000	108 - 1,638	-	-	27 - 500
Lattice Semiconductor	CrossLink NX	17,000 - 39,000	1,024 - 2,560	-	-	40 - 180
Microsemi	PolarFire	109,000 - 481,000	7.6 - 33000	-	-	284 - 584

Tabla 2.1: Comparación de los FPGA existentes en el mercado.

Empresa	Familia	Celdas lógicas	Memoria (kb)	DSP Slices	Transectores	Pines I/O
Efinix	Trion	3,888 - 192,000	78,848 - 14,131,000	-	-	55 - 415
Quicklogic	PolarPro 3	1,019	73,000	-	-	46

2.1.2. Tarjetas de Desarrollo

Después de realizar la investigación de los FPGA que hay actualmente en el mercado, se realizó una investigación de tarjetas y kits de desarrollo que incluyen los FPGA encontrados, esta información se muestra en la tabla 2.2.

Con base a la investigación presentada, se eligió la tarjeta de desarrollo amiba 2 debido al costo y a que se adapta a los recursos a requeridos en la fase de diseño, misma que incluye un FPGA Spartan 6 (XC6SLX9). Este FPGA tiene 9,152 celdas lógicas, 576 kb de memoria, 16 DSP slices y 200 pines I/O y un oscilador de 50 MHz. Además, la tarjeta tiene un convertidor USB/RS-232, 51 pines I/O de propósito general, 12 leds, 9 switches de dos posiciones, 6 pulsadores, 8 desplegadores de 7 segmentos, alimentación principal de 5 V y alimentación secundaria de 3.3 V.

Tabla 2.2: Comparación de las tarjetas de desarrollo que contienen los FPGA de la tabla 2.1

Empresa	FPGA	Nombre del Kit	Precio (USD)
Xilinx	Spartan 6	SP605 Evaluation Kit	$650
Xilinx	Spartan-7	SP701 FPGA Evaluation Kit	$645
Xilinx	Virtex-7	FPGA VC707 Evaluation Kit	$3,495
Xilinx	Zynq-7000	ZC702 Evaluation Kit	$1,609.90
Intel	Arria 10	SoC Development Kit	$4,495
Intel	Cyclone V GX	SX SoC Development Board	$1,795
Intel	Cyclone IV GX	DE2-i150 FPGA Development Kit	$700
Intel	Max 10	Neek	$395
Lattice Semiconductor	CrossLink NX	LIF-MD6000-ML-EVN	$170.69
Microsemi	PolarFire	MPF300-VIDEO-KIT	$1,078.80
Efinix	Trion	T8F81C	$100
Intesc	Spartan 6	Amiba 2	$40

2.2. Trabajos relacionados a la Unidad de Punto Flotante (FPU)

En circuitos digitales y sistemas empotrados son importantes las operaciones de punto flotante, sin embargo, estas operaciones consumen bastantes recursos computacionales. No obstante, los avances tecnológicos han guiado un incremento dramático en la densidad y velocidad de operaciones en FPGA [24].
En [24], se menciona que más del 94 % de operaciones de punto flotante usadas son la suma/resta y la multiplicación, por este motivo se enfocan en la implementación de una FPU de alto desempeño sobre estas operaciones. Para la implementación de la suma/resta se emplea el algoritmo estándar conocido como "algoritmo LOD", por otra parte, para implementar la multiplicación emplearon un algoritmo basado en la separación de los operandos que serán multiplicados en bloques más pequeños. El diseño que presentan otorga un resultado después de 22 ciclos de reloj tanto para la suma/resta como para la multiplicación. Ang Lv *et al.* [25], presenta una unidad de punto flotante diseñada para realizar cálculos en redes neuronales. Su diseño presenta un formato de datos de 32 bits personalizado que permite cambiar la cantidad de cálculos mediante el cambio de la estructura de los datos. Las operaciones disponibles en esta referencia son la multiplicación, suma y resta. El formato que se emplea consiste en 1 bit de signo, 14 bits para el exponente y 17 bits de mantisa, de esta manera los autores maximizan la velocidad de las operaciones y reducen los recursos multiplicadores requeridos. Además, aseguran que el uso de su estructura en redes neuronales permite procesar los datos con menos potencia y mayor velocidad. Los resultados que obtienen muestran que el diseño personalizado hace uso de menos recursos del FPGA, y la velocidad de ejecución de operaciones es más rápida.
La referencia [20] enfatiza el hecho de que una red neuronal necesita una precisión óptima en la FPU. El estudio implementa un detector de rostros basado en FPGA usando redes neuronales. Se usa una FPU para representar el sistema numérico y proporciona un rango dinámico y reduce los bits de la unidad aritmética más que el método de punto fijo. Estas características llevan a la reducción en la memoria de tal forma que resulta eficiente para un sistema de redes neuronales con un gran tamaño de bits de datos. El sistema procesa una imagen en 1.7 ms y es 38 veces más rápido que los 50 ms que tarda un procesador Pentium 4 con velocidad de 1GHz. El detector basado en FPGA también requiere del 33 % del área total en FPGA (Xilinx XC3S1500). Para determinar los bits del FPU, se examinó cómo la representación del error afecta la tasa de detección. La unidad aritmética ocupa el 84 % del área disponible. La reducción del FPU de 32 a 24 bits reduce en un 25 % el tamaño de la memoria y de la unidad aritmética teniendo sólo el 0.32 % de deterioro en la tasa de detección. La FPU tiene dos módulos, un sumador y un multiplicador. Se emplea un IP Core de tipo hard[1] en el FPGA para el multiplicador con el fin de mejorar la velocidad. El incremento del número de segmentación permite velocidades más rápidas, pero también incrementa área y potencia.
En la literatura [26] se emplean dos algoritmos con el fin de comparar el desempeño que se tiene en la implementación de un multiplicador de punto flotante, estos algoritmos son Booth y Karatsuba (normal y recursivo). La implementación del multiplicador se realiza en precisión simple y precisión doble, definidas

[1]Estos cores usualmente se usan para diseño de bajo nivel, como en lógica análoga. Son llamados hard cores porque la función del chip no puede ser modificada de ninguna manera significativca por los diseñadores una vez que ha sido creado.

con el estándar IEEE-754. El algoritmo Booth tiene como propósito reducir los recursos de hardware empleados y los tiempos de retraso. Por otra parte, el algoritmo Karatsuba se define para la multiplicación de grandes enteros y es considerado como uno de los más rápidos y mejores, además, toma menos tiempo de propagación de señales.

En [14], se pretende reducir el área y el retraso de ruta combinatoria para mejorar la velocidad de operaciones de punto flotante con formato de 32 bits, la cual, es alcanzada por el paralelismo en el multiplicador. Se usa el algoritmo Booth, donde el número de productos parciales se reducen y aceleran la velocidad de la multiplicación. El módulo se implementa en un Spartan 6. Los resultados muestran que se obtiene un 59 % de optimización en el retardo de la multiplicación y un 50 % en la suma. En este trabajo, se menciona que el gran uso de la unidad de punto flotante se debe a tres razones, como son, las restricciones efectivas de rango y los problemas de compatibilidad correlacionados con la unidades de punto fijo, también, se debe a la mejora en tecnología VLSI que se puede asignar hardware a la inclusión de operaciones en punto flotante y por último, se encuentra la implementación de una unidad de punto flotante donde el cálculo de punto flotante y de punto fijo pueden operar concurrentemente.

[27] presenta el diseño de una Unidad de Punto Flotante de velocidad y area eficiente. Se emplea el algoritmo modificado de Mitchell para la multiplicación y el algoritmo Euclidiano para la división de punto flotante. Se muestra la mejora en área del 21.2 % en términos de *slices*, 27.4 % en términos de LUTs y 36.3 % en términos de IOBs con respecto a su diseño anterior.

Así como en el trabajo explicado anteriormente, [28] presenta un multiplicador basado en el algoritmo Mitchell, el cual tiene la posibilidad de alcanzar una precisión arbitraria. El multiplicador está basado sobre la misma idea de representación de números que el algoritmo de Mitchell, pero con la diferencia de que no usa aproximación logarítmica. Esta implementación hace uso de circuitos paralelos para la corrección de errores. Se menciona que en aplicaciones concernientes al procesamiento de señales, los resultados aritméticos pueden no ser exactamente precisos.

La referencia [29] presenta un módulo divisor de precisión simple basado en el algoritmo Goldschmidt. El algoritmo se implementa usando un multiplicador y restador de 32 bits. La característica que predomina en esta propuesta es que el módulo que realiza el cálculo de la mantisa está diseñado usando la técnica de multiplicación Vedic de 24 bits. La técnica Vedic da lugar a una mayor velocidad de cálculo y es usada para incrementar el desempeño del divisor de punto flotante. Para sintetizar el diseño se usó un Spartan 6 SP605. Se obtuvo una reducción en el consumo de potencia y tiempo de latencia del 26 % y del 42 % respectivamente.

En [30] se implementa la suma, resta, multiplicación y división de acuerdo al formato IEEE 754. La FPU se diseñó bajo el lenguaje de descripción de hardware VHDL usando Xilinx 13.3 y trabaja con 32 bits, de los cuales son 1 bit de signo, 8 bits de exponente y 23 bits de mantissa contemplando un bit oculto.

A diferencia de [14], [24]-[26], [28], [29], se implementa un módulo de 24 bits la cual permite tener un rango similar a la representación de 32 bits pero sacrificando la precisión. En comparación con la FPU de [20], este trabajo tiene 8 bits en lugar de 6 bits para el exponente y 15 bits en lugar de 17 bits para la mantisa. Esto

permite contar con el rango del formato de precisión simple float32 y el formato de Tensorflow bfloat16 [15] pero incrementa la precisión comparado con bfloat16 y reduce la memoria comparado con float32.

2.3. Trabajos relacionados a la capa convolucional

La convolución es una operación que se encarga de extraer características de una imagen haciendo uso de un kernel. De acuerdo con [3] el kernel estará conectado a regiones locales de la imagen y realizará un producto punto entre sus pesos y la región en la que está posicionado. En [31], [32], se menciona que la convolución de 2 dimensiones es una de las operaciones más usadas en aplicaciones de procesamiento de video e imágenes.

En la referencia [17] se presenta una red neuronal convolucional (CNN) de estructura Z de 3 dimensiones, con el propósito de resolver el problema de eficiencia y velocidad de los algoritmos de aprendizaje profundo. En años recientes, con el rápido desarrollo de FPGA de alto desempeño, los investigadores prestan mayor atención al bajo consumo de potencia y diseños de arquitecturas flexibles. La figura 2.1 muestra la estructura que implementaron los autores del trabajo.

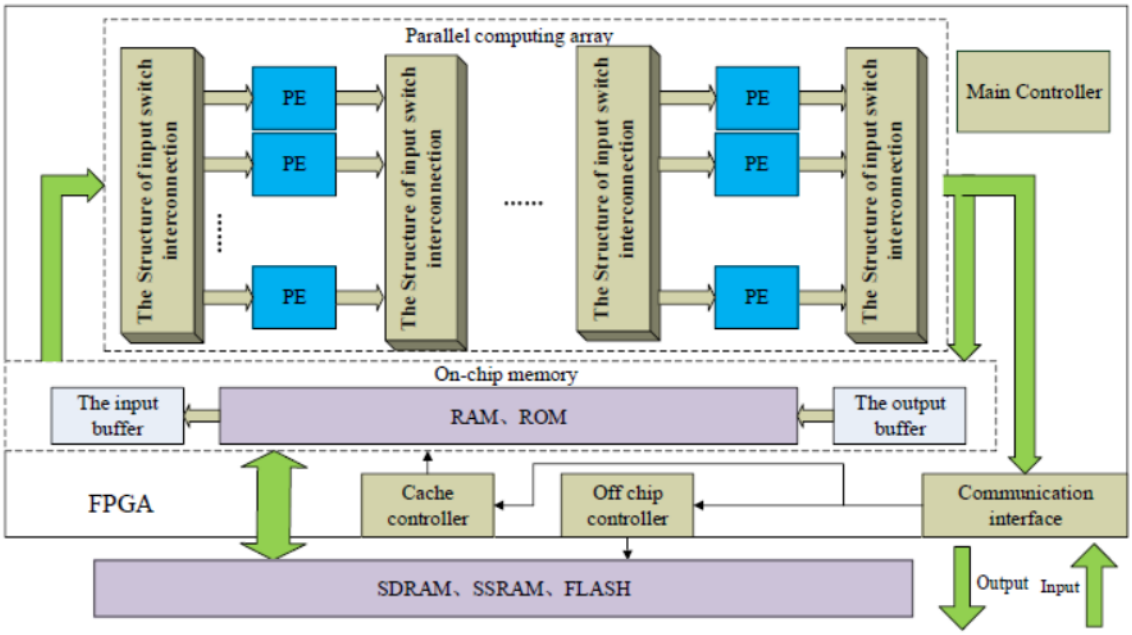


Figura 2.1: Estructura de una red neuronal profunda. Imagen obtenida de [17].

Los pesos sinápticos requeridos por la red son encontrados en una PC, después el sistema transmite los pesos a la red encontrada en el FPGA a través de PCI-e. Después de que el FPGA habilita el programa principal, se transfiere la imagen almacenada en la memoria hacia el array paralelo.

En [32] se presenta el diseño e implementación de un núcleo que realiza la operación convolucional en 2 dimensiones para aplicaciones optimizadas de video, donde se hace uso de un FPGA SpartanXL. El núcleo

está parametrizado y es escalable con respecto al tamaño de la ventana, la longitud de bits del píxel de entrada y el tamaño de la imagen. Las operaciones efectuadas entre la ventana y la superficie de la imagen son representadas como una suma de potencias de dos en la representación "CSD "(Canonical Signed Digit), lo que significa que la operación de multiplicación puede ser fácilmente implementada por un número pequeño de operaciones de suma acumulables, garantizando ahorro en recursos de hardware. Los bloques que implementan son la multiplicación paralela de bits, la suma paralela de bits y bloques para retrasar los píxeles.

En [31] se propone una arquitectura digital de alto desempeño para procesar convoluciones de dos dimensiones usando el cuadrante de simetría del kernel. Los píxeles en los cuatro cuadrantes de la región del kernel con respecto al píxel de la imagen son considerados simultáneamente para procesar los productos parciales de la suma de la convolución. También, se propone una estrategia para identificar los datos que serán introducidos en diferentes elementos de procesamiento, ayudando a reducir los requerimientos en almacenamiento de los datos. Su diseño ayuda a reducir en un 75 % el uso de multiplicadores y en un 50 % el uso de sumadores. La arquitectura que proponen, es capaz de manejar operaciones con un kernel de 14×14 a una tasa de 57 cuadros de 1024×1024 píxeles por segundo en un FPGA Virtex 2.

En la referencia [33] se propone un modelo mejorado RLeNet, el cual optimiza los parámetros y usa tecnología paralela para acelerar redes neuronales convolucionales implementadas en hardware. Los parámetros del modelo mejorado son reducidos en un 3.2 % con respecto al modelo original. Así mismo, su uso para realizar reconocimiento con el conjunto de datos MNIST tiene una precisión del 98.5 %. El modelo se implementa en el FPGA Artix 7 de Xilinx. Cuando el sistema hace uso de un reloj de 200MHz, el tiempo para procesar una imagen es de $31.8\mu s$. Con respecto al entrenamiento o obtención de los pesos sinápticos del modelo, se hace una conversión de tipo de float32 a int9 para reducir los recursos del FPGA que usa el sistema y acelerar el tiempo de procesamiento, de tal manera que se usan números de punto fijo y no números de punto flotante. La imagen de entrada tiene una resolución de 28×28 píxeles y está en formato de escala de grises. Por último, se emplea una memoria RAM desplazadora para realizar la convolución.

En [34] se presenta la implementación de un acelerador para operaciones de convolución a través de una arquitectura basada en un arreglo sistólico implementada en la tarjeta Zedboard de Xilinx. Se usa una multiplicación de matrices para llevar a cabo la convolución. Tal multiplicación se obtiene mediante el reordenamiento de las estructuras para poder mapear la convolución a una multiplicación de matrices, la cual proporciona dos ventajas: flexibilidad y continuidad. Además, hacen uso de la red LeNet 5 para realizar pruebas experimentales. En cuanto a los resultados, se obtuvo que el arreglo sistólico consume 64 secciones DSP para operaciones de convolución.

La referencia [35] se centra en el desarrollo de una arquitectura eficiente que implemente el algoritmo de convolución de dos dimensiones usando bloques de control. Proponen la implementación con un uso reducido de registros desplazadores, multiplicadores, sumadores y bloques de control, dando lugar a un ahorro de recursos de hardware así como el uso de pocos LUTs. La implementación del sistema se realiza en el FPGA Spartan 3E de Xilinx.

En el trabajo propuesto se hace uso de una Unidad de Punto Flotante (FPU) diseñada en el mismo. La FPU

se emplea para realizar el producto punto entre los valores obtenidos de la imagen almacenada en una memoria RAM y los valores del Kernel. Para realizar el desplazamiento del kernel sobre la imagen se emplean n apuntadores que mantienen la distancia y el formato del kernel. Para controlar los apuntadores se implementó una máquina de estados finitos que realiza un conteo tomando como referencia las dimensiones de la imagen, de esta manera se sabe cuántos desplazamientos horizontales y verticales debe realizar antes de cubrir la superficie total de la imagen. Los datos obtenidos se almacenan en otra estructura de datos.

2.4. Trabajos relacionados a la capa totalmente conectada

Una capa totalmente conectada está formada por perceptrones interconectados. En la literatura se han encontrado métodos para la extracción de rasgos, como es el caso de [23], donde se propone un método preciso, mismo que está basado en FPGA y su aplicación específica es el reconocimiento de dígitos persas escritos a mano. El método de extracción de rasgos es apropiado para FPGA porque es posible implementarlo sólo con operaciones de suma y resta. El método propuesto es usado para extraer los rasgos de imágenes normalizadas de dígitos escritos a mano de tamaño 40 * 40 píxeles. El sistema es simple, más preciso y menos complejo que otros sistemas similares. El proyecto emplea 18,000 imágenes binarias normalizadas de dimensiones 40*40 pixeles para entrenar la red y 2,000 para ponerla a prueba.

Por otra parte, [21] propone métodos para implementar redes neuronales artificiales (ANN) basadas en controladores PID. Las redes neuronales pueden implementarse de dos formas: análogas y digitales. Es conocido que los sistemas digitales tienen características más ventajosas que los sistemas análogos en los siguientes aspectos: mayor precisión, mejor repetibilidad, baja sensibilidad al ruido, mejor adaptación para realizar pruebas, alta flexibilidad y compatibilidad con otros tipos de preprocesadores.

En una perspectiva más amplia, las implementaciones de hardware se clasifican como: basadas en FPGA, basadas en DSP y basadas en ASIC. Una ventaja que ofrecen las implementaciones en FPGA es que pueden preservar la arquitectura paralela de las neuronas en una capa y también ofrecen flexibilidad en la reconfiguración. El problema fundamental que limita el tamaño de una Red Neuronal Artificial (ANN) basada en FPGA es el costo de implementación de las multiplicaciones asociadas con las conexiones sinápticas debido a que las ANN paralelas requieren un gran número de multiplicaciones. Una forma de reducir el número de multiplicadores es compartiendo un multiplicador con todas las entradas de las neuronas.

Otro método para reducir la circuitería necesaria para la multiplicación se basa en técnicas de procesamiento estocástico de bits en serie. La complejidad del sumador depende de la precisión de las entradas de la sinapsis y en el número de entradas para cada neurona. Los sumadores deben ser compartidos a través de las entradas con resultados intermedios siendo guardados en un acumulador.

El trabajo presentado en [22] muestra otra aplicación de un perceptrón multicapa (MLP) que consiste en un sistema de reconocimiento de gases, el cual es capaz de discriminar especies limitadas de gas industrial. Le denominan “nariz electrónica”y consiste en un arreglo de 8 microplacas basadas en sensores de gas con una delgada capa de oxido de estaño con diferentes patrones de selectividad, una unidad colectora de señales y

una señal de reconocimiento de patrones y decisiones en un chip basado en lógica reconfigurable.
Se implementó en FPGA una red neuronal basada en Back Propragation (BP) con un perceptrón multicapa. La red contiene 8 unidades de entrada, 4 neuronas en la capa oculta y 5 neuronas regulares (Regular Neurons) en la capa de salida. La "nariz electrónica "clasificó 5 tipos de gases industriales en la simulación por computadora.
Para las pruebas de validación se empleó la tarjeta APS X208. También, se describe una propuesta de solución a la implementación de MLP en hardware. Esta propuesta se desarrolla como una aplicación de tiempo real mediante el uso del lenguaje de descripción de hardware VHDL para procesar la entrada de las señales de 8 sensores de gas.
En [19] se implementa una ANN básica en FPGA. La implementación en FPGA puede utilizar paralelismo para acelerar el tiempo de procesamiento. Adicionalmente, la implementación en hardware puede ahorrar potencia a comparación con una CPU/GPU. La ANN implementada en este trabajo reduce el rango de error de 10^{-2} a 10^{-4}, para el problema de la compuerta XOR.
En [2] se presenta un proceso que incluye la adquisición de la imagen, el preprocesamiento de la imagen, un módulo para la red MLP y la salida del reconocimiento resultante. El preprocesamiento de la imagen y la red MLP implementada en hardware son la parte clave del proyecto. La imagen original se convierte en un vector de características de 256 espacios. Una red MLP puede aprender y guardar una gran cantidad de mapeos de modelo entrada-salida. La topología de la estructura de una red MLP consta de 3 capas: capa de entrada, capa oculta y capa de salida. También tiene 2 partes: una es cuando avanza la información de entrada y otra cuando se propaga el error hacia atrás. Además, incluye dos fases, la fase de entrenamiento offline y la fase de reconocimiento online. Para cada neurona en la capa oculta la entrada es la misma. Los valores de entrada son los valores de los pixeles. Los pesos y umbrales de cada neurona son diferentes. La función de la neurona es medir la integración de las 256 entradas. La neurona implementada recibe un valor para cada entrada, lo multiplica y después lo agrega a un registro que acumula el valor de las multiplicaciones totales, para después enviarlo a la función de activación.
La función de activación se implementa con una aproximación lineal por partes, este método lo requiere muchos multiplicadores así como registro y ahorra bastantes recursos lógicos.
El modelo que se propone en este trabajo de tesis se caracteriza por su fácil configuración gracias a la implementación unitaria del perceptrón, el cual hace uso de la unidad de punto flotante implementada en este trabajo, una vez configurada la primer neurona, es posible replicarla para así poder conectarla con otras neuronas y de esta manera obtener la MLP deseada.
Una de las principales diferencias entre el trabajo propuesto y aquellos mencionados anteriormente, los cuales, forman parte del estado del arte es que en este trabajo se propone la implementación de cada módulo que forma parte de una red neuronal convolucional, además, se realiza la implementación de una unidad de punto flotante de 24 bits para que lleve a cabo las operaciones necesarias en cada capa. En relación a los trabajos de punto flotante, la diferencia radica en la modificación del número de bits que forman parte de la mantisa, donde se disminuyen de 23 a 15 bits. También se hace uso del sumador LOD definido en el formato

IEEE-754 [36]. Por otra parte, el multiplicador hace uso de un proceso encontrado en [24] que consiste en separar los bits correspondientes al signo, el exponente y la mantisa para procesar cada fragmento de manera separada.

