

UNIVERSIDAD POLITÉCNICA DE  
MADRID

ESCUELA TÉCNICA SUPERIOR DE  
INGENIEROS DE TELECOMUNICACIÓN



TRABAJO DE FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN  
EN FPGA DE ARQUITECTURAS  
NEUROMÓRFICAS  
CONFIGURABLES BASADAS EN  
COMPUTACIÓN APROXIMADA

NICOLAS LAMARLERE HABCHI

JUNIO 2022



# TRABAJO DE FIN DE GRADO

Título: DISEÑO E IMPLEMENTACIÓN EN FPGA DE ARQUITECTURAS

NEUROMÓRFICAS CONFIGURABLES

Autor: NICOLAS LAMARLERE HABCHI

Tutor: PABLO ITUERO HERRERO

Departamento: DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

## COMITÉ DE EVALUACIÓN

Presidente: .....

Vocal: .....

Secretario: .....

Los miembros del tribunal arriba nombrados acuerdan otorgar la calificación de: .....

Madrid, a ..... de ..... de 2022



UNIVERSIDAD POLITÉCNICA DE MADRID  
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE  
TELECOMUNICACIÓN



TRABAJO DE FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN  
EN FPGA DE ARQUITECTURAS  
NEUROMÓRFICAS  
CONFIGURABLES BASADAS EN  
COMPUTACIÓN APROXIMADA

AUTOR: NICOLAS LAMARLERE HABCHI

TUTOR: PABLO ITUERO HERRERO

JUNIO 2022



*Quiero dedicar este trabajo de fin de grado y este fin de una etapa de mi vida a mi familia por apoyarme en los momentos de estrés, a mi pareja por entenderme y alegrarme las semanas y a mis amigos por sufrir las penas juntos con una cerveza en la mano.*

*Por último quiero dar las gracias a Pablo Ituero Herrero por darme esta oportunidad de desarrollar el proyecto junto a él, por enseñarme y por aguantarme.*



# Abstract

Neural networks, machine learning and deep learning are becoming increasingly important in our societies and in research. In this project we will investigate how we can use the elements that FPGAs offer us and create architectures that take advantage of the flexibility of such systems. The most demanding operations within artificial neurons are the multiplication and the computation of inputs by an activation function, which is why this work focuses on optimizing the operations by creating a trade off between precision, area, frequency and delay.

To achieve this configurable neuron, a multiplier block based on the Booth Algorithm is developed. This multiplier has the ability to vary its accuracy through 4 levels of precision with different impacts on area and maximum frequency. In addition, due to the implementation of serial and parallel architectures, the block can be reconfigured in latency. An activation function approximation block is also developed. This block offers 4 different implementations of the activation function with programmable resource utilisation and latency. Using these blocks, the FPGA configurable artificial neuron is formed which has configurable elements both in synthesis and in real time.

The architecture has an exhaustive characterization of the blocks in area, frequency and latency. The standard metrics in the evaluation of approximate computing systems are also developed. With each block a Matlab code is generated which, given the network parameters, automatically generates the files necessary for the operation of the neuron in VHDL syntax.

The project is developed around a Nexys 4 DDR development board manufactured by Digilent. This board has an Artix-7 FPGA from Xilinx and as programming environment we will use Vivado and the VHDL language.

For the simulations and verification of results we will use the simulations offered by Vivado and Matlab. Finally, a configurable and fully parameterizable artificial neuron is formed, with a maximum frequency of 112 MHz and a resource utilization of 186 LUTs and 14 Flip Flops.

## **Key Words**

FPGA, VHDL, Approximate computing, Neural nets, Artificial neurons, Configurable architectures.

# Resumen

Las redes neuronales, *machine learning* y *deep learning* están tomando cada vez más importancia en nuestras sociedades y en la investigación. En este proyecto vamos a investigar como podemos usar los elementos que nos dan las FPGA y crear arquitecturas que aprovechen la flexibilidad de estos sistemas. Las operaciones más costosas dentro de las neuronas artificiales son la multiplicación y el cómputo mediante función de activación, es por esto que, este trabajo se centra en optimizar las operaciones creando un compromiso entre precisión, área, latencia y retardo que permita adaptar las redes mediante el uso de neuronas configurables.

Para lograr esta neurona configurable se desarrolla un bloque multiplicador basado en el Algoritmo de Booth. Éste tiene la capacidad de variar su precisión en 4 niveles con diferentes impactos en área y frecuencia máxima, además, gracias a la implementación de arquitecturas en serie y en paralelo, se puede configurar el bloque en latencia. Se desarrolla también, un bloque generador de aproximaciones de la función de activación, se ofrecen 4 implementaciones con una utilización de recursos y latencia programables. Haciendo uso de estos bloques se forma la neurona artificial configurable en FPGA que tendrá elementos de configurabilidad tanto en síntesis como en tiempo real.

La arquitectura cuenta con una caracterización exhaustiva de los bloques en área, frecuencia y latencia, se desarrolla también el cálculo de métricas estándar en evaluación de sistemas basados en computación aproximada. Junto a cada bloque se ha diseñado un código en *Matlab* que, dados los parámetros de la red, genera automáticamente los archivos necesarios para el funcionamiento de la neurona en sintaxis VHDL.

El proyecto se desarrolla en torno a una placa de desarrollo *Nexys 4 DDR* fabricada por *Digilent*, esta cuenta con una FPGA *Artix-7* de *Xilinx* y como entorno de programación vamos a utilizar Vivado y el lenguaje VHDL. Para las simulaciones y comprobación de resultados utilizaremos las simulaciones que ofrece Vivado y *Matlab*.

Para finalizar se expone una neurona artificial configurable y completamente parametrizable con una frecuencia máxima de 112 MHZ y una ocupación de 186 LUTs y 14 *Flip Flops*.

## **Palabras clave**

FPGA, VHDL, Computación aproximada, Redes neuronales, Neuronas artificiales, Arquitecturas configurables.

# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Introducción a redes neuronales . . . . .	1
1.2. ¿Qué es una FPGA? . . . . .	3
1.3. ¿Por qué hacer una implementación en FPGA? . . . . .	5
1.4. Computación aproximada . . . . .	6
1.5. Configurabilidad en la red . . . . .	6
1.6. Programación hardware . . . . .	6
1.7. Objetivos . . . . .	7
1.8. Contribuciones . . . . .	7
1.9. Desarrollo temporal del trabajo . . . . .	8
1.10. Estructura del documento . . . . .	9
1.11. Glosario . . . . .	9
<b>2. Multiplicadores</b>	<b>11</b>
2.1. Multiplicación binaria clásica . . . . .	12
2.2. Algoritmo de Booth Modificado . . . . .	12
2.2.1. Funcionamiento . . . . .	13
2.2.2. Variantes del algoritmo de Booth . . . . .	15
2.3. Mejoras al algoritmo . . . . .	17

2.3.1. Extensión de signo . . . . .	18
2.3.2. Vector de corrección de errores . . . . .	19
2.4. Configurabilidad . . . . .	21
2.4.1. Codificación de Booth aproximada . . . . .	21
2.4.2. Eliminación del último subvector de error . . . . .	23
2.5. Multiplicador completo . . . . .	24
<b>3. Caracterización del multiplicador</b>	<b>27</b>
3.1. Comparativas en área, latencia, frecuencia máxima y throughput . . .	27
3.1.1. Implementación paralela . . . . .	27
3.1.2. Implementación serie . . . . .	29
3.2. Comparativas en precisión . . . . .	30
3.3. Análisis de datos y recomendaciones . . . . .	31
<b>4. Función de activación</b>	<b>33</b>
4.1. LUT . . . . .	34
4.2. Expansión de Taylor . . . . .	34
4.2.1. Rango intermedio . . . . .	35
4.2.2. Rango superior . . . . .	35
4.3. SONF . . . . .	36
4.4. Agrupación de valores de la LUT . . . . .	36
<b>5. Caracterización del bloque de función de activación</b>	<b>39</b>
5.1. Comparativas en área, latencia, frecuencia máxima y throughput . . .	40
5.1.1. Área . . . . .	40
5.1.2. Latencia . . . . .	40

5.1.3. Frecuencia máxima . . . . .	41
5.2. Comparativas en precisión . . . . .	41
5.3. Análisis de datos y recomendaciones . . . . .	41
<b>6. Neurona completa</b>	<b>43</b>
6.1. Análisis de datos . . . . .	44
6.2. Área . . . . .	45
6.3. Latencia . . . . .	45
6.4. Frecuencia máxima y throughput . . . . .	45
<b>7. Conclusiones</b>	<b>47</b>
7.1. Conclusiones . . . . .	47
7.2. Futuras líneas de investigación . . . . .	50
<b>A. Presupuesto</b>	<b>51</b>
A.1. Recursos materiales . . . . .	51
A.1.1. Recursos hardware . . . . .	51
A.1.2. Recursos software . . . . .	51
A.2. Remuneración . . . . .	52
A.3. Aplicación de impuestos y coste total . . . . .	52
<b>B. Aspectos éticos, económicos, sociales y ambientales</b>	<b>53</b>
B.1. Aspectos éticos . . . . .	53
B.2. Aspectos sociales . . . . .	54
B.3. Impacto económico . . . . .	54
B.4. Impacto ambiental . . . . .	54

<b>C. Gráficas</b>	<b>55</b>
C.1. Gráficas comparativas de la sigmoide y errores . . . . .	55
C.1.1. Implementación con una LUT . . . . .	55
C.1.2. Implementación con la expansión de Taylor . . . . .	56
C.1.3. Implementación con SONF . . . . .	56
C.1.4. Implementación con batch . . . . .	57
<b>D. Planificación temporal y asignación de registros</b>	<b>59</b>
D.1. Rango intermedio Taylor . . . . .	60
D.2. Rango superior Taylor . . . . .	62

# Índice de figuras

1.1.	Estructura de un perceptrón [1]	2
1.2.	Función Sísmoide	2
1.3.	Estructura de una red neuronal por capas [2]	3
1.4.	Estructura de una FPGA [3]	3
1.5.	Estructura de una FPGA [3]	4
1.6.	Nexys 4 DDR [4]	4
1.7.	Cronograma	8
2.1.	Coste energético de operaciones en 45nm 0.9V [5]	11
2.2.	Multiplicación binaria de $8 \cdot 8$ bits [6]	12
2.3.	Codificación por bloques del multiplicador en Radix 2[7]	13
2.4.	Codificación por bloques del multiplicador en Radix 4[8]	14
2.5.	Suma de productos parciales de Booth[9]	15
2.6.	Arquitectura serie del algoritmo de Booth	16
2.7.	Análisis de recursos implementación serie	16
2.8.	Arquitectura paralela algoritmo del de Booth de 6 bits	17
2.9.	Análisis de recursos implementación paralela	17
2.10.	Extensión de signo optimizada[9]	18
2.11.	Implementación codificación de signo	19

2.12. Arquitectura del bloque de extensión de signo generada por Vivado . . . . .	19
2.13. Vector de corrección de error[9] . . . . .	19
2.14. Tabla de la verdad vector corrección de errores . . . . .	20
2.15. Descripción lógica vector de corrección . . . . .	20
2.16. Generación de productos parciales . . . . .	21
2.17. Expresión lógica precisión completa[10] . . . . .	21
2.18. Mapa de Karnaugh precisión completa . . . . .	22
2.19. Mapa de Karnaugh precisión modificada[10] . . . . .	22
2.20. Expresión lógica reducida[10] . . . . .	22
2.21. Mapa de Karnaugh precisión modificada[10] . . . . .	22
2.22. Expresión lógica reducida[10] . . . . .	23
2.23. Generador de bits de producto parcial . . . . .	23
2.24. Suma del vector de error[9] . . . . .	24
2.25. Suma del vector de error[9] . . . . .	24
2.26. Bloque multiplicador completo . . . . .	25
2.27. Arquitectura del bloque generador de productos parciales . . . . .	25
 3.1. Comparativa entre área y precisión . . . . .	28
3.2. Comparativa entre frecuencia/throughput y precisión . . . . .	29
3.3. Comparativa entre área y precisión . . . . .	29
3.4. Comparativa entre frecuencia/throughput y precisión . . . . .	30
3.5. Métricas computación aproximada . . . . .	31
 4.1. Arquitectura función de activación basada en LUT . . . . .	34
4.2. Expansión de Taylor la función Sigmoide[11] . . . . .	35
4.3. Aproximación de segundo orden no lineal[12] . . . . .	36

4.4.	Argupación de valores en <i>batch</i> [13] . . . . .	36
4.5.	Arquitectura agrupación de valores de la LUT . . . . .	37
5.1.	Comparativas entre área y aproximación . . . . .	40
5.2.	Comparativa entre frecuencia/throughput y aproximación . . . . .	41
5.3.	Métricas computación aproximada . . . . .	41
6.1.	Esquema de la neurona completa . . . . .	43
6.2.	Simulación de la neurona completa . . . . .	44
6.3.	Análisis del área ofrecido por la herramienta de síntesis . . . . .	45
6.4.	Área neurona mejorada . . . . .	46
7.1.	Impacto de los distintos parámetros en el bloque multiplicador . . . . .	48
7.2.	Impacto de los distintos parámetros en el bloque función de activación	49
C.1.	Gráfica implementación con una LUT . . . . .	55
C.2.	Gráfica implementación con la expansión de Taylor . . . . .	56
C.3.	Gráfica implementación con SONF . . . . .	56
C.4.	Gráfica implementación con batch . . . . .	57
C.5.	Gráfica implementación con batch . . . . .	57
C.6.	Gráfica implementación con batch . . . . .	58
D.1.	Planificación temporal y asignación de registros . . . . .	60
D.2.	Implementación . . . . .	61
D.3.	Planificación temporal y asignación de registros . . . . .	62
D.4.	Implementación . . . . .	63



# Índice de cuadros

A.1. Recursos hardware. . . . .	51
A.2. Recursos de software. . . . .	52
A.3. Remuneración. . . . .	52
A.4. Coste total del trabajo de fin de grado . . . . .	52



# Capítulo 1

## Introducción

El objeto de este trabajo de fin de grado es el análisis de uno de los modelos computacionales más utilizados en la actualidad: las redes neuronales; en concreto nos centraremos en el estudio en las neuronas que las conforman.

### 1.1. Introducción a redes neuronales

Uno de los motivos principales que han convertido las redes neuronales en el modelo de preferencia frente a ciertos problemas es su semejanza con el funcionamiento de un cerebro biológico, ya que emulan como se comunican entre sí las neuronas transmitiendo información de una a otra en una estructura de capas. Esto las convierte en extremadamente útiles para afrontar problemas de la vida real donde un modelo clásico o una estrategia de fuerza bruta no darían resultado o resultarían ineficientes.[1]

La computación convencional está enfocada al tratamiento de datos mediante una serie de instrucciones y procesos fijos. En la computación basada en redes neuronales se busca lograr la solución a un problema mediante el aprendizaje previo a través de ejemplos de forma análoga a como lo realizaría un cerebro humano. Esto ha inspirado numerosos estudios que investigan los procesos neuronales biológicos e intentos de aplicarlos a modelos computacionales.

La unidad básica de una red neuronal se denomina perceptrón[1]; en su forma más simple puede modelarse como una serie de entradas  $[x_1, x_2, x_3..., x_n]$  multiplicadas por una serie de pesos  $[w_1, w_2, w_3..., w_n]$ . Estas señales son llevadas a un sumatorio y evaluadas mediante una función de activación.

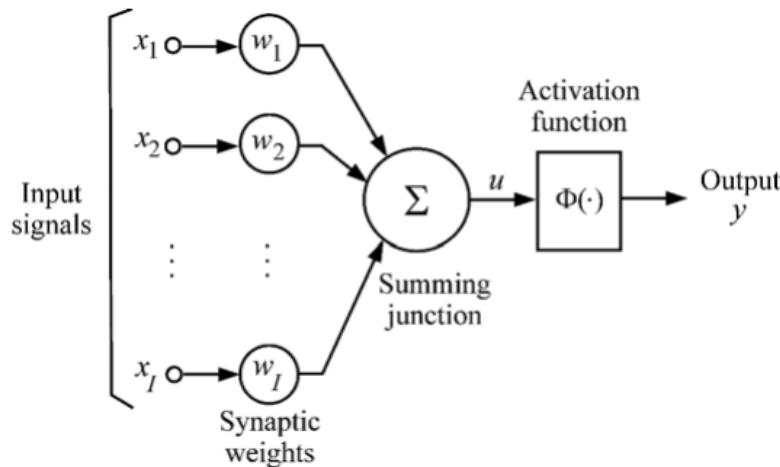


Figura 1.1: Estructura de un perceptrón [1]

Estas entradas  $x_n$  pueden ser, por ejemplo, escalas de gris codificadas de un píxel, el tamaño de un tumor, la frecuencia de una pista de audio...

Los pesos  $w_n$  son una serie de variables que, durante la fase de entrenamiento de la red, se han ido modificando en función de los resultados obtenidos y los esperados mediante un proceso llamado regresión[1].

El último paso del perceptrón es la función de activación, en esta se recibe como entrada la salida del sumatorio y se obtiene un número entre 0 y 1 a la salida. Existen múltiples implementaciones de la función, RELU, Leaky RELU, Tanh, binarias... [14] Pero en este trabajo nos vamos a centrar en la Sigmoidal caracterizada por la función  $f(x) = \frac{1}{1+e^{-x}}$ .

