UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Ingeniería Electrónica

Diseño y Verificación de una red neuronal recurrente de tipo GRU y su integración en un analizador de impedancias

Trabajo Fin de Máster

Máster Universitario en Ingeniería de Sistemas Electrónicos

AUTOR/A: Ferrero Mancheño, Pablo

Tutor/a: Gadea Gironés, Rafael

Cotutor/a: Monzó Ferrer, José María

Cotutor/a externo: Fe, Jorge Deolindo

CURSO ACADÉMICO: 2024/2025

Resumen

El presente Trabajo Fin de Máster aborda la implementación de una unidad de red neuronal recurrente del tipo Gated Recurrent Unit (GRU) en una FPGA de Xilinx, utilizando Vitis HLS como herramienta de síntesis de alto nivel. Se analiza el rendimiento del diseño en términos de eficiencia computacional y precisión de los resultados, comparando su ejecución en hardware con simulaciones en software. La implementación se ha llevado a cabo en la plataforma RedPitaya, integrando una FPGA con un procesador ARM para gestionar el flujo de datos. A lo largo del desarrollo se han superado múltiples desafíos técnicos, destacando la curva de aprendizaje en el uso de herramientas de Xilinx y la optimización de recursos en la FPGA. Los resultados obtenidos demuestran que el procesamiento en hardware permite una reducción significativa de latencia y un mejor aprovechamiento de la capacidad de cómputo en paralelo, lo que hace de esta solución una alternativa viable en entornos embebidos.

Resum

Aquest Treball de Fi de Màster tracta sobre la implementació d'una unitat de xarxa neuronal recurrent del tipus Gated Recurrent Unit (GRU) en una FPGA de Xilinx, utilitzant Vitis HLS com a eina de síntesi d'alt nivell. Es realitza una anàlisi del rendiment del disseny en termes d'eficiència computacional i precisió dels resultats, comparant la seva execució en hardware amb simulacions en software. La implementació s'ha dut a terme en la plataforma RedPitaya, integrant una FPGA amb dos processador ARM per gestionar el flux de dades. Al llarg del desenvolupament s'han superat múltiples obstacles tècnics, destacant la corba d'aprenentatge en l'ús d'eines de Xilinx i l'optimització de recursos a la FPGA. Els resultats obtinguts demostren que el processament en hardware permet una reducció significativa de latència i un millor aprofitament de la capacitat de càlcul en paral·lel, fet que fa que aquesta solució siga una alternativa viable en entorns embeguts.

Abstract

This Master's Thesis focuses on the implementation of a Gated Recurrent Unit (GRU) recurrent neural network unit in a Xilinx FPGA using Vitis HLS as a high-level synthesis tool. The performance of the design is analyzed in terms of computational efficiency and result accuracy, comparing its execution in hardware with software simulations. The implementation has been carried out on the RedPitaya platform, integrating an FPGA with an ARM processor to manage data flow. Throughout the development, multiple technical challenges have been overcome, highlighting the learning curve in the use of Xilinx tools and resource optimization in the FPGA. The results obtained show that hardware processing allows for a significant reduction in latency and better utilization of parallel computing capacity, making this solution a viable alternative in embedded environments.

Quiero expresar mi mas profundo y sincero agradecimiento a todas las personas que han contribuido de manera invaluable al éxito de este proyecto de fin de máster.

En primer lugar, agradecer a mi familia por su apoyo incondicional a lo largo de mi formación académica, quienes siempre han creído en mi y han estado a mi lado en cada paso del camino.

Agradecer a mis amigos, de la infancia y de la universidad, quienes han hecho que el esfuerzo y las largas horas de dedicación fueran mas llevaderos. Su compañía ha enriquecido mi experiencia no solo a nivel académico, sino también a nivel social y personal.

Agradecer a la Universidad Politécnica de Valencia por brindarme la oportunidad de formarme como profesional y adquirir los conocimientos necesarios para afrontar este proyecto.

Por último, quiero expresar mi más sincero agradecimiento a mis tutores, Rafael Gadea y Jose Maria Monzo, por su valiosa orientación, dedicación y motivación en todo momento. Su apoyo ha sido fundamental para el logro de este trabajo. Me considero afortunado por haber tenido la oportunidad de trabajar con su respaldo, y espero aplicar todo lo aprendido en futuros proyectos.

Índice general

Índice de figuras

Índice de tablas

Listado de siglas empleadas

ACP Accelerator Coherency Port.

AI Artificial Intelligence.

ALU Arithmetic Logic Unit.

ANN Artificial Neural Network.

AP Arbitrary Precision.

APU Application Processor Unit.

ARM Advanced RISC Machine.

ASIC Application-Specific Integrated Circuit.

BSP Board Support Package.

CDFG Control and Data Flow Graph.

CLB Configurable Logic Blocks.