3.

Análisis de técnicas usadas para implementar las capas de una red neuronal convolucional secuencial en lógica reconfigurable

A continuación, se presentan conceptos y técnicas que fueron llevadas a cabo y tomadas en cuenta durante la elaboración de este proyecto de investigación. Su comprensión es factor clave para el entendimiento de este trabajo.

3.1. Estándar IEEE 754-2008

El estándar IEEE 754-2008 [36] incluye casi todas las definiciones del estándar de 1985 y 1987. La mejora principal en el nuevo estándar es la definición de representaciones y operaciones decimales de punto flotante. El estándar define formatos aritméticos y de intercambio, algoritmos de redondeo, operaciones aritméticas y manejo de excepciones.
Este formato permite representar un subconjunto finito de números reales. Los números finitos pueden ser representados en base 2 y en base 10.
Cada número se describe por 3 enteros: el signo (0 o 1), el significando o mantisa *s* y el exponente *e*. El formato también permite la representación de números infinitos y de valores especiales, llamados NaN (Not a Number) para representar valores inválidos.
Los valores que pueden ser representados, están determinados por la base B, el número de dígitos del significando (*p*) y el máximo y mínimo de valores *e* (e_{min} y e_{max}). De esta manera, *s* es un entero que pertenece al rango 0 a B^{p-1} y *e* es un entero tal que $e_{min} \leq e \leq e_{max}$.

3.1.1. Aritmética en punto flotante

Con esta notación es posible representar un amplio rango de valores numéricos, tanto positivos como negativos centrados en el cero. Un número cualquiera *X* expresado en notación exponencial 3.1, se puede expresar como:

$$X = M \times B^E \tag{3.1}$$

Si se quiere expresar en un registro de *n* bits, se utilizarán *p* bits para la mantisa *M* y *q* bits para el exponente *E*, además, un bit de signo *s*. Siendo *B* la base del exponente y cumpliéndose que $n = p + q + 1$ [37]-[39]. Las operaciones de suma y resta, así como la multiplicación y la división pueden producir reboses, ya sea por resultados demasiado grandes (desbordamientos) o demasiado pequeños (subdesbordamientos).
Hay 4 tipos de reboses posibles:

- Desbordamiento del exponente: Es cuando un exponente positivo *E* excede de su valor máximo permitido.
- Subdesbordamiento del exponente: Es cuando un exponente negativo *E* excede de su valor mínimo permitido. Esto significa que el número *x* es demasiado pequeño y se puede considerar como igual a cero.
- Desbordamiento de mantisa: En la suma de dos mantisas del mismo signo, se puede producir un arrastre del bit más significativo. Esto se soluciona mediante la renormalización, desplazando a la derecha un bit la mantisa y ajustando el exponente.
- Subdesbordamiento de mantisa: En el proceso de alineación de las mantisas, si los dígitos se desplazan hacia la derecha más allá de su bit menos significativo, lo que sucede es que estos bits se pierden y el proceso es similar a un redondeo del resultado.

Suma y resta

Cuando se suman o restan dos números en punto flotante, se deben comparar los exponentes e igualarlos, por lo que se debe desplazar o alinear el más pequeño respecto al más grande. Dados dos números en representación en punto flotante, como se muestra en las ecuaciones 3.2 y 3.3.

$$x = m_x \times 2^{xe} \tag{3.2}$$

$$y = m_y \times 2^{ye} \tag{3.3}$$

Las operaciones de suma y resta se definen de la siguiente manera, suponiendo que $x_e < y_e$:

$$x + y = (m_x \times 2^{x_e - y_e} + m_y) \times 2^{y_e} = (m_x + m_y \times 2^{y_e - x_e}) \times 2^{x_e} \tag{3.4}$$

$$x - y = (m_x \times 2^{x_e - y_e} - m_y) \times 2^{y_e} = (m_x - m_y \times 2^{y_e - x_e}) \times 2^{x_e} \tag{3.5}$$

Multiplicación y división

La multiplicación y la división son más sencillas de realizar. La fórmula que permite realizar estas operaciones manualmente son:

$$x * y = (m_x \times m_y) \times 2^{x_e + y_e} \tag{3.6}$$

$$x/y = (m_x/m_y) \times 2^{x_e - y_e} \tag{3.7}$$

3.2. CNN (Red Neuronal Convolucional)

Los modelos llamados redes neuronales [1], están inspirados en una forma simplificada del funcionamiento de las neuronas. Actualmente, las redes neuronales profundas, han sido muy exitosas en tareas de alta complejidad, como son, la identificación de objetos en imágenes (Figura 3.1), el reconocimiento del lenguaje y el reconocimiento de patrones.

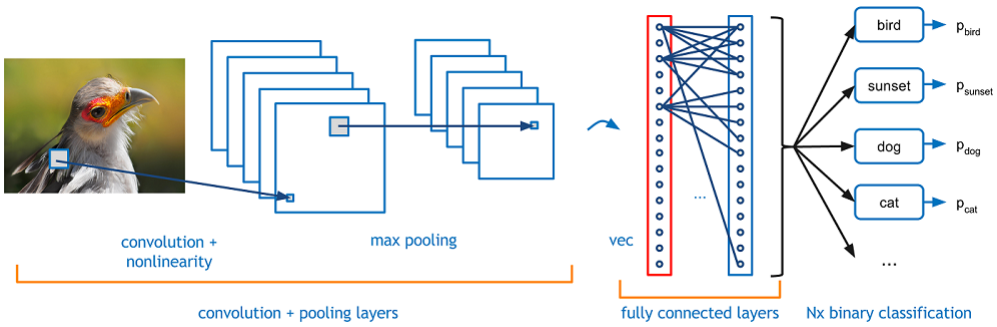


Figura 3.1: Representación del reconocimiento de objetos en una imagen por una red neuronal convolucional [40].

Algunas de las innovaciones más influyentes que han surgido en la rama de la visión por computadora han sido las redes neuronales convolucionales (CNN) [40]. Las CNN asumen que las entradas son imágenes, lo que permite codificar ciertas propiedades dentro de la arquitectura. Esto permite que la implementación de las siguientes funciones sean más eficientes y reduce enormemente la cantidad de parámetros en la red [3].

3.2.1. Arquitectura de una CNN

Las Redes Neuronales Convolucionales (CNN) toman ventaja de que sus parámetros de entrada son imágenes y restringen su arquitectura de forma más sensible [3]. Las neuronas de una CNN (Figura 3.2) están colocadas en 3 dimensiones: ancho, alto y profundidad. Las neuronas en una capa sólo estarán conectadas a una pequeña región de la capa que la precede, y no a todas las neuronas como en una capa totalmente conectada. Al final de la arquitectura de una CNN, se reducirá la imagen completa a un vector que contiene el puntaje obtenido para cada clase, el cual estará organizado a lo largo de la dimensión de profundidad.

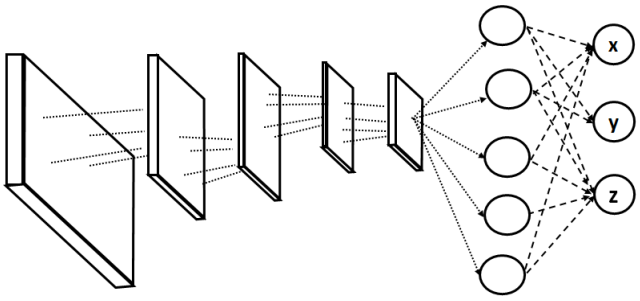


Figura 3.2: Ejemplo de la arquitectura de una red neuronal convolucional.

Capas usadas para construir CNN's

Una CNN [3] es una secuencia de capas, y cada capa de una CNN transforma un volumen de activación en otro, a través, de una función diferencial. Para construir una CNN, se usan 3 tipos principales (Figura 3.3) de capas:

- Convolutional Layer
- Pooling Layer
- Fully-Connected layer

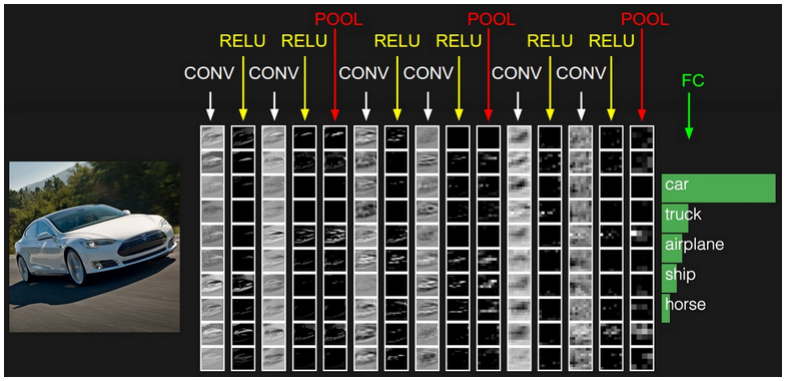


Figura 3.3: Ejemplo del proceso para reconocer un objeto dentro de una imagen [3].

A continuación, se presenta la descripción de algunas de las capas que forman parte de la arquitectura de una CNN:

- **INPUT** Son los valores de los píxeles de la imagen.
- **CONV LAYER (Figura 3.4)** Calcula la salida de las neuronas que están conectadas a las regiones locales en la entrada, cada una calcula un producto punto entre sus pesos y la región a la que están conectados del volumen de entrada.

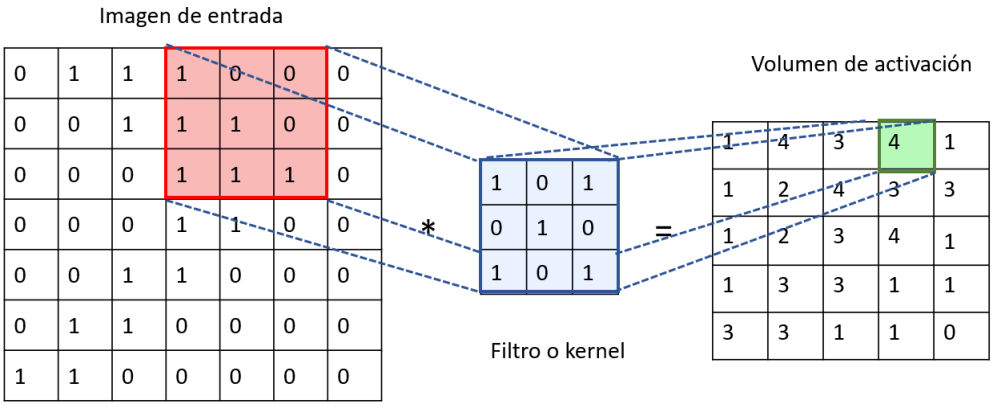


Figura 3.4: Representación de una capa convolucional.

- **RELU LAYER** Aplica la función de activación elemento a elemento, de la forma: $max(0, x)$.

- **POOL LAYER (Figura 3.5)** Su función es reducir progresivamente el tamaño del volúmen de activación (información relevante en la imagen), para reducir la cantidad de parámetros y cálculos en la red, de esta manera se controla el sobre ajuste. Llevará a cabo una operación de muestreo a lo largo de las dimensiones espaciales (ancho y alto).

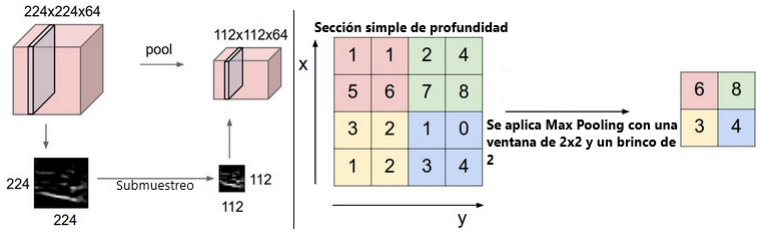


Figura 3.5: Ejemplo de la capa de pooling en una red neuronal convolucional [3].

- **FULLY-CONNECTED LAYER** Calculará los porcentajes de relación a cada clase del objeto reconocido. Está formada por múltiples perceptrones interconectados y al final de esta capa se encuentra la función de activación SOFTMAX para el caso de la clasificación, la cual determina el porcentaje de parentesco del objeto clasificado con cada clase.

3.2.2. Perceptrón

Un perceptrón [11], toma un vector de entrada de valores reales, calcula una combinación lineal de estas entradas y después tiene a la salida *"1"* si el resultado es mayor a un umbral y *"-1"* de otra manera (ecuación 3.8), donde cada w_i es una variable[1] de tipo real, también considerada peso sináptico que determina

[1]Esto sólo ocurre cuando la función de activación del perceptrón es la función signo, debido a que se pueden tener otras funciones de activación.

la contribución de la entrada x_i a la salida del perceptrón. Cabe aclarar que la cantidad $(-w_0)$ es un umbral que la combinación de las entradas ponderadas $w_1 * x_1 + ... + w_n * x_n$ debe sobrepasar para que el perceptrón de una salida de 1.

$$o(x_1, ..., x_n) = \begin{cases} 1 & si \quad w_0 + w_1 * x_1 + ... + w_n * x_n > 0 \\ -1 & si \quad no \end{cases} \tag{3.8}$$

Se puede ver al perceptrón (figura 3.6) como la representación de un hiper plano de una superficie de decisión en un espacio de instancias de n-dimensiones. El perceptrón da como resultado un 1 para instancias que se encuentran en un lado del hiper plano y -1 para instancias que se encuentran en otro lado.

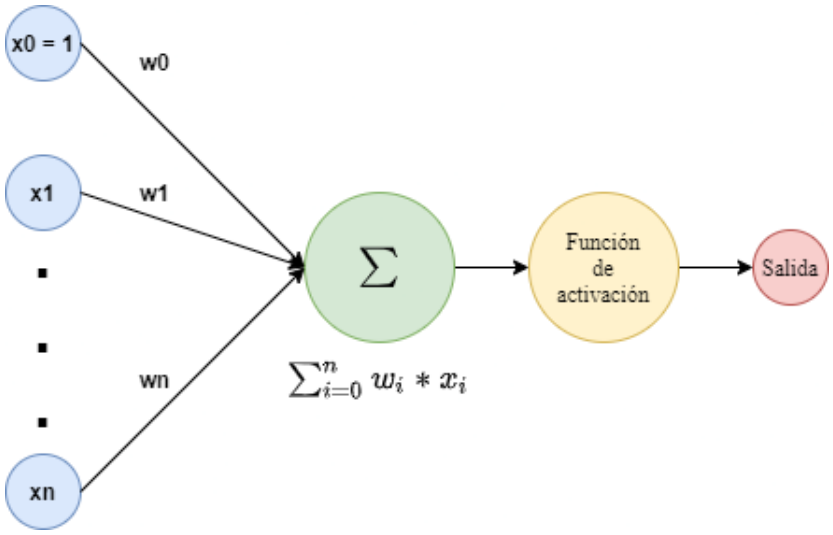


Figura 3.6: Representación del perceptrón.

3.3. Circuitos síncronos

Un circuito síncrono [41] (Figura 3.7), hace uso de una serie de elementos para su correcto funcionamiento, los cuales son listados a continuación:

- Registro de estado
- Salida del registro
- Lógica del estado siguiente
- Lógica de salida
- Señal externa de entrada

El elemento de memoria, mejor conocido como registro de estado, es una colección de Flip-Flop tipo D, sincronizados por una señal de reloj global. La salida del registro, que es la señal *state_reg* (el contenido guardado en el registro), representa el estado interno del sistema. La lógica del estado siguiente es un

circuito combinacional que determina el estado siguiente del sistema. La lógica de salida es otro circuito combinacional que genera la señal de salida externa. Nótese que la salida depende de la señal de entrada externa y del estado actual del registro.

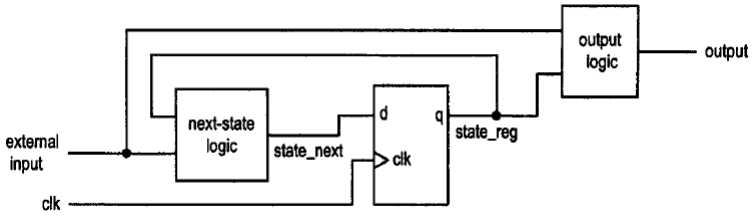


Figura 3.7: Diagrama conceptual de un circuito secuencial síncrono [41].

A continuación, se indica el funcionamiento del circuito:

- En el flanco de subida de la señal del reloj, el valor de la señal *state_next* es muestreada y propagada hacia el puerto *q*, el cual se convierte en el nuevo valor de la señal *state_reg*. El valor también se almacena en un flip-flop y permanece sin cambios por el resto del periodo del reloj. Representa el estado actual del sistema.
- Basándonos en el valor de la señal *state_reg* y la entrada externa, la lógica del estado siguiente (*next-state logic*) opera el valor de la señal *state_next* y la lógica de salida (*output logic*) opera el valor de la salida externa (*output*).
- En el siguiente flanco de subida del reloj, el nuevo valor de la señal *state_next* es muestreado y la señal *state_reg* es actualizada. Después, el proceso se repite.

Existen varias ventajas del diseño síncrono. Primero, simplifica la sincronización del circuito. Satisfacer las restricciones de sincronización es una de las tareas más difíciles de diseño. Debido a que en un circuito síncrono, todos los flip-flops son controlados bajo la misma señal de reloj, el muestreo de los flancos del reloj ocurre simultáneamente. Sólo se necesita considerar las restricciones de tiempo de un componente con memoria simple. Segundo, el modelo síncrono claramente separa el circuito combinacional y el elemento de memoria. Fácilmente, se puede separar la parte combinacional del sistema y diseñarlo y analizarlo como un circuito combinacional regular. Tercero, el diseño síncrono puede adaptarse fácilmente a los riesgos de sincronización.

3.3.1. UART (Transmisor-Receptor Asíncrono Universal)

El UART [42] (Figura 3.8), es un circuito que recibe datos paralelos y los envía a través de un puerto de forma serial. Los UARTs son usados frecuentemente en conjunto con el estándar RS-232 de la EIA (Alianza de Industrias Electrónicas), el cual, especifica las características eléctricas, mecánicas, funcionales y procedimentales entre dos equipos de comunicación de datos.

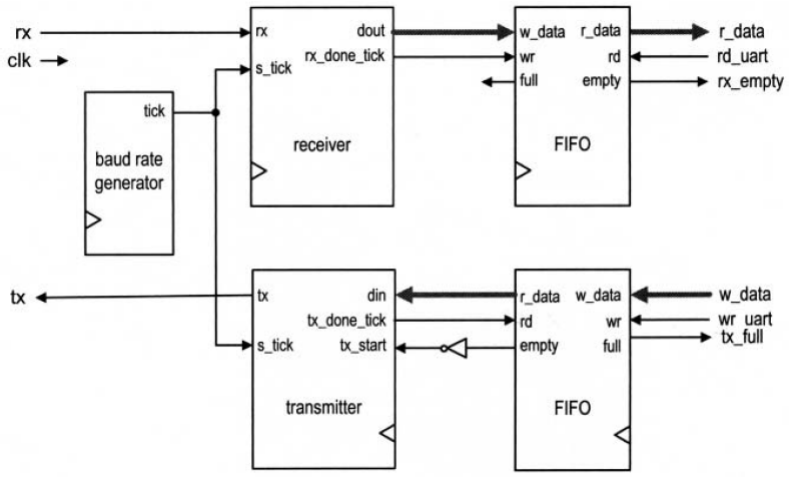


Figura 3.8: Diagrama de bloques de un UART completo [42].

El UART incluye un transmisor y un receptor. El transmisor es un circuito desplazador especial que carga datos en paralelo y después los mueve hacia fuera bit a bit a una frecuencia determinada. Por otro lado, el receptor mueve datos hacia dentro bit a bit y después reúne la información.
La línea serial (Figura 3.9) es '1' cuando se encuentra en el estado **idle**. La transmisión comienza con el bit de **start**, el cual es '0', seguido de los bits de **data** y un bit opcional de paridad y finaliza con un bit '1' de **stop**.

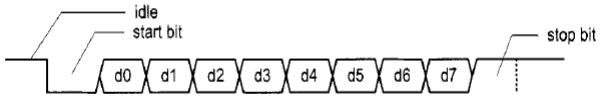


Figura 3.9: Línea serial para la transmisión de un byte [42].

Antes de que comience la transmisión, el transmisor y el receptor deben acordar un conjunto de parámetros listados a continuación:

- Frecuencia de baudios (número de bits por segundo)
- Número de bits de datos y bits de **stop**
- Bit de paridad.

3.3.2. FSM (Máquina de Estados Finitos)

Una FSM [42], es usada para modelar un sistema que transita entre un número finito de estados internos. Las transiciones dependen del estado actual y una entrada externa. A diferencia de un circuito secuencial

regular, la transición de los estados de una FSM no exhibe un patrón simple y repetitivo. Su lógica de estado siguiente es conocida como lógica aleatoria. En la práctica, la aplicación principal de una FSM es actuar como el controlador de un sistema digital grande, el cual examina los comandos y estados externos y activa las señales de control adecuadas para la operación de control de una ruta de datos, la cual, usualmente está compuesta de componentes secuenciales estándar.
Existen dos formas de representar la salida de una FSM (Figura 3.10), las cuales, son las listadas a continuación:

- **Salida Moore**: La salida es sólo una función del estado actual
- **Salida Mealy**: La salida es una función del estado actual y una entrada externa

Ambos tipos de salida pueden existir en una FSM compleja.

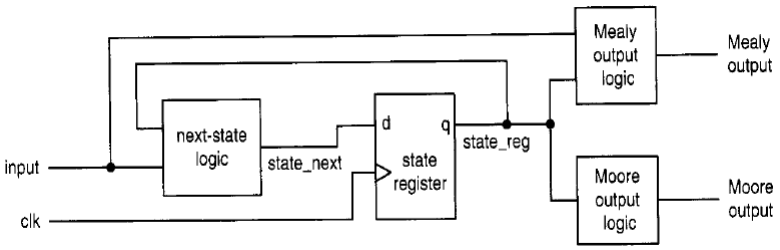


Figura 3.10: Diagrama de bloques de una FSM síncrona [42].

4.