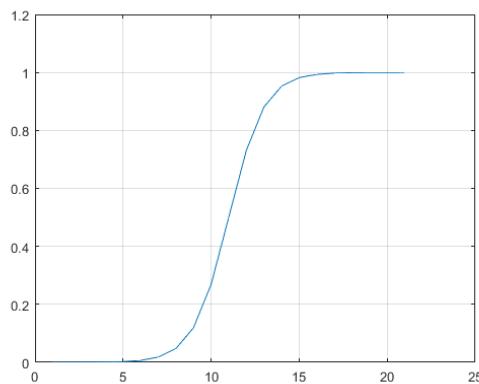


Figura 1.2: Función Sigmoidal

Con la estructura del perceptrón definida podemos alejar el enfoque y formar una red neuronal multicapa completa. En este tipo de redes los perceptrones se colocan en una estructura de capas. La primera recibe los estímulos de la red y los computa, la salida se conecta a la siguiente capa y así sucesivamente. Las capas posteriores a la primera se conocen como capas ocultas. Así la información viaja a través de la red hasta llegar a la capa de salida que nos indica la estimación de la red[2].

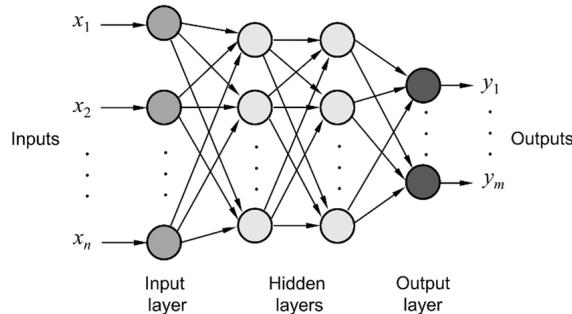


Figura 1.3: Estructura de una red neuronal por capas [2]

## 1.2. ¿Qué es una FPGA?

Las FPGAs (*Field Programmable Gate Array*) son sistemas que pertenecen a la familia de los componentes lógicos programables.

En el interior de las FPGAs encontramos una matriz de CLBs (*Configurable Logic Blocks*) que se comunican mediante una red de interconexiones reprogramables.[3]. De esta manera, mediante un lenguaje de programación *hardware* (VHDL) podemos indicar a nuestro compilador y sintetizador (Vivado) como deben realizarse estas interconexiones para obtener la funcionalidad deseada.

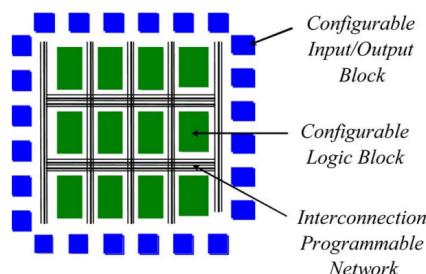


Figura 1.4: Estructura de una FPGA [3]

Los CLBs internos de las FPGAs suelen estar formados por un número variable de células lógicas, estas células lógicas a su vez están formadas por una LUT(*Look*

## CAPÍTULO 1. INTRODUCCIÓN

---

*Up Table*) que puede configurarse como una memoria o como una función, un elemento de *Carry* para las operaciones aritméticas y un Flip Flop para registrar el resultado[3].

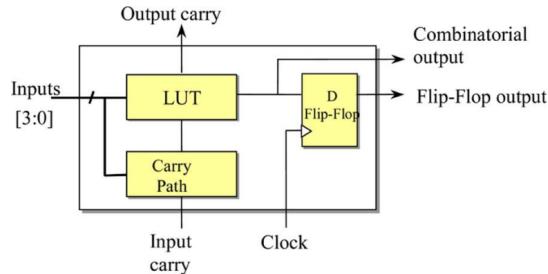


Figura 1.5: Estructura de una FPGA [3]

Es este elemento de reconfigurabilidad el que vamos a aprovechar para crear arquitecturas implementables en redes neuronales que permitan a usuarios de más alto nivel adaptar la red a los requerimientos.

Las FPGAs suelen venir en placas de desarrollo donde, en un mismo circuito, encontramos elementos de memoria DDR, controladores y diferentes puertos como ethernet, PCIE, VGA...

En este trabajo vamos a implementar las arquitecturas en una Nexys 4 DDR de Digilent basado en la FPGA Artix-7 de Xilinx.

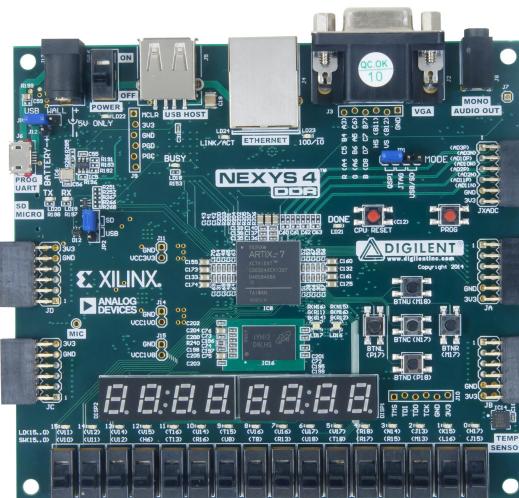


Figura 1.6: Nexys 4 DDR [4]

### 1.3. ¿Por qué hacer una implementación en FPGA?

En la implementación y ejecución de redes neuronales, las FPGAS son una alternativa muy interesante frente a los ASIC, los procesadores de uso general y los procesadores gráficos[15].

En los procesadores de uso general o CPU encontramos sistemas de muy altas velocidades pero, debido a la naturaleza paralela de las RN, no pueden hacer uso de esta potencia al completo. Las CPUs cuentan con un número limitado de núcleos optimizados para ejecutar operaciones en serie [15]. En estas implementaciones un sistema basado en FPGA puede trabajar en tandem con el procesador para hacerse cargo de las operaciones más costosas.

Los procesadores gráficos o GPU sí cuentan con esta optimización en la ejecución paralela de datos [15], lo que los hace la opción mas viable por el momento; sin embargo no vienen sin sus desventajas. De los sistemas propuestos, éste es el que mayor consumo energético tiene [16]. Encontramos también una alta latencia debido a la compleja jerarquía de acceso a datos dentro del sistema [17] y no se puede hacer una implementación únicamente con una GPU, se tiene que contar con un procesador aparte para el tratamiento de datos.

Los sistemas ASIC son muy similares a los basados en FPGA, con la diferencia que estos carecen del elemento de flexibilidad por bloques. Estas implementaciones, sin embargo, ofrecen unos rendimientos y consumos excelentes pero pueden ser extremadamente costosos en fabricaciones de media/baja producción y tienen tiempos de desarrollo y simulación muy elevados[18].

Como punto intermedio a estos sistemas entran las FPGAs que, aunque tengan un rendimiento moderado, ofrecen tiempos de producción rápidos y una flexibilidad que les permite adaptarse a las diferentes necesidades de las redes manteniendo un precio bajo [16]. La flexibilidad de estos sistemas permiten configurabilidad tanto en síntesis como en tiempo real [17].

## 1.4. Computación aproximada

La computación aproximada es un modelo de computación basado en la idea de que, en ciertas aplicaciones, un cálculo aproximado puede darnos un resultado válido para nuestro objetivo [19]. Al relajar las restricciones de las arquitecturas en términos de precisión se encuentran mejoras considerables en otros parámetros como el área, consumo o retardo.

Esta estrategia se ha visto acentuada con el aumento de aplicaciones que requieren un tratamiento de datos intensivo como el análisis de big data, procesado de imágenes y vídeo o inteligencia artificial.

Es por esto que el uso de modelos aproximados en redes neuronales puede darnos grandes ventajas sin que haya un impacto significativo en la precisión de la red. Esto se debe a la naturaleza inherente de las redes neuronales que las hace tolerantes a errores[20].

## 1.5. Configurabilidad en la red

Hemos hablado de la flexibilidad de los sistemas FPGA y como permiten a las RN tener un elemento de configurabilidad. ¿Pero por qué es una ventaja?

Si se analizan las implementaciones de RN se observa que todas son diferentes, ya sea por el objetivo de la red, la latencia requerida, la precisión, el tipo de red...

Esto supone un problema cuando a los programadores se les ofrecen arquitecturas fijas que pueden ser insuficientes o poco eficaces.

Es por esto que este trabajo se centra en crear una serie de bloques bien definidos y genéricos en los que usuarios de más alto nivel puedan especificar de forma sencilla los parámetros más relevantes y conseguir un aumento en la eficacia del uso de los recursos que nos brindan las FPGAs.

## 1.6. Programación hardware

Actualmente nos encontramos frente a un nuevo paradigma en la computación, específicamente en las plataformas utilizadas. Cada innovación en los sistemas de procesamiento de uso general nos recuerda al inminente fin de la Ley de Moore. Esto junto a las ineficiencias en algunas tareas de los procesadores genéricos ha promovido un resurgimiento en el diseño y utilización de sistemas de procesado con fin específico[21].

Los HDL (*Hardare Description Language*) son lenguajes diseñados para la descripción de procesos y estructuras físicas en diferentes niveles de abstracción. Se dividen en lenguajes descriptivos analógicos, digitales y mixtos[22], para el desarrollo en este trabajo vamos a utilizar un lenguaje descriptivo digital VHDL, y como plataforma de desarrollo vamos a utilizar la herramienta *Vivado*.

## 1.7. Objetivos

- Investigación sobre el diseño y funcionamiento de redes neuronales artificiales.
- Estudio de la literatura existente sobre las técnicas de computación aproximada aplicables a redes neuronales. En concreto el foco estará en la multiplicación y la función de activación.
- Diseño, descripción, desarrollo y simulación de los operadores tolerantes a errores en VHDL.
- Optimización arquitectural parametrizable haciendo hincapié en el compromiso entre utilización de recursos, precisión/tolerancia a errores y retardo.
- Implementación de neurona artificial parametrizable.
- Extracción de conclusiones, estadísticas y casos de uso.

Como resultado final se obtendrá una neurona artificial configurable por el usuario que proporcionará un compromiso entre precisión, utilización de recursos y latencia.

## 1.8. Contribuciones

Este trabajo ofrece la implementación *hardware* de una neurona completa en una FPGA con las siguientes partes:

- Algoritmo de multiplicación de Booth configurable en área, retardo y precisión. El bloque multiplicador cuenta con 4 niveles de precisión además de una implementación serie y paralela
- Implementación hardware de 4 aproximaciones de la función sigmoide. Esta implementación nos da 4 elecciones sobre la aproximación a elegir además de hacer uso del bloque multiplicador.
- Extracción de datos y métricas sobre operadores basados en computación aproximada. Se ha realizado un análisis exhaustivo de las implementaciones que ofrece el sistema para ayudar a los desarrolladores a tomar decisiones sobre cuál elegir.
- Desarrollo de una neurona completa. Con los bloques diseñados se implementa una neurona configurable en precisión y latencia en FPGA.
- Se ha generado un código en *Matlab* para la generación de los parámetros necesarios en todos los bloques con sintaxis VHDL, haciendo el sistema totalmente parametrizable.

Todo el código en VHDL y Matlab puede encontrarse en Github:  
<https://github.com/NicLamrlr/Configurable-Neuron-VHDL>

## 1.9. Desarrollo temporal del trabajo

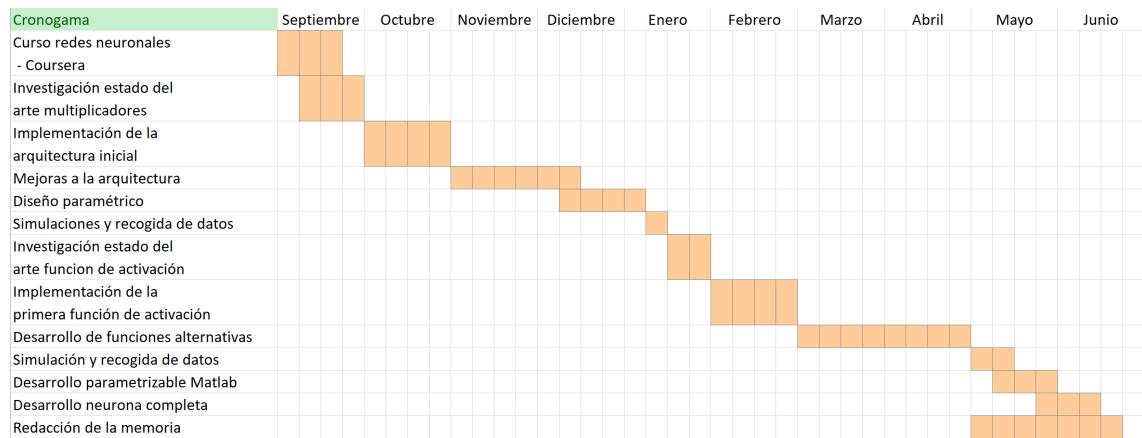


Figura 1.7: Cronograma

Este trabajo se desarrolla desde septiembre de 2021 hasta finales de julio de 2022. Comienza con un curso ofrecido por la web *Coursera* para asentar las bases teóricas en relación a las redes neuronales. En paralelo comienza un estudio sobre las distintas partes que conforman una neurona y como se pueden implementar en plataformas *hardware*. En octubre comienza la parte práctica con las primeras implementaciones del algoritmo de multiplicación, en primer lugar se realiza una implementación inicial de un multiplicador basado en el Algoritmo de Booth. A lo largo del mes de noviembre y diciembre se implementan varias mejoras al algoritmo y se añaden los niveles de configurabilidad en precisión. Con la arquitectura final terminada se desarrolla un código a través de *Matlab* para generar los parámetros necesarios y obtener así un sistema totalmente parametrizable. A principios de año se realizó un análisis exhaustivo del multiplicador para su caracterización de manera que sea fácil para los usuarios determinar la arquitectura necesaria en su implementación. A mediados de enero se comienza una investigación sobre las diferentes formas de implementar la función de activación en la literatura. Entre febrero y abril se desarrolla una primera aproximación a la función y varias alternativas con el objetivo de poder modificar el bloque función de activación dando la elección sobre cuál utilizar. Durante el mes de mayo, de la misma manera que en el bloque multiplicador, se desarrolla un código que genera los parámetros necesarios para la implementación en sintaxis VHDL. Finalmente entre mayo y junio se desarrolla la implementación de una neurona completa configurable con todos los elementos implementados en los meses anteriores, paralelamente, durante las últimas 7 semanas del proyecto, se realiza la redacción de la memoria.

## 1.10. Estructura del documento

De aquí en adelante encontraremos 6 capítulos. El capítulo 2 se centra en el desarrollo del multiplicador configurable, el capítulo 3 muestra los datos y simulaciones sobre la arquitectura del capítulo anterior, en el capítulo 4 describe la implementación de la función de activación, el capítulo 5 se caracteriza el bloque función de activación, en el capítulo 6 se describe la implementación de la neurona completa y análisis y el capítulo 7 está dedicado a conclusiones y futuras líneas de investigación.

## 1.11. Glosario

**LUT** → *Look Up Table*

**FPGA** → *Field Programmable Gate Array*

**Ca2** → *Complemento a 2*

**RN** → *Red Neuronal*

**pp** → *Producto Parcial*

**CLB** → *Configurable Logic Block*

**ASIC** → *Application Specific Integrated Circuit*

**CPU** → *Central Processing Unit*

**GPU** → *Graphics Processing Unit*

**HDL** → *Hardware Description Language*

**FA** → *Función de Activación*

**SONF** → *Second Order Non Linear Function*

**FF** → *Flip Flop*

## CAPÍTULO 1. INTRODUCCIÓN

---

## Capítulo 2

# Multiplicadores

La multiplicación digital es un elemento fundamental en los sistemas de procesamiento de señal. En sistemas de alto rendimiento las multiplicaciones y sumas de números binarios es un componente central en los procesos aritméticos. Encontramos estudios que demuestran que mas del 70 % de las instrucciones en sistemas de procesado de datos son multiplicaciones y sumas.[23]

Si analizamos en términos de coste energético de estas operaciones vemos como la multiplicación supera con creces a la suma: [5]

Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3.1pJ

Figura 2.1: Coste energético de operaciones en 45nm 0.9V [5]

En términos de latencia, con el aumento de arquitecturas basadas en procesamiento de una gran cantidad de datos como las *Deep Neural Networks* o aprendizaje profundo encontramos la necesidad de tener una cantidad significativa de multiplicadores que sean eficientes a la hora de realizar los cálculos, ya que la velocidad de las multiplicaciones suele ser la mayor limitación en términos de frecuencia y profundidad de segmentación[24].

En este trabajo vamos a abordar el problema de crear un compromiso entre área, latencia y precisión en el que, en todo momento, el usuario sea capaz de modificar los parámetros en función de las necesidades.

## 2.1. Multiplicación binaria clásica

La multiplicación binaria tradicional consiste en la extracción de productos parciales, su desplazamiento y su suma consecutiva[6].

							a7	a6	a5	a4	a3	a2	a1	a0	
			X		b7	b6	b5	b4	b3	b2	b1	b0			
					P15	P14	P13	P12	P11	P10	P9	P8	X		
				P23	P22	P21	P20	P19	P18	P17	P16	X	X		
				P31	P30	P29	P28	P27	P26	P25	P24	X	X	X	
				P39	P38	P37	P36	P35	P34	P33	P32	X	X	X	
				P47	P46	P45	P44	P43	P42	P41	P40	X	X	X	
				P55	P54	P53	P52	P51	P50	P49	P48	X	X	X	
				P63	P62	P61	P60	P59	P58	P57	P56	X	X	X	
C	S14	S13	S12	S11	S10	S9	S8	S7	S6	S5	S4	S3	S2	S1	S0

Figura 2.2: Multiplicación binaria de 8 · 8 bits [6]

Esta implementación es relativamente simple, pero puede llegar a ser muy costosa en términos de área y/o latencia ya que requiere la suma de n productos parciales en una multiplicación  $n \cdot n$ . [25]

En este contexto, se va a plantear un estudio sobre los diferentes algoritmos e implementaciones de la multiplicación en punto fijo manteniendo siempre el objetivo de la configurabilidad.

## 2.2. Algoritmo de Booth Modificado

Nos hemos decantado por la implementación del Algoritmo de Booth Modificado y sus variantes para este trabajo. Principalmente por su extendido uso, eficacia y la existencia de aproximaciones que nos permiten tomar el control sobre área, precisión y retardo de la multiplicación.