CNN Convolutional Neural Network.

CPU Central Processing Units.

DDR Double Data Rate.

DMA Direct Memory Access.

DNN Deep Neural Networks.

DSP Digital Signal Processor.

ECC Error-Correcting Code.

ELF Executable and Linkable Format.

ENOB Effective Numbers of Bits.

FC Fully Connected Layer.

FF Flip-Flop.

FIFO First In, First Out.

FPGA Field Programmable Gate Array.

GNU General Public License.

GPU Graphics Processing Unit.

GRU Gated Recurrent Unit.

HBM High Bandwidth Memory.

HDL Hardware Description Language.

HLS High Level Synthesis.

IDE Integrated Development Environment.

IOP Input/Output Peripheral.

IP Intellectual Property.

LSB Least Significant Bit.

LSTM Long Short-Term Memory.

LUT Look Up Table.

MLP Multilayer Perceptron.

OCM On-Chip Memory.

OS Operating System.

PL Programmable Logic.

PS Processing System.

RAM Random Access Memory.

RNN Recurrent Neural Network.

ROM Read-Only Memory.

RTL Register Transfer Level.

SCU Snoop Control Unit.

SDR Software Defined Radio.

SoC System on Chip.

TLP Task-Level Parallelism.

VHDL VHSIC Hardware Description Language.

XSA Xilinx Shell Archive.

Parte I

Memoria

Capítulo 1

Introducción

En la actualidad, la optimización del procesamiento de redes neuronales en *hardware* especializado ha cobrado una gran relevancia debido a la creciente demanda de sistemas eficientes para inteligencia artificial y aprendizaje profundo. Las unidades de procesamiento programables, como las FPGA (*Field Programmable Gate Arrays*), han emergido como una solución altamente versátil para la aceleración de cálculos computacionales intensivos. Su capacidad para realizar operaciones en paralelo con un alto grado de eficiencia las convierte en una alternativa atractiva frente a las soluciones tradicionales basadas en CPU y GPU.

En este contexto, el presente Trabajo Fin de Máster se centra en la implementación de una célula de red neuronal recurrente del tipo *Gated Recurrent Unit* (GRU) en una FPGA de Xilinx, utilizando el lenguaje de alto nivel Vitis HLS para su desarrollo. La GRU es un modelo recurrente ampliamente utilizado en aplicaciones de procesamiento de lenguaje natural y análisis de series temporales, donde la memoria y la capacidad de aprender dependencias a largo plazo juegan un papel fundamental. Su implementación en *hardware* requiere una optimización cuidadosa de los recursos disponibles para garantizar un rendimiento adecuado sin comprometer la precisión de los cálculos.

El objetivo principal de este trabajo es implementar en *hardware* una GRU y analizar su rendimiento, evaluando tanto su eficiencia computacional como la precisión de los resultados obtenidos en comparación con simulaciones en software. Para ello, se ha utilizado la plataforma RedPitaya, una herramienta versátil que permite integrar el diseño de *hardware* con la programación embebida. Esta plataforma está basada en una FPGA Xilinx Zynq, la cual integra un procesador ARM de doble núcleo junto con la lógica programable, permitiendo ejecutar software de control mientras se aceleran los cálculos mediante hardware dedicado. La implementación de este sistema supone un reto significativo debido a la complejidad de las operaciones matemáticas involucradas y la necesidad de optimizar el uso de los recursos *hardware* disponibles.

En este contexto, uno de los principales objetivos es garantizar que la implementación de la GRU se ajuste a los recursos de la FPGA, asegurando que el diseño sea viable en términos de ocupación y que, idealmente, pueda integrarse sobre un proyecto ya existente sin comprometer su funcionamiento. Además, se establece una restricción en la latencia del sistema, la cual no debe superar los 150 ciclos de reloj para garantizar una ejecución eficiente. Finalmente, se define como requisito fundamental la utilización de *High-Level Synthesis* (HLS) para el desarrollo de la red neuronal, lo que permitirá una mayor flexibilidad en el diseño y facilitará la optimización del

hardware generado.

Este trabajo también aborda el proceso de desarrollo desde una perspectiva pragmática, documentando la metodología seguida y los problemas encontrados, así como las estrategias empleadas para su resolución. A lo largo de este estudio, se analizan las ventajas del uso de herramientas de síntesis de alto nivel (HLS) en comparación con la programación tradicional en lenguajes de descripción de *hardware* como VHDL o Verilog, destacando las mejoras en la eficiencia del diseño y en la productividad del desarrollo. Además, se presentan consideraciones sobre la adaptabilidad del diseño a futuras mejoras y su viabilidad en aplicaciones reales. Es importante destacar que el tamaño de esta implementación es considerablemente elevado, ya que el procedimiento seguido ha sido detallado con el objetivo de servir como una guía estructurada para futuros trabajos, proporcionando una referencia clara y comprensible para quienes deseen abordar desarrollos similares en el ámbito del diseño de redes neuronales en *hardware*.