Desarrollo e Implementación del proyecto

En este capítulo se presentan los bloques desarrollados para llevar a cabo la implementación del proyecto. En la figura 4.1 se ilustra el diagrama de bloques del sistema donde se muestra de forma abstracta cada elemento que lo componte, dando lugar a un diagrama representativo del funcionamiento que se tiene dentro del FPGA.

Se muestra una comunicación entre el FPGA contenido en una tarjeta de desarrollo amiba 2 y una PC que se comunica con un módulo convertidor de protocolos (FTDI) por USB y el FTDI se comunica con el FPGA por medio de RS-232. También, incorpora el protocolo JTAG, empleado para configurar el FPGA con el diseño implementado en VHDL.

La comunicación serial se implementa con el fin de poder recibir los valores de los píxeles de las imágenes que serán procesadas a través de los distintos bloques implementados, contenidos en el módulo acelerador. Los píxeles se reciben en formato de punto flotante de 24 bits y son almacenados temporalmente en la memoria externa hasta que el proceso de recepción de la imagen esté completo.

Se implementa el transmisor y receptor del circuito UART, esto con el fin de recibir las imágenes que serán usadas por el sistema y también para enviar los resultados obtenidos hacia la PC, donde su visualización y análisis será realizado de forma óptima.

El módulo acelerador contiene la operación de convolución, la función de activación ReLU, el operador pooling, el perceptrón y la unidad de punto flotante conformada por las operaciones suma, resta y multiplicación. La figura 4.2 muestra el diagrama de bloques del contenido del módulo acelerador, así como los bloques que serán implementados de color verde. También, en color rojo, se muestra la secuencia de las capas de la red neuronal convolucional.

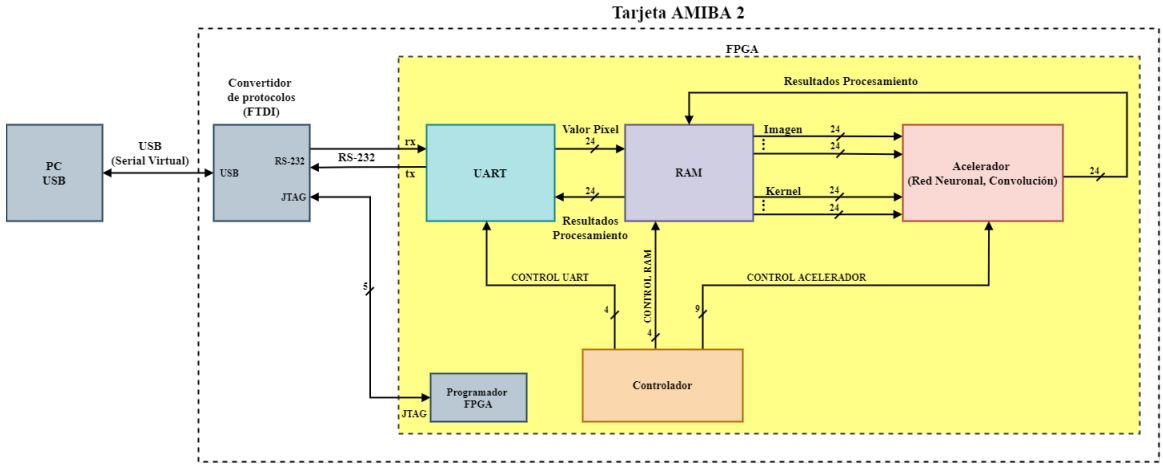


Figura 4.1: Diagrama de bloques de una Red Neuronal Convolucional implementada en un FPGA.

Dentro de cada bloque se explota la concurrencia que ofrece el FPGA y se tiene un proceso secuencial fuera de cada uno. El bloque de color gris, denominado como función de activación SOFTMAX no será implementado debido a la complejidad que representa, ya que entre las operaciones que necesita se encuentra el cálculo de una división de exponenciales.

Además, se puede observar que se tiene una secuencia de capas interconectadas, cada una con la capa que la precede.

Los datos de salida de cada conjunto de capas irán seleccionando características más significativas y también se irá reduciendo el volumen de datos que entran a una nueva capa de convolución, de tal manera que los datos que pasan por el Perceptrón Multicapa son los más representativos de la imagen de entrada.

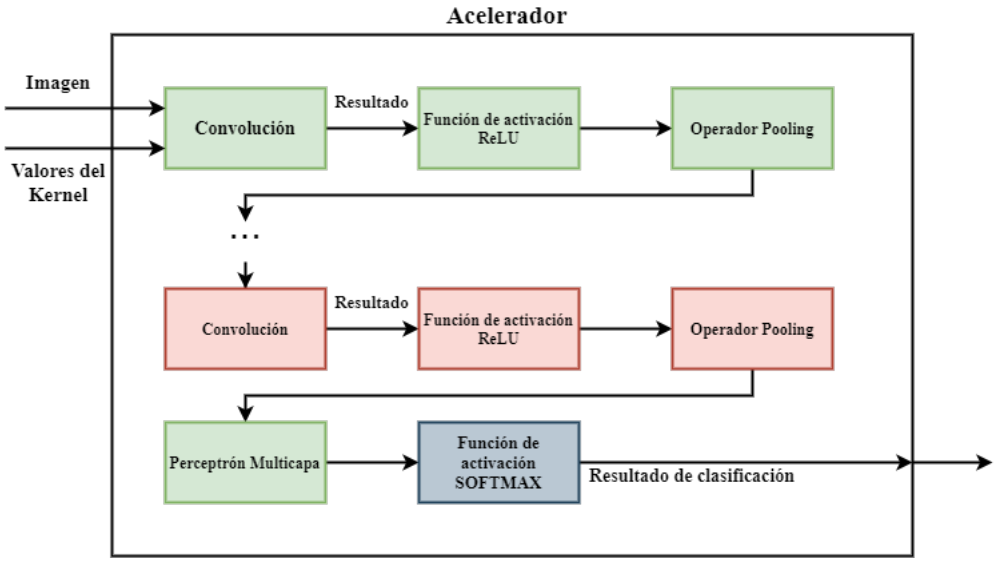


Figura 4.2: Diagrama de bloques del contenido del módulo acelerador.

El bloque ilustrado en la figura 4.3 describe el proceso propuesto para realizar la operación de convolución y a primera instancia, describe el funcionamiento de barrido del operador pooling. La imagen recibida es una imagen de forma cuadrada con un tamaño de $L \times L$ píxeles y la memoria RAM que almacena la imagen tendrá 24 columnas y $L \times L$ filas, manteniendo una estructura considerada como vector columna. Se mantiene un número P de apuntadores que es definido por el número de pesos K del kernel y se realiza un salto de tamaño L entre cada apuntador, para representar el cambio de columna del Kernel por la superficie de la imagen. El módulo *Producto punto* contiene los multiplicadores, sumadores de punto flotante y aplica la función de activación ReLU al valor obtenido. Por último el módulo *Volumen de Activación* almacena el resultado obtenido para que sea transferido a un nuevo conjunto de capas.

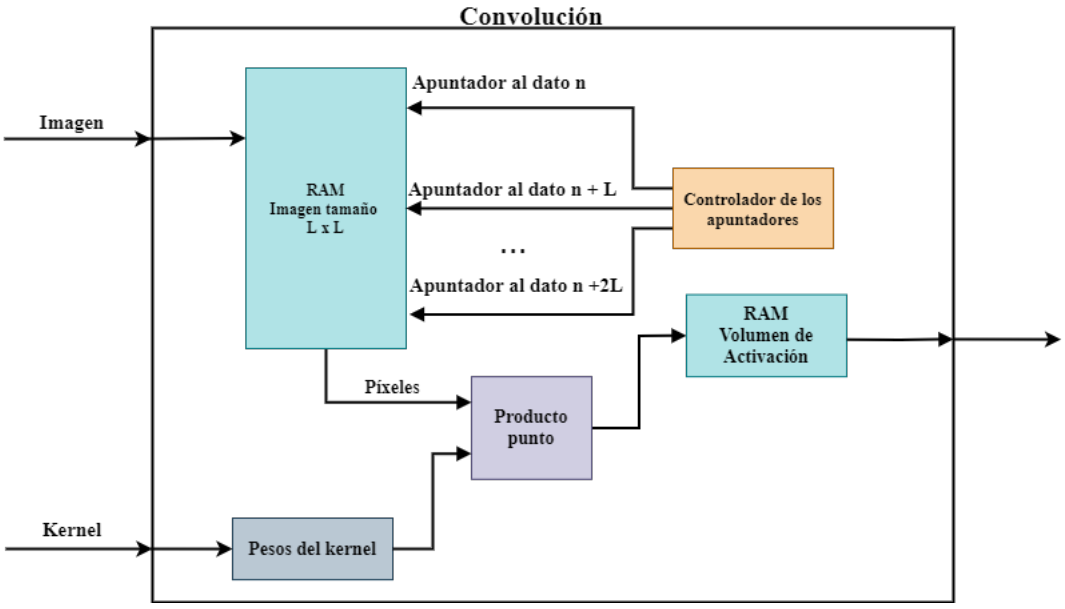


Figura 4.3: Diagrama de bloques de la operación de convolución.

4.1. Comunicación Serial

A continuación, se describe el proceso que fue llevado a cabo para implementar cada módulo que forma parte de la comunicación serial, además, se incluye la implementación de una interfaz gráfica bajo el lenguaje de programación python para gestionar la comunicación en una computadora.

En la PC se realiza la normalización de la imagen que será utilizada en el FPGA, el proceso se describe en la imagen 4.4.

La normalización consiste en tener los valores de los píxeles entre 0 y 1, los cuales, antes de la normalización se encuentran en la escala 0 a 255, puesto que están en el formato de escala de grises. Los valores de los píxeles de la imagen normalizada se representan en formato de punto flotante de 24 bits.

En el FPGA se recibe la imagen en formato de punto flotante y se almacena en memoria externa, donde será accesible para cualquier módulo que necesite hacer operaciones sobre la misma.

Por último, cuando se haya terminado el proceso de operaciones aplicadas sobre la imagen, el resultado obtenido será enviado hacia la PC para validar la efectividad de las operaciones.
En la figura 4.5, se muestra el diagrama esquemático del circuito que se encarga de gestionar la comunicación, dentro del mismo, se incluye un módulo divisor de frecuencia denominado *Pulso_Muestreo* que permite tener una velocidad de transferencia a 19,200 baudios.

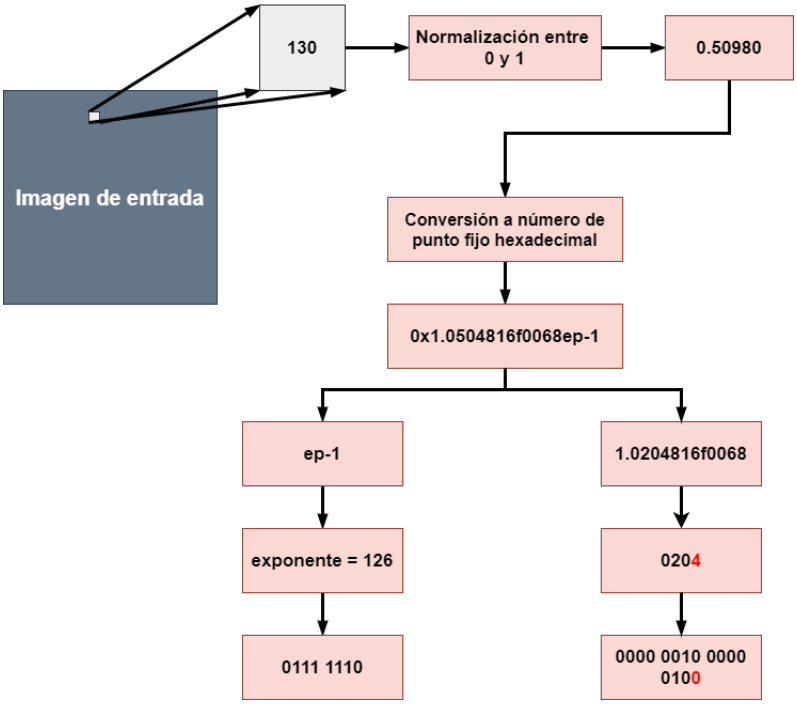


Figura 4.4: Proceso que se lleva a cabo para normalizar la imagen con valores de punto flotante entre 0 y 1.

El módulo *IP_UART* contiene los circuitos receptor y transmisor del circuito UART y cada uno de estos circuitos hacen uso de un registro que los ayuda a almacenar el dato de 24 bits en 3 datos de 8 bits, de tal manera que la transmisión y recepción de la información sea posible, pues sólo es posible enviar hasta 8 bits en cada transmisión, los registros son *Reg_Transmisor* y *Reg_Receptor*.
La recepción se realiza en dos etapas, la primera consiste en recibir la cantidad de píxeles contenidos en la imagen, es decir, el tamaño de la imagen, y la segunda consiste en recibir los datos que forman parte de la imagen, es decir, los valores de los píxeles. Una vez que se ha recibido el primer dato, éste sale por la señal *size* y los datos que serán recibidos después son considerados como *data* y se transfieren al registro del módulo receptor, donde se espera a que el registro esté lleno para que a la salida se tenga el dato de 24 bits.

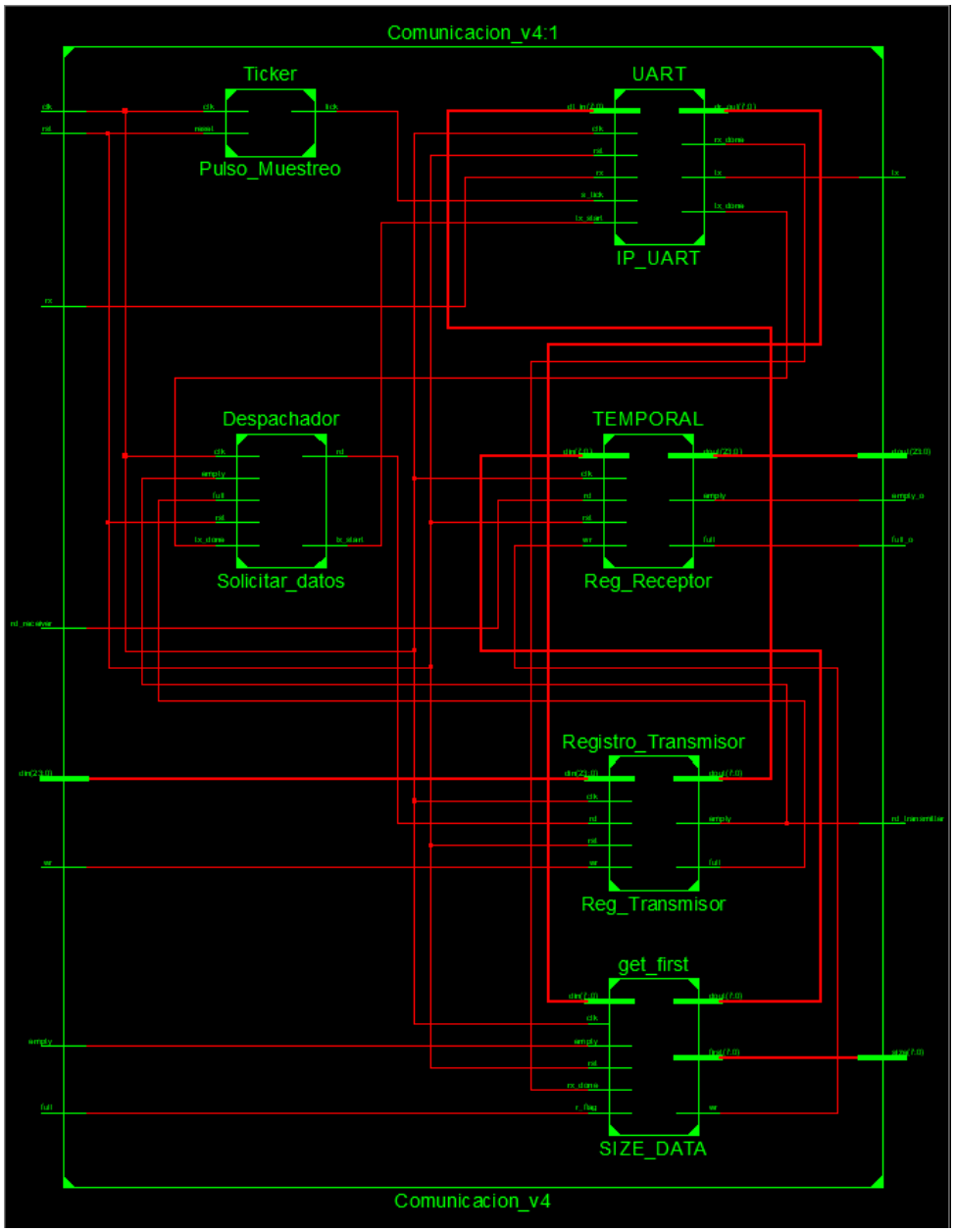


Figura 4.5: Diagrama esquemático del módulo que se encarga de la comunicación entre el FPGA y la CPU para envío y recepción de imágenes en formato de punto flotante de 24 bits.

Para la transmisión de la imagen, se recibe un dato de 24 bits, se divide en 3 bytes y se almacena en 3 localidades de 1 byte cada una, con ayuda del módulo *Solicitar_datos* se tiene el control de la transmisión de los 3 bytes que forman parte del dato, puesto que cada vez que se termine de transmitir un byte, se solicita el siguiente hasta que el registro se encuentre vacío; cuando esto pasa se solicita un nuevo dato para ser transmitido.

4.1.1. Implementación del UART

La comunicación serial se implementó para comunicar el FPGA con una computadora, con el motivo de poder enviar las imágenes y los filtros, ambos en formato de punto flotante de 24 bits. También se emplea para recibir los resultados de cada bloque implementado y de esta manera poder transmitirlos hacia la computadora.

La figura 4.6 muestra la implementación del circuito *IP_UART*, el cual contiene el módulo transmisor y receptor.

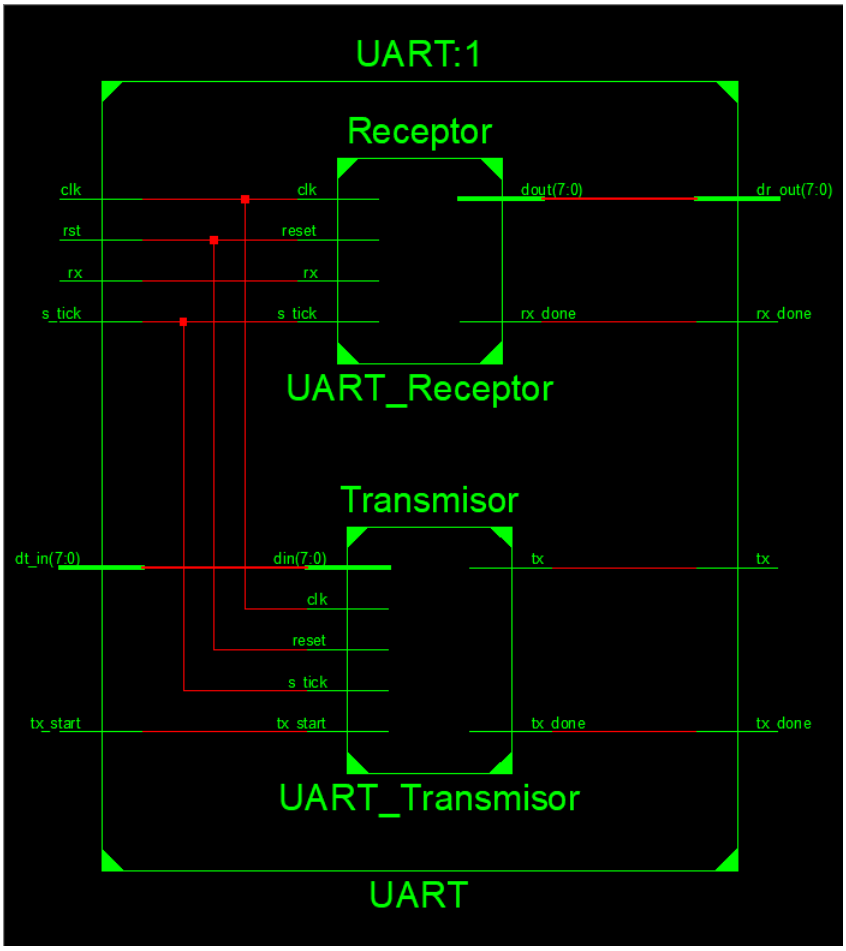


Figura 4.6: Diagrama esquemático del circuito UART.

Circuito Receptor

Para realizar pruebas con el circuito receptor, se implementó un circuito que incluye el divisor de frecuencia, el circuito receptor y el registro para almacenar los datos recibidos. El esquemático del circuito receptor con los componentes se muestra en la figura 4.7. El circuito *Ticker* proporciona un pulso que sincroniza el circuito receptor para que sea capaz de realizar un muestreo de 16 veces al bit de entrada. La velocidad de transferencia asignada al diseño es de 19,200 baudios, lo que significa que en un segundo es

capaz de transmitir 19,200 bits. La ecuación 4.1 determina la frecuencia con la que se envía un pulso hacia el circuito receptor.

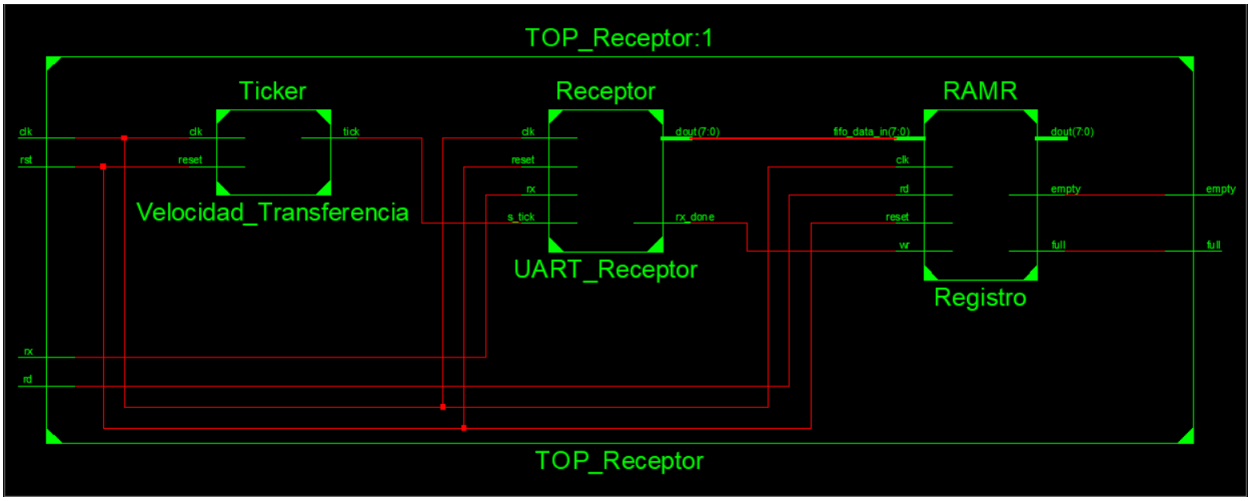


Figura 4.7: Esquemático del circuito receptor de la comunicación UART.