El Algoritmo de Booth original o Radix 2 descrito por Andrew Donald Booth en 1950 [26], se enfoca en la optimización de la multiplicación de números binarios en Ca2. Esta optimización se consigue mediante la recodificación del multiplicador tomando cadenas de '1' y '0' consecutivas y actuando sobre la suma cuando haya

cambios.

El funcionamiento es el siguiente[27]:

- Se concatena un '0' al LSB del multiplicando y se agrupa en bloques de 2 bits.
- Se evalúa cada bloque.
  - "00" o "11" no se hace nada, cadena sin cambio
  - "01" fin de una cadena de '1', se suma el multiplicando al acumulador
  - "10" comienzo de una cadena de '1', se resta el multiplicando al acumulador

$Y_i$	$Y_{i-1}$	Partial Product
0	0	<b>0xMultiplicand</b>
0	1	<b>1xMultiplicand</b>
1	0	<b>-1xMultiplicand</b>
1	1	<b>0xMultiplicand</b>

Figura 2.3: Codificación por bloques del multiplicador en Radix 2[7]

Esta versión tiene algunos inconvenientes, por un lado el número de operaciones con productos parciales es variable aumentando la complejidad en implementaciones paralelas y además la presencia de '1' aislados hace que el algoritmo sea ineficiente [28].

Con el objetivo de mejorar, las prestaciones se han diseñado diferentes versiones del algoritmo, en este trabajo vamos a implementar el Algoritmo de Booth modificado de Radix 4.

### 2.2.1. Funcionamiento

#### Generación de productos parciales

En esta implementación, en lugar de utilizar un bit del multiplicador y aplicarlo al multiplicando se toman grupos de 3 bits y se pasan por un codificador de la siguiente manera:

Block	Partial product (operation)
000	0
001	+1*multiplicand
010	+1*multiplicand
011	+2*multiplicand
100	-2*multiplicand
101	-1*multiplicand
110	-1*multiplicand
111	0

Figura 2.4: Codificación por bloques del multiplicador en Radix 4[8]

Para la generación de los productos parciales se toma el multiplicador. Si tiene un número de bits par se le concatena un cero por la derecha, y se toman los 3 LSBs. En función de este código aplicamos la operación correspondiente al multiplicando y se suma en un acumulador. Tras la obtención del producto parcial desplazamos el multiplicando dos posiciones a la derecha y se vuelven a tomar los 3 LSBs de éste para su cómputo.

Dado un multiplicando  $A = [a_0, a_1, a_2, a_3]$  y un multiplicador  $B = [b_0, b_1, b_2, b_3]$ , tomamos como primer bloque  $[b_2 \ b_3 \ 0]$  y aplicamos la operación correspondiente a  $A$  generando el primer producto parcial. Para la generación del segundo producto parcial, tomamos  $[b_0 \ b_1 \ b_2]$  y aplicamos la tabla 2.4 de la misma manera.

### Suma de productos parciales

La suma de los productos parciales es muy similar a la de la multiplicación binaria clásica, con la diferencia de que, a la hora de hacer la suma de los productos parciales, estos se posicionan desplazados dos posiciones respecto al producto anterior.

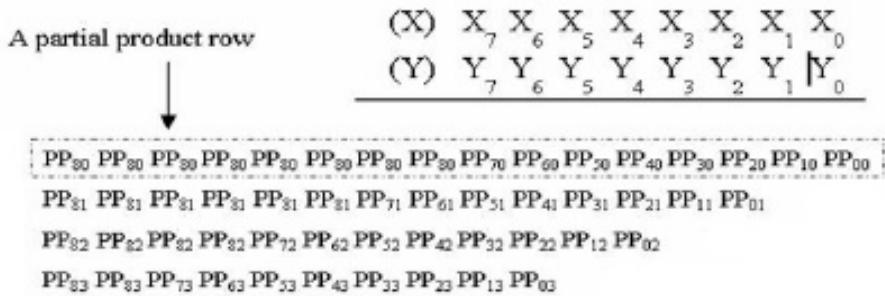


Figura 2.5: Suma de productos parciales de Booth[9]

Cabe destacar que al ser una multiplicación de números en Ca2 es necesaria la extensión de signo de cada pp.

### 2.2.2. Variantes del algoritmo de Booth

Como hemos comentado antes, este algoritmo nos da la posibilidad de hacer diferentes implementaciones con las que podemos añadir configurabilidad al diseño.

Como primera aproximación hemos hecho dos implementaciones, paralelo y serie, del algoritmo siguiendo los pasos descritos antes.

#### Implementación serie

Para la implementación serie vamos a tomar los 3 LSBs del multiplicando y generamos el producto parcial acorde a la tabla 2.3. Este producto se lleva a un acumulador y se desplaza el multiplicando dos posiciones a la derecha.

## CAPÍTULO 2. MULTIPLICADORES

---

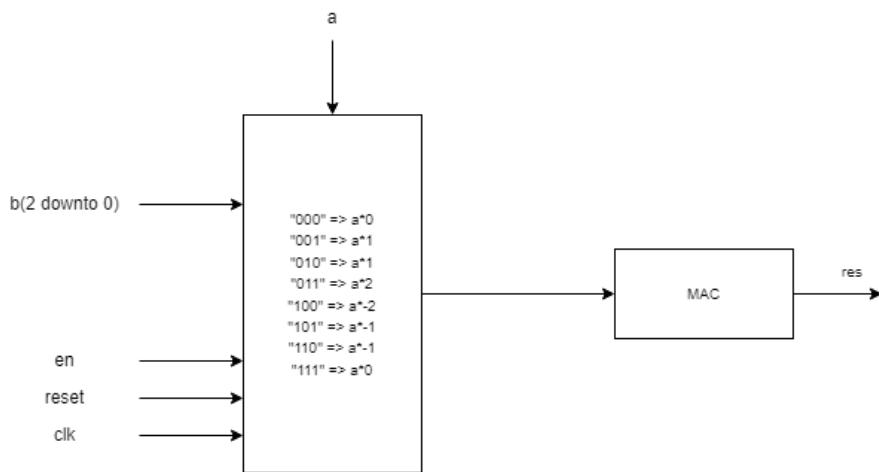


Figura 2.6: Arquitectura serie del algoritmo de Booth

Con esta implementación obtenemos una latencia de  $\frac{n}{2} + 1$  ciclos de reloj, siendo  $n$  el número de bits, y el siguiente análisis de recursos:

Utilization		Post-Synthesis   Post-Implementation	
Resource	Utilization	Available	Utilization %
LUT	62	63400	0.10
FF	33	126800	0.03
IO	27	210	12.86
BUFG	1	32	3.13

Figura 2.7: Análisis de recursos implementación serie

### Implementación paralela

En la implementación paralela se han creado dos bloques, un codificador de Booth y un sumador serie:

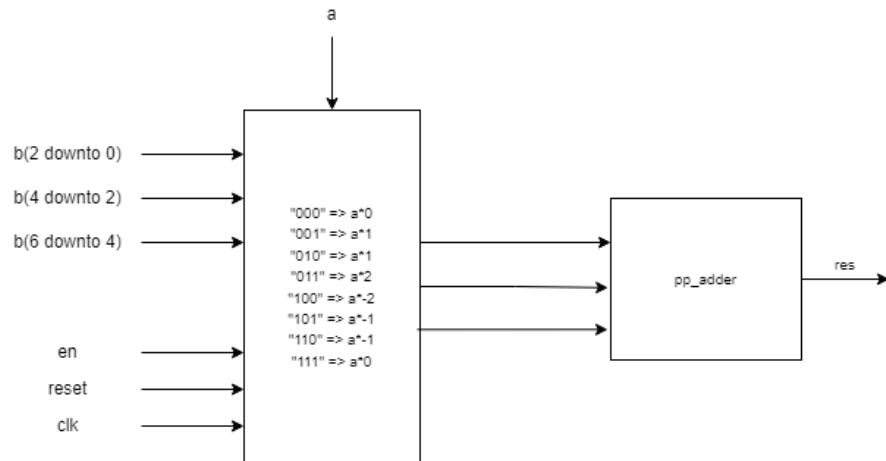


Figura 2.8: Arquitectura paralela algoritmo del de Booth de 6 bits

Esta arquitectura nos da una latencia de un ciclo y esta utilización de recursos:

Utilization		Post-Synthesis   Post-Implementation		
		Graph   Table		
Resource	Utilization	Available	Utilization %	
LUT	29	63400	0.05	
IO	24	210	11.43	

Figura 2.9: Análisis de recursos implementación paralela

### 2.3. Mejoras al algoritmo

Estas implementaciones ya nos ofrecen alguna mejora en área o latencia, pero seguimos sin tener ese elemento de configurabilidad en el que se centra este trabajo. Además, en el procesamiento de las señales hay algunos pasos inefficientes como la multiplicación del dato de entrada por un número negativo que requiere la negación y la suma de 1 al dato cuando el código de entrada es "100", "101", o "110" o la extensión de signo de cada producto parcial para la suma.

Vamos a implementar 2 optimizaciones del algoritmo y 4 niveles de precisión que tendrán impacto directo en el área y latencia de la arquitectura.

### 2.3.1. Extensión de signo

Como se comentaba antes, cada producto parcial necesita una extensión de signo de longitud variable en función del estado de la multiplicación en el que nos encontramos. Esto añade complejidad al diseño ya que tenemos productos parciales de diferentes tamaños, además de la lógica extra necesaria para la extensión.

Para optimizar esta operación vamos a utilizar un método para la extensión de signo descrito en la literatura [9]. En esta implementación, en lugar de extender el pp hasta el MSB del resultado vamos a concatenar una serie de bits específicos a la izquierda del pp los cuales codifican el signo del producto a la hora de sumar de la siguiente manera:

$$\begin{array}{r}
 \begin{array}{cccccccccc|c}
 (X) & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 & \\
 (Y) & Y_7 & Y_6 & Y_5 & Y_4 & Y_3 & Y_2 & Y_1 & | & Y_0
 \end{array} \\
 \hline
 \end{array}$$
  

$$\begin{array}{l}
 \overline{\text{PP}_{80}} \text{ PP}_{80} \text{ PP}_{80} \text{ PP}_{80} \text{ PP}_{70} \text{ PP}_{60} \text{ PP}_{50} \text{ PP}_{40} \text{ PP}_{30} \text{ PP}_{20} \text{ PP}_{10} \text{ PP}_{00} \\
 1 \quad \overline{\text{PP}_{81}} \text{ PP}_{81} \text{ PP}_{71} \text{ PP}_{61} \text{ PP}_{51} \text{ PP}_{41} \text{ PP}_{31} \text{ PP}_{21} \text{ PP}_{11} \text{ PP}_{01} \\
 1 \quad \overline{\text{PP}_{82}} \text{ PP}_{82} \text{ PP}_{72} \text{ PP}_{62} \text{ PP}_{52} \text{ PP}_{42} \text{ PP}_{32} \text{ PP}_{22} \text{ PP}_{12} \text{ PP}_{02} \\
 1 \quad \overline{\text{PP}_{83}} \text{ PP}_{83} \text{ PP}_{73} \text{ PP}_{63} \text{ PP}_{53} \text{ PP}_{43} \text{ PP}_{33} \text{ PP}_{23} \text{ PP}_{13} \text{ PP}_{03}
 \end{array}$$

Figura 2.10: Extensión de signo optimizada[9]

Al primer pp generado se le extiende el bit de signo dos veces afirmado y una vez negado. A los siguientes se les extiende una vez negado y se le concatena un '1'. Para esta implementación se ha decidido concatenar un 0 más a los pp posteriores al primero, para que todos tengan el mismo tamaño y así facilitar la arquitectura de la suma.

Este proceso se implementa con un bloque llamado sign\_extension\_trick el cual recibe el pp y una señal de control para indicar si se trata del primer producto parcial o posteriores, este bloque nos devuelve el producto parcial con la codificación de signo concatenada.

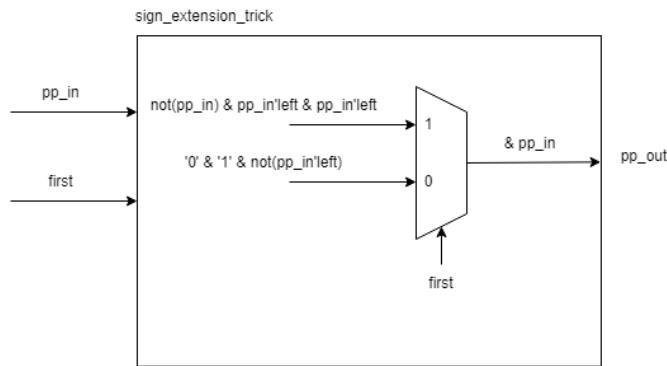


Figura 2.11: Implementación codificación de signo

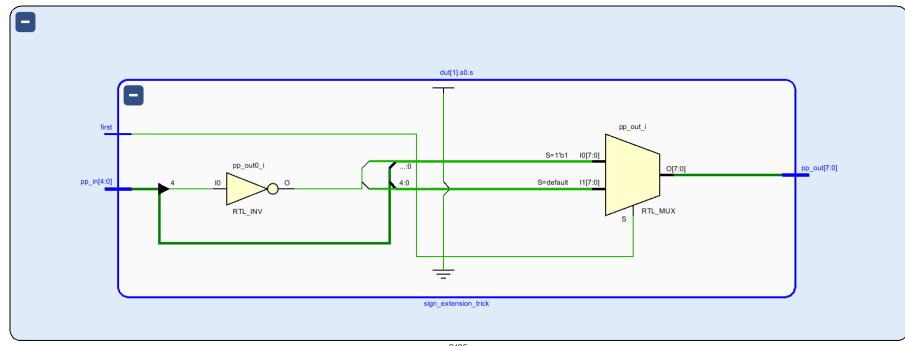


Figura 2.12: Arquitectura del bloque de extensión de signo generada por Vivado

### 2.3.2. Vector de corrección de errores

Como se muestra en la tabla 2.3, puede darse la situación en la que haya que multiplicar el multiplicando por -1 o por -2. En la implementación actual cuando se da este caso se niega el dato y se le suma 1 cada vez que se requiera. Esto puede añadir varias sumas intermedias a lo largo de la operación. Basándonos en una implementación descrita en la literatura [9], vamos a generar un vector de corrección de errores a lo largo de la operación que se sumará en el último paso junto a los pps.

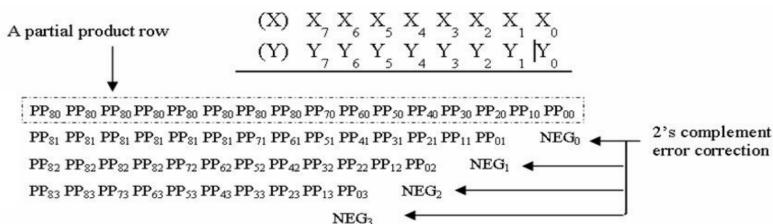


Figura 2.13: Vector de corrección de error[9]

Las variables  $NEG_n$  valdrán 0 cuando el producto parcial sea el resultado de una operación positiva y 1 en caso contrario. De esta manera, todas las sumas de la negación que habría que realizar en cada paso se computan al final de la operación simultáneamente.

Este vector se irá generando junto a los pps aplicando operaciones lógicas a los 3 bits de codificación del multiplicador. Sabemos que el vector será necesario cuando existan operaciones con números negativos, es decir, en los códigos "100", "101", o "110". Con esta información se generó una tabla de la verdad que cubriera estos casos.

	<b>A</b>	<b>B</b>	<b>C</b>	<b>Y</b>
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figura 2.14: Tabla de la verdad vector corrección de errores

Con la tabla generada vemos que aplicando la siguiente ecuación a los bits de codificación obtenemos la parte del vector correspondiente a ese pp.

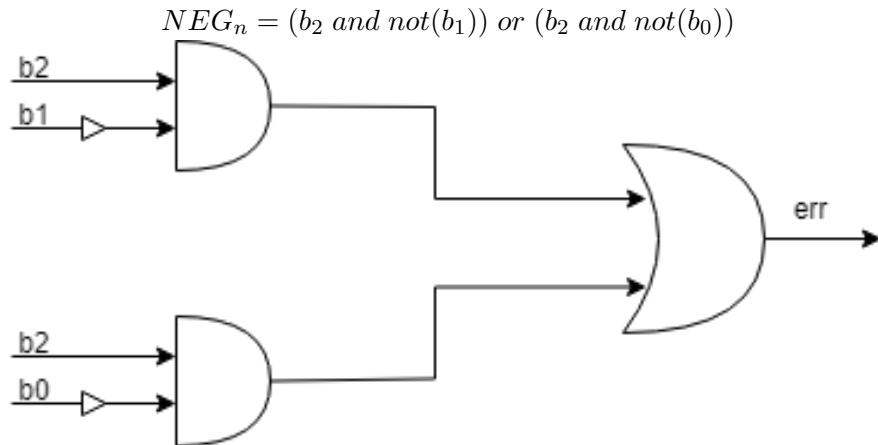


Figura 2.15: Descripción lógica vector de corrección

A la salida del sistema le añadimos un 0 por la izquierda para compensar el desplazamiento de dos posiciones entre pps, y cada vector de error generado por cada pp se concatena en una señal de longitud  $2*(\text{número de productos parciales})$  que se sumará al final  $NEG = [NEG_n \ NEG_{n+1} \ NEG_{n+2} \dots]$

## 2.4. Configurabilidad

### 2.4.1. Codificación de Booth aproximada

Con la implementación actual los pps se generan mediante un multiplexor y nos dan el resultado completo, por lo que no podemos añadir ningún elemento de computación aproximada en el sistema.