1.1. Objetivos

El presente trabajo tiene como objetivo principal la implementación en *hardware* de una GRU utilizando HLS, así como el análisis de su rendimiento en términos de eficiencia computacional y precisión de los resultados en comparación con simulaciones en software.

Para alcanzar este propósito, se establecen los siguientes objetivos específicos:

- Diseño e Implementación en *Hardware*.
  - Desarrollar la implementación de la GRU en una FPGA Xilinx Zynq mediante Vitis HLS.
  - Optimizar el diseño para asegurar un uso eficiente de los recursos disponibles en la FPGA.
  - Garantizar que la latencia del sistema no supere los 150 ciclos de reloj.
  - Minimizar el uso de recursos para permitir la integración de la GRU con otras implementaciones existentes en la RedPitaya.
- Análisis del rendimiento.
  - Evaluar la precisión de los resultados obtenidos en la implementación *hardware* en comparación con simulaciones en *software*.
  - Analizar el consumo de recursos y el tiempo de ejecución para determinar la eficiencia computacional del diseño.
- Integración en la RedPitaya.
  - Implementar la GRU en la FPGA de la RedPitaya.
  - Verificar el correcto funcionamiento del diseño en un entorno de *hardware* real.
- Documentación.
  - Describir el proceso de implementación y las estrategias de optimización aplicadas.
  - Proporcionar una base metodológica para futuros desarrollos en la implementación de redes neuronales en *hardware*.

1.2. Estructura del trabajo

El presente documento se organiza en los siguientes capítulos:

- **Capítulo 1: Introducción.** Presenta el contexto y los objetivos del trabajo, así como la relevancia del problema abordado y las herramientas utilizadas. Se expone la motivación para la realización del proyecto y su posible impacto en aplicaciones futuras.
- **Capítulo 2: Marco teórico.** Se introduce el concepto de redes neuronales artificiales, en particular las redes neuronales recurrentes y la arquitectura GRU. También se describen los fundamentos de la síntesis de alto nivel (HLS) y su aplicación en el diseño de *hardware*. Se contextualizan estos temas en el ámbito de la implementación en FPGA y se revisa el estado del arte.
- **Capítulo 3: Métodos e instrumentos** Se describen los elementos de *hardware* empleados en la implementación, incluyendo la FPGA Xilinx Zynq y la plataforma RedPitaya. Además, se detallan las herramientas de desarrollo utilizadas, como Vivado y Vitis HLS, así como las estrategias adoptadas para la optimización del diseño y la implementación de la IP.
- **Capítulo 4: Desarrollo, implementación y resultados.** Se expone el proceso de diseño e implementación de la GRU en **hardware**, incluyendo la integración de la IP en la FPGA y la programación de la RedPitaya. Se presentan y analizan los resultados obtenidos, comparando las diferentes versiones implementadas y evaluando la eficiencia en tiempo de ejecución y consumo de recursos.
- **Capítulo 5: Conclusiones y líneas futuras.** Se sintetizan los principales hallazgos del trabajo y se proponen posibles mejoras y extensiones futuras. Se incluyen también reflexiones sobre la experiencia adquirida y las lecciones aprendidas en el proceso.

Capítulo 2

Marco Teórico

En este capítulo se introducen los principios básicos de la base de este trabajo, los cuales servirán para comprender e interpretar el motivo de utilizar este tipo de red neuronal y las herramientas para llevar a cabo esta.

2.1. Introducción a las Redes Neuronales Artificiales

2.1.1. Inspiración biológica y el origen de las redes neuronales artificiales

Las redes neuronales artificiales son algoritmos de software cuyo fundamento se inspira en el comportamiento de las neuronas del cerebro humano (Figura 2.1). En nuestro cerebro ocurren millones de reacciones químicas que comunican nuestras células cerebrales, o neuronas, unas con otras, mediante estructuras especializadas.

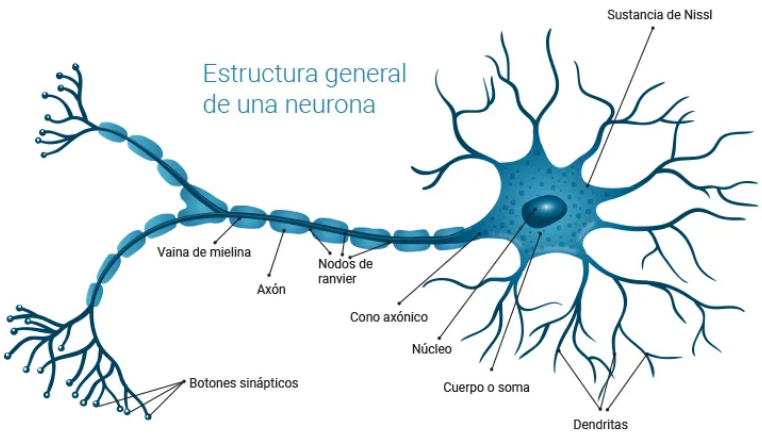

Figura 2.1: Estructura neurona biológica

Las neuronas se conectan entre sí por medio de conexiones llamadas sinapsis (Figura 2.2), las cuales son intercambios eléctricos entre el botón sináptico de una neurona (emisora o presináptica)