Como se mencionó anteriormente, es necesario mantener un muestreo del bit que está siendo recibido para incorporarlo al byte final cuando el bit recibido se encuentra a la mitad de la transferencia y garantizar que se toma en el punto más estable.

$$f = \frac{V}{B \times F_m} \tag{4.1}$$

Donde, V es la frecuencia de trabajo de la tarjeta que se va a utilizar, B es la velocidad de transferencia en baudios y F_m es el factor de muestreo, que en este caso será igual a 16.
De la ecuación 4.1 se tiene:

$$f = \frac{50MHz}{19200 \times 16} \tag{4.2}$$

dando como resultado una $f = 163$, lo que significa que el circuito *Ticker* emitirá un pulso cada 163 ciclos del reloj. Como la tarjeta con la que se han realizado pruebas tiene un reloj de 50MHz, tiene un ciclo de trabajo de 20 ns, de tal manera que el pulso que se emitirá por el circuito *Ticker* será cada 3.26 μs.
En la figura 4.8 se muestra la simulación del circuito receptor, como se puede observar en la señal x_dout se muestran 10 dígitos que han sido recibidos, lo que indica que el circuito funciona adecuadamente.

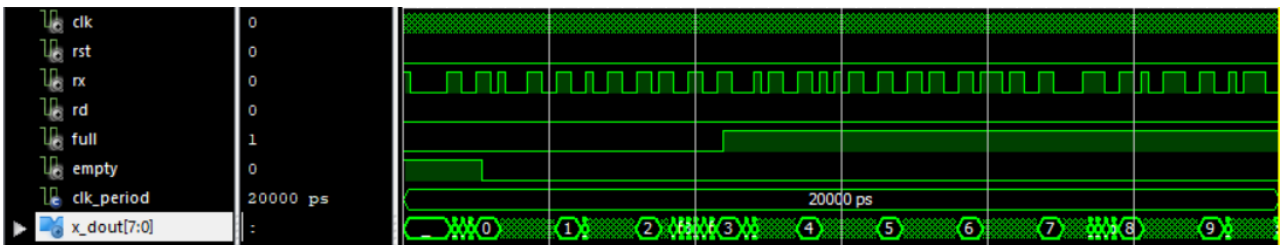


Figura 4.8: Simulación del circuito receptor.

Circuito Transmisor

De la misma manera, se realizó la parte transmisora del circuito UART. La figura 4.9 muestra el esquemático del circuito transmisor. Donde se tienen 4 circuitos interconectados, de la misma manera que con el circuito receptor, se mantiene un divisor de frecuencia denominado *Ticker* con las mismas características del que fue usado con el módulo receptor, el circuito divisor de frecuencia permite sincronizar los circuitos.
También, se tiene una máquina de estados (FSM) para comprobar la transmisión de los datos. La FSM consta de 4 estados que se encargan de mandar 1 caracter en cada ciclo de reloj, mismos que forman la palabra **"HOLA"**.
El caracter enviado de la FSM hacia el transmisor es almacenado en un registro, una vez que el registro contenga un dato se emitirá una señal hacia el circuito transmisor, mismo que tomará el dato contenido en el registro y comenzará a transmitirlo bit a bit a través de la señal *tx*.

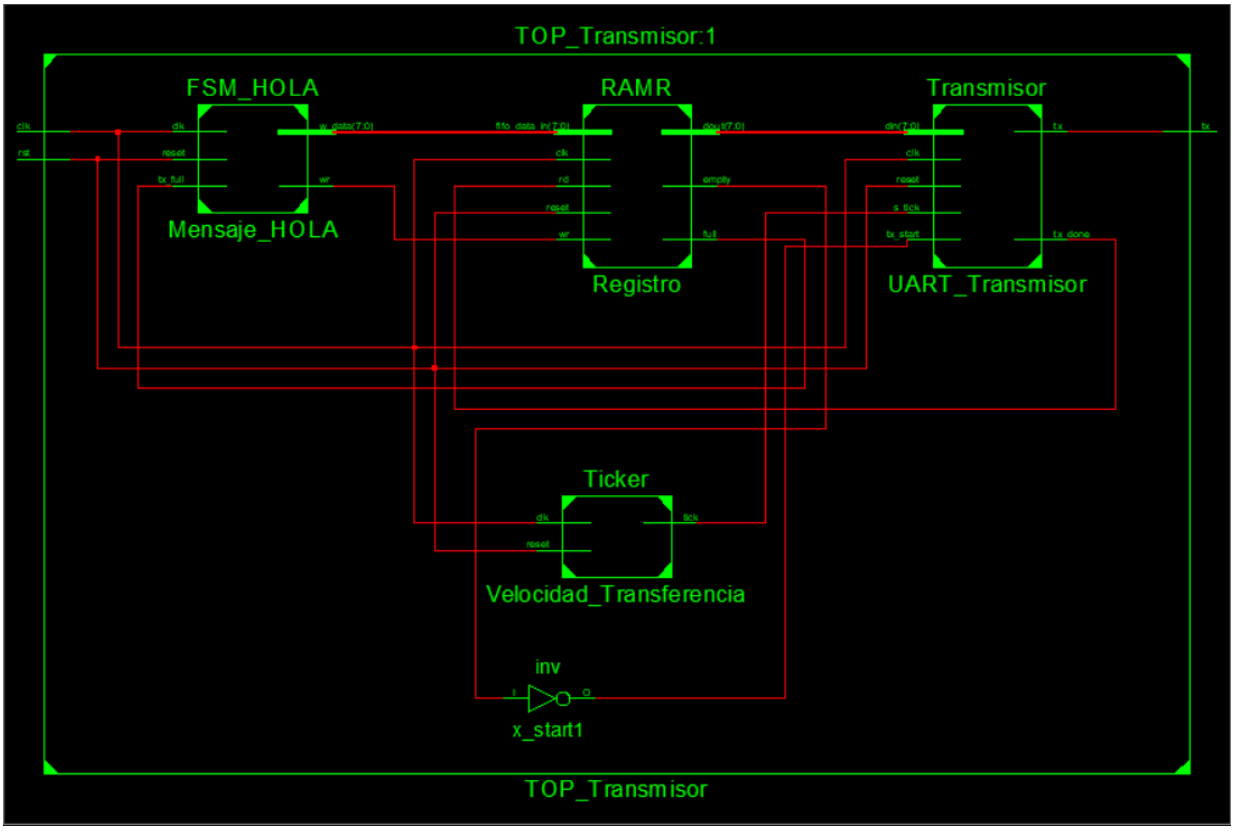


Figura 4.9: Esquemático del circuito transmisor de la comunicación UART.

En la figura 4.10 se muestra la simulación del funcionamiento adecuado del circuito transmisor. Se puede observar que la señal *din* recibe un caracter cada que se cumplen ciertas condiciones establecidas en el módulo transmisor, pues tiene que enviar cada bit que forma parte del caracter, una vez que se ha completado la transferencia, se enciende la señal *tx_done*, además, se observa que la transmisión del caracter se realiza

durante el estado data de la máquina de estados finitos (FSM) que se tiene en el circuito transmisor.
De esta manera, se puede avalar que el funcionamiento del circuito transmisor se realiza de manera adecuada.

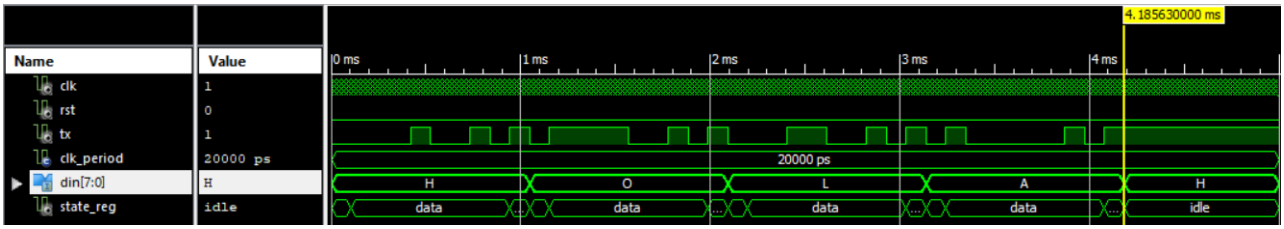


Figura 4.10: Simulación del circuito transmisor.

Registros para el transmisor y el receptor

Al inicio de esta sección, se mencionó que el módulo encargado de la comunicación de datos contenía un registro tanto para el receptor como para el transmisor.

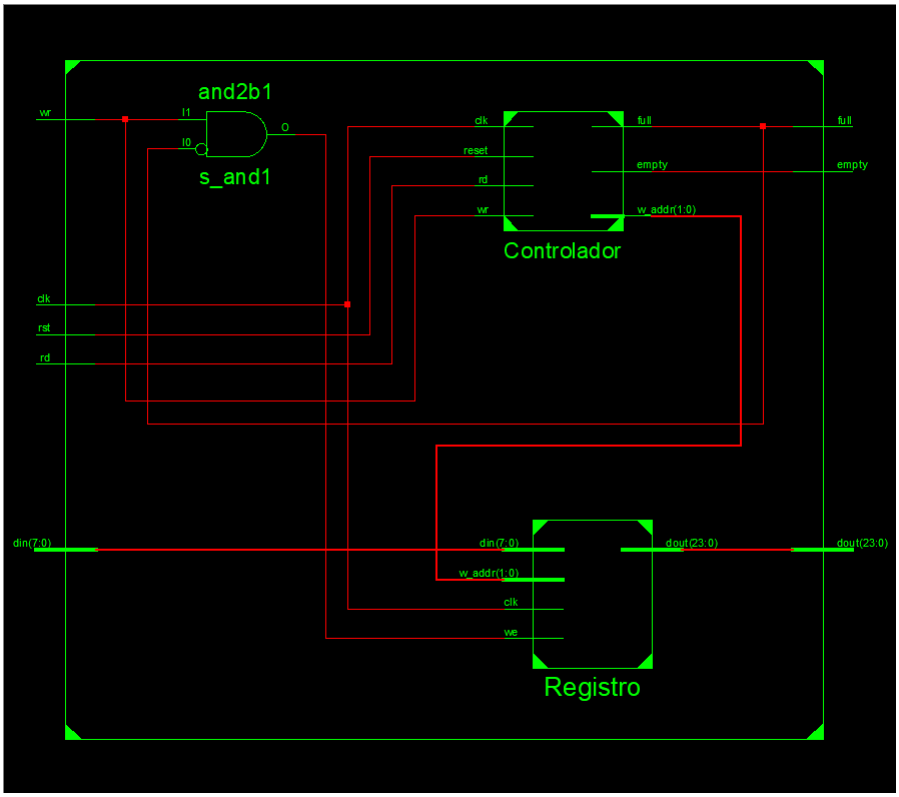


Figura 4.11: Diagrama esquemático del registro encargado de almacenar los datos recibidos.

El diseño del módulo se realizó para que fuera posible recibir números de punto flotante en formato de 24 bits. Es bien sabido que el módulo UART sólo recibe y envía datos de hasta 8 bits a la vez. Por lo tanto, fue necesario diseñar un circuito capaz de juntar los datos recibidos que forman parte de un número en la etapa de recepción y también, de separar en 3 bytes los datos que van a ser enviados en la etapa de la transmisión.

A continuación, se presentan los dos registros diseñados para poder enviar y recibir números de punto flotante con ayuda del módulo UART.
La figura 4.11, representa el circuito del registro que se encarga de almacenar los datos recibidos. Consta de 5 señales de entrada y 3 señales de salida. Entre las señales de entrada, se encuentra una señal de reloj, una señal de reinicio, una señal para escribir datos, otra para leer datos y por último, una señal para recibir datos de 8 bits. Las señales de salida se componen por una señal para indicar que el registro se encuentra lleno, una señal para indicar que el registro se encuentra vacío y una señal de 24 bits que proporciona el número de punto flotante recibido.

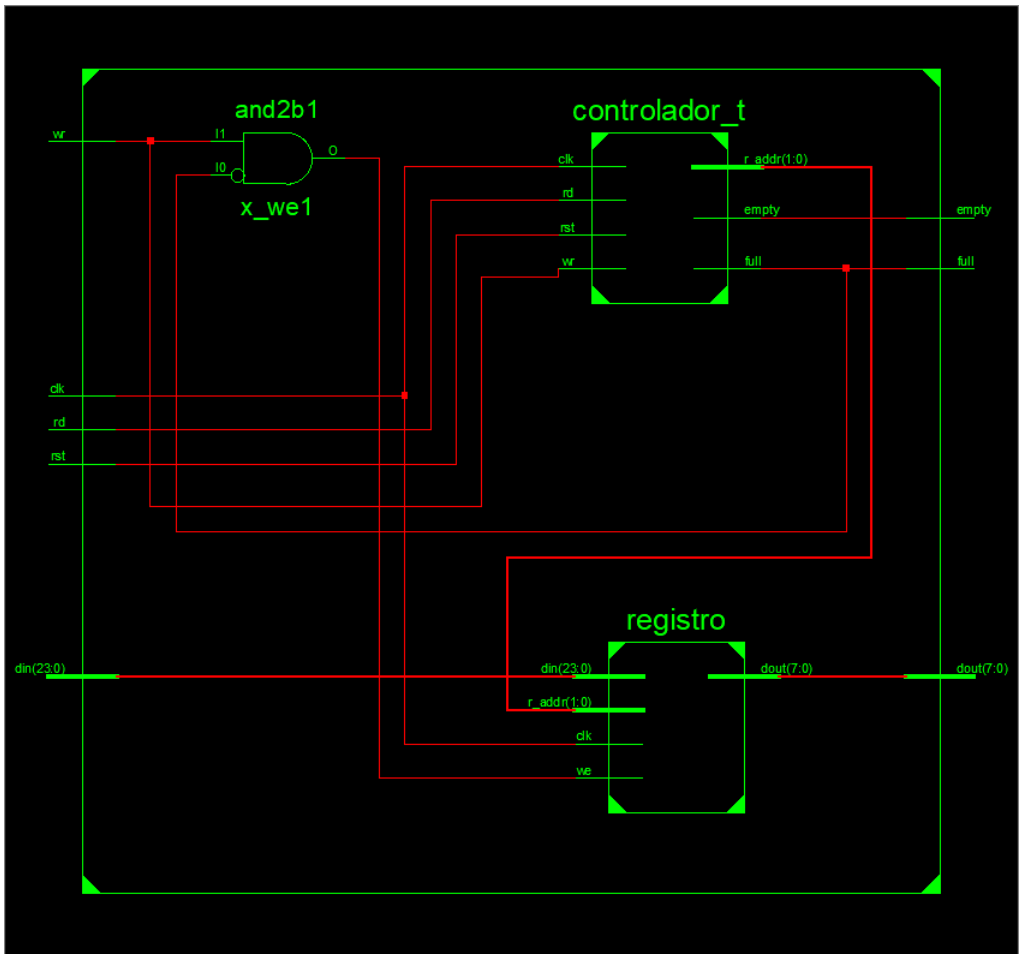


Figura 4.12: Diagrama esquemático del registro encargado de almacenar los datos que serán transmitidos.

El funcionamiento del registro receptor se pensó específicamente para recibir números de punto flotante de 24 bits a través de la línea serial. El registro se encarga de almacenar y retener conjuntos de 3 datos de 8 bits y concatenarlos para proporcionar un dato de 24 bits a la salida una vez que la señal que indica que el registro se ha llenado presente un valor igual a *'1'*.
Cuando esto pasa, se realiza una solicitud de lectura, se toma el dato de 24 bits y se le asigna un valor de *'1'* a la señal que indica que el registro está vacío, y el valor de la señal que indica que el registro está lleno se

cambia a *'0'*. Por otra parte, se tiene el registro encargado de almacenar los datos que serán transmitidos, el diagrama esquemático de este registro se presenta en la figura 4.12.
El registro que almacena temporalmente los datos que serán transmitidos tiene 5 señales de entrada, entre las que se encuentran, una señal de reloj, una señal de reinicio, una señal para solicitar la escritura de datos, una señal para solicitar la lectura de datos y por último, una señal de entrada de datos de 24 bits. A la salida se tiene una señal para indicar que el registro está vacío y otra para indicar que se encuentra lleno, y una señal de salida de datos de 8 bits.
El funcionamiento de este registro consiste en recibir un dato de 24 bits, internamente el dato se separa en 3 bytes y se almacenan en 3 localidades del registro. Cada vez que se lee un dato, el registro se establece con la señal de lleno en *'1'*. El módulo *despachador de datos* se encarga de vaciar el registro y será explicado más adelante. Cuando el registro se ha vaciado, la señal de vacío se establece en *'1'* y se espera hasta que se solicite la escritura de un nuevo dato. Como se puede hacer notar, el funcionamiento de los registros se diseñó e implementó pensando en el envío y recepción de datos de 24 bits, esto para facilitar el uso de números de punto flotante en el FPGA y sacando ventaja del uso de la PC para la conversión tanto de valores decimales a punto flotante para las imágenes enviadas hacia el FPGA, como de valores de punto flotante a valores decimales para analizar los resultados obtenidos.

Circuito despachador de datos

Para garantizar un almacenamiento y transmisión adecuados se diseñó una máquina de estados, cuyo funcionamiento tiene como propósito esperar a que se reciban los datos y sean almacenados en la memoria RAM.

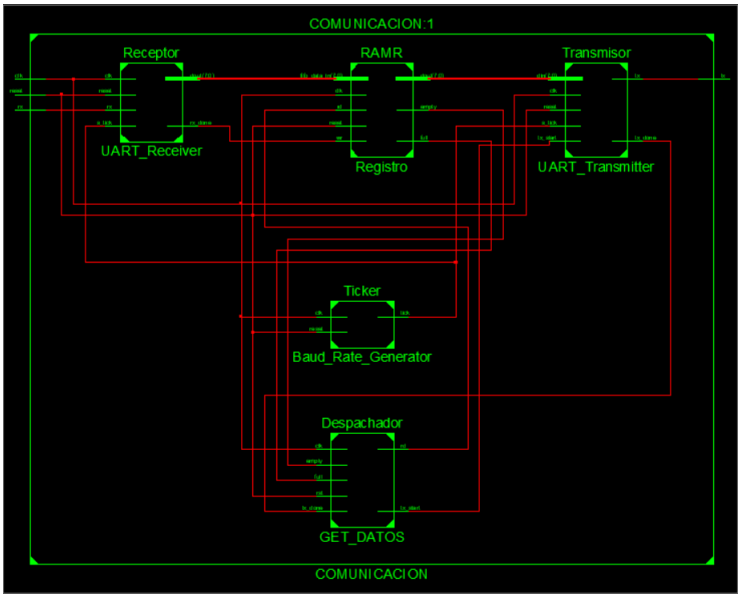


Figura 4.13: Diagrama esquemático del UART con el circuito que lee los datos de la memoria RAM.

Una vez que la memoria RAM está llena se envía una señal hacia la máquina de estados por medio del

puerto “Full”, cuando la máquina de estados recibe la señal, ésta comienza a extraer los datos hasta que se enciende la señal “Empty”, entonces regresa al estado inicial y espera a que la señal “Full” se encienda otra vez. El módulo añadido se nombró **“Despachador”**, como se muestra en la figura 4.13.
Así mismo, se muestra la carta ASM (Figura 4.14) que sigue la máquina de estados diseñada para extraer los datos, la cual está contenida en el circuito despachador. La máquina de estados tiene 3 estados, los cuales son *Idle, Solicitar y Vaciar*.

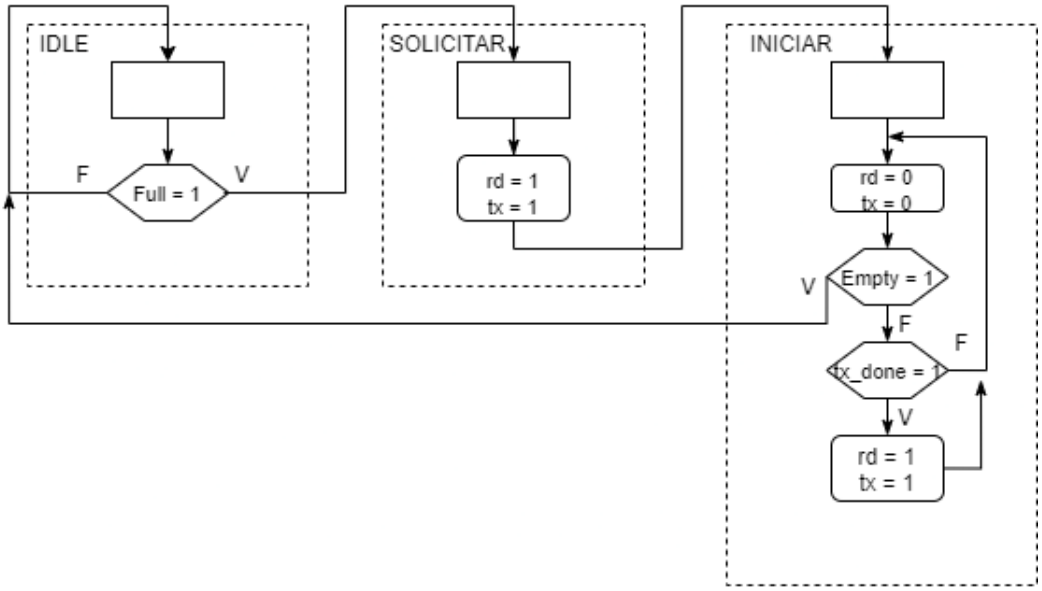


Figura 4.14: Carta ASM de la máquina de estados que controla el circuito “Despachador”.

El estado *Idle* se encarga de mantener la máquina de estados en espera hasta que la señal *Full*, proveniente de la memoria RAM, se encienda. Una vez que la señal *Full* de la memoria RAM se enciende, significa que la imagen se ha almacenado completamente y ya no se reciben más datos; llegando a este punto la memoria RAM está lista para entregar los datos que contiene y se pasa al siguiente estado. Cuando la máquina de estados pasa al estado *Solicitar* ésta realiza una lectura de la memoria RAM y también activa la señal que le indica al circuito transmisor que debe iniciar la transmisión de un dato, después de realizar esta acción se pasa al siguiente estado, denominado como *Iniciar*.
Cuando se finaliza la transmisión del dato, la señal *tx_done* del circuito transmisor es activada y es empleada por la máquina de estados para poder solicitar otro dato a la memoria RAM y así iniciar una nueva transmisión. Una vez que se han leído todos los datos contenidos en la memoria RAM, la señal *Empty* se activa y le indica a la máquina de estados que ya no hay más datos que leer, entonces ésta regresa al estado inicial y vuelve a esperar hasta que la memoria RAM esté llena.
El circuito despachador se diseñó con la finalidad de garantizar una extracción y transmisión de datos del registro, que en este caso es una memoria RAM implementada con los recursos que ofrece el FPGA. Resulta un módulo muy útil, pues permite solicitar datos a de memoria externa cada vez que el registro encienda la

señal que indica que se encuentra vacío y trabaja en conjunto con el módulo transmisor del circuito UART para transmitir los bytes contenidos del número de punto flotante solicitado.