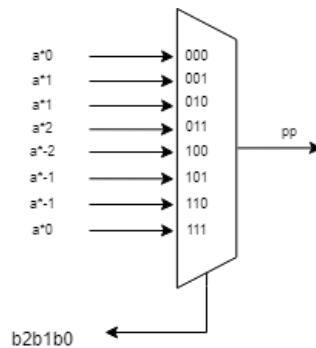


Figura 2.16: Generación de productos parciales

Vamos a realizar la implementación basándonos en un diseño propuesto en la literatura [10]. Esta arquitectura se basa en la generación de los productos parciales bit por bit, para ello primero se extrae la expresión lógica del pp, extracción directa en función de la figura 2.3.

$$pp_{ij} = (b_{2i} \oplus b_{2i-1})(b_{2i+1} \oplus a_j) + \\ \overline{(b_{2i} \oplus b_{2i-1})(b_{2i+1} \oplus b_{2i})(b_{2i+1} \oplus a_{j-1})}.$$

Figura 2.17: Expresión lógica precisión completa[10]

Con este diseño podemos implementar una estructura por bloques que reciba 2 bits del multiplicando y los 3 bits de codificación del multiplicador para generar un bit del pp. Esta estructura por bloques es dinámica y completamente parametrizable en función del tamaño de los datos de entrada. Éste desarrollo nos da como resultado un procesado exacto de la multiplicación

## CAPÍTULO 2. MULTIPLICADORES

---

Para modificar la precisión vamos a realizar una serie de cambios en el mapa de Karnaugh 2.18 y así generar una expresión lógica reducida.

$b_{2i+1}b_{2i}$	000	001	011	010	110	111	101	100
$a_ja_{j-1}$	00	0	0	0	1	0	1	1
	01	0	0	1	0	1	0	0
	11	0	1	1	1	0	0	0
	10	0	1	0	1	0	0	1

Figura 2.18: Mapa de Karnaugh precisión completa

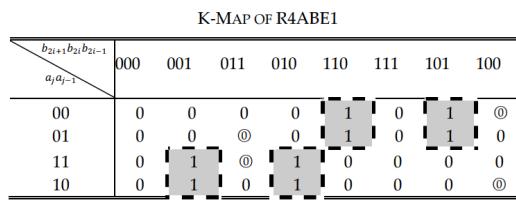


Figura 2.19: Mapa de Karnaugh precisión modificada[10]

La figura 2.19 muestra cómo en las posiciones marcadas con un 0 redondeado se ha sustituido el 1 por 0, esto reduce significativamente la complejidad de la expresión lógica a dos puertas OR y una AND.

$$\begin{aligned}
 pp_{ij} &= a_j \overline{b_{2i+1}} \overline{b_{2i}} b_{2i-1} + a_j \overline{b_{2i+1}} b_{2i} \overline{b_{2i-1}} \\
 &\quad + \overline{a_j} b_{2i+1} b_{2i} \overline{b_{2i-1}} + \overline{a_j} b_{2i+1} \overline{b_{2i}} b_{2i-1} \\
 &= (b_{2i} \oplus b_{2i-1})(b_{2i+1} \oplus a_j).
 \end{aligned}$$

Figura 2.20: Expresión lógica reducida[10]

Este paso se puede extender más con otra modificación del mapa, según se muestra en la figura 2.21.

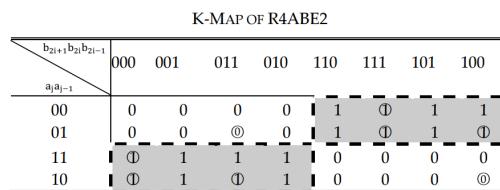


Figura 2.21: Mapa de Karnaugh precisión modificada[10]

Los bits redondeados indican que han sido modificados respecto al original. Este mapa nos deja con la siguiente expresión lógica que hace uso de un bit del multiplicando y multiplicador y una única puerta OR.

$$pp_{ij} = a_j \overline{b_{2i+1}} + \bar{a}_j b_{2i+1} = b_{2i+1} \oplus a_j.$$

Figura 2.22: Expresión lógica reducida[10]

Esta modificación se implementa en los bloques generadores de pps en función de una señal de control de precisión que tiene 3 niveles (3 - precisión total, 2 - precisión media, 1 - precisión baja) y que se implementa en el momento de la síntesis. Tras la implementación de los diferentes niveles de precisión en Vivado podemos generar el circuito lógico correspondiente y visualizarlo 2.23.

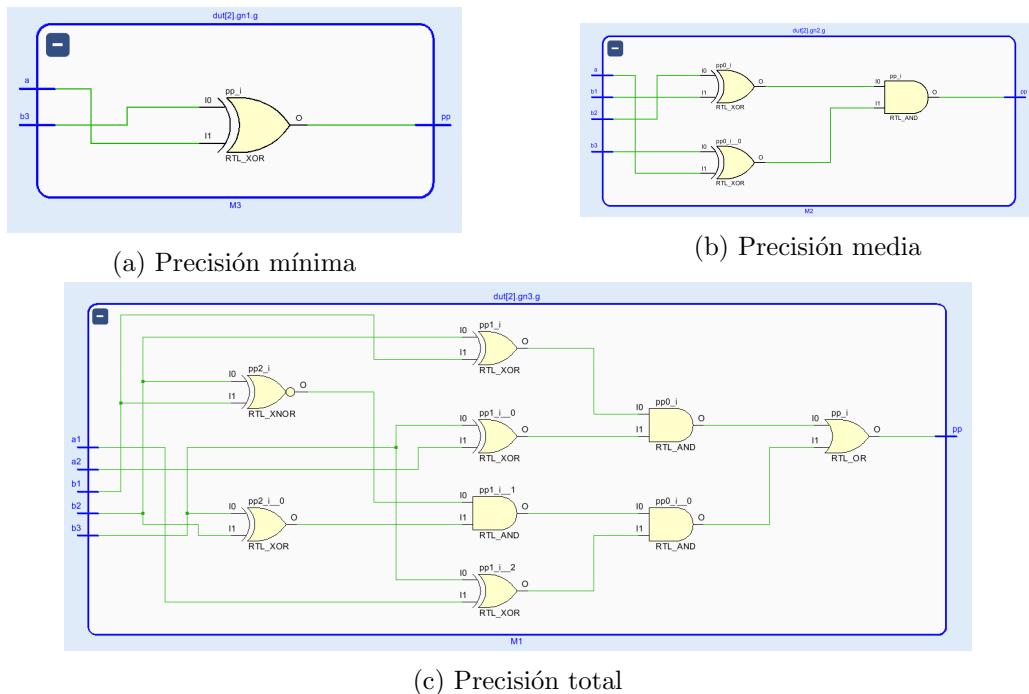


Figura 2.23: Generador de bits de producto parcial

#### 2.4.2. Eliminación del último subvector de error

En la tabla 2.13 se observa cómo, el último subvector de corrección de error  $NEG_3$ , se encuentra aislado en una fila. La solución propuesta hasta ahora se basa en sumar todos los subvectores concatenados al final de la generación de productos parciales.

A partial product row

$\downarrow$

(X)    X <sub>7</sub> X <sub>6</sub> X <sub>5</sub> X <sub>4</sub> X <sub>3</sub> X <sub>2</sub> X <sub>1</sub> X <sub>0</sub>		(Y)    Y <sub>7</sub> Y <sub>6</sub> Y <sub>5</sub> Y <sub>4</sub> Y <sub>3</sub> Y <sub>2</sub> Y <sub>1</sub>   Y <sub>0</sub>
PP <sub>80</sub> PP <sub>70</sub> PP <sub>60</sub> PP <sub>50</sub> PP <sub>40</sub> PP <sub>30</sub> PP <sub>20</sub> PP <sub>10</sub> PP <sub>00</sub>		
PP <sub>81</sub> PP <sub>81</sub> PP <sub>81</sub> PP <sub>81</sub> PP <sub>81</sub> PP <sub>71</sub> PP <sub>61</sub> PP <sub>51</sub> PP <sub>41</sub> PP <sub>31</sub> PP <sub>21</sub> PP <sub>11</sub> PP <sub>01</sub>		
PP <sub>82</sub> PP <sub>82</sub> PP <sub>82</sub> PP <sub>82</sub> PP <sub>72</sub> PP <sub>62</sub> PP <sub>52</sub> PP <sub>42</sub> PP <sub>32</sub> PP <sub>22</sub> PP <sub>12</sub> PP <sub>02</sub>		
PP <sub>83</sub> PP <sub>83</sub> PP <sub>73</sub> PP <sub>63</sub> PP <sub>53</sub> PP <sub>43</sub> PP <sub>33</sub> PP <sub>23</sub> PP <sub>13</sub> PP <sub>03</sub>		
0 NEG <sub>3</sub> 0 NEG <sub>2</sub> 0 NEG <sub>1</sub> 0 NEG <sub>0</sub>		

Figura 2.24: Suma del vector de error[9]

En la literatura científica [9], se ha propuesto la eliminación del último elemento  $NEG_3$  y concatenar los subvectores restantes a los pps sucesivos según se expone en al figura 2.25.

A partial product row

$\downarrow$

(X)    X <sub>7</sub> X <sub>6</sub> X <sub>5</sub> X <sub>4</sub> X <sub>3</sub> X <sub>2</sub> X <sub>1</sub> X <sub>0</sub>		(Y)    Y <sub>7</sub> Y <sub>6</sub> Y <sub>5</sub> Y <sub>4</sub> Y <sub>3</sub> Y <sub>2</sub> Y <sub>1</sub>   Y <sub>0</sub>
PP <sub>80</sub> PP <sub>70</sub> PP <sub>60</sub> PP <sub>50</sub> PP <sub>40</sub> PP <sub>30</sub> PP <sub>20</sub> PP <sub>10</sub> PP <sub>00</sub>		
PP <sub>81</sub> PP <sub>81</sub> PP <sub>81</sub> PP <sub>81</sub> PP <sub>81</sub> PP <sub>71</sub> PP <sub>61</sub> PP <sub>51</sub> PP <sub>41</sub> PP <sub>31</sub> PP <sub>21</sub> PP <sub>11</sub> PP <sub>01</sub> 0 NEG <sub>0</sub>		
PP <sub>82</sub> PP <sub>82</sub> PP <sub>82</sub> PP <sub>82</sub> PP <sub>72</sub> PP <sub>62</sub> PP <sub>52</sub> PP <sub>42</sub> PP <sub>32</sub> PP <sub>22</sub> PP <sub>12</sub> PP <sub>02</sub> 0 NEG <sub>1</sub>		
PP <sub>83</sub> PP <sub>83</sub> PP <sub>73</sub> PP <sub>63</sub> PP <sub>53</sub> PP <sub>43</sub> PP <sub>33</sub> PP <sub>23</sub> PP <sub>13</sub> PP <sub>03</sub> 0 NEG <sub>2</sub>		

Figura 2.25: Suma del vector de error[9]

De esta manera, en la implementación paralela evitamos realizar la suma final del vector y en la implementación serie el último ciclo dedicado a la suma desaparece. La implementación de esta modificación supone una reducción en la precisión baja, ya que afecta solo al último elemento y nos da una probabilidad de error del 37.5% [9].

## 2.5. Multiplicador completo

Todas las optimizaciones y variaciones descritas se han implementado en un bloque multiplicador completamente parametrizable y configurable por el usuario a través de señales de control.

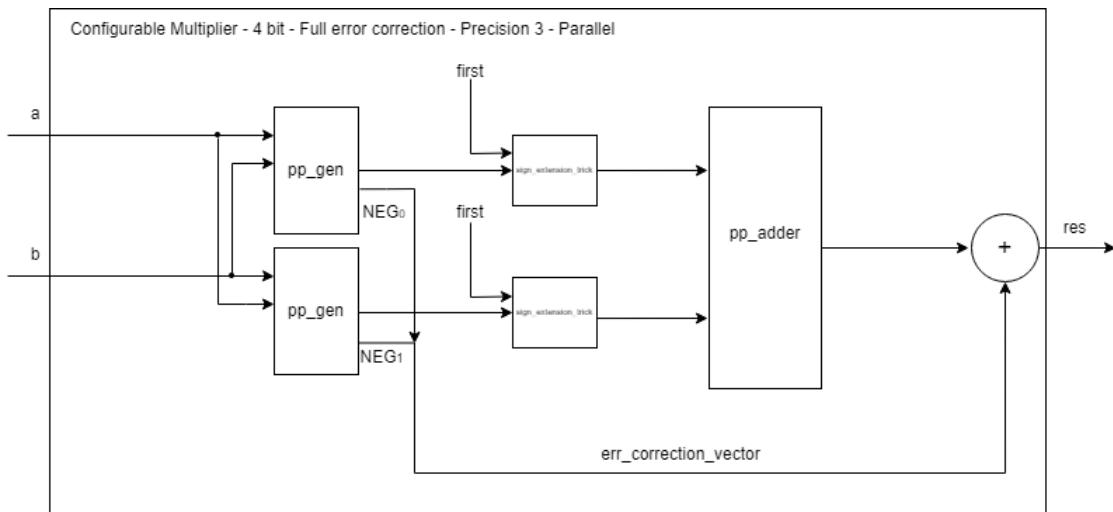


Figura 2.26: Bloque multiplicador completo

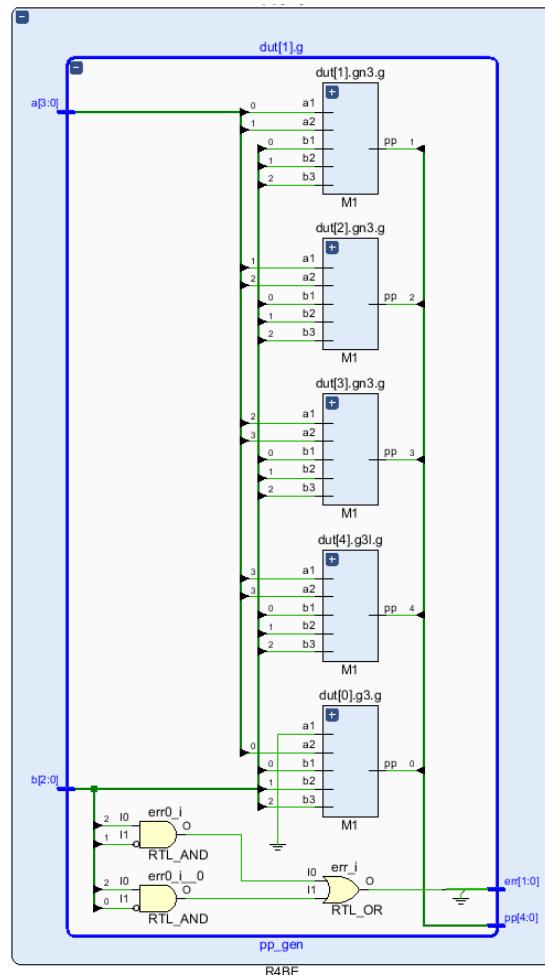


Figura 2.27: Arquitectura del bloque generador de productos parciales

## CAPÍTULO 2. MULTIPLICADORES

---

## Capítulo 3

# Caracterización del multiplicador

Con el bloque multiplicador completado pasamos a evaluar su rendimiento en diferentes escenarios. Se ha realizado un estudio exhaustivo con 4, 6, 8 y 12 bits para poder comparar las variaciones de área de las diferentes arquitecturas. El proceso ha consistido en la generación de combinaciones de números para multiplicar en *Matlab* y su cálculo mediante las simulaciones en *Vivado*, con los resultados obtenidos se muestra la comparativa entre la simulación de la arquitectura y los generados por *Matlab* en todas las combinaciones de precisión y tamaño posibles.

### 3.1. Comparativas en área, latencia, frecuencia máxima y throughput

#### 3.1.1. Implementación paralela

##### Área

Para la evaluación de área hemos utilizado la herramienta de síntesis de Vivado, esta realiza automáticamente las interconexiones de los elementos de *hardware* en función de los parámetros que le demos.

precisión	bits	<b>neg = 1</b>	<b>neg = 0</b>
		LUTS	
<b>P3</b>	4	18	11
	6	27	26
	8	42	41
	12	88	87
<b>P2</b>	4	18	8
	6	21	20
	8	38	37
	12	82	81
<b>P1</b>	4	9	5
	6	16	15
	8	31	30
	12	71	70

Figura 3.1: Comparativa entre área y precisión

En la tabla 3.1 analizamos el área que nos ocupan las diferentes arquitecturas. En la columna precisión hacemos el análisis con los tres niveles descritos en la sección 2.4.1 siendo P3 la implementación exacta y P1 el nivel más bajo de precisión. La columna neg nos indica si hemos tenido en cuenta el último subvector de corrección ( $neg = 1$ ) o no ( $neg = 0$ ), como se describe en la sección 2.4.2.

### Latencia

Al ser una implementación paralela todos los resultados se obtienen tras un ciclo de reloj.

$$\text{Latencia} = \text{clock\_period}$$

### Frecuencia máxima

Para la obtención de la frecuencia máxima vamos a utilizar las herramientas de análisis que nos ofrece Vivado. Vamos a aumentar progresivamente la frecuencia de la arquitectura para comprobar los límites del sistema.

			<b>neg = 1</b>	<b>neg = 0</b>
<b>precisión</b>		<b>bits</b>	Frecuencia máxima (MHz)/Throughput (MOp/s)	
<b>P3</b>	4	4	<b>232/232</b>	<b>330/330</b>
	6	6	<b>169/169</b>	<b>217/217</b>
	8	8	<b>135/135</b>	<b>163/163</b>
	12	12	<b>95/95</b>	<b>109/109</b>
<b>P2</b>	4	4	<b>357/357</b>	<b>344/344</b>
	6	6	<b>175/175</b>	<b>222/344</b>
	8	8	<b>137/137</b>	<b>166/166</b>
	12	12	<b>97/97</b>	<b>111/111</b>
<b>P1</b>	4	4	<b>435/435</b>	<b>416/416</b>
	6	6	<b>188/188</b>	<b>250/250</b>
	8	8	<b>147/147</b>	<b>181/181</b>
	12	12	<b>101/101</b>	<b>116/116</b>

Figura 3.2: Comparativa entre frecuencia/throughput y precisión

En esta tabla reflejamos los diferentes niveles de precisión y tamaño de bits y como afectan a la frecuencia máxima y al *throughput*, medido en  $10^6$  operaciones por segundo. Dado que la latencia es de 1 ciclo en este caso son iguales.