y el botón dendrítico de otra (receptora o postsináptica). Dichas conexiones permiten que una neurona mande descargas eléctricas a las células con las que se encuentra conectada. La descarga eléctrica viaja desde el cuerpo o soma de una neurona hasta el Axón, el cual realiza la función de un cañón eléctrico a través de los neurotransmisores. Cuando una neurona recibe un estímulo, su electroquímica se altera, acumulando energía. Cuando esa energía se acumula en cierta cantidad, la neurona la descarga por medio del Axón hacia las neuronas vecinas, formando una conexión sináptica con ellas.

Figura 2.2: Sinapsis neuronal

Una neurona como tal es diminuta en sí, pero cuando muchas se encuentran interconectadas, pueden formar toda una red de comunicaciones que pueden resolver problemas muy complejos. Por ejemplo, el cerebro de una persona contiene billones de neuronas. A esta comunicación entre neuronas se le denomina entonces una red neuronal.

Una vez explicada la parte biológica de la que provienen, se puede decir que una red neuronal artificial puede considerarse como un sistema de inteligencia artificial (AI) complejo que combina e interconecta un gran número de unidades de procesamiento no lineal llamadas neuronas.

El algoritmo del perceptrón [1] fue el primer intento de modelar numéricamente el comportamiento de una neurona biológica, funciona recibiendo múltiples entradas binarias y generando una única salida binaria.

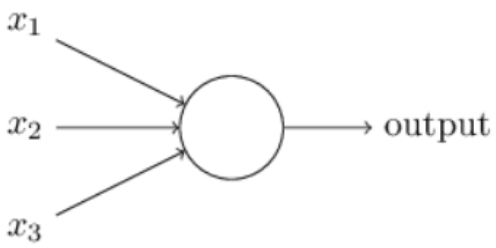

Figura 2.3: Perceptron [2]

El creador de este primer algoritmo, Frank Rosenblatt [3],propuso una regla simple para calcular la salida, la cual consistía en introducir unos pesos, $w_1, w_2, \ldots$, números reales que expresan la importancia de las respectivas entradas para la salida. La salida de la neurona, 0 o 1, se determina según si la suma ponderada $\sum_j w_j x_j$ es menor o mayor que un cierto valor umbral. Al igual que los pesos, el umbral es un número real que es un parámetro de la neurona. Para expresarlo en términos algebraicos más precisos:

$$salida = \begin{cases} 0 & \text{si } \sum_j w_j x_j \leq \text{umbral} \\ 1 & \text{si } \sum_j w_j x_j > \text{umbral} \end{cases} \tag{2.1}$$

Una forma de entender el perceptrón es como un dispositivo que toma decisiones basadas en la ponderación de diferentes evidencias o entradas.

El comportamiento de los perceptrones puede ser entendido como una forma de tomar decisiones basadas en la información de entrada. Por ejemplo, un perceptrón podría usarse para resolver problemas simples como decidir si aprobar o rechazar una solicitud de préstamo, basado en datos binarios como “ingresos adecuados” o “historial de crédito satisfactorio”. A cada uno de estos datos el banco le puede dar un peso, el cual indica la importancia o la irrelevancia de estos para la toma de decisiones. Y el umbral seria el número que debe superar la combinación de estas entradas con sus pesos para aprobar o rechazar el préstamo.

Aunque el perceptrón no refleja todo el proceso de toma de decisiones humanas, sirve para mostrar cómo se pueden combinar distintos factores para llegar a una decisión. Al evaluar las entradas según sus pesos, el perceptrón puede decidir cuál es la salida más adecuada en función de la información disponible. Por lo tanto, sería razonable pensar que una red compleja de perceptrones podría tomar decisiones más complejas y detalladas.