4.1.2. Interfaz Gráfica

Con el motivo de facilitar el envío y la recepción de los datos, se implementó una interfaz gráfica que permite visualizar las imágenes enviadas y el resultado de las operaciones aplicadas en el FPGA. En la figura 4.15 se muestra el diseño inicial del flujo que seguiría la interfaz para establecer la comunicación.
Para poner a prueba la comunicación serial simplemente se envió una imagen con un tamaño de 60×60 píxeles, lo que da un total de 3,600 píxeles enviados. La interfaz gráfica se implementó en el lenguaje de programación Python haciendo uso del IDE de programación Spyder.
La comunicación consiste en abrir un puerto serial, se colocaron algunos elementos que ayudan a seleccionar los parámetros de configuración de la comunicación incluyendo el puerto COM por el cual se realizará la comunicación. Una vez que se ha establecido la conexión, el botón **conectar** permanecerá inhabilitado, como símbolo de que la comunicación ha sido iniciada.
Después se selecciona la imagen para ser enviada al FPGA; el botón **enviar** tiene la tarea, como su nombre lo indica, de enviar la imagen seleccionada a través del puerto serial.

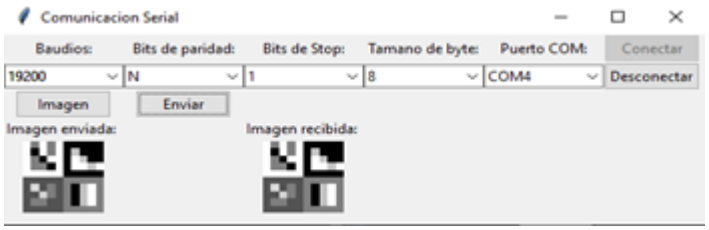


Figura 4.15: Interfaz gráfica implementada para gestionar la comunicación serial.

4.2. Unidad de Punto Flotante

La unidad de punto flotante consta de 4 operaciones fundamentales, como son: suma, resta, multiplicación y división. En este apartado se describe la implementación de cada una de estas operaciones. En la tabla 4.1 se describe el número de bits que necesita cada parte de la representación del número en punto flotante así como el rango de números que pueden ser representados con cada formato.

Tabla 4.1: Rango de formatos de punto flotante [43].

Formato	Bits totales	Bits significativos	Bits exponente	No. menor	No. mayor
Precisión sencilla	32	23 + 1 signo	8	1.2 e^{-38}	3.4 e^{38}
Precisión doble	64	52 + 1 signo	11	5.0 e^{-324}	1.8 e^{308}
bfloat16	16	7 + 1 signo	8	1.166 e^{-38}	1.68 e^{38}
float24	24	15 + 1 signo	8	1.175 e^{-38}	1.7 e^{38}

4.2.1. Suma/Resta

En la figura 4.16, se muestra el esquemático y cada uno de los módulos que forman parte del circuito sumador y restador. Este circuito consta de 5 módulos que permiten realizar las operaciones de manera adecuada. El circuito recibe 2 números de 24 bits, donde se incluye el bit de signo, 8 bits para el exponente y 15 bits para la mantisa o fracción.

La primera etapa consiste en verificar qué número es más grande, el circuito **"Mayor"** se encarga de comparar los bits del 22 al 0 y también divide los bits para que los números salgan en el formato de signo, exponente y mantisa.

La etapa siguiente consta del módulo que se encarga de alinear los números, el circuito se denomina **"Alinear"**. Primero, realiza una resta de los exponentes con el fin de conocer cuántas unidades es más grande el exponente mayor del exponente menor y el número de ceros que se agregan a la izquierda del número menor es igual al resultado de la resta, de esta manera, los números han sido alineados y su exponente es el mismo. Una vez que se ha terminado el proceso, el número alineado es pasado al circuito que se encarga de realizar la operación, denominado **"Suma_Resta"**, la operación que se realiza, ya sea suma o resta, está relacionada con el signo de los números, de tal manera que si los signos son iguales se realiza una suma y si los signos son diferentes entonces se realiza una resta. Se debe hacer notar que el resultado de la operación se almacena en una señal con un bit más que los que ocupan los números que están involucrados en la operación, en este caso los números ocupan 15 bits, por lo que el resultado se almacena en 16 bits. El siguiente módulo, denominado **"Normalización"**, se encarga de que el resultado esté normalizado, esto quiere decir que el bit más significativo de la mantisa debe contener un uno. Si existe un acarreo, es decir, que el bit más significativo del resultado sea 1, entonces se toman los 15 bits más significativos y el exponente se conserva, de otra manera, se verifica que el número de ceros que se encuentran en los bits más significativos no sea mayor al exponente, de ser así, todo el resultado es igualado a cero; si ninguna de estos dos caso es cumplido, entonces se cuenta el número de ceros que se encuentran en los bits más significativos y se agregan los mismos ceros a la izquierda, para conservar los 15 bits que forman parte de la mantisa. El número de ceros resultante es restado al exponente y de esta forma se tiene el exponente y la mantisa de forma normalizada.

Por último, se unen todos los bits con ayuda del circuito **"UnionBits"** y se proporciona el resultado en formato de 24 bits.

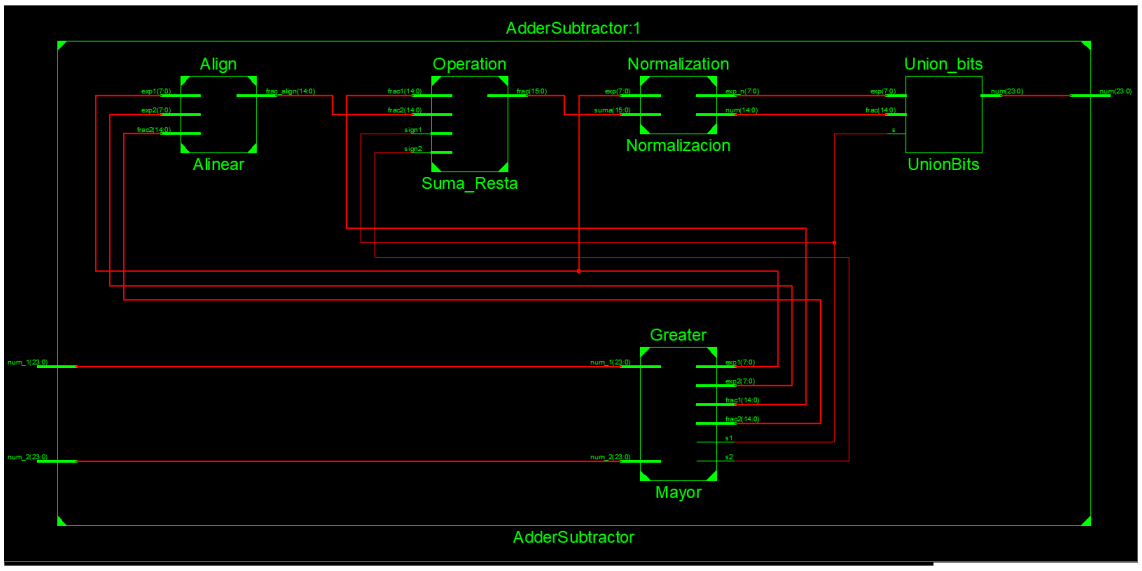


Figura 4.16: Esquemático del circuito suma/resta.

4.2.2. Multiplicación

El circuito multiplicador consta de 8 módulos internos, el primero se encarga de separar los bits, puesto que la entrada se basa en la recepción de dos señales de 24 bits, es necesario separar los bits para que se pueda procesar con el formato de signo, exponente y mantisa. De forma concurrente, se revisa que ninguno de los números sea cero para que no se realice el proceso y simplemente se mande una señal con ceros a la salida. Si ningún número es igual a cero, entonces se procede con las etapas que son ejecutadas de forma concurrente denominadas como **"Signo"**,**"Exponente"** y **"Mantisa"**.

La etapa de **"Signo"** se encarga de obtener el signo del resultado a partir de los signos de los dos números, éste se obtiene mediante una compuerta lógica xor. La etapa denominada como **"Exponente"** realiza una suma de los dos exponentes. A uno de los exponentes se le resta un 127 antes de realizar la suma, con el fin de mantener las características del formato definidas para el exponente. La etapa denominada como **"Mantisa"** se encarga de la multiplicación de la mantisa de los dos números, el resultado se almacena en una señal de 30 bits.

En la etapa posterior, se realiza la selección de los bits que formarán parte de la mantisa, específicamente en la etapa de **"Normalizacion"**. La etapa de normalización se encarga de truncar el resultado de 30 bits a los 18 bits que contienen más significado del número resultante, es decir, verifica el bit más significativo diferente de cero y a partir de ese bit toma los 18 bits que continúan en la señal. Después, se pasa a la etapa de redondeo, donde se comparan los 3 bits menos significativos y se le suma un uno en aquella posición menos significativa que contenga un 1, de esta manera se garantiza que esa información, proporcionada por los bits menos significativos, estará contenida en la cadena de bits final.

A esta técnica de redondeo se le conoce como "redondeo al número más cercano". Por último, la normalización se encarga de verificar que en los bits más significativos no se contenga un 0, de ser así, se realiza

el mismo proceso que en el módulo de la suma-resta, donde se cuenta el número de ceros que se contienen en los bits más significativos y se ajusta el exponente para desplazar el primer 1 hasta la posición del bit más significativo, después se agrega el número de ceros equivalentes al número de posiciones que han sido desplazadas.

Una vez que se han realizado las operaciones y se obtiene el resultado final, éste se pasa a un módulo denominado **"Union"**, donde se unen el bit obtenido de signo, los bits obtenidos del exponente y los bits obtenidos de la mantisa, la cadena de 24 bits se pasa al módulo **"Selector "** para elegir el resultado final.

El resultado final depende del módulo que se encuentra en las primeras etapas, encargado de verificar que ningún número sea igual a cero. Cuando ningún número es igual a cero se realiza el proceso descrito anteriormente y el resultado calculado es la salida que se obtiene de este circuito.

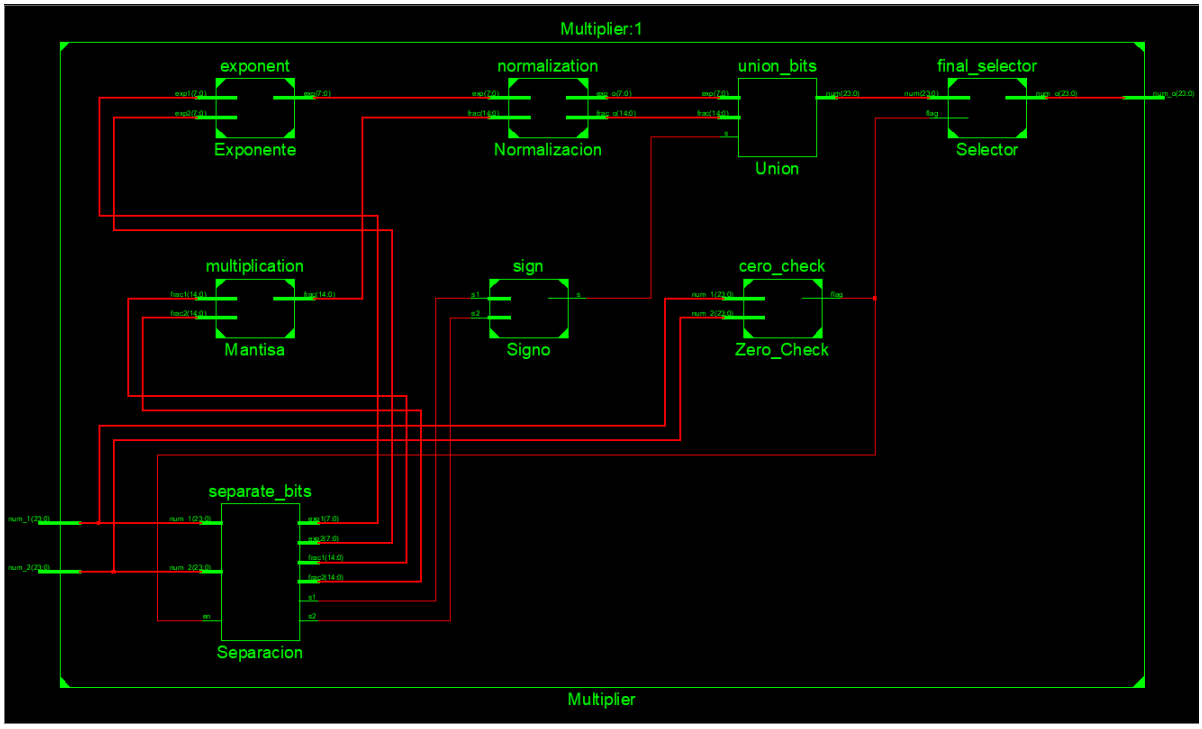


Figura 4.17: Esquemático del circuito multiplicador.

4.3. Función de Activación ReLU

La función de activación Rectified Linear Unit (ReLU) (figura 4.18), está constituida por un multiplexor cuyo selector es el bit más significativo de la señal de entrada, la cual consta de 24 bits. El bit más significativo representa al signo del número de punto flotante que se tiene a la entrada, de tal manera que se tiene un número positivo si el bit 23 es igual a 0 y si se tiene un número negativo entonces el bit 23 es igual a 1.

La función de activación retorna el número que entra si el número es positivo y retorna una señal de 24 ceros si el número que entra es negativo.

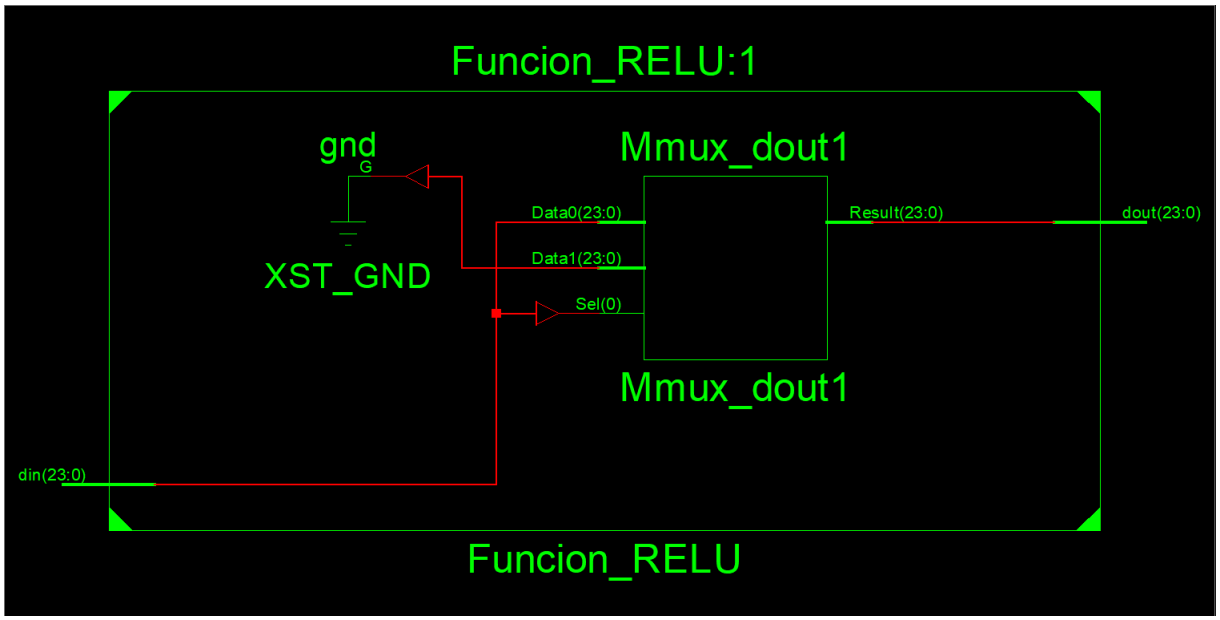


Figura 4.18: Diagrama esquemático de la función de activación ReLU.

La simulación de la función de activación ReLU (figura 4.19), muestra el funcionamiento adecuado del circuito. Como se representa en la simulación, el valor de entrada se mantiene si el bit más significativo es igual a cero, si es de otra manera, la salida del circuito otorga una señal constituida por 24 ceros.

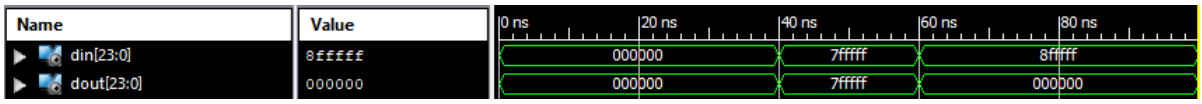


Figura 4.19: Simulación de la función de activación ReLU.

4.4. Operación de Convolución

El módulo que se encarga de realizar la operación de convolución se muestra en la figura 4.20. El módulo emplea una memoria ROM para almacenar los valores del kernel y otra para almacenar la imagen recibida, también se tiene un módulo denominado producto punto que alberga los módulos diseñados en la unidad de punto flotante con el motivo de realizar las operaciones necesarias y un módulo controlador de la convolución que se encarga de extraer los datos de la memoria que almacena los valores de los píxeles de la imagen y adquirir el resultado del producto punto, además se encarga de realizar el desplazamiento tanto horizontal como vertical una vez que el kernel ha alcanzado los límites de la imagen. La tarea que realiza el controlador es gestionada por una máquina de estados que será explicada más adelante.

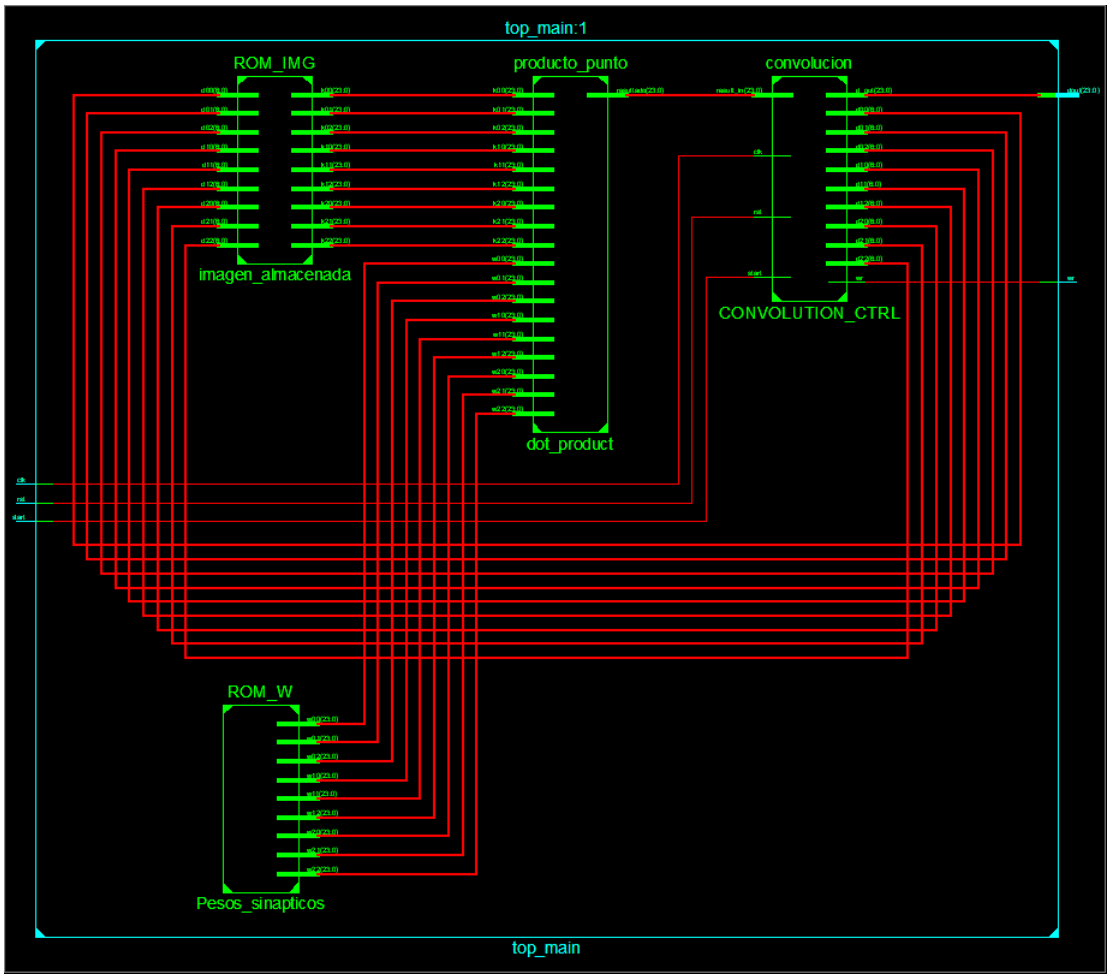


Figura 4.20: Esquemático de la operación de convolución.

La figura 4.21 muestra el proceso que se lleva a cabo para realizar el desplazamiento horizontal, debido a que se tiene una memoria con la forma de un arreglo columna, se decidió realizar el diseño de tal manera que no se modificara la distribución espacial de la memoria, es decir, que no se convirtiera en una matriz de tamaño $n \times n$, así que se realizó un mapeo de los datos tomando en cuenta las dimensiones de la imagen y el tamaño del kernel. Para realizar el diseño, se consideró una imagen de tamaño 10×10 y un kernel de tamaño 3×3.

Con la ecuación 4.3 se obtuvo el tamaño del arreglo resultando así como el número de desplazamientos que se iban a hacer tanto horizontal como verticalmente. Los parámetros que se toman en cuenta en la ecuación son los siguientes: W es el ancho de la imagen, F es la dimensión del kernel, P es el espacio de relleno en los bordes y S es el desplazamiento que tiene el kernel sobre la superficie de la imagen.

$$\frac{(W - F + 2 * P)}{S + 1} \tag{4.3}$$

En el diseño, los valores obtenidos son los siguientes:

- W = 10

- F = 3
- P = 0
- S = 1

Lo que nos da como resultado un vector resultante de 8×8 y un total de 7 desplazamientos horizontales y 7 verticales.