### 3.1.2. Implementación serie

En la implementación serie seguimos las mismas metodologías para evaluar al multiplicador.

#### Área

<b>precisión</b>	<b>bits</b>	<b>neg = 1</b>		<b>neg = 0</b>	
		LUTS	REGISTROS	LUTS	REGISTROS
<b>P3</b>	4	37	39	46	45
	6	46	48	57	63
	8	69	67	89	81
	12	79	95	115	117
<b>P2</b>	4	37	39	40	45
	6	51	53	55	63
	8	69	67	85	81
	12	70	95	112	117
<b>P1</b>	4	32	39	37	41
	6	48	53	58	63
	8	63	67	66	74
	12	69	95	113	117

Figura 3.3: Comparativa entre área y precisión

### Latencia

En esta arquitectura la latencia depende del numero de bits a multiplicar, en este caso  $(\frac{n}{2} + 1) * \text{clock\_period}$  si  $\text{neg} = 1$  pero, si desactivamos la señal de control  $\text{neg}$  reducimos en un ciclo la ejecución.

$$\text{Latencia} = \begin{cases} (\frac{n}{2} + 1) * \text{clock\_period} & \text{si } \text{neg} = 1 \\ (\frac{n}{2}) * \text{clock\_period} & \text{si } \text{neg} = 0 \end{cases}$$

### Frecuencia máxima

precisión	bits	Frecuencia máxima (MHz)/Throughput (MOp/s)	
		neg = 1	neg = 0
P3	4	266/ <b>66</b>	212/ <b>70</b>
	6	256/ <b>64</b>	204/ <b>51</b>
	8	208/ <b>41</b>	192/ <b>38</b>
	12	188/ <b>26</b>	175/ <b>25</b>
P2	4	294/ <b>98</b>	256/ <b>84</b>
	6	227/ <b>56</b>	204/ <b>51</b>
	8	208/ <b>41</b>	192/ <b>38</b>
	12	188/ <b>26</b>	178/ <b>25</b>
P1	4	294/ <b>98</b>	344/ <b>81</b>
	6	227/ <b>56</b>	188/ <b>47</b>
	8	208/ <b>41</b>	192/ <b>38</b>
	12	188/ <b>26</b>	178/ <b>25</b>

Figura 3.4: Comparativa entre frecuencia/throughput y precisión

### 3.2. Comparativas en precisión

Para realizar las comparativas en precisión hemos seguido la metodología definida para evaluación de sistemas basados en computación aproximada [29]. Siguiendo un método establecido podemos realizar comparativas entre las variaciones dentro de nuestra arquitectura y también frente a otros sistemas basados en computación aproximada. Al utilizar los mismos bloques configurables en precisión para la implementación serie y paralelo la tabla de errores es la misma para ambos

Se han extraído los siguientes indicadores:

- Error Rate (ER): Porcentaje de resultados erróneos frente al total de resultados.
- Error Distance (ED): La diferencia entre el valor exacto y el aproximado.

- Mean Error Distance (MED): Valor medio de los EDs.
- Relative Error Distance (RED): El ratio entre el ED y el valor esperado.
- Mean Relative Error Distance (MRED): El valor medio de los REDs.
- Normalized Mean Error Distance (NMED): Normalización del MED por el valor máximo obtenible de la operación exacta.

Estos valores quedan expuestos en la siguientes tabla:

		neg = 1				neg = 0			
precisión	bits	ER (%)	MED	MRED	NMED	ER(%)	MED	MRED	NMED
P3	4	0	0	0	0	37.6471	1.5059	0.2563	0.0235
	6	0	0	0	0	37.5092	6.0015	0.2504	0.0059
	8	0	0	0	0	37.5006	24.0004	0.2500	0.0015
	12	0	0	0	0	37.5000	384.0000	0.2500	9.1553e-05
P2	4	43.53	9.043	0.0439	0.1413	61.5686	9.8275	0.1284	0.1536
	6	57.6557	147.0422	0.0550	0.1436	71.5751	149.4115	0.0926	0.1459
	8	68.3070	2.3600e+03	0.0558	0.1441	78.8220	2.3682e+03	0.0842	0.1445
	12	82.1984	6.0493e+05	0.0503	0.1442	88.1295	6.0499e+05	0.0766	0.1442
P1	4	70.9804	9.0431	1.6488	0.1413	83.9216	9.7647	1.5590	0.1526
	6	86.3736	146.0425	6.0144	0.1426	92.8449	147.5223	5.7074	0.1441
	8	93.4768	2.3400e+03	22.7227	0.1428	96.6735	2.3430e+03	22.0710	0.1430
	12	98.4234	5.9918e+05	346.7499	0.1429	99.2096	5.9920e+05	344.0967	0.1429

Figura 3.5: Métricas computación aproximada

### 3.3. Análisis de datos y recomendaciones

Con todas las estadísticas extraídas podemos sacar algunas conclusiones sobre cuales serían las mejores implementaciones en función de los requisitos.

La primera elección que se le puede dar al programador es si desea hacer una implementación serie o paralela.

Si analizamos en términos de área se observa que para tamaños de bits menores de 12 la implementación paralela consume menos recursos por tanto sería la implementación óptima; sin embargo, se puede intuir que para números más grandes (32, 64, 128... bits) si estamos limitados en área, la ejecución paralela podría no ser una opción.

Si observamos la primera fila de precisión, en paralelo 3.1 el consumo de recursos al pasar de 4 bits a 12 bits aumenta en un 489% mientras que en serie 3.3 aumenta un 213%, es decir, menos de la mitad.

En términos de frecuencia máxima se observa como la implementación serie, al tener caminos críticos mas cortos tiende a tener una frecuencia máxima mayor, especialmente en longitudes de bits grandes. Al aumentar la longitud vemos como los valores

de la ejecución en serie no sufren esos cambios tan bruscos en la frecuencia, lo que sí ocurre en paralelo. La mayor diferencia entre frecuencias en ejecución paralela es del 430 % y en serie del 193 %.

Si la latencia es un elemento crítico del sistema la implementación a realizar tiene que ser paralela, sin importar el tamaño de la multiplicación, ya que esta sí va a aumentar linealmente con el número de bits en implementación serie mientras que en paralelo se mantiene constante a un ciclo.

La siguiente elección que se le da al programador es el nivel de precisión necesario, para esto vamos a evaluar la tabla 3.5.

Si es necesaria la precisión completa se debe elegir  $precision = 3$  y  $neg = 1$ , si podemos asumir algún nivel de error podemos sacar algunas conclusiones interesantes.

Si se decide poner  $neg$  a cero, es decir, obviar el ultimo subvector de error 2.4.2, obtenemos el resultado mas cercano al exacto, tanto en probabilidad de error como en error máximo, este paso puede ser una opción muy interesante en implementaciones serie ya que nos reduce en un ciclo la computación y aumenta el *throughput* sin tener un impacto demasiado significativo en el resultado final. En paralelo también nos da un incremento en la frecuencia máxima al evitarnos una suma extra en el paso final. Se caracteriza este paso como muy recomendable para cualquier tipo de implementación.

Encontramos también 3 niveles distintos de precisión 2.4.1. Observando la tabla para los niveles imprecisos ( $precision = 2$  y  $precision = 3$ ) vemos como la probabilidad de error aumenta al disminuir la precisión y aumentar el numero de bits, de la misma manera aumenta el error medio; sin embargo, el valor normalizado *NMED* no sufre variaciones significativas, lo que nos indica que el tamaño de los errores se mantiene correlado con el número de bits.

Evaluando el *MED* se puede concluir que para  $precision = 2$  se obtienen errores mayores que al hacer la implementación con  $precision = 1$ , esto puede parecer contraintuitivo pero relacionándolo con la probabilidad de error vemos que éste aumento del error se compensa con una frecuencia de errores menor. Queda a cargo del programador elegir la opción que más convenga al diseño.

## Capítulo 4

# Función de activación

Con el objetivo de realizar la implementación de una neurona completa vamos a desarrollar un bloque de función de activación siguiendo los esquemas de configurabilidad en los que se enmarca este trabajo.

La función de activación nos permite tomar las entradas de la neurona y generar una salida que actuará como estímulo de las neuronas situadas en la siguiente capa. La precisión de una red neuronal depende del número de capas de ésta y de la función de activación utilizada [30].

Para nuestra implementación hemos decidido generar nuestro bloque con 4 desarrollos distintos de la función de activación basados en la función Sigmoide.

Una propiedad de la función Sigmoide que utilizaremos en todas las implementaciones es su simetría en torno al 0:

$$f(x) = \begin{cases} s(x) & \text{si } x \geq 0 \\ 1 - s(-x) & \text{si } x < 0 \end{cases}$$

Siendo  $s(x) = \frac{1}{1+e^{-x}}$

De esta manera si el dato de entrada es positivo se computa sin cambios, pero si es negativo se niega la entrada y tras el cálculo la salida será  $1 - resultado$ .

Vamos a presentar todas las implementaciones para posteriormente hacer un estudio en términos de precisión, área, latencia y frecuencia máxima.

## 4.1. LUT

La implementación más directa de la FA es el uso de una LUT que contenga todos los valores de la función para su extracción con el dato de entrada.

Para la implementación, hemos programado un código en *Matlab* que nos generará un archivo de texto. Este archivo se inicializa en nuestro diseño y lo usaremos para la extracción de valores de la función de activación.

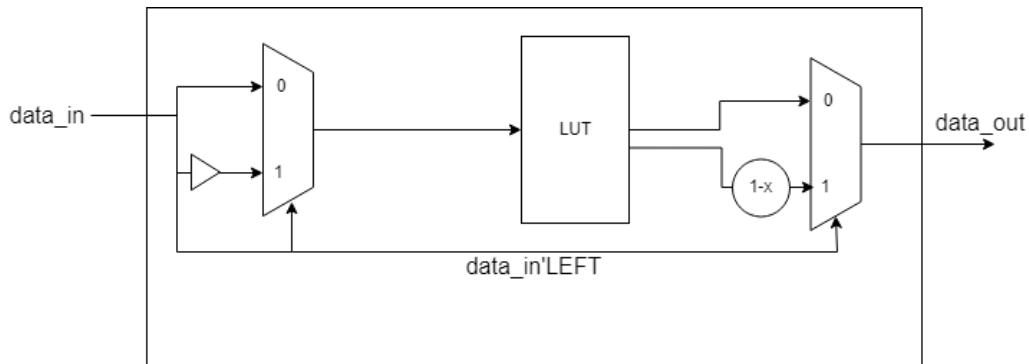


Figura 4.1: Arquitectura función de activación basada en LUT

Dado que la función Sigmoide nos da valores entre 0 y 1, sólo guardaremos en la LUT los bits fraccionarios para luego concatenar un 0 por la izquierda tras realizar la búsqueda en la tabla.

## 4.2. Expansión de Taylor

En esta implementación vamos a tomar como referencia la expansión de Taylor de la función para implementarla siguiendo el modelo *datapath*. Basándonos en la expansión expuesta en la literatura [11] procedemos a hacer la implementación hardware:

$$y = \begin{cases} 0.571859 + (0.392773)x + (0.108706)x^2 + \\ (0.014222)x^3 + (0.000734)x^4 & -\infty < x \leq -1.5 \\ \frac{1}{2} + \frac{1}{4}x - \frac{1}{48}x^3 + \frac{1}{480}x^5 & -1.5 < x < 1.5 \\ 0.428141 + (0.392773)x - (0.108706)x^2 + \\ (0.014222)x^3 - (0.000734)x^4 & 1.5 \leq x < \infty \end{cases}$$

Figura 4.2: Expansión de Taylor la función Sigmoide[11]

#### 4.2.1. Rango intermedio

Para la implementación del rango intermedio  $(-1.5, 1.5)$  hemos desarrollado la ecuación de la siguiente manera:

$$y = \frac{1}{2} + x \left[ \frac{1}{4} + x^2 \left( \frac{-1}{48} + \frac{1}{480}x^2 \right) \right] \quad (4.1)$$

De esta manera el tamaño de las multiplicaciones entre los datos de entrada será, como máximo, de  $2n$ .

Las constantes las hemos representado en formato punto fijo  $< 1.12 >$  ya que nos da una buena precisión para las constantes más pequeñas.

Con la ecuación desarrollada pasamos a realizar la planificación temporal y asignación de registros D.1.

#### 4.2.2. Rango superior

Para el rango de valores de entrada fuera de  $(-1.5, 1.5)$  sólo vamos a analizar el superior, dada la propiedad de simetría de la Sigmoide 4. Añadiremos lógica al circuito para negar los números negativos en este rango y hacer la resta final. Como se realizó en el rango intermedio, vamos a desarrollar la ecuación de  $[1.5, \infty)$ . Vamos a implementar la ecuación en la misma arquitectura que en el rango intermedio para reducir el área.

$$y = 0.428141 + x[0.392773 + x(-0.108706 + x(0.014222 - 0.000734))]$$

Para la implementación del bloque completo habrá una señal de control *mid* que nos indicará en qué rango nos encontramos y asignará las entradas de los multiplexores D.2.

### 4.3. SONF

Vamos a implementar una aproximación no-lineal de segundo orden a la Sigmoide[12], la utilización de recursos será menor que la de Taylor al necesitar solo de un multiplicador, dos desplazamientos y una resta.

$$f(x) = \begin{cases} 1 - \frac{1}{2} \left(1 - \frac{1}{4} |x|\right)^2, & 0 < x \leq 4 \\ \frac{1}{2} \left(1 - \frac{1}{4} |x|\right)^2, & -4 < x \leq 0 \\ 1, & 4 < x \\ 0, & x \leq -4 \end{cases}$$

Figura 4.3: Aproximación de segundo orden no lineal[12]

### 4.4. Agrupación de valores de la LUT

Para esta implementación vamos a modificar el diseño del apartado 4.1 basándonos en un estudio sobre las redundancias de la función Sigmoide al cuantificarla[13]. Si observamos la tabla generada con los valores de la Sigmoide encontramos valores que se repiten dentro de la misma. Vamos a realizar agrupaciones de valores que llamaremos *batch*, de cada *batch* guardamos el primer valor en una LUT y guardaremos el error entre el primer y el último valor del *batch* en una segunda LUT.

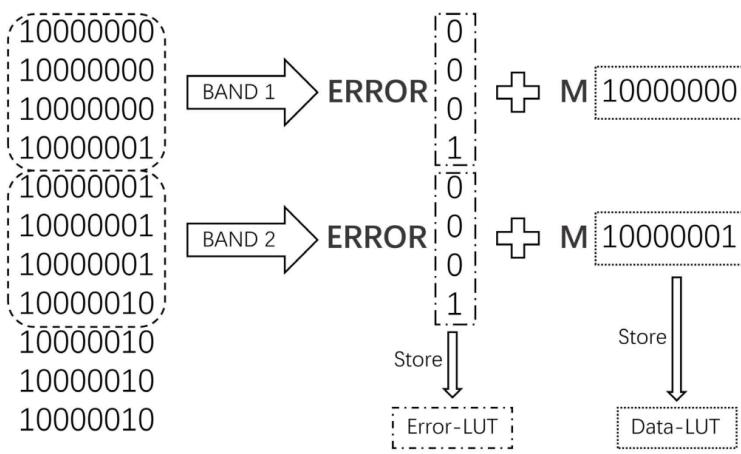


Figura 4.4: Argupación de valores en *batch*[13]

El tamaño del *batch* es variable, pudiendo tomar valores de potencias de 2 [2 4 8 16...]. Con las tablas de datos y errores formadas tomaremos el dato de entrada y lo dividiremos en función del tamaño del *batch* mediante desplazamientos a la izquierda y con el haremos una búsqueda en las dos LUTS para luego restar los resultados.

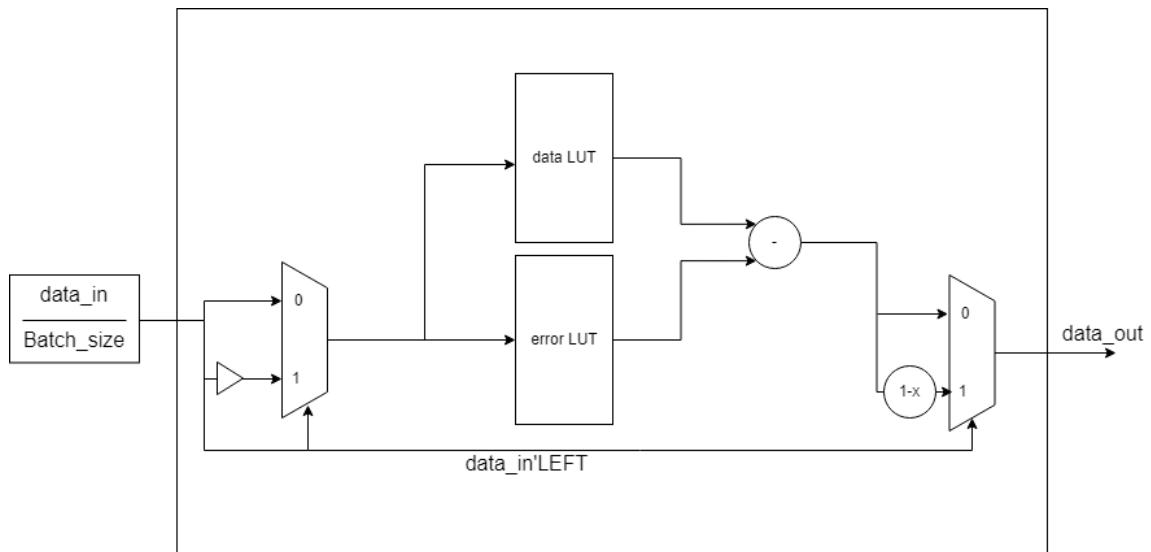


Figura 4.5: Arquitectura agrupación de valores de la LUT

De esta manera reducimos el tamaño de la tabla de valores de datos a coste de pérdida de precisión.