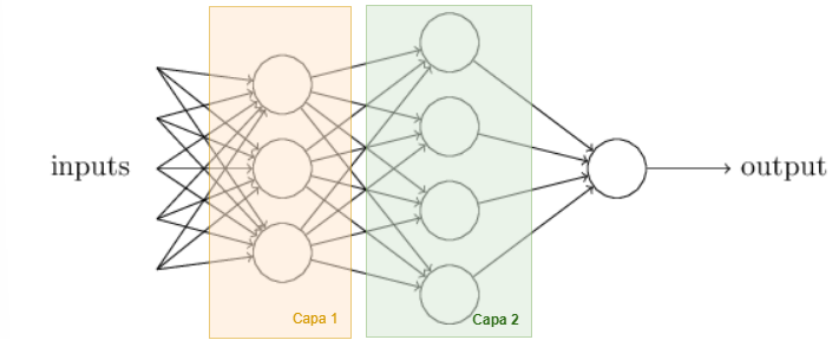

Figura 2.4: Red de perceptrones

En esta red, la primera capa de perceptrones, que consta de una serie de unidades, toma decisiones simples basadas en la evaluación de la evidencia de entrada. En cuanto a los perceptrones de la segunda capa, cada uno de ellos toma una decisión evaluando los resultados obtenidos de la primera capa. De esta forma, un perceptrón en la segunda capa puede realizar decisiones más complejas y abstractas en comparación con los de la primera capa. A su vez, los perceptrones de capas posteriores pueden tomar decisiones aún más sofisticadas. De este modo, una red de perceptrones con múltiples capas es capaz de llevar a cabo un proceso de toma de decisiones cada vez más avanzado y detallado.

Para simplificar la descripción de los perceptrones, podemos realizar algunas modificaciones en la notación que nos permitan expresar de manera más concisa su funcionamiento. La condición $\sum_j w_j x_j >$ umbral es algo engorrosa, por lo que podemos introducir dos cambios en la notación para facilitar su comprensión. El primer cambio consiste en representar la suma ponderada $\sum_j w_j x_j$ como un producto punto, $w \cdot x \equiv \sum_j w_j x_j$, donde w y x son vectores cuyas componentes corresponden a los pesos y las entradas, respectivamente. El segundo cambio es trasladar el umbral al otro lado de la desigualdad, y sustituirlo por lo que se conoce como el sesgo del perceptrón, $b \equiv -$umbral. De este modo, utilizando el sesgo en lugar del umbral, la regla del perceptrón puede ser reescrita de la siguiente forma:

$$salida = \begin{cases} 0 & \text{si } w \cdot x + b \leq 0 \\ 1 & \text{si } w \cdot x + b > 0 \end{cases} \tag{2.2}$$

El sesgo puede considerarse como una medida de la facilidad con la que el perceptrón puede generar una salida de 1. O, para expresarlo de manera más biológica, el sesgo es una medida de cuán fácil es que el perceptrón se active. En el caso de un perceptrón con un sesgo muy alto, resulta extremadamente sencillo que el perceptrón produzca una salida de 1. Sin embargo, si el sesgo es muy negativo, se vuelve difícil que el perceptrón emita una salida de 1. Aunque la introducción del sesgo representa un cambio pequeño en la forma en que describimos los perceptrones, veremos más adelante que facilita simplificaciones adicionales en la notación.

La universalidad computacional de los perceptrones es, al mismo tiempo, alentadora y limitada. Es alentadora porque demuestra que las redes de perceptrones pueden alcanzar la misma capacidad computacional que cualquier otro dispositivo de cómputo. Sin embargo, esta perspectiva puede parecer limitada, ya que sugiere que los perceptrones podrían ser simplemente una representación diferente de funciones lógicas fundamentales, lo cual no implicaría una innovación significativa.

No obstante, la situación trasciende esta interpretación inicial. Es posible desarrollar algoritmos de aprendizaje capaces de ajustar automáticamente los pesos y sesgos de una red de neuronas artificiales en respuesta a estímulos externos, sin necesidad de una intervención directa del programador. Este enfoque permite utilizar las neuronas artificiales de manera radicalmente distinta a los circuitos de lógica convencional. En lugar de diseñar explícitamente un circuito con funciones lógicas predefinidas, las redes neuronales tienen la capacidad de aprender a resolver problemas, incluso aquellos para los que sería extremadamente complejo diseñar un circuito lógico convencional.

De forma general, una red neuronal artificial (ANN) es un modelo computacional inspirado en la estructura y funcionamiento del cerebro humano. Su base fundamental radica en un conjunto de entradas, denotadas como x, que son procesadas a través de conexiones ponderadas por valores w, conocidos como pesos. Además, cada neurona incorpora un término adicional, el sesgo (b), que permite ajustar la salida del modelo.

El proceso de cálculo en una red neuronal se desarrolla en capas. En cada capa, las neuronas reciben una combinación lineal de las entradas y sus respectivos pesos, a lo que se suma el sesgo. Posteriormente, este resultado es transformado mediante una función de activación, cuyo propósito es introducir no linealidad en el modelo y, de este modo, habilitar la capacidad de la red para resolver problemas complejos. Las funciones de activación juegan un papel esencial en el aprendizaje y el comportamiento de las redes neuronales, ya que determinan cómo se propagan las señales a través de las capas y cómo se obtienen las salidas finales.