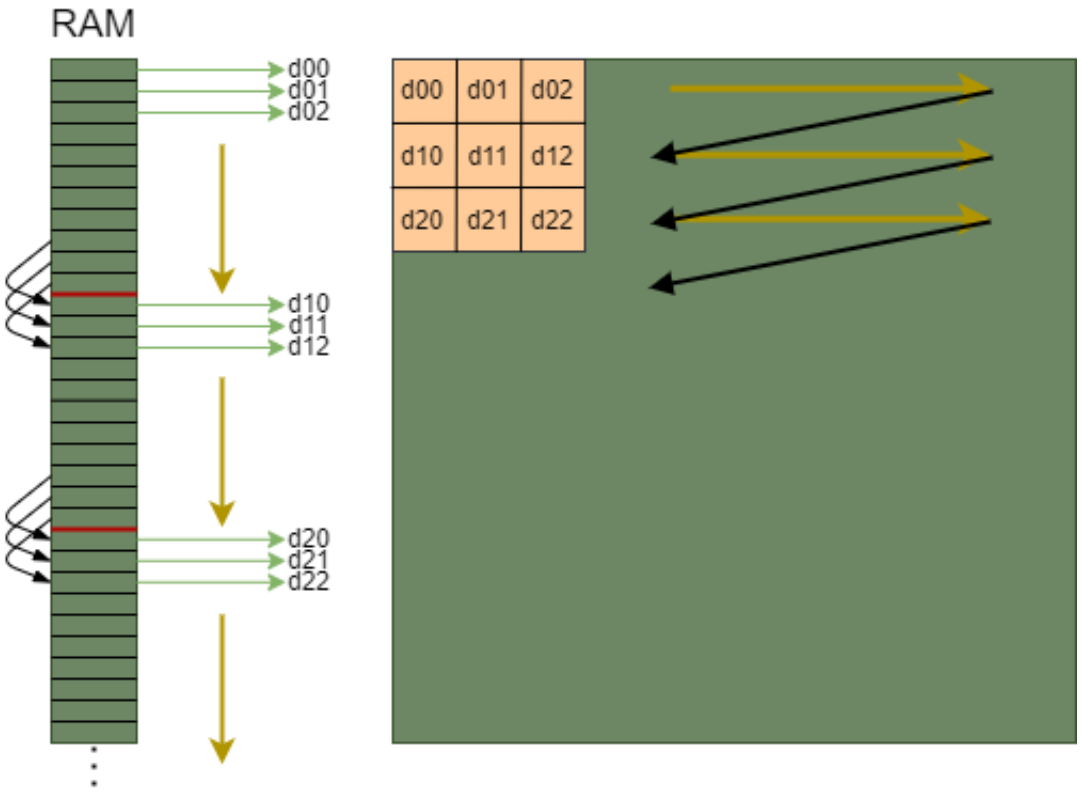


Figura 4.21: Representación del recorrido que se realiza sobre la estructura de la memoria RAM para llevar a cabo los desplazamientos horizontal y vertical.

La máquina de estados finitos (figura 4.22) mantiene un contador de desplazamientos, una vez que se sobrepasa el límite establecido, se recurre a un desplazamiento vertical y se reinicia el contador. La distancia que se mantiene entre cada píxel de la ventana es 1 cuando forman parte de la misma fila, cuando forman parte de otra fila pero se encuentran en la misma columna la distancia debe ser igual al número total de píxeles contenidos en una fila de la imagen. Esta relación debe mantenerse durante todo el proceso de convolución para no tener pérdida de datos. Cuando se quiere realizar un desplazamiento vertical, una vez que la máquina de estados detecta que se ha llegado al límite de desplazamientos horizontales, se suma a cada dirección contenida en la ventana del kernel el tamaño del kernel, en este caso es igual a 3 y el contador vertical debe ser incrementado.

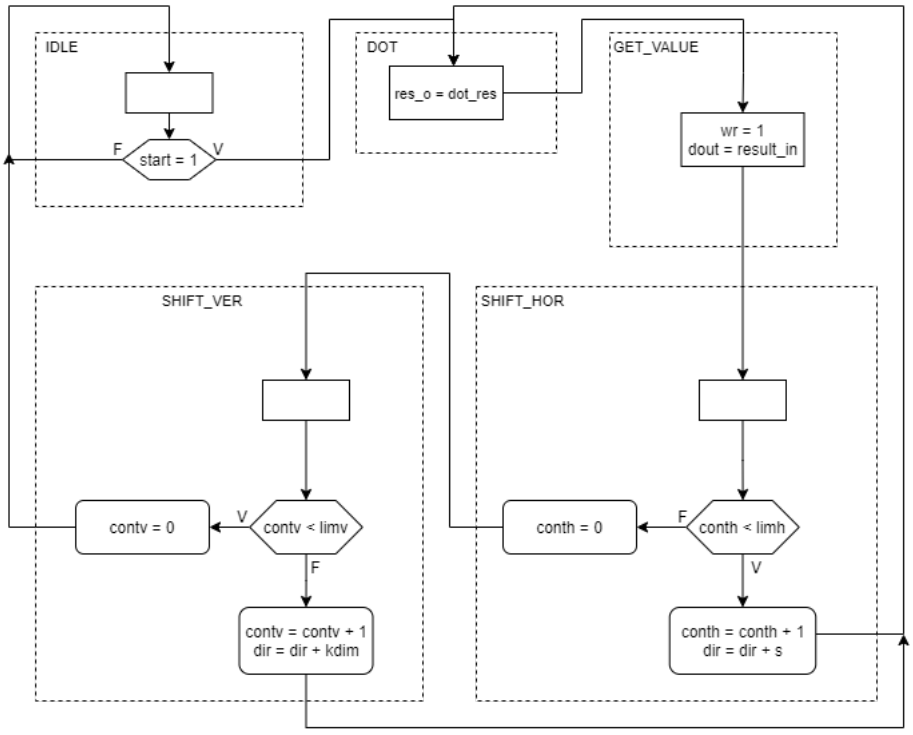


Figura 4.22: Carta ASM de la máquina de estados finitos que controla la operación de convolución.

Dentro del módulo considerado como DOT, cuyo propósito es el de realizar las operaciones necesarias para obtener el resultado del producto punto, también se incluye el bias, considerado como una no linealidad la cual permite alejar del origen la recta clasificadora de los datos. Antes de devolver el valor obtenido en la convolución, se aplica la función de activación ReLU para almacenar en la siguiente estructura de datos el valor que será utilizado en el operador pooling.

En la figura 4.23 se realiza el proceso que lleva a cabo el módulo de la operación de convolución donde se puede observar por la señal **wr** que se tienen 8 resultados en el proceso de desplazamiento horizontal, también se muestra que el contador vertical se incrementa hasta que se llega a 7 desplazamientos horizontales y en el siguiente ciclo de reloj se muestra un incremento de 3 a cada señal que se encarga de extraer los datos en las direcciones de memoria correspondientes. El proceso termina cuando el contador vertical alcanza 7 desplazamientos. La operación de convolución con los parámetros mencionados anteriormente, se realiza en un total de $3{,}95\mu s$.

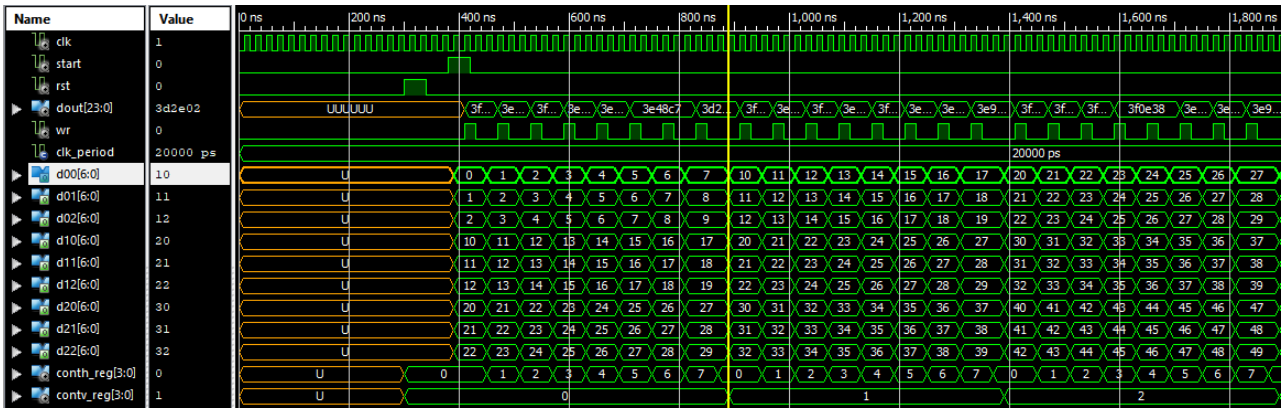


Figura 4.23: Simulación del módulo que realiza la operación de convolución.

4.4.1. Diseño Concurrente

Para aprovechar la capacidad de concurrencia del FPGA en la operación de convolución, se propone la computación de la imagen con múltiples ventanas que se desplazan simultáneamente por diferentes áreas de la imagen respetando el proceso que se llevaría con un solo kernel, es decir, manteniendo el movimiento del barrido pero adelantando las ventanas unos cuántos píxeles para generar un proceso concurrente.
En la figura 4.24 se tiene el componente ROM_IMG que almacena los valores de los píxeles de la imagen, una memoria ROM que almacena un conjunto de valores que representan el kernel o ventana, 4 controladores del barrido que realiza la operación de convolución y 4 módulos que realizan la operación producto punto, los cuales se encargan de realizar la extracción y multiplicación de una sección de la imagen con los valores del kernel de tal manera que los 4 conjuntos de datos abarquen el área de la imagen y realicen la operación de convolución. Dentro del módulo de producto punto denominado como **DOT**, se contempla el bias que es sumado al resultado del producto punto y antes de devolver el resultado con el bias aplicado se tiene la función de activación ReLU para arrojar los valores listos para que el operador pooling sea aplicado.

$$r = f\_sel + k\_pad \tag{4.4}$$

La distribución de las ventanas sobre la superficie de la imagen se calcula mediante la ecuación 4.3, dividiendo el resultado de la ecuación entre el número de ventanas que quieren colocarse, en este caso se colocaron 4 ventanas dando como resultado que cada una realiza el cálculo de 2 filas de píxeles.
Para saber el número de filas que abarca cada ventana, se hace uso de la ecuación 4.4, donde **r** es el número de filas que abarcará cada ventana, sin embargo, las ventanas afectadas por la convolución están determinadas por el valor f_sel, determinando el número de filas sobre las que será posicionada la casilla central de cada kernel y k_pad es el número de casillas de relleno que tiene el valor central del kernel, en el caso de un kernel de 3×3, el valor de k_pad es igual a 1.

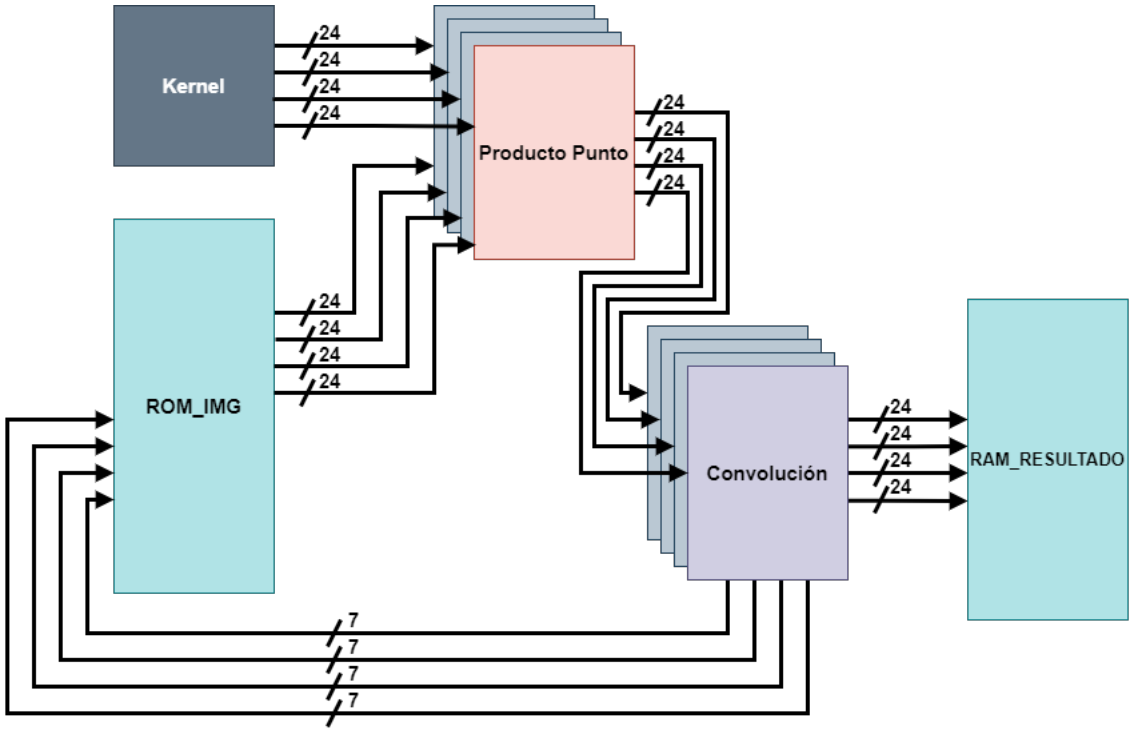


Figura 4.24: Diagrama que representa el proceso convolucional realizado por múltiples kernels.

Con los parámetros establecidos anteriormente, se configura cada controlador convolucional, los cuales se van a encargar de extraer los valores de los píxeles correspondientes involucrados en cada operación de producto punto.

A continuación, se muestran las simulaciones realizadas para cada ventana o kernel en las figuras 4.25, 4.26, 4.27 y 4.28. Es importante mencionar que el tiempo de ejecución se realiza en 950 ns, también, cada vez que se obtiene un dato se activa el flanco de subida de las señales wr lo que significa que se pasa el dato a una memoria ram que almacena los valores.

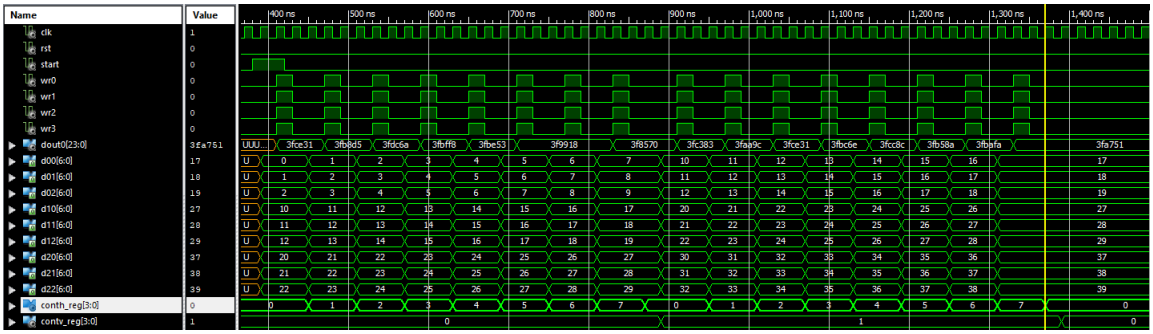


Figura 4.25: Simulación del desplazamiento del primer kernel.

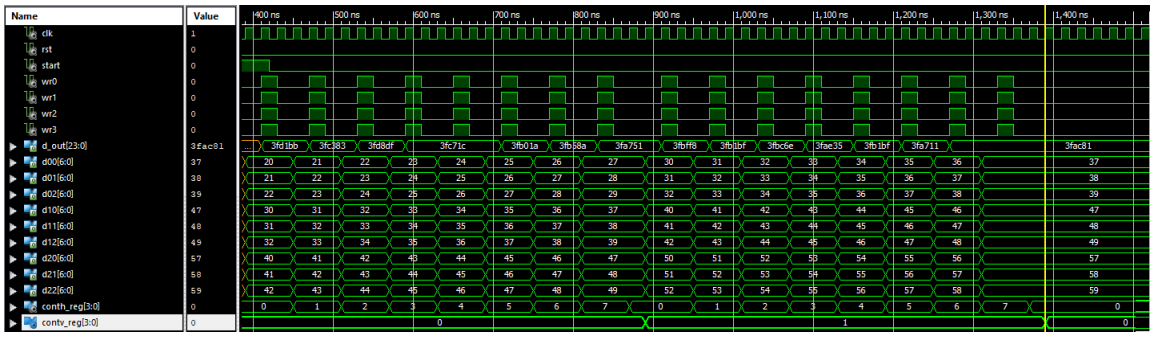


Figura 4.26: Simulación del desplazamiento del segundo kernel.

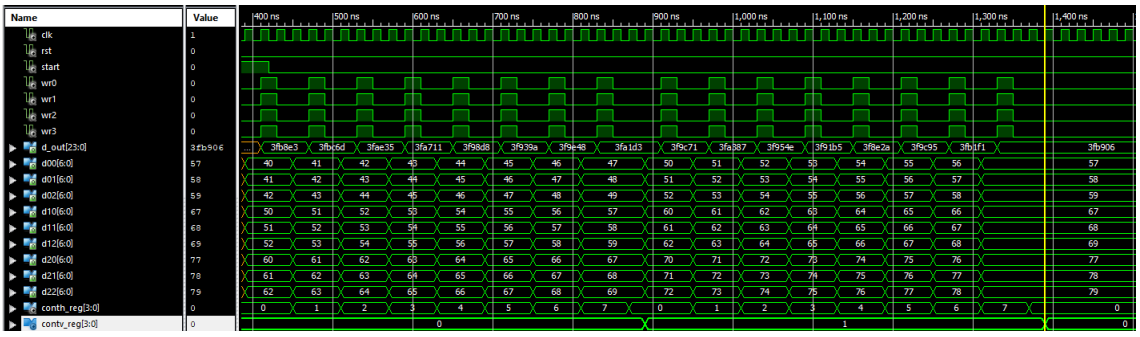


Figura 4.27: Simulación del desplazamiento del tercer kernel.

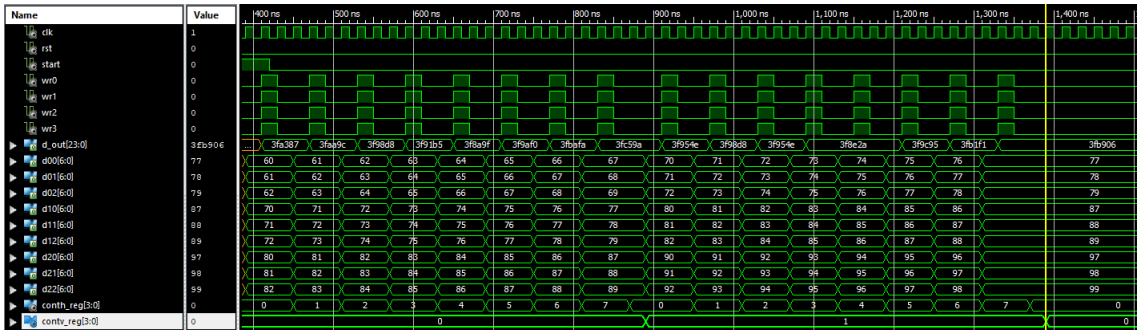


Figura 4.28: Simulación del desplazamiento del cuarto kernel.

4.5. Operador Pooling

La figura 4.29 muestra los bloques contenidos dentro del módulo que se encarga de realizar la operación pooling. Dentro de este módulo se tiene la imagen que contiene el volúmen de activación de la capa convolucional, un módulo para extraer el valor más grande contenido en la ventana que se desplazará sobre la superficie de la imagen y un controlador que se encarga de realizar el desplazamiento de la ventana de tal manera que mantiene la lectura de datos con las direcciones en memoria que deben ser extraídos.

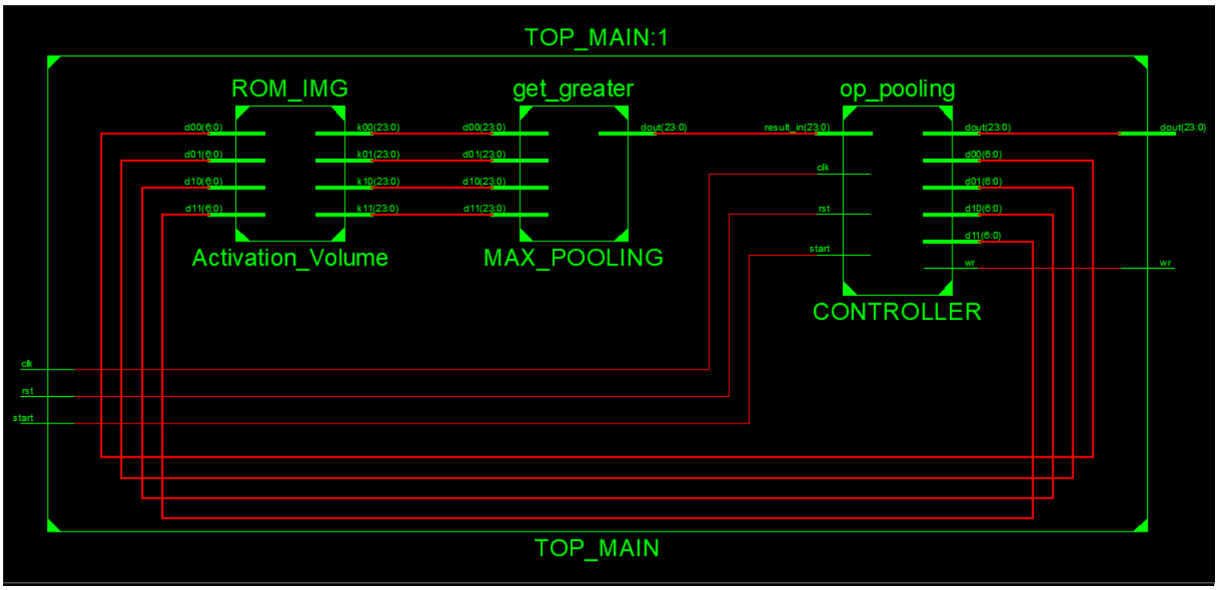


Figura 4.29: Esquemático del operador Pooling.

La máquina de estados utilizada en este módulo es la misma que se usa en el módulo convolucional, puesto que se trata de una ventana que realiza un barrido sobre la superficie de la estructura, la única diferencia que existe es la configuración de la ventana y el salto que da la ventana en cada desplazamiento. Los valores que se tienen en esta configuración que se tiene es un kernel de 2×2, un salto de 2, un relleno de 0 y una imagen de 10×10.