De la misma manera que en el apartado 4.1 eliminamos los ceros a la izquierda en ambas tablas para reducir el tamaño.



## Capítulo 5

# Caracterización del bloque de función de activación

Con las diferentes aproximaciones a la sigmoide implementadas y simuladas vamos a programar un bloque que las agrupe a todas dando al usuario la elección sobre cual utilizar.

Este bloque recibe como parámetro de entrada una variable *fun* que indica a la herramienta de síntesis que implementación generar.

- $fun = 0 \rightarrow LUT$
- $fun = 1 \rightarrow Taylor$
- $fun = 2 \rightarrow SONF$
- $fun = 3 \rightarrow Batch$

Para las simulaciones se ha generado un código en *Matlab* que toma como entrada el tamaño de la palabra, el número de bits fraccionarios, el bloque a implementar y el valor de saturación de la función. Este código se encarga de generar todas las entradas posibles al sistema, las LUTs en caso de que sean necesarias y un archivo de configuración en formato *.txt* con la sintaxis en VHDL que debe ser copiado en la librería del proyecto.

De esta manera tenemos una implementación configurable y totalmente parametrizable. Con los archivos generados por *Matlab* realizamos la simulación en *Vivado* y realizamos un análisis de datos como se hizo en el Capítulo 3. Para las simulaciones hemos tomado como entrada una palabra de 8 bits en formato punto fijo  $< 3.5 >$ , una saturación de la función igual a 4, es decir, para valores mayores de 4 la función satura a 1 y para valores menores de -4 la función satura a 0 y, para las implementaciones que requieran multiplicaciones, hemos implementado el multiplicador desarrollado en el capítulo 2.

## CAPÍTULO 5. CARACTERIZACIÓN DEL BLOQUE DE FUNCIÓN DE ACTIVACIÓN

---

### 5.1. Comparativas en área, latencia, frecuencia máxima y throughput

#### 5.1.1. Área

Aproximación	LUTs	Registros
<b>LUT</b>	21	1
<b>Taylor</b>	184	28
<b>SONF</b>	43	14
<b>Tamaño Batch</b>	23	7
	22	7
	10	7

Figura 5.1: Comparativas entre área y aproximación

Para la implementación batch 4.4, hemos simulado para agrupaciones de 2, 4 y 8 valores.

#### 5.1.2. Latencia

Las implementaciones basadas en LUT, SONF, y batch tienen una latencia de un ciclo de reloj, mientras que Taylor, como denota la planificación D.3, tiene una latencia de 4 ciclos de reloj.

$$Latencia = \begin{cases} clock\_period & si \quad fun = 0, 2, 3 \\ 4 * clock\_period & si \quad fun = 1 \end{cases}$$

## CAPÍTULO 5. CARACTERIZACIÓN DEL BLOQUE DE FUNCIÓN DE ACTIVACIÓN

---

### 5.1.3. Frecuencia máxima

Aproximación	Frecuencia Máxima (MHz)/Throughput(MOp/s)
LUT	455/ <b>455</b>
Taylor	75/ <b>18</b>
SONF	137/ <b>137</b>
<b>Tamaño Batch</b>	
<b>2</b>	455/ <b>455</b>
<b>4</b>	455/ <b>455</b>
<b>8</b>	455/ <b>455</b>

Figura 5.2: Comparativa entre frecuencia/throughput y aproximación

### 5.2. Comparativas en precisión

Siguiendo la metodología del apartado 3.2 evaluamos la precisión de la arquitectura y sus variantes.

Aproximación	ER(%)	MED	MRED	NMED
LUT	99.6094	0.0061	0.0338	0.0061
Taylor	100	0.0036	0.0327	0.0036
SONF	100	0.0220	0.1785	0.0220
<b>Tamaño Batch</b>				
<b>2</b>	100	0.0115	0.0410	0.0115
<b>4</b>	100	0.0188	0.0643	0.0188
<b>8</b>	100	0.0412	0.1457	0.0412

Figura 5.3: Métricas computación aproximada

### 5.3. Análisis de datos y recomendaciones

La aproximación de la función a elegir dependerá de los criterios del diseñador. Si buscamos la máxima precisión, según la tabla 5.3 la mejor opción sería la implementación con la expansión de Taylor aunque la opción de utilizar una LUT y *batch* de tamaño 2, también nos da buenos resultados para una cuantificación de salida de 7 bits fraccionarios.

Analizando el área de la arquitectura en términos de LUTs y registros, la implementación con una única LUT es la que menos recursos consume; sin embargo, para implementaciones con un mayor número de bits puede que no sea viable ya que el tamaño de la LUT será  $2^{n-1} \cdot (n - 1)$  bits siendo  $n$  la longitud de la palabra. La aproximación que mas elementos lógicos requiere es Taylor seguido de SONF, aunque en implementaciones con palabras de longitud mayor, son los que mantendrían un

## CAPÍTULO 5. CARACTERIZACIÓN DEL BLOQUE DE FUNCIÓN DE ACTIVACIÓN

---

tamaño más reducido, y la aproximación mediante *batch* supone un buen equilibrio entre las anteriores.

Si la latencia es una limitación las implementaciones basadas en LUTs y SONF son las más indicadas por tener una latencia de 1 ciclo.

En términos de frecuencia máxima y *throughput* las LUTs tienen los caminos críticos y latencias más pequeñas, superando con creces la velocidad del multiplicador. La implementación SONF en este aspecto tiene un rendimiento moderado y Taylor es la más limitada debido al aumento de la complejidad de las operaciones.

# Capítulo 6

## Neurona completa

Con todos los bloques que la conforman implementados y simulados, vamos a realizar las conexiones necesarias para diseñar la arquitectura de una neurona en la FPGA.

Como entrada la neurona tomará un *array* de vectores de tamaño variable. Éstos serán multiplicados por sus correspondientes pesos guardados en un archivo de texto *weights.txt*. Los resultados son llevados a un sumador cuya salida se conecta al bloque función de activación.

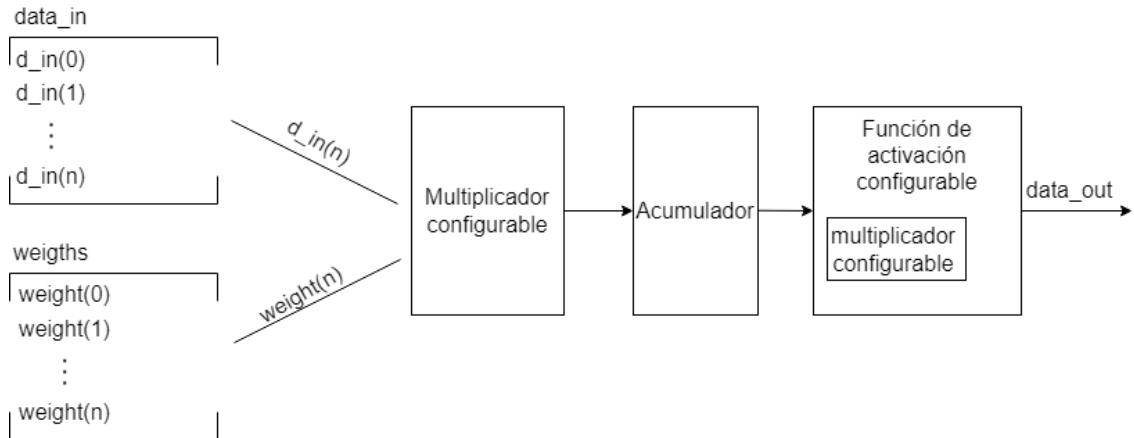


Figura 6.1: Esquema de la neurona completa

Siguiendo los pasos del capítulo 5 se ha generado en *Matlab* un código donde el usuario introduce el tamaño de la palabra, la longitud de los bits fraccionarios, el tipo de función de activación a utilizar, el nivel de saturación, la precisión del multiplicador y el número de entradas de la neurona.

Con estos datos el código genera automáticamente todos los parámetros que debe utilizar la neurona escritos en sintaxis VHDL.

## 6.1. Análisis de datos

Para la evaluación hemos generado una neurona con 8 entradas en formato  $<3.5>$ . Los valores de las entradas se han asignado de forma arbitraria al igual que los pesos. Para la implementación se han marcado los siguientes parámetros:

- $wl = 8 \rightarrow$  Longitud total de 8 bits
- $fl = 5 \rightarrow$  Longitud fraccionaria de 5 bits
- $fun = 3 \rightarrow$  Función de activación : Batch de tamaño 2
- $sat = 4 \rightarrow$  Saturación = 4
- $neg = 0 \rightarrow$  No se tiene en cuenta el último subvector de error
- $pres = 3 \rightarrow$  Precisión = 3 (máxima)
- $serpar = 0 \rightarrow$  Implementación paralela
- $n = 8 \rightarrow$  Número de entradas = 8

Con los parámetros definidos pasamos a evaluar la precisión de la neurona.

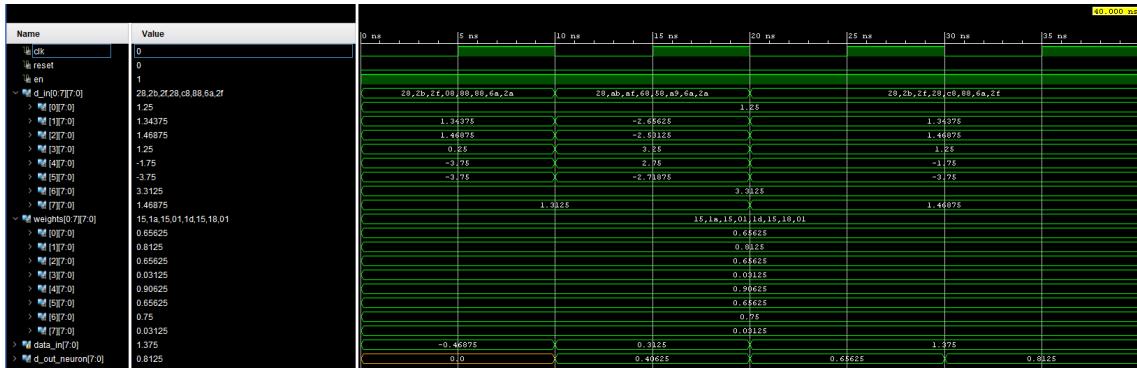


Figura 6.2: Simulación de la neurona completa

Analizando la imagen 6.2 en 10 ns vemos que tenemos las siguientes entradas en el array, denotadas en  $d\_in$  [1.25 1.34375 1.46875 0.25 -3.75 -3.75 3.3125 1.3125] y como pesos  $weights$  [0.65625 0.8125 0.65625 0.03125 0.90625 0.65625 0.75 0.03125]. Si multiplicamos cada entrada por su peso correspondiente y se hace la suma, con precisión completa, nos queda -0.4502. En nuestro caso la salida del acumulador denotada como  $data\_in$  obtenemos el resultado de -0.46875. Esto se debe al truncamiento que se hace a la entrada del bloque función de activación. La salida de la neurona, denotada como  $d\_out\_neuron$  toma un valor de 0.40625, que es el resultado de aplicar -0.46875 a la aproximación de la función Sigmoide *batch* de tamaño 2. Si

comparamos el valor en precisión completa con el obtenido mediante computación aproximada resulta un error de :  $(0.38931 - 0.40625) = -0.016$  lo que entra dentro del rango de valores de errores esperados.

Vamos a realizar también un análisis en área, latencia, frecuencia máxima y *throughput* como se realizó en los capítulos 3 y 5.

## 6.2. Área

Resource	Estimation	Available	Utilization %
LUT	252	63400	0.40
FF	7	126800	0.01
IO	76	210	36.19
BUFG	1	32	3.13

Figura 6.3: Análisis del área ofrecido por la herramienta de síntesis

Como se observa en el cuadro 6.3 tenemos una ocupación de 252 LUTs, lo que supone el 0.4 % del área total de la FPGA *Artix - 7*. Con estas condiciones la FPGA podría sintetizar unas 250 neuronas.

## 6.3. Latencia

La latencia viene definida por la implementación elegida, tanto en el bloque multiplicador y en el bloque función de activación.

En la implementación actual, con 8 bits y haciendo uso de LUTs, nuestra latencia es de 1 ciclo de reloj, pero como máximo puede llegar hasta 25 ciclos con 8 bits utilizando implementaciones serie.

## 6.4. Frecuencia máxima y throughput

Para evaluar los límites de la implementación vamos a aumentar la frecuencia del sistema comprobando que se cumplen las condiciones de temporización. Con los parámetros actuales *Vivado* reporta una frecuencia máxima de 75 MHz. Este resultado es bajo para las velocidades de los componentes que se han desarrollado, tras estudiar el camino crítico de la arquitectura se descubre que esto se debe en gran medida al sumador del resultado de las multiplicaciones, que actualmente está en una estructura de cascada y nos provoca un camino crítico largo. Si programamos manualmente una estructura de sumas en árbol y añadimos un FF entre la salida del sumador y la entrada del bloque función de activación, la frecuencia aumenta hasta

## CAPÍTULO 6. NEURONA COMPLETA

---

112 MHz. Como mejora a futuro sería interesante implementar una estructura en árbol parametrizable. Con esta implementación encontramos también una mejora en el área que ocupa el circuito:

Resource	Estimation	Available	Utilization %
LUT	186	63400	0.29
FF	14	126800	0.01
IO	52	210	24.76
BUFG	1	32	3.13

Figura 6.4: Área neurona mejorada

Esto nos permitiría desarrollar 340 neuronas en la misma FPGA.

# Capítulo 7

## Conclusiones

### 7.1. Conclusiones

Este trabajo de fin de grado se ha enfocado en el diseño e implementación en FPGA de los bloques principales que conforman a las neuronas artificiales. Cada decisión de diseño que se ha tomado ha tenido en cuenta el objetivo fundamental de este trabajo, la configurabilidad.

De esta manera se han generado operadores aproximados configurables que crean un compromiso entre área, latencia y precisión.

En primera instancia, se ha desarrollado la implementación de un multiplicador basado en el Algoritmo de Booth. Como primer nivel de configurabilidad se ha hecho una implementación paralela y una serie con latencias de 1 ciclo de reloj y  $(\frac{n}{2} + 1)$  ciclos de reloj respectivamente. Con los diferentes tamaños de palabra se ha demostrado que, en las implementaciones serie, el área tiene un crecimiento más reducido al aumentar el número de bits frente a alternativas paralelas. Se deja al programador la elección sobre qué estructura implementar en función de las limitaciones y características de la red.

Se han implementado también 4 niveles de precisión dentro del multiplicador. Mediante simulaciones y análisis se ha comprobado que una reducción en la precisión tiene un impacto directo en área, latencia y *throughput*. Algunas de estas mejoras tienen un impacto también en la latencia reduciendo en un ciclo de reloj las implementaciones serie.

El multiplicador se ha desarrollado para ser completamente parametrizable, aceptando cualquier tamaño de palabra. Conjuntamente se ha desarrollado un código para generar archivos con los parámetros y valores necesarios para la implementación y las pruebas.

De esta manera se ha llegado a un multiplicador que, al multiplicar palabras de 4 bits, se sintetiza en una arquitectura con una ocupación mínima de 9 LUTs, una frecuencia máxima de 435 MHz y una latencia de 1 ciclo.

Siguiendo los principios de diseño del multiplicador se ha generado un bloque que implementa la función Sigmoide. Dentro de éste se han implementado 4 aproximaciones a la función diferentes: el uso de una LUT que codifica los valores de la función, el desarrollo de la función mediante la expansión de Taylor, una aproximación de segundo orden no lineal y la agrupación de la LUT en sub-bloques cuyo tamaño es configurable por el usuario reduciendo el área al aumentar el tamaño pero, también aumentando el error. Junto al bloque configurable, que da la opción al programador de elegir que aproximación utilizar, se genera también un código en *Matlab* que con la información de las características de la neurona genera un texto con la sintaxis de VHDL con todos los parámetros necesarios y las LUTS.

Al tener 4 implementaciones, más la variabilidad del tamaño de las agrupaciones en la implementación con LUTs reducidas, damos al programador una gran flexibilidad sobre cual elegir en función de los requisitos. Además se le dan herramientas para la generación de todas las constantes y valores necesarios para cada una de las implementaciones.

Esta flexibilidad se traduce en una latencia variable, de 1 a 4 ciclos, y diversos valores de área y precisión. Además, el multiplicador desarrollado en la primera parte del proyecto se ha implementado también en las aproximaciones de la función de activación que lo requieran, añadiendo mas configurabilidad al bloque.

Para la caracterización de ambos bloques en precisión se han usado métricas de estudio de computación aproximada haciendo que las arquitecturas se puedan comparar con otros sistemas basados en estos principios.

Se muestra una tabla con los distintos parámetros de configurabilidad y su efecto sobre la arquitectura.

	Área	Latencia	Frecuencia	Precisión
<b>Serie</b>	▲ *	▼	▲ *	
<b>Paralelo</b>	▼	▲	▼	
<b>Precisión = 3</b>	—	—	—	—
<b>Precisión = 2</b>	▼	—	▲	▼
<b>Precisión = 1</b>	▼	—	▲	▼
<b>Neg = 1</b>	—	—	—	—
<b>Neg = 0</b>	▲	▲	▲	—

Figura 7.1: Impacto de los distintos parámetros en el bloque multiplicador

\* Para longitudes de bit pequeñas la implementación paralela tiene mejores resultados que la serie en términos de área y frecuencia.