En este trabajo, se emplean las funciones de activación sigmoide y tangente hiperbólica (*tanh*), ya que ambas son ampliamente utilizadas en redes neuronales debido a su capacidad para modelar relaciones no lineales de manera eficiente. En el siguiente apartado se realizará una breve explicación sobre el porque son necesarias estas funciones de activación y su origen.

2.1.1.1. Funciones de activación

El desarrollo de algoritmos de aprendizaje representa un avance significativo en el diseño de redes neuronales artificiales. Para entender cómo una red neuronal aprende, es fundamental analizar la forma en que los cambios en los parámetros internos, como los pesos y los sesgos, afectan su funcionamiento. Por ejemplo, al realizar una pequeña modificación en un peso o sesgo dentro de la red, idealmente, dicha alteración debería generar únicamente un cambio proporcional y controlado en la salida. Esta propiedad es esencial para permitir que la red ajuste sus parámetros progresivamente y mejore su desempeño en una tarea específica.

Las **funciones de activación** surgen como un elemento clave para superar estas limitaciones. Estas funciones introducen no linealidad en la red, permitiendo modelar relaciones complejas entre las entradas y las salidas [4]. Además, regulan cómo las neuronas de una capa interactúan con las siguientes, haciendo posible el aprendizaje progresivo.

En los perceptrones tradicionales, la **función escalón** se utilizaba como activación. Esta función genera una salida binaria (por ejemplo, 0 o 1), dependiendo de si la suma ponderada de las entradas supera un umbral. Aunque útil en tareas simples, su naturaleza no diferenciable y sus saltos abruptos limitan su capacidad de aprendizaje en redes profundas.

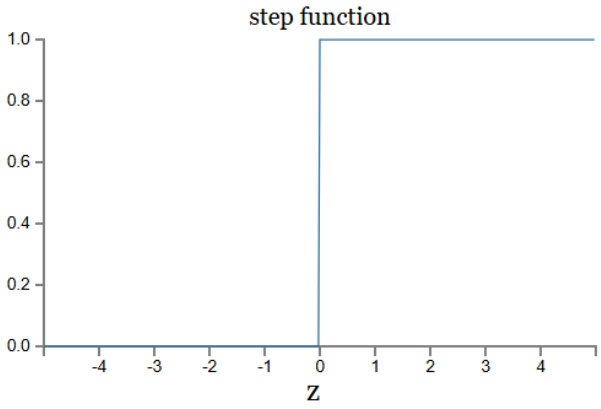

Figura 2.5: Función escalón [2]

La **función sigmoide**, una alternativa más suave, ofrece una transición continua entre los valores de salida, lo que facilita el ajuste de los parámetros de la red. Matemáticamente, se define como:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

donde z es una combinación lineal de las entradas ponderadas y un sesgo. La sigmoide conserva la similitud conceptual con la función escalón, en el sentido de que transforma las entradas en una salida acotada. Sin embargo, al ser diferenciable y continua, permite que las pequeñas modificaciones en los pesos y los sesgos se reflejen de manera proporcional en la salida, habilitando un aprendizaje más eficiente.

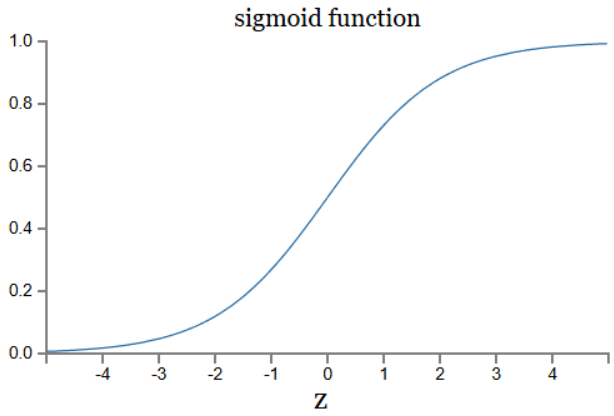

Figura 2.6: Función sigmoide [2]

Por otro lado, la **tangente hiperbólica (tanh)** es otra función de activación ampliamente utilizada, que proporciona una salida acotada entre -1 y 1. Esta propiedad la hace más adecuada para

tareas donde las salidas negativas tienen relevancia. Se define como:

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.4}$$

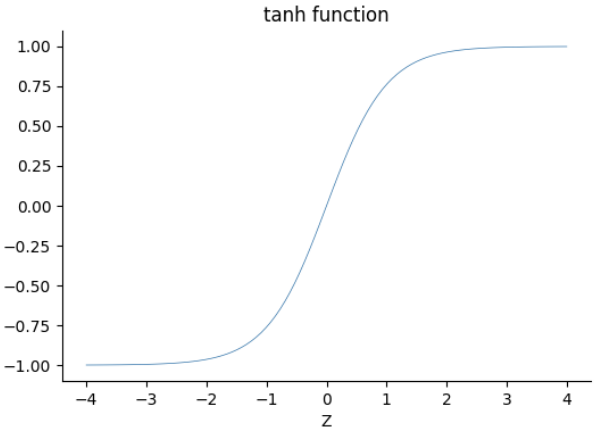

Figura 2.7: Función tangente hiperbólica (*tanh*)

En resumen, la función escalón, aunque sencilla y binaria, no es diferenciable, lo que limita su aplicabilidad en el entrenamiento de redes neuronales. Por otro lado, las funciones sigmoide y *tanh* introducen características de suavidad y diferenciabilidad, lo que facilita un ajuste más preciso de los modelos y las hace fundamentales en redes neuronales modernas. Si bien existen otras funciones de activación populares, como la ReLU, en este trabajo nos hemos centrado en la sigmoide y la *tanh* debido a su relevancia para el contexto de nuestra investigación, lo que justifica su explicación.

2.1.2. Tipos de Redes Neuronales Artificiales

Existen diversos tipos de redes neuronales, cada una diseñada para tareas específicas dentro del aprendizaje automático. A continuación, se describen los tipos más relevantes: Perceptrones Multicapa (MLP), Redes Neuronales Convolucionales (CNN) y Redes Neuronales Recurrentes (RNN). Cada tipo presenta características y arquitecturas particulares que los hacen aptos para distintos tipos de problemas[5].

2.1.2.1. Perceptrones multicapa (MLPs)

Las redes MLP son un tipo de red neuronal artificial feedforward, utilizada como arquitectura base para redes neuronales profundas (DNN) [6]. Este tipo de red es un enfoque supervisado, donde la red aprende a partir de datos etiquetados. El modelo básico de un MLP consiste en tres capas: la capa de entrada, la capa de salida y una o más capas ocultas. Cada neurona de una capa está conectada a todas las neuronas de la siguiente capa, formando una red completamente conectada.

En un MLP, la capa de entrada recibe los datos, y luego estos son procesados por las capas ocultas, que pueden ser varias. Finalmente, la capa de salida realiza las predicciones basadas en los datos procesados [7]. La Ecuación 2.5 muestra cómo se realiza la activación de las neuronas en un MLP, donde Φ es la función de activación. Esta ecuación establece que la salida Y es el resultado de aplicar la función de activación a la combinación lineal de las entradas multiplicadas por sus respectivos pesos, más un sesgo.

$$Y = \Phi(W \times X + b) \tag{2.5}$$

En esta ecuación, los términos X, W, b y Y representan el vector de entrada, el vector de pesos, el sesgo y el valor de salida, respectivamente. La Figura 2.8 muestra la estructura de un modelo de perceptrón multicapa. Matemáticamente, cada elemento y_j de la salida Y se calcula como:

$$y_j = \Phi\left(\sum_{i=1}^{N} W_i^j \cdot x_i + b_j\right) \tag{2.6}$$

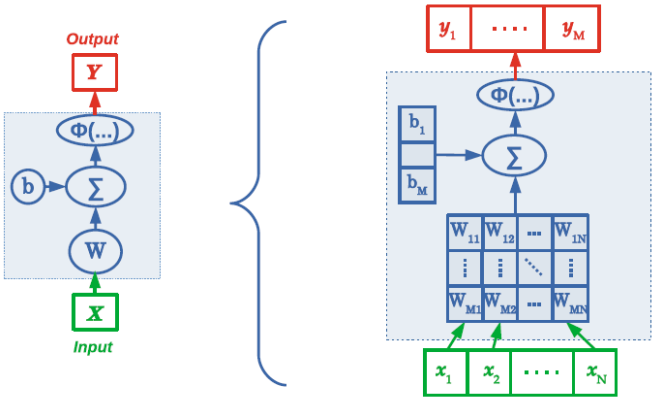

Figura 2.8: Estructura de un MLP [8]

2.1.2.2. Redes neuronales convolucionales (CNN)

Las redes CNN son un tipo de red neuronal *feedforward*, muy utilizadas en tareas como la visión por computadora, clasificación de imágenes, reconocimiento de objetos, y predicción de series temporales [9]. Se basan en la idea de convolución para extraer características automáticamente de los datos de entrada, reduciendo la necesidad de un preprocesamiento manual. La CNN típicamente se compone de tres tipos de capas principales: la capa convolucional, la capa de agrupamiento y la capa totalmente conectada. Las capas convolucionales extraen características locales, mientras que las capas de agrupamiento (*pooling*) se encargan de reducir la dimensionalidad, favoreciendo una mayor eficiencia computacional.