La simulación del módulo se presenta en la figura 4.30, en la cual se puede observar el desplazamiento de la ventana mediante el incremento de los apuntadores, nótese que en cada incremento se tiene un aumento de 2 y que la ventana sólo realiza 5 cálculos antes de saltar a la siguiente fila. Cuando la ventana cambia de fila, el aumento es igual al número de columnas más el tamaño del kernel, en este caso se incrementa 12 a cada apuntador de la ventana cuando ésta llega al límite establecido.

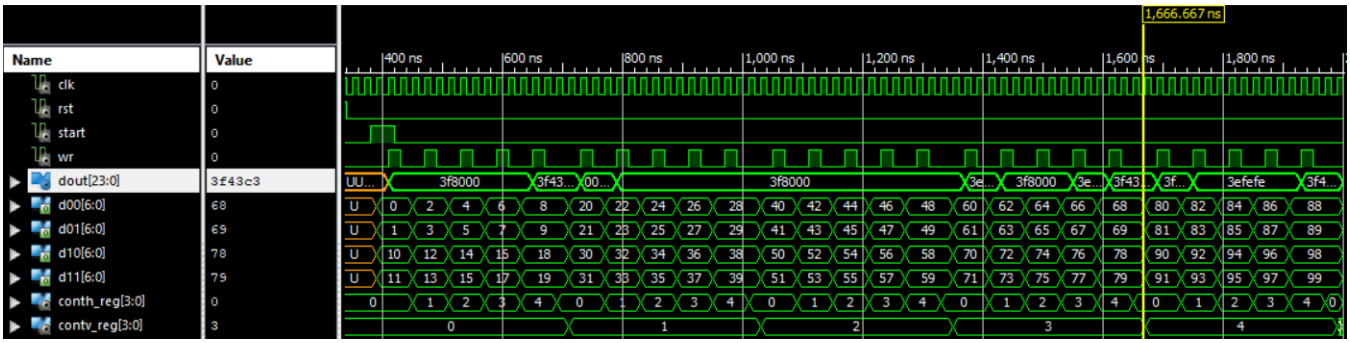


Figura 4.30: Simulación del módulo encargado de realizar la operación Pooling.

La figura 4.31 muestra el proceso que lleva a cabo el operador pooling. La ventana en la posición inicial toma 4 datos y éstos son leídos por el módulo MAX_POOLING el cual, se encarga de elegir el mayor de

los 4 datos, el dato mayor es elegido y se pasa a una siguiente estructura donde será almacenado el resultado del operador pooling. Una vez que el dato mayor en la posición actual de la ventana ha sido tomado, ésta da un salto de dos posiciones tomando un nuevo conjunto de 4 datos, el proceso se repite hasta llegar al límite horizontal, después da un salto vertical eligiendo un nuevo conjunto de filas.

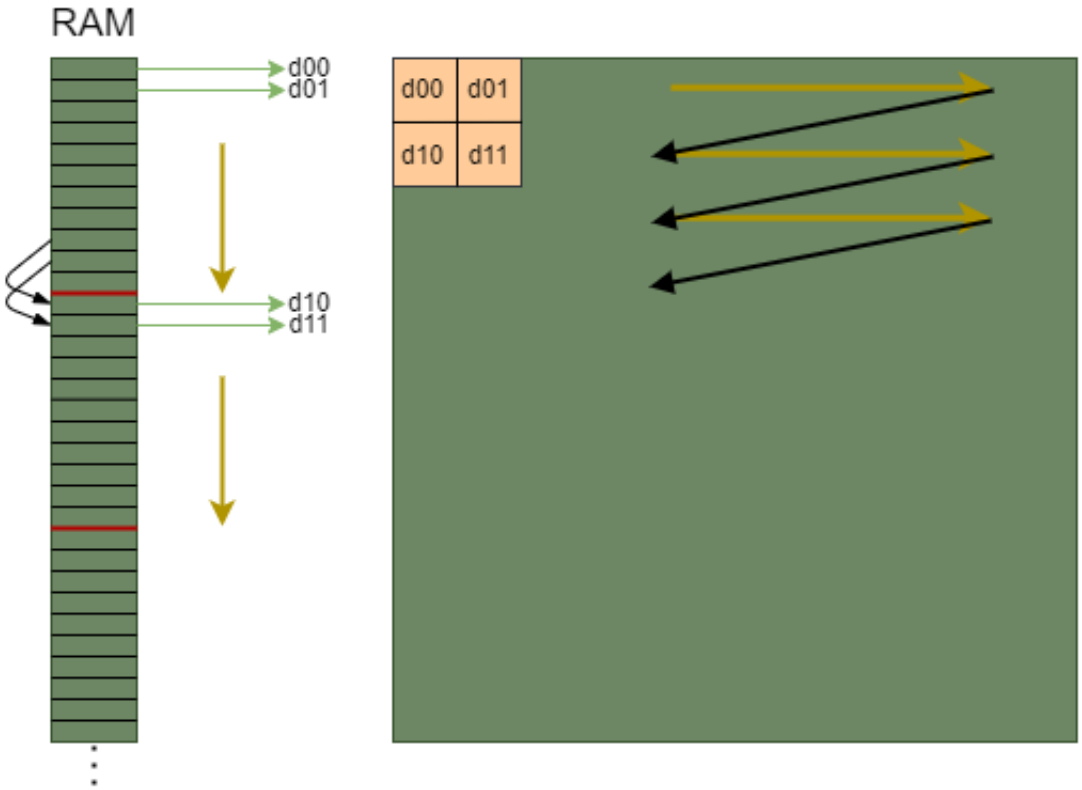


Figura 4.31: Representación de la operación pooling.

4.6. Perceptrón

El diagrama esquemático del perceptrón se ilustra en la figura 3.6. El circuito recibe como parámetros de entrada 3 números y un dato denominado como bias, a su vez, se tienen los pesos sinápticos propios del perceptrón. Cada dato de entrada se multiplica por el peso sináptico correspondiente con ayuda del módulo multiplicador de punto flotante y los resultados se suman con el módulo sumador/restador de punto flotante, el resultado obtenido es evaluado por la función de activación que da a la salida un 1 si el número es positivo o un -1 si el número es negativo.

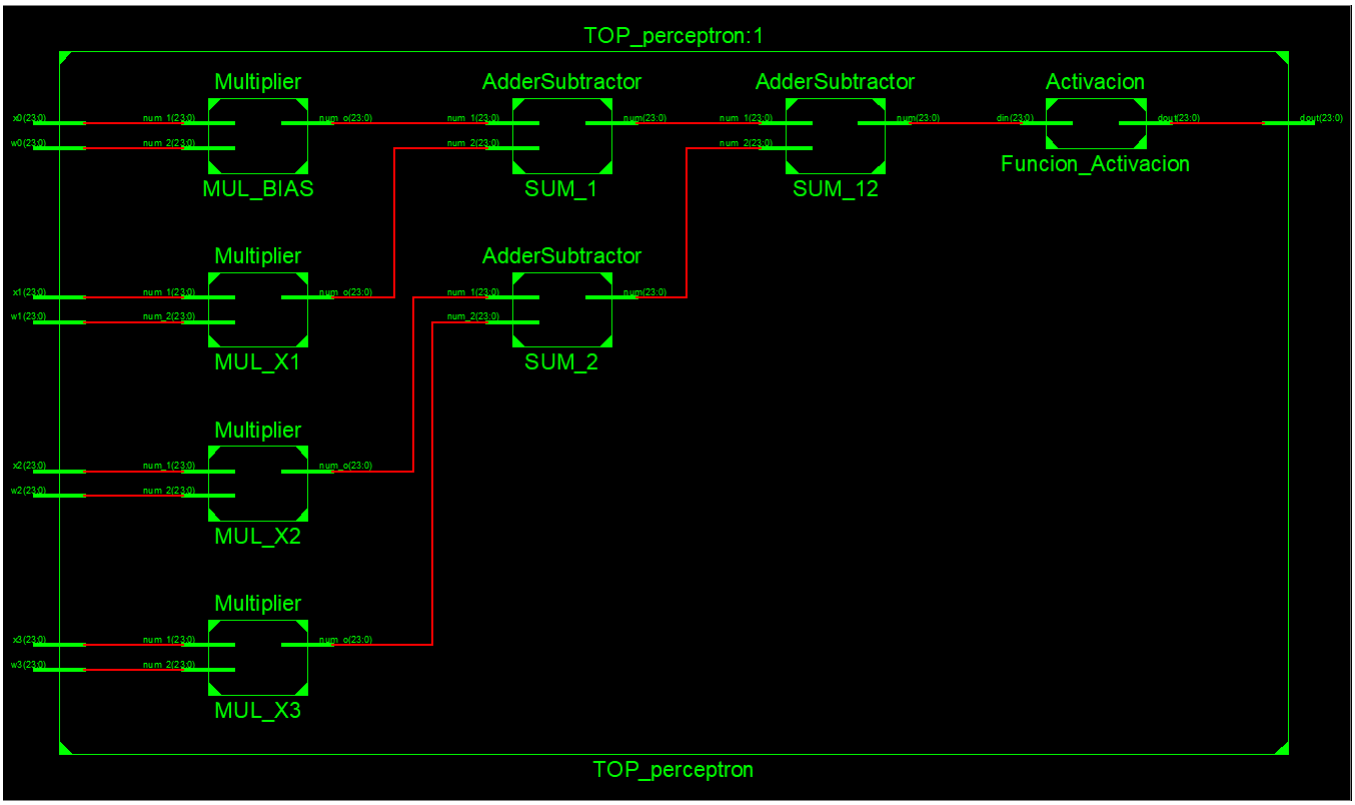


Figura 4.32: Diagrama esquemático del perceptrón.

Gracias a la implementación de la unidad de punto flotante que contiene las operaciones tales como suma, resta y multiplicación es posible dinamizar el perceptrón de tal manera que se configure de acuerdo a las características que se necesiten, es decir, añadiendo o limitando el número de multiplicadores y sumadores.

5.

Resultados

En este capítulo se presentan los resultados obtenidos tras la implementación de los módulos que se describen en el capítulo anterior. Por parte de la unidad de punto flotante se muestran algunas operaciones simuladas que fueron realizadas con el módulo sumador-restador y el multiplicador, así como una comparación con las mismas operaciones realizadas en un ordenador. También, se presentan los resultados obtenidos con el módulo de convolución, el operador pooling y el perceptrón. Dentro de la operación de convolución, se valida la función de activación ReLU, puesto que dentro del módulo que realiza el producto punto se añade con el fin de obtener el resultado que será tomado por el operador Pooling. Además, se verifica el funcionamiento de la unidad de punto flotante, pues se trabaja con números decimales que se encuentran entre 0 y 1, de tal manera que las multiplicaciones son efectuadas con números con exponente negativo. El conjunto de resultados arrojados nos ayudan a determinar la exactitud que presenta cada uno de los módulos implementados.

5.1. Operaciones de punto flotante

Para validar que las operaciones realizadas con la Unidad de Punto Flotante (FPU) fueron adecuadas, se realizaron las operaciones en un procesador de 64 bits y en un FPGA con la unidad de punto flotante diseñada de 24 bits. Los resultados de cada unidad de punto flotante con precisión diferente fueron comparados entre sí. Se usó un procesador Ryzen 7 con una FPU con precisión de 64 bits y la simulación de la FPU diseñada bajo el lenguaje de descripción de hardware VHDL con el IDE de desarrollo ISE DESIGN SUITE en su versión 14.6.

La tabla 5.1 muestra las operaciones realizadas con la calculadora de la PC. Una vez que se establecieron los números que iban a operarse en forma de suma, resta y multiplicación y los resultados de las operaciones realizadas con tales números, se realizó su conversión en punto flotante y se obtuvo el número representado en 24 bits, como se muestra en la tabla 5.3.

Sin embargo, la representación obtenida de los números es una aproximación al valor real, así como los resultados obtenidos con los resultados esperados. En la tabla 5.1 se muestra el valor obtenido después de

realizar la operación en la calculadora de la PC, comparado con el valor obtenido de la tabla 5.3 se tiene una diferencia de resultados que como se mencionó anteriormente el resultado obtenido con la unidad de punto flotante es una aproximación al resultado obtenido con la calculadora de la PC.
Además, se tiene un resultado obtenido al realizar las operaciones de los números representados en 24 bits con la calculadora de la PC, lo que se puede considerar como el resultado real de la representación en 24 bits.

Tabla 5.1: Operaciones realizadas con la calculadora de la PC.

Número 1	Op.	Número 2	Resultado obtenido
-14.06	*	49.116	-690.57096
-123.456	*	-39.27	4,848.11712
219.794	*	49.116	10,795.402104
111.222	+	329.151	440.373
-810.22	+	-14.06	-824.28
534.664	+	-1,000.16	-465.496

La diferencia entre el resultado obtenido y el resultado esperado se debe a dos puntos importantes, el primero es que la precisión de la unidad de punto flotante de 24 bits es menor a la unidad de punto flotante de 64 bits, por lo tanto se tiene un número aproximado al valor del número que se quiere operar, el segundo se debe a la técnica de redondeo aplicada en la unidad de punto flotante diseñada.

Tabla 5.3: Operaciones realizadas con la unidad de punto flotante diseñada de 24 bits.

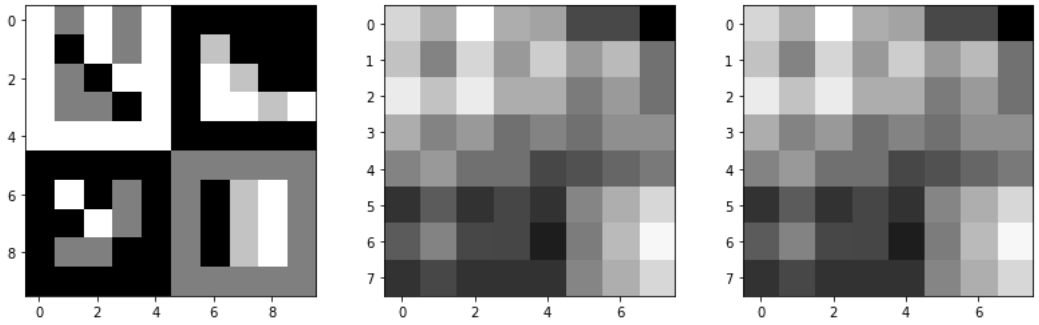
Número Representado 1	Op.	Número Representado 2	Resultado Obtenido	Resultado Esperado
-14.0595703125	*	49.115234375	-690.83845	690.5390911102294921875
-123.453125	*	-39.26953125	4,848	4,847.946635009765625
219.7890625	*	49.115234375	10,795	10,794.9913177490234375
111.21875	+	329.140625	440.359375	440.359375
-810.15625	+	-14.0595703195	-824.25	-824.21582031250000625
534.65625	+	-1,000.15625	-465.5	-465.5

La tabla 5.3 muestra en la columna de **Resultado Obtenido** el resultado de la operación ejecutada en la simulación y en la columna de **Resultado Esperado** el resultado que arroja la operación realizada en

la calculadora de la PC. Se puede observar que hay un pequeño error en el resultado obtenido debido a la precisión de 24 bits con la que fue implementada la unidad de punto flotante, con lo cual se puede determinar que el funcionamiento de la unidad de punto flotante es óptimo.
Las operaciones realizadas y que se muestran en las tablas fueron ejecutadas en la simulación del circuito con el fin de validar que los resultados arrojados por la unidad de punto flotante son adecuados, su funcionamiento se determina más adelante en su incorporación en la operación de convolución, donde se efectúa la operación producto punto, también, dentro del perceptrón.

5.2. Convolución

Para validar la operación de convolución se usó el simulador que contiene el IDE de desarrollo VIVADO en su versión 2020.2 bajo el lenguaje de descripción de hardware VHDL. Además, para comparar los resultados obtenidos en el simulador, se realizó la operación de convolución en una PC con un procesador AMD Ryzen 7 de 64 bits con 8 núcleos y con una frecuencia de 2.3 GHz.
Inicialmente se tomó la imagen (a) mostrada en la figura 5.1 cuyos valores representados con 8 bits se encuentran en el rango de 0 a 255 y fueron normalizados entre 0 y 1. Los valores normalizados fueron convertidos al formato de punto flotante de 24 bits con ayuda de un programa [44] implementado en el lenguaje de programación Python, mismo que concatena los valores en una cadena con un formato específico para poder ser copiada y utilizada en una memoria ROM contenida en el circuito encargado de realizar la convolución en un módulo diseñado en VHDL.



(a) Imagen sobre la que se realizó la convolución.

(b) Imagen que representa la convolución en la PC.

(c) Imagen que representa la convolución realizada en simulación.

Figura 5.1: Resultados obtenidos posteriores a la realización de la convolución con un filtro promedio.

Como ya se mencionó, los números de punto flotante obtenidos se utilizaron dentro de la memoria ROM_IMG, los cuales representan los valores de los píxeles en un formato de punto flotante de 24 bits.
Una vez que se han establecido los valores que contiene la memoria ROM_IMG, se inicia la simulación del circuito encargado de realizar la convolución. El circuito controlador de la convolución se encarga de extraer los datos de la memoria ROM_IMG de acuerdo al funcionamiento planteado por la máquina de estados

representada en la figura 4.22 y mandarlos al módulo encargado de realizar el producto punto y obtener el resultado. Los resultados de cada producto punto realizado en el proceso convolucional se almacenaron en una memoria RAM, la cual muestra los valores obtenidos por cada operación.

	0	1	2	3	4	5	6	7
0	1.6662284	1.554682	1.7773391	1.5551177	1.5294098	1.2810447	1.2810447	1.084967
1	1.6104552	1.4435713	1.6662284	1.4997802	1.6405205	1.5032661	1.5882331	1.3921554
2	1.7215658	1.6104552	1.7220016	1.5555534	1.5555534	1.4182991	1.5032661	1.3921554
3	1.5551177	1.444007	1.4997802	1.3886695	1.444007	1.3882338	1.4732008	1.4732008
4	1.4444427	1.4997802	1.3886695	1.3882338	1.2771231	1.306317	1.3620901	1.4174277
5	1.2222214	1.3328964	1.2217857	1.2766874	1.22135	1.4466214	1.5581678	1.6688428
6	1.3328964	1.4435713	1.2771231	1.2766874	1.1660124	1.4209135	1.5882331	1.7542455
7	1.2217857	1.2771231	1.2217857	1.22135	1.22135	1.4466214	1.5581678	1.6688428

Figura 5.2: Valores resultantes de la convolución realizada en la PC.

Para poder visualizar los resultados obtenidos por la simulación del circuito que realiza la convolución, se tomaron los valores contenidos en la memoria RAM y se escribieron manualmente dentro de otro programa [45] que también se desarrolló bajo el lenguaje de programación Python, el cual se encarga de convertir los datos a su representación flotante en base 10 y después muestra la imagen con ayuda de la librería matplotlib. La imagen obtenida después de interpretar los datos obtenidos en la simulación llevada a cabo se muestra en la figura 5.1(c).

Por otro lado, para obtener la imagen mostrada en la figura 5.1(b), se realizó la convolución en la PC, tomando los mismos datos de la imagen que se muestra en la figura 5.1(a) y el kernel promedio.

La operación de convolución se validó comparando los resultados obtenidos del procesamiento de la imagen (figura 5.1(a)) en una computadora y el procesamiento de la imagen con el módulo implementado, a continuación, se presentan las imágenes obtenidas respectivamente; también, se muestra la imagen utilizada para realizar la prueba.

	0	1	2	3	4	5	6	7
0	1.6661987	1.554657	1.7773132	1.5550842	1.5293884	1.2810364	1.2810364	1.0849609
1	1.6104126	1.4435425	1.6661987	1.4997559	1.6405029	1.5032349	1.5881958	1.3921204
2	1.7215271	1.6104431	1.7219849	1.555542	1.555542	1.4182739	1.5032349	1.3921509
3	1.5551147	1.4440002	1.4997559	1.3886414	1.4440002	1.3882141	1.473175	1.473175
4	1.4444275	1.4997559	1.3886414	1.3882141	1.2770996	1.3063049	1.3620605	1.4174194
5	1.2221985	1.3328857	1.2217712	1.2766724	1.221344	1.4465942	1.558136	1.6687927
6	1.3328857	1.4435425	1.2770996	1.2766724	1.1659851	1.4208984	1.5881958	1.7542114
7	1.2217712	1.2770996	1.2217712	1.221344	1.221344	1.4465942	1.558136	1.6687927

Figura 5.3: Valores resultantes de la convolución realizada en la simulación.

Es importante mencionar que mediante la validación de este módulo, también se está validando la unidad de punto flotante, puesto que los valores de los píxeles utilizados están normalizados en un rango de 0 a 1, además, el kernel utilizado fue el kernel promedio, debido a la precisión que se tiene con la FPU de 24 bits, el valor de $1/9$ se representa en valor decimal como: 0.111110687256. Las multiplicaciones realizadas en este proceso operan con números de punto flotante con exponente negativo, lo que indica que el resultado obtenido es equivalente a realizar una división.

A pesar de que las imágenes obtenidas son similares, los valores decimales varían debido a la precisión que presentan los diferentes formatos utilizados, en la PC se hace uso de un formato de 64 bits y la convolución realizada en el módulo implementado se tiene una precisión de 24 bits.

La figura 5.2 muestra los resultados numéricos obtenidos posteriormente a realizar la operación de convolución en la PC. Y la figura 5.3 muestra los resultados numéricos obtenidos después de realizar la simulación del módulo de la operación de convolución que fue implementada.

Como se puede observar, los datos obtenidos son iguales en los primeros 4 decimales y varían a partir del quinto decimal. Esto se debe a la precisión que se tiene en los diferentes formatos utilizados, en la PC se tiene un formato de 64 bits y en la simulación se usa el formato de 24 bits implementado.

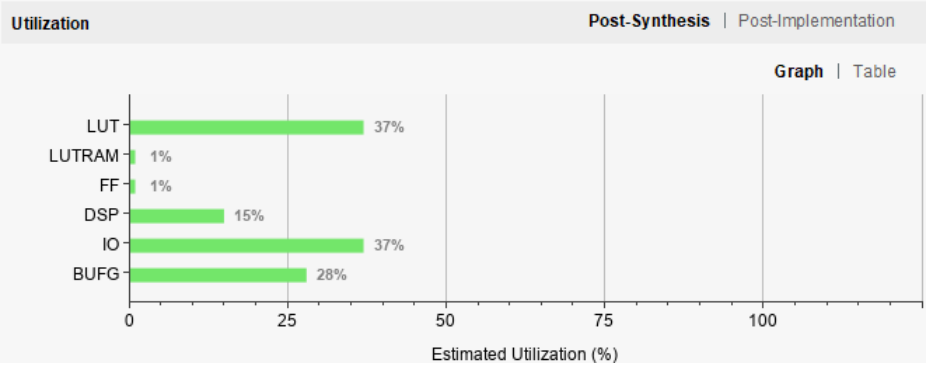


Figura 5.4: Recursos utilizados de un FPGA Artix-7.