	Área	Latencia	Frecuencia	Precisión
LUT	✓	↑	↑	—
Taylor	✓	✓	✓	↑
SONF	—	↑	—	—
Batch	↑ **	↑	↑	—

Figura 7.2: Impacto de los distintos parámetros en el bloque función de activación

\*\* En longitudes de bit grandes, *batch* puede no ser eficiente en términos de área.

Una línea horizontal naranja indica que no hay cambio o no es significativo respecto a las demás implementaciones.

En este trabajo nos hemos centrado en ofrecer herramientas que permitan la configurabilidad de las neuronas. Para la elección de los parámetros se ha realizado una caracterización extensa sobre las combinaciones de arquitecturas para que el desarrollador pueda tomar una decisión sobre cuál elegir.

Como parte final del proyecto hemos cumplido el objetivo de este trabajo de fin de grado, que era el desarrollo de una neurona completa configurable en una FPGA. Ésta se ha formado con los bloques desarrollados en los primeros capítulos del trabajo. Se han mantenido los elementos de parametrización y configurabilidad en los que se centra el desarrollo. Cómo resultado final obtenemos una neurona con una frecuencia máxima de 112 MHz, capaz de computar un número predefinido de entradas y proporcionando una salida aproximada configurable en precisión por el usuario. Como se realizó con los demás bloques, esta neurona viene acompañada de un código en *Matlab* que genera todos los parámetros necesarios para las diferentes configuraciones.

## 7.2. Futuras líneas de investigación

Como futuras líneas de investigación sería interesante hacer un análisis de la neurona con todas las combinaciones de precisión y tipo de aproximación extrayendo resultados para optimizarla en términos de latencia y frecuencia máxima.

Si se implementa una red neuronal completa con neuronas configurables la variación de los parámetros entre capas podría estudiarse de forma algorítmica para realizar implementaciones más eficientes en función del objetivo.

El consumo es un tema que se ha mencionado en este trabajo brevemente pero no se le ha hecho un análisis. Una evaluación experimental sobre el gasto energético frente a las diferentes arquitecturas podría ser muy útil para aquellas redes neuronales basadas en sistemas con batería como podrían ser los asistentes de voz de un terminal móvil.

Para concluir, este trabajo ha sido muy enriquecedor personalmente, permitiéndome aprender nuevas técnicas de diseño hardware, mejorando mis conocimientos en VHDL y FPGA, y técnicas de simulación y obtención de métricas. Además el trabajo de investigación previo ha hecho que aprenda a buscar en la literatura, implementar conceptos expuestos y mejorar mi redacción de documentos técnicos. También me ha permitido profundizar en temas relativos a las redes neuronales y entenderlas desde una perspectiva de bajo nivel.

# Apéndice A

## Presupuesto

Para la preparación del presupuesto se han tenido en cuenta recursos materiales, software, tiempo invertido e impuestos.

### A.1. Recursos materiales

#### A.1.1. Recursos hardware

Para realizar el desarrollo e implementación de los sistemas descritos en este trabajo se ha utilizado un ordenador y la placa de desarrollo Nexys 4 DDR A.1.

Cuadro A.1: Recursos hardware.

Recursos hardware	Precio (€)	Cantidad (-)	Tiempo de uso (meses)	Tiempo de amortización (años)	total (€)
Ordenador	1200	1	10	3	333.33
Nexys 4 DDR	330	1	10	3	91.66
<b>Total Cost (€):</b>					<b>425</b>

#### A.1.2. Recursos software

En la tabla A.2 se listan todos los programas de software utilizado. Vivado, Matlab y la suite de Office se renuevan sus licencias cada año por tanto se estima su amortización en 1 año.

## APÉNDICE A. PRESUPUESTO

---

Cuadro A.2: Recursos de software.

Recursos Software	Precio (€)	Cantidad (-)	Tiempo de amortización (años)	Tiempo de uso (meses)	Total (€)
Vivado	2665	1	1	10	2495.83
MATLAB	800	1	1	10	666.66
Windows 10	145	1	-	10	145
Microsoft Office	69	1	1	10	57.5
<b>Total Cost (€):</b>					<b>3364.99</b>

## A.2. Remuneración

Este trabajo se ha realizado entre septiembre de 2021 y junio de 2022, con una duración aproximada de 380 horas. Se impone una compensación de 15€/h. Estos datos se reflejan en la tabla A.3.

Cuadro A.3: Remuneración.

Categoría laboral	Salario mensual (€)	Número de horas (h)	Coste por hora (€/h)	Total (€)
Ingeniería	2760	380	15	5700
<b>Coste total (€):</b>				<b>5700</b>

## A.3. Aplicación de impuestos y coste total

Se aplica la tasa de impuestos estándar del 21 %. El coste total queda reflejado en la tabla A.4.

Cuadro A.4: Coste total del trabajo de fin de grado

Descripción	Coste (€)
Recursos hardware	425
Recursos software	3364.99
Remuneración	5700
Subtotal	9489.99
Aplicación de impuestos (21 %)	1992.9
<b>Coste total (€):</b>	<b>11482.88</b>

## Apéndice B

# Aspectos éticos, económicos, sociales y ambientales

En este anexo se discute el impacto ético, económico, social y ambiental de este trabajo.

### B.1. Aspectos éticos

En lo relativo a las redes neuronales, se puede realizar un análisis ético a ellas mismas más que al uso que se les puede dar. En la mayoría de los casos las redes serán entrenadas con ejemplos de la vida real y sobre nuestras sociedades, por tanto, si la sociedad no es ética la red neuronal tampoco lo será, con la diferencia que los circuitos carecen de ese elemento de pudor que tenemos los humanos a la hora de expresar opiniones o tomar decisiones.

Esto se demuestra, por ejemplo, con el aviso que ofrece la herramienta *DALL·E*, una red neuronal dedicada a la generación de imágenes en función una entrada de texto:

*"While the capabilities of image generation models are impressive, they may also reinforce or exacerbate societal biases. While the extent and nature of the biases of the DALL-E mini model have yet to be fully documented, given the fact that the model was trained on unfiltered data from the Internet, it may generate images that contain stereotypes against minority groups. Work to analyze the nature and extent of these limitations is ongoing, and will be documented in more detail in the DALL-E mini model card.[31]"*

Otro aspecto muy relevante es el debate sobre si las redes neuronales podrían alcanzar la conciencia, si esto llegase a ocurrir habría un cambio en el paradigma sobre como entendemos la computación y a nosotros mismos.

## B.2. Aspectos sociales

En el aspecto social las redes neuronales jugarán un papel fundamental en el futuro. Actualmente son beneficiosas en campos como la medicina, la conducción autónoma, asistentes de voz... Pero, de la misma manera, pueden tener impactos negativos. Podría darse la situación en la que solo algunos individuos tengan acceso a las redes mas potentes y se beneficien de esta ventaja creando una brecha social. De la misma manera la seguridad y privacidad pueden verse comprometidas con algoritmos de reconocimiento facial o la desencriptación de los sistemas de seguridad.

## B.3. Impacto económico

En este trabajo se ha desarrollado una opción para implementación de redes neuronales que intenta optimizar los recursos disponible. Las FPGA como plataforma ofrecen coste reducidos frente a las otras opciones disponibles.

## B.4. Impacto ambiental

La configurabilidad que ofrecemos hace que pueda haber distintas implementaciones dentro de una misma plataforma, sin necesidad de tener que fabricar nuevos productos específicos. Además puede ofrecer una reducción en los consumos energéticos.

# Apéndice C

## Gráficas

### C.1. Gráficas comparativas de la sigmoide y errores

#### C.1.1. Implementación con una LUT

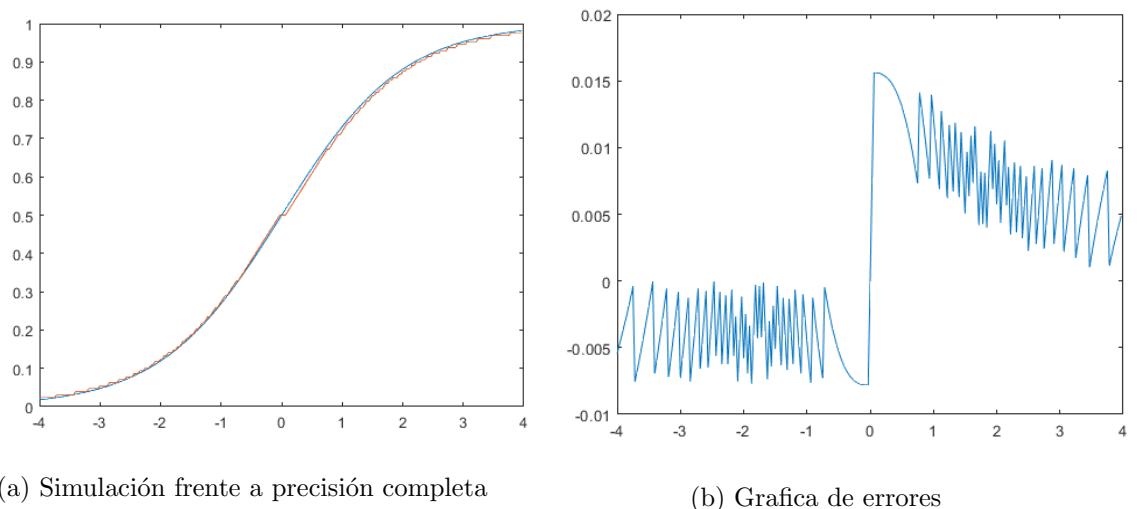


Figura C.1: Gráfica implementación con una LUT

### C.1.2. Implementación con la expansión de Taylor

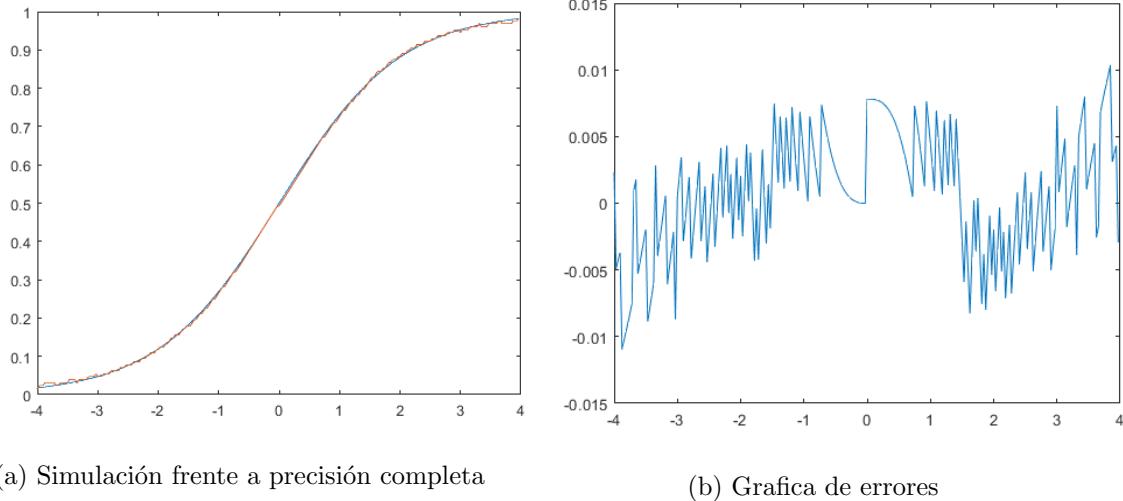


Figura C.2: Gráfica implementación con la expansión de Taylor

### C.1.3. Implementación con SONF

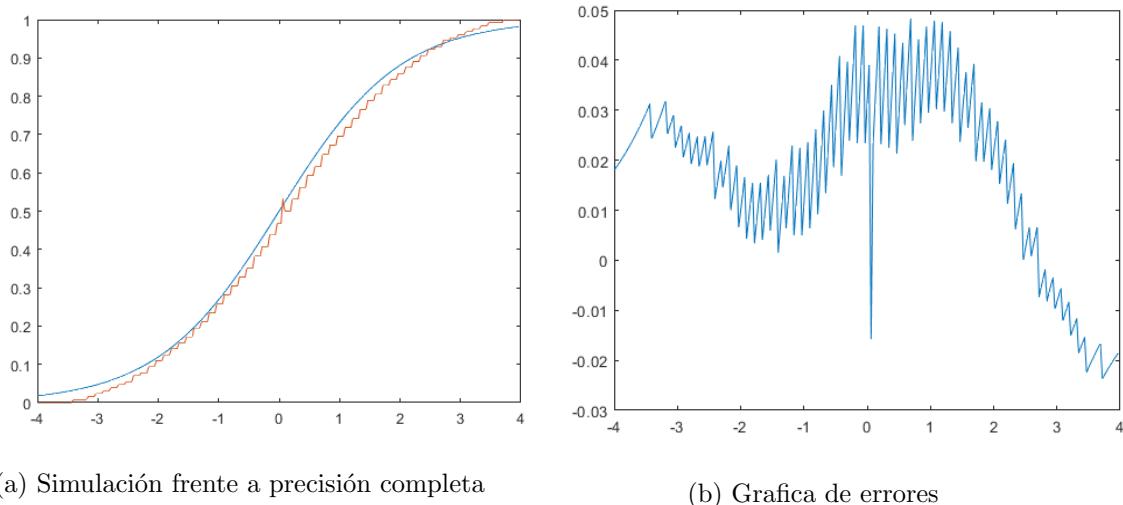
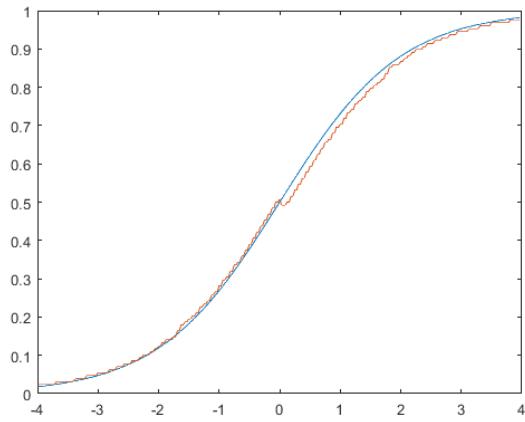


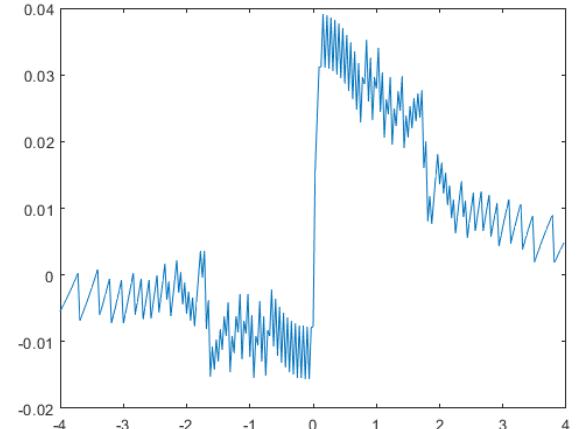
Figura C.3: Gráfica implementación con SONF

### C.1.4. Implementación con batch

#### Tamaño 2



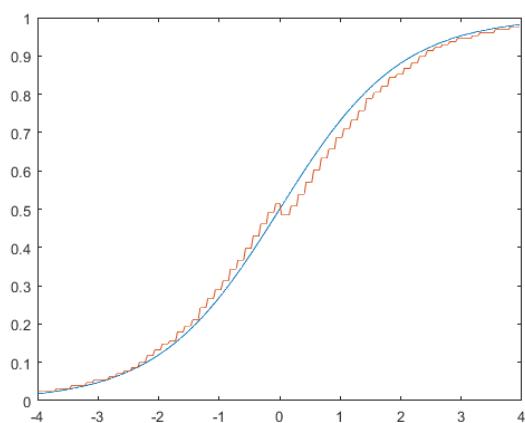
(a) Simulación frente a precisión completa



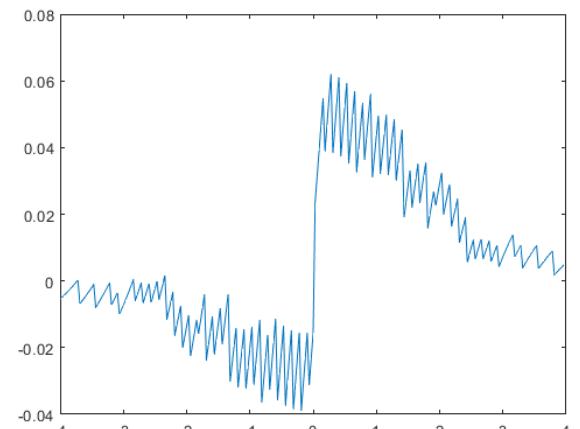
(b) Grafica de errores

Figura C.4: Gráfica implementación con batch

#### Tamaño 4



(a) Simulación frente a precisión completa



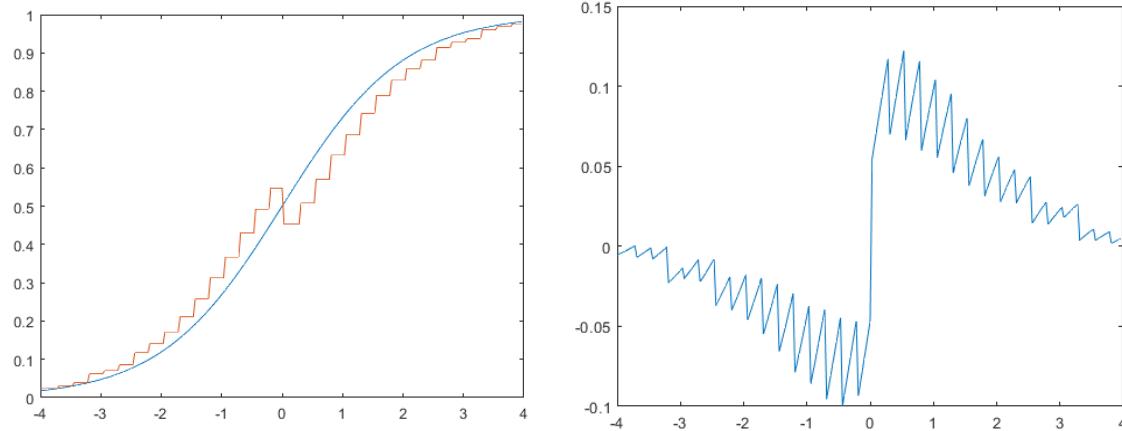
(b) Grafica de errores

Figura C.5: Gráfica implementación con batch

## APÉNDICE C. GRÁFICAS

---

### Tamaño 8



(a) Simulación frente a precisión completa

(b) Grafica de errores

Figura C.6: Gráfica implementación con batch



## Apéndice D

# Planificación temporal y asignación de registros

### D.1. Rango intermedio Taylor

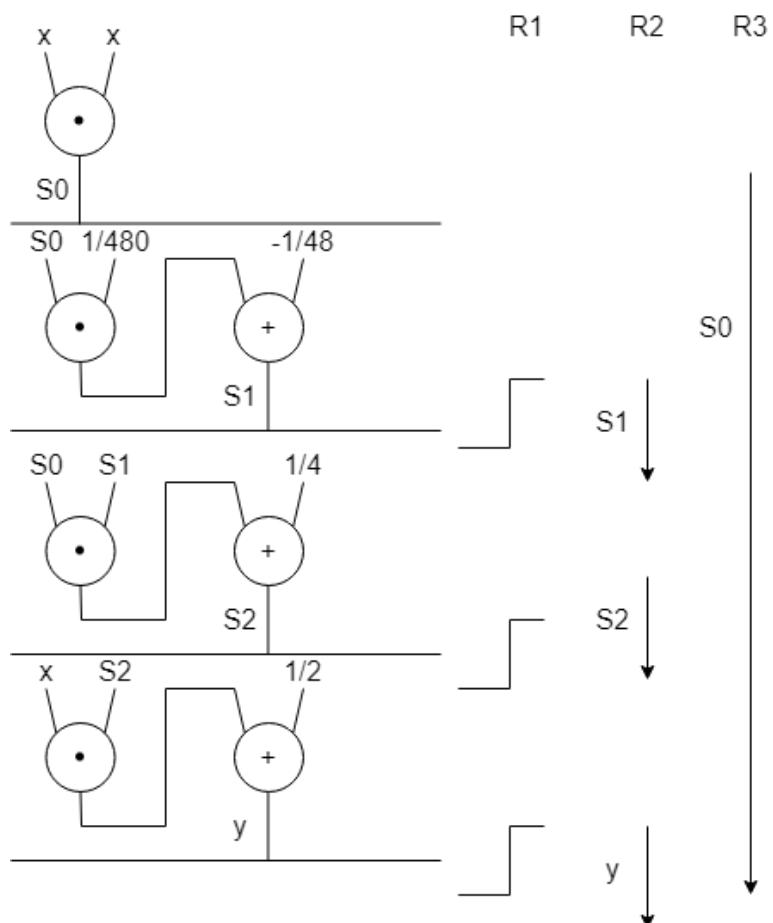


Figura D.1: Planificación temporal y asignación de registros  
60

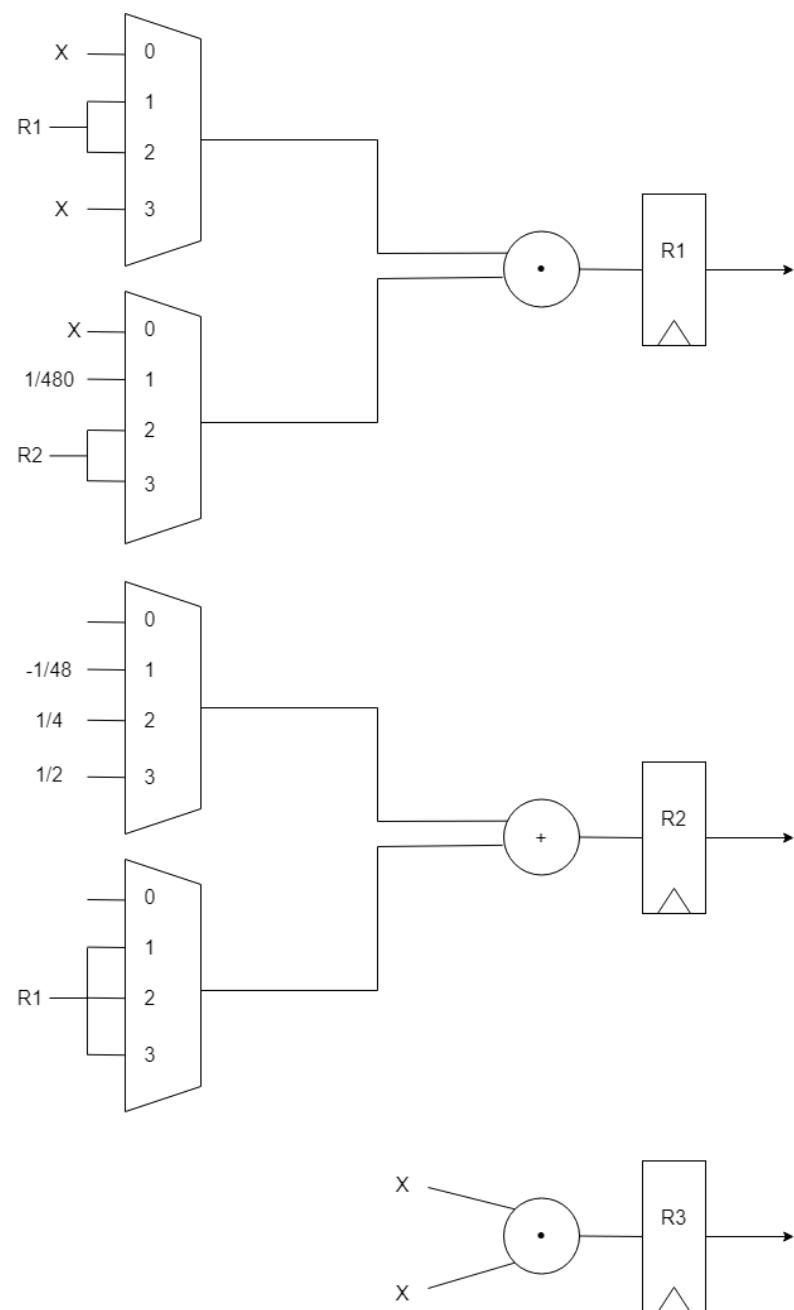


Figura D.2: Implementación

## D.2. Rango superior Taylor

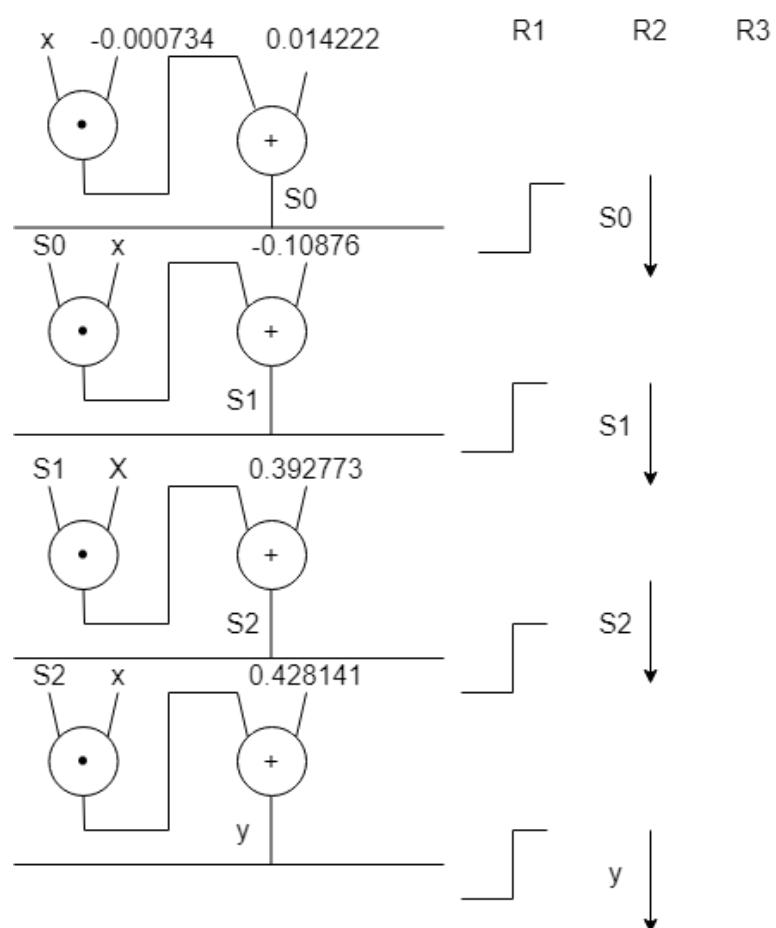


Figura D.3: Planificación temporal y asignación de registros

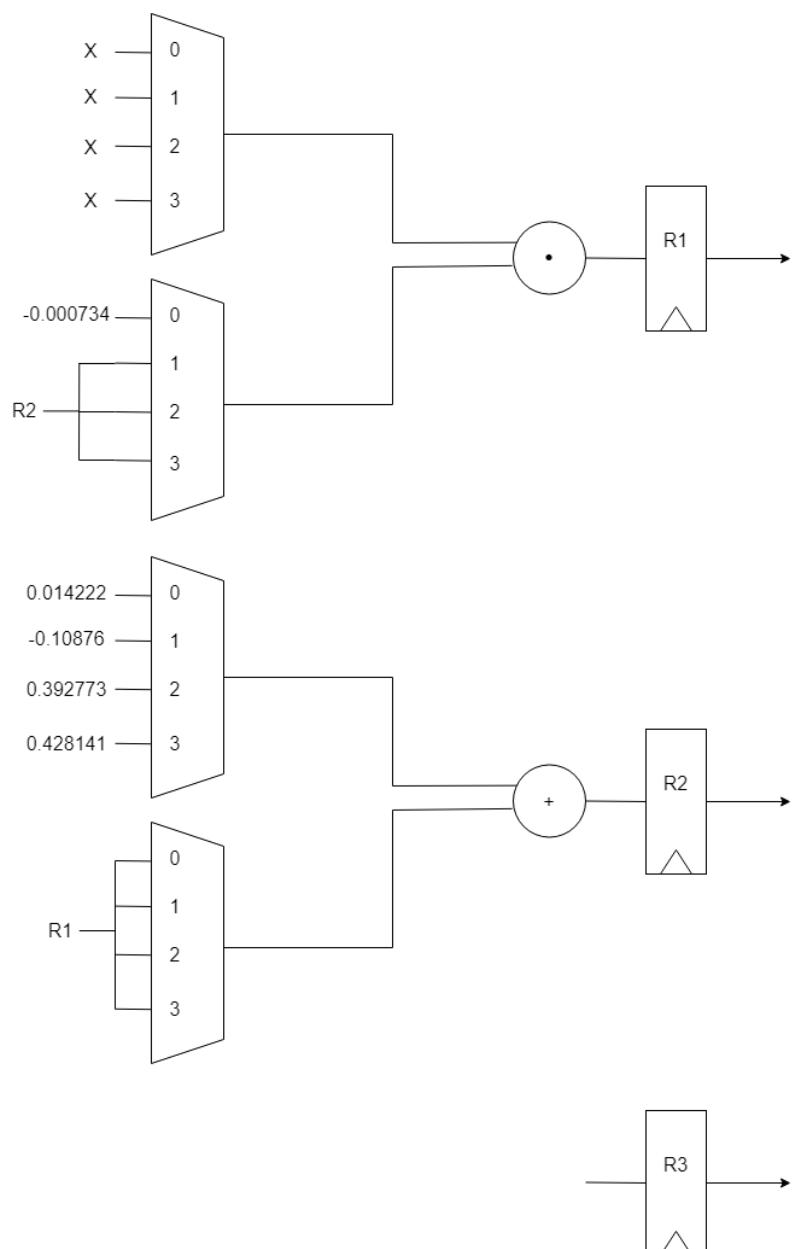


Figura D.4: Implementación

## APÉNDICE D. PLANIFICACIÓN TEMPORAL Y ASIGNACIÓN DE REGISTROS

# Bibliografía

- [1] Amos R Omondi and Jagath Chandana Rajapakse. *FPGA implementations of neural networks*, volume 365. Springer, 2006.
- [2] Abraham Pouliakis, Efrossyni Karakitsou, Niki Margari, Panagiotis Bountris, Maria Haritou, John Panayiotides, Dimitrios Koutsouris, and Petros Karakitsos. Artificial neural networks as decision support tools in cytopathology: past, present, and future. *Biomedical engineering and computational biology*, 7:BECB-S31601, 2016.
- [3] Eric Monmasson and Marcian N Cirstea. Fpga design methodology for industrial control systems—a review. *IEEE transactions on industrial electronics*, 54(4):1824–1842, 2007.
- [4] Digilent. Nexys 4 ddr.
- [5] Mark Horowitz. 1.1 computing’s energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14. IEEE, 2014.
- [6] Koyel Dey and Sudipta Chattopadhyay. Design of high performance 8 bit binary multiplier using vedic multiplication algorithm with 16 nm technology. In *2017 1st International Conference on Electronics, Materials Engineering and Nano-Technology (IEMENTech)*, pages 1–5. IEEE, 2017.
- [7] Nader Sharifi Gharabaghlo and Tohid Moradi Khaneshan. Performance analysis of high speed radix-4 booth encoders in cmos technology. *Majlesi Journal of Electrical Engineering*, 13(3):49–57, 2019.
- [8] Neeta Pandey and Saurabh Gupta. Design and implementation of novel multiplier using barrel shifters. *International Journal of Image, Graphics and Signal Processing*, 7(8):28–34, 2015.
- [9] Razaidi Hussin, Ali Yeon Md Shakaff, Norina Idris, Zaliman Sauli, Rizalafande Che Ismail, and Afzan Kamarudin. An efficient modified booth multiplier architecture. In *2008 International Conference on Electronic Design*, pages 1–4. IEEE, 2008.

## BIBLIOGRAFÍA

---

- [10] Weiqiang Liu, Liangyu Qian, Chenghua Wang, Honglan Jiang, Jie Han, and Fabrizio Lombardi. Design of approximate radix-4 booth multipliers for error-tolerant computing. *IEEE Transactions on Computers*, 66(8):1435–1441, 2017.
- [11] Onursal Çetin, Feyzullah Temurtaş, and Şenol Gülgönül. An application of multilayer neural network on hepatitis disease diagnosis using approximations of sigmoid activation function. *Dicle Tip Dergisi*, 42(2):150–157, 2015.
- [12] Syahrulanuar Ngah and Rohani Abu Bakar. Sigmoid function implementation using the unequal segmentation of differential lookup table and second order nonlinear function. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, 9(2-8):103–108, 2017.
- [13] Yusheng Xie, Alex Noel Joseph Raj, Zhendong Hu, Shaohaohan Huang, Zhun Fan, and Miroslav Joler. A twofold lookup table architecture for efficient approximation of activation functions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(12):2540–2550, 2020.
- [14] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *towards data science*, 6(12):310–316, 2017.
- [15] Avnet. Fpga vs. gpu vs. cpu – hardware options for ai applications.
- [16] Intel. Fpga vs. gpu for deep learning.
- [17] Aldec. Fpga vs gpu for machine learning applications: Which one is better?
- [18] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. You cannot improve what you do not measure: Fpga vs. asic efficiency gaps for convolutional neural network inference. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11(3):1–23, 2018.
- [19] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. Approximate computing: A survey. *IEEE Design & Test*, 33(1):8–22, 2015.
- [20] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. Approxann: An approximate computing framework for artificial neural network. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 701–706. IEEE, 2015.
- [21] Lenny Truong and Pat Hanrahan. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, volume 136 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:21, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [22] Ernst Christen and Kenneth Bakalar. Vhdl-ams-a hardware description language for analog and mixed-signal applications. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 46(10):1263–1272, 1999.

- [23] Suresh Kumar Soniya. A review of different type of multipliers and multiplier-accumulator unit. *International Journal of Emerging Trends & Technology in Computer Science (IJETTCS)*, 2(4):364–368, 2013.
- [24] Fayez Elguibaly. A fast parallel multiplier-accumulator using the modified booth algorithm. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(9):902–908, 2000.
- [25] S Arish and RK Sharma. An efficient binary multiplier design for high speed applications using karatsuba algorithm and urdhva-tiryagbhyam algorithm. In *2015 Global Conference on Communication Technologies (GCCT)*, pages 192–196. IEEE, 2015.
- [26] Andrew D Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.
- [27] Ashwini K Dhumal and SS Shirgan. Comparison between radix-2 and radix-4 based on booth algorithm. *Electronic and Telecommunication Department, International Journal of Advanced Research in Computer and Communication Engineering*, 5(12), 2016.
- [28] Sukhmeet Kaur, Manpreet Signh Manna Suman, and Signh Manna. Implementation of modified booth algorithm (radix 4) and its comparison with booth algorithm (radix-2). *Advance in Electronic and Electric Engineering*, 3(6):683–690, 2013.
- [29] Mahmoud Masadeh, Osman Hasan, and Sofiene Tahar. Comparative study of approximate multipliers. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pages 415–418, 2018.
- [30] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.
- [31] Huggin Face. Dalle.

## BIBLIOGRAFÍA

---