La comparación de los resultados obtenidos permite validar el funcionamiento de la unidad de punto flotante y del módulo convolucional en sí, puesto que la diferencia en los resultados se debe a la precisión de cada formato, como se mencionó anteriormente.

Para conocer el desempeño que se tiene en el módulo implementado se realizó el cálculo del error cuadrático medio (RMSE), el cual mide la cantidad de error que hay entre dos conjuntos de datos. Esta métrica arrojó un valor de 2.452409143921587e-05 y permite determinar la distancia promedio entre los dos conjuntos de datos. El RMSE obtenido al aplicarse al conjunto de datos que forman parte de los resultados adquiridos con la convolución realizada con el tipo de dato *numpy.float32* y los resultados de la convolución con el tipo de dato *numpy.float64* es .003297962345842096e-05.

Se puede observar que el error obtenido por el RMSE aplicado a la convolución realizada con 64 y 32 bits es aún más pequeño a pesar de que los datos en 64 bits tienen doble precisión en comparación de los datos

de 32 bits. Esto se debe a que la diferencia se encuentra en los bits menos significativos, por lo que a medida que los decimales se alejan del punto decimal hacia la derecha van a influir menos en el valor del número resultante.
Por lo tanto, se determina que el RMSE obtenido es óptimo teniendo en consideración que la precisión de los valores se reduce 8 bits en comparación de los datos de 32 bits y 40 bits en comparación con los datos de 64 bits.

Resource	DSP	BRAM	LUT	FF
Available	220	140	53200	106400
Used	64	69	28861	41828
Utilization (%)	29.09	49.29	54.250	39.31

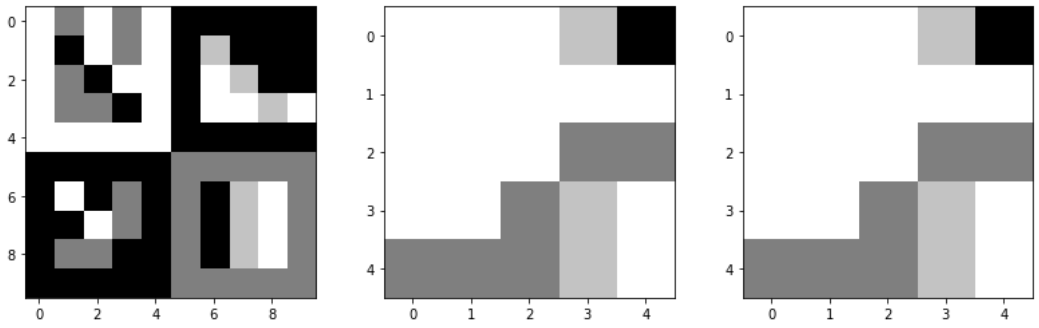
Figura 5.5: Recursos utilizados por el trabajo [34] implementado en un FPGA Zynq-7000.

Los recursos que se utilizan en la simulación se muestran en la figura 5.4, se estableció el FPGA Artix-7 en el IDE debido a los recursos que el diseño convolucional necesita. Como se puede observar, en la figura 5.5 se muestran los recursos utilizados por el desarrollo de [34], su diseño convolucional se implementó en un FPGA Zynq-7000 la cual, en comparación con la Artix-7 tiene más recursos lógicos.
ES importante destacar el porcentaje de LUT utilizado en el Artix-7 del diseño implementado en este trabajo, puesto que sólo usa el 32 % y el 1 % de FF para procesar una imagen de 10×10, por otro lado [34] emplea el 54.25 % de LUT y el 39.31 % de FF en el FPGA Zynq-7000, la cual es un FPGA con mayores prestaciones que la Artix-7.
El tiempo de ejecución en la PC fue de 2.995 ms, mientras que el tiempo que muestra la ejecución en simulación es de .00095 ms, lo cual nos dice que es considerablemente más rápido realizar el procesamiento de esta estructura en el FPGA, gracias a la explotación de la característica de concurrencia que ofrece.

5.3. Pooling

Para realizar la simulación del operador Pooling se hizo uso del IDE de desarrollo VIVADO en su versión 2020.2 y una PC con un procesador AMD Ryzen 7 de 64 bits con 8 núcleos y con una frecuencia de 2.3 GHz para aplicar el operador Pooling.
El operador Pooling implementado fue el MaxPooling, el cual consiste en tomar el valor más grande contenido en un conjunto de datos. Para validar este módulo se hizo uso de la misma máquina de estados que controla el desplazamiento del kernel sobre la imagen, con la única diferencia de que el kernel tomado fue de 2×2 y el salto que se daba fue de 2, de tal manera que la ventana no repite valores de datos en cada paso que da. En la imagen 5.6 se muestra la imagen que fue tomada para aplicar el operador, como se puede observar es la misma sobre la que se aplicó la convolución. Para obtener los resultados de este módulo, se tomó como punto de partida la misma imagen que se tomó en la operación de convolución, por lo tanto, se reutilizó la memoria RAM del módulo convolucional. Se configuró la máquina de estados para que tomara 4 datos que pasan por un módulo que se encarga de extraer el dato mayor y al final lo almacena en una memoria RAM. El

conjunto de datos resultante, se pasó manualmente al programa [45] que se encarga de convertirlos en base 10 y mostrar la imagen resultante, misma que se muestra en la figura 5.6(c). Una vez que los datos han sido representados en la imagen resultante, se realizó el proceso del operador Pooling en la PC, los resultados se muestran en la figura 5.6(b).



(a) Imagen sobre la que se aplicó el operador pooling.

(b) Imagen que representa el resultado del operador pooling en la PC.

(c) Imagen que representa el resultado del operador pooling en simulación.

Figura 5.6: Resultados obtenidos tras la aplicación del operador pooling.

Se observan los resultados obtenidos tanto con la PC como con la simulación. Se puede notar que ambas imágenes presentan los mismos tonos en cada posición, lo cual indica que tanto el barrido sobre la superficie de la imagen, como la aplicación del operador MaxPooling es adecuada. Por último, el tiempo de ejecución del MaxPooling sobre la imagen de 10×10 con una ventana de 2×2 se realiza en .0006 ms.

5.4. Perceptrón

Para realizar la validación del perceptrón, se utilizó el IDE de desarrollo VIVADO en su versión 2020.2 y una PC una PC con un procesador AMD Ryzen 7 de 64 bits con 8 núcleos y con una frecuencia de 2.3 GHz.

Con el fin de saber si el perceptrón diseñado e implementado en VHDL presenta un buen funcionamiento, se realizó un programa en Python para obtener un perceptrón o neurona entrenada. La función que realiza esta neurona consiste en simular el comportamiento de una compuerta lógica *OR*.

Se corrió una etapa de entrenamiento con un conjunto de datos etiquetados y una etapa de prueba posteriormente al entrenamiento. Una vez que finalizaron las dos etapas se obtuvieron los siguientes pesos finales del perceptrón:

- W_0: -0.26094833
- W_1: 0.39356209

- W_2: 0.00820866

Como se ha mencionado, es necesario representar los valores decimales en el formato de punto flotante de 24 bits. Los valores que tendrá como pesos sinápticos el perceptrón implementado en VHDL son los siguientes:

- W_0: -0.260948181152
- W_1: 0.3935546875
- W_2: 0.00820851325989

Se probaron 10 datos en la etapa de validación del perceptrón, los cuales arrojaron los datos mostrados en la figura 5.7. Se puede observar que de los datos que fueron clasificados sólo uno tuvo un error.

```
prediccion: 1  objetivo:  [-1]
prediccion: 1  objetivo:  [1]
prediccion: -1  objetivo:  [-1]
prediccion: -1  objetivo:  [-1]
prediccion: -1  objetivo:  [-1]
prediccion: -1  objetivo:  [-1]
prediccion: 1  objetivo:  [1]
prediccion: -1  objetivo:  [-1]
prediccion: -1  objetivo:  [-1]
prediccion: -1  objetivo:  [-1]
```

Figura 5.7: Resultados obtenidos en la validación del perceptrón.

En la simulación, se tienen los mismos resultados, que en la PC. En la figura 5.8 se muestran los resultados en formato hexadecimal. El valor BF8000 representa el -1 y el valor 3F8000 representa el valor 1.

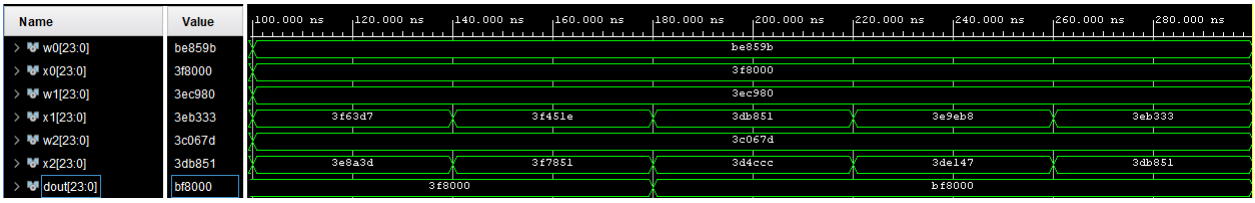


Figura 5.8: Resultados de la simulación del perceptrón con los valores obtenidos del perceptrón entrenado en la PC.

El tiempo de procesamiento del perceptrón es de .0003 ms. Sin embargo, el procesamiento de este perceptrón no es muy pesado como lo es la operación de convolución, debido a que sólo se realizan 3 multiplicaciones y 2 sumas.
Una vez finalizadas las pruebas, se puede determinar que tanto la unidad de punto flotante, como el método propuesto para realizar la convolución y el perceptrón funcionan adecuadamente y proporcionan un funcionamiento óptimo. Gracias al desarrollo de la unidad de punto flotante y a las operaciones que constituyen

una red neuronal convolucional se pueden realizar implementaciones que contribuyan al análisis de datos y a los sistemas embebidos.

6.

Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones del trabajo realizado con los diferentes métodos propuestos que fueron explicados previamente. Como se observó, los resultados obtenidos con el proceso llevado a cabo por la operación de convolución fueron adecuados y similares a los resultados que presenta la operación realizada en la PC.
En la siguiente sección, se lista un conjunto de mejoras propuestas para trabajo a futuro, tal como, agregar la división a la unidad de punto flotante.

6.1. Conclusiones

En este proyecto de investigación se propone la implementación de las principales capas de una red neuronal convolucional, como son, la operación de convolución, la función de activación ReLU, el operador Pooling y el perceptrón. Para poder realizar las operaciones que se necesitan dentro de la operación de convolución y el perceptrón, se diseñó, implementó y validó una unidad de punto flotante con un formato de 24 bits.
La implementación de la unidad de punto flotante incluye las operaciones de suma, resta y multiplicación. Dentro de la multiplicación se pueden realizar operaciones con números que tengan exponente negativo, lo cual permite realizar una división.
La operación de convolución se implementó y validó satisfactoriamente de acuerdo a los resultados obtenidos, pues se obtuvo una imagen similar a la obtenida en una convolución realizada en la PC con una precisión de 64 bits.
La función de activación ReLU se incluyó en la operación de convolución, se encuentra conectada en la salida para que sea posible su aplicación al resultado de la convolución y de esta manera el dato que pase a la siguiente etapa pueda ser utilizado directamente por el módulo que corresponde al operador Pooling.
El operador Pooling se implementó utilizando el controlador que realiza el desplazamiento de la ventana sobre la superficie de la imagen, de acuerdo a los resultados obtenidos se puede determinar que su funcionamiento se realiza adecuadamente.

El perceptrón se validó con los datos obtenidos de un perceptrón entrenado en la PC, fue configurado con los pesos sinápticos del perceptrón entrenado y se le pasaron los mismos valores, el perceptrón validado en la simulación presentó los mismos resultados que se obtuvieron por el perceptrón entrenado, por lo tanto, se determina que su funcionamiento es adecuado.
El mayor reto se encuentra en la capa convolucional, puesto que la operación de convolución, además de ser una operación secuencial, hace uso de un número considerable de sumadores y multiplicadores de punto flotante que vuelven bastante pesada su computación. Mediante el uso del FPGA, es posible acelerar el procesamiento de las operaciones necesarias, sin embargo, debido a la gran cantidad de recursos utilizados y a la tarjeta de desarrollo que se tiene, no fue posible su implementación, por lo tanto, los resultados obtenidos fueron tomados desde el proceso de simulación.
En la simulación, fue posible validar que el proceso que se lleva a cabo por cada uno de los módulos tiene un funcionamiento correcto.
Es importante contar con una tarjeta de desarrollo que incluya un FPGA con mayores características a las que se tiene con el Spartan 6 incluido en la tarjeta de desarrollo amiba2.
Gracias a la implementación de la unidad de punto flotante, es posible la implementación de los módulos, ya que es utilizada en la capa convolucional y en la implementación del perceptrón, además, proporciona libertad para implementar un perceptrón con distintas características puesto que sólo se tiene que agregar o retirar tantos multiplicadores y sumadores sean necesarios.
La máquina de estados implementada para llevar a cabo el barrido de un kernel sobre una estructura de datos bidimensional puede ser configurada de acuerdo a las necesidades de cada implementación, permitiendo ser reutilizada en distintas situaciones, como es, haciendo uso de estructuras con una mayor resolución y también haciendo uso de kernels con distinto tamaño.
Por último, para poder realizar la implementación de un modelo de red neuronal, es importante añadir la división a la unidad de punto flotante así como la función de activación SOFTMAX.

6.2. Trabajo futuro

Como trabajo futuro se plantea la implementación en un FPGA con mayores recursos a los que se tiene en el Spartan 6.
También, es importante añadir la división a la unidad de punto flotante para poder implementar la función de activación SOFTMAX.
Por otro lado, una vez que se tenga la implementación de todas las capas de una red neuronal convolucional se puede probar implementando un modelo en el FPGA, de tal manera que se comience a realizar reconocimiento de objetos.
Gracias a la implementación de un modelo de red neuronal convolucional, se puede añadir la característica de reconocimiento de objetos a un sistema empotrado.

Referencias

[1] B. P. Orozco, "Inteligencia Artificial", *FCCyT*, 2018.

[2] C. Zhang, Y. Wang, J. Guo y H. Zhang, "Digital Recognition Based on Neural Network and FPGA Implementation", en *2017 9th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, IEEE, ago. de 2017. DOI: 10.1109/ihmsc.2017.71.

[3] S. University, *CS231n Convolutional Neural Networks for Visual Recognition*, Http://cs231n.github.io/convolutional-networks/.

[4] F. Chollet, *Deep Learning with Python*. Manning, 30 de nov. de 2017, ISBN: 1617294438. dirección: https://www.ebook.de/de/product/28930398/francois_chollet_deep_learning_with_python.html.

[5] S. Russel y P. Norvig, *Artificial Intelligence A Modern Approach*. Prentice Hall, 2003.

[6] A. M. Turing, "Computational Machinery and Intelligence", *Mind*, n.º 49, págs. 433-460, 1950.

[7] R. S. Sutton y A. G. Barto, *Reinforcement Learning: an Introduction*. The MIT Press, 2017.

[8] I. Goodfellow, Y. Bengio y A. Courville, *Deep Learning (Adaptive Computation and Machine Learning series)*. The MIT Press, 2016, ISBN: 9780262337373. dirección: https://www.amazon.com/Deep-Learning-Adaptive-Computation-Machine-ebook/dp/B01MRVFGX4?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B01MRVFGX4.

[9] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.", *Psychological review*, vol. 65, n.º 6, pág. 386, 1958.

[10] D. Calvo, *Perceptrón - Red Neuronal*, Http://www.diegocalvo.es/perceptron/.

[11] T. M. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[12] *¿Qué es un FPGA?*, https://vhdl.es/fpga/.

[13] B. Liu, D. Zou, L. Feng, P. Fu y J. Li, "An FPGA-Based CNN Accelerator Integrating Depthwise Separable Convolution", *Electronics*, 2019.

[14] S. Palekar y N. Narkhede, "High Speed and Area Efficient Single Precision Floating Point Arithmetic Unit", *IEEE International Conference On Recent Trends In Electronics Information Communication Technology*, 2016.

[15] Google, *Usar bfloat16 con modelos de TensorFlow*, Https://cloud.google.com/tpu/docs/bfloat16?hl=es-419.

[16] H. Das, *Cloud Computing for Geospatial Big Data Analytics*. Springer-Verlag GmbH, 11 de dic. de 2018, ISBN: 3030033597. dirección: https://www.ebook.de/de/product/35048264/cloud_computing_for_geospatial_big_data_analytics.html.

[17] M. Zhu, Q. Kuang, J. Lin, Q. Luo, C. Yang y M. Liu, "A Z Structure Convolutional Neural Network Implemented by FPGA in Deep Learning", en *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, IEEE, 2018. DOI: 10.1109/iecon.2018.8592775.

[18] Y. Chen y W. du Plessis, "Neural network implementation on a FPGA", en *IEEE AFRICON. 6th Africon Conference in Africa,*, IEEE, 2002. DOI: 10.1109/afrcon.2002.1146859.

[19] S. Li, K. Choi e Y. Lee, "Artificial neural network implementation in FPGA: A case study", en *2016 International SoC Design Conference (ISOCC)*, IEEE, 2016. DOI: 10.1109/isocc.2016.7799795.

[20] Y. Lee y S. bum Ko, "FPGA Implementation of a Face Detector using Neural Networks", en *2006 Canadian Conference on Electrical and Computer Engineering*, IEEE, 2006. DOI: 10.1109/ccece.2006.277536.

[21] V. Gupta, K. Khare y R. Singh, "FPGA Design and Implementation Issues of Artificial Neural Network Based PID Controllers", en *2009 International Conference on Advances in Recent Technologies in Communication and Computing*, IEEE, 2009. DOI: 10.1109/artcom.2009.182.

[22] F. Benrekia, M. Attari, A. Bermak y K. Belhout, "FPGA implementation of a neural network classifier for gas sensor array applications", en *2009 6th International Multi-Conference on Systems, Signals and Devices*, IEEE, 2009. DOI: 10.1109/ssd.2009.4956804.

[23] M. Moradi, M. A. Poormina y F. Razzazi, "FPGA Implementation of Feature Extraction and MLP Neural Network Classifier for Farsi Handwritten Digit Recognition", en *2009 Third UKSim European Symposium on Computer Modeling and Simulation*, IEEE, 2009. DOI: 10.1109/ems.2009.13.

[24] L. S. A. Hamid, K. A. Shehata, H. El-Ghitani y M. ElSaid, "Design of Generic Floating Point Multiplier and Adder/Subtractor Units", *12th International Conference on Computer Modelling and Simulation*, 2010.

[25] A. Lv, C. Wang, L. Hou, Z. Zeng, J. Guo y N. Jiang, "An Arithmetic Unit and Multiplying Accumulation Unit of a Custom Floating Point Data Format", *International Conference on Integrated Circuits and Microsystems*, 2018.

[26] R. K. Kodali, S. K. Gundabathula y L. Boppana, "FPGA Implementation of IEEE-754 Floating Point Karatsuba Multiplier", *International Conference on Control, Instrumentation, Communication and Computational Technologies*, 2014.

[27] S. Rani, "Area and Speed Efficient Floating Point Unit", *ICRAIE*, 2014.

[28] Z. Babic, A. Avramovic y P. Bulic, "An Iterative Mitchell's Algorithm Based Multiplier", *IEEE*, 2008.

[29] N. Singh y T. Sasamal, "Design and Synthesis of Goldschmidt Algorithm based Floating Point Divider on FPGA", *ICCSP*, 2016.

[30] N. Arora, G. Sharma y S. Kumar, "VHDL implementation of 32-bit floating point unit (FPU)", *International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE)*, vol. 4, jul. de 2015, ISSN: 2278-909X.

[31] M. Z. Zhang, H. T. Ngo, A. R. Livingston y V. K. Asari, "An Efficient VLSI Architecture for 2-D Convolution with Quadrant Symmetric Kernels", *Proceedings of the IEEE Computer Society Annual Symposium on VLSI New Frontiers in VLSI Desing*, 2005.

[32] K. Benkrid y S. Belkacemi, "DESIGN AND IMPLEMENTATION OF A 2D CONVOLUTION CORE FOR VIDEO APPLICATIONS ON FPGAs", *International Workshop on Digital and Computational Video*, 2002.

[33] L. Kang, H. Li, X. Li y H. Zheng, "Design of Convolution Operation Accelerator based on FPGA", *International Conference on Machine Learning, Big Data and Business Intelligence*, 2020.

[34] Y. Cao, X. Wei, T. Qiao y H. Chen, "FPGA-based accelerator for convolution operations", 2019.

[35] M. Sreenivasulu y T. Meenpal, "Efficient Hardware Implementation of 2D Convolution on FPGA for Image Processing Application", 2019.

[36] M. S. Committee, *IEEE Standard for Floating-Point Arithmetic*, IEEE, ed. IEEE, 2008.

[37] J.-P. Deschamps, G. D. Sutter y E. Cantó, *Guide to FPGA Implementation of Arithmetic Functions*. Springer-Verlag GmbH, 2 de abr. de 2012, ISBN: 9400729871. dirección: https://www.ebook.de/de/product/19111721/jean_pierre_deschamps_gustavo_d_sutter_enrique_canto_guide_to_fpga_implementation_of_arithmetic_functions.html.

[38] J. M. Muller, N. Brisebarre, F. de Sinechin, C.-P. Jeannerod y V. Lefevre, *Handbook of Floating-Point Arithmetic*. Springer Basel AG, 11 de nov. de 2009, ISBN: 9780817647056. dirección: https://www.ebook.de/de/product/13923544/handbook_of_floating_point_arithmetic.html.

[39] M. L. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*. siam, 2001.

[40] A. Deshpande, *A beginner's guide to understanding convolutional neural networks*, $Https : //adeshpande3.github.io/A - Beginner's - Guide - To - Understanding - Convolutional - Neural - Networks/$, jul. de 2016.

[41] P. P. Chu, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. JOHN WILEY & SONS INC, 1 de abr. de 2006, 669 págs., ISBN: 0471720925.

[42] ——, *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*. JOHN WILEY & SONS INC, 7 de jul. de 2011, 440 págs., ISBN: 0470185317. dirección: `https://www.ebook.de/de/product/6622077/pong_p_chu_fpga_prototyping_by_vhdl_examples_xilinx_spartan_3_version.html`.

[43] M. Borgwardt, *The Floating-Point Guide*, Https://floating-point-gui.de/.

[44] *Programa para crear una cadena en formato VHDL*, $https : //github.com/OkalebH/formatoVHDL.$

[45] *Programa para visualizar los resultados obtenidos en VHDL*, $https : //github.com/OkalebH/CONVOLUTIONTESTER.$