La Figura 2.9 ilustra el flujo de procesamiento de una CNN, detallando cómo las características de los datos se extraen y se procesan a través de cada una de estas capas.

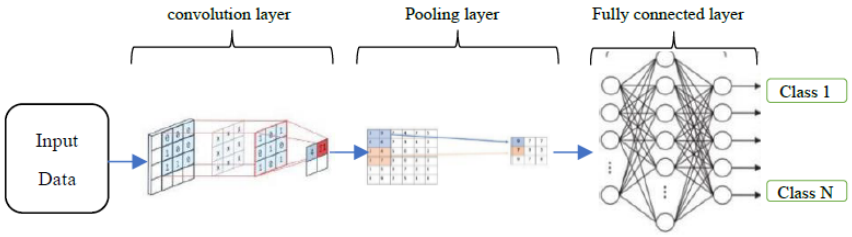

Figura 2.9: Cadena de procesamiento de una CNN

- **Capa Convolucional:** Esta capa es esencial en las redes convolucionales, ya que realiza la operación de convolución sobre los datos de entrada. Las primeras capas convolucionales extraen características simples, como bordes y texturas, mientras que las capas más profundas capturan características más complejas.
- **Capa de Agrupamiento (*Pooling*):** La capa de agrupamiento reduce la dimensionalidad de los mapas de características, disminuyendo la carga computacional. Existen varias técnicas, como *Max Pooling* y *Average Pooling*, que permiten mantener las características más importantes y reducir la posibilidad de sobreajuste.
- **Capa Totalmente Conectada (FC):** Al final de la red, la capa completamente conectada conecta cada neurona con todas las neuronas de la capa anterior. Su principal función es realizar las predicciones o clasificaciones finales a partir de las características extraídas.

2.1.2.3. Redes Neuronales Recurrentes (RNN)

Las Redes Neuronales Recurrentes (RNN) son una clase de modelos de aprendizaje profundo que poseen memoria interna, lo que les permite capturar dependencias secuenciales. A diferencia de las redes neuronales tradicionales, que tratan las entradas como entidades independientes, las RNN consideran el orden temporal de las entradas, lo que las hace especialmente adecuadas para tareas que involucran información secuencial. Al emplear un bucle recurrente, las RNN aplican la misma operación a cada elemento en una serie, de modo que el cálculo en un paso temporal depende tanto de la entrada actual X_t como de la salida del paso anterior Y_{t-1}, lo que introduce un mecanismo de "memoria"que captura la dependencia temporal.

La capacidad de las RNN para utilizar información contextual es particularmente valiosa en tareas como el procesamiento de lenguaje natural, la clasificación de videos y el reconocimiento de voz. Por ejemplo, en la modelización del lenguaje, comprender las palabras anteriores en una oración es crucial para predecir la siguiente palabra. Las RNN sobresalen en la captura de tales dependencias debido a su arquitectura recurrente. Este enfoque recurrente, que introduce un bucle de retroalimentación, está ilustrado en la Figura 2.10.

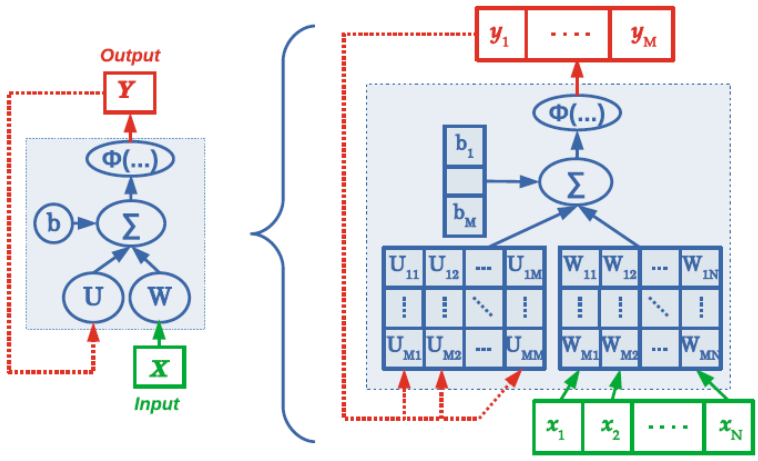

Figura 2.10: Representación gráfica de una RNN [8]

En este tipo de arquitectura, considerando t como el incremento en los pasos de ejecución, la salida Y_t se genera en función de la entrada X_t del paso actual y también de la salida del paso de ejecución previo Y_{t-1}. Las ecuaciones correspondientes son:

$$Y_t = \Phi(W \times X_t + U \times Y_{t-1} + b) \tag{2.7}$$

$$y_j^t = \Phi\left(\sum_{i=1}^{N} W_i^j \cdot x_i^t + \sum_{j=1}^{M} U_j^j \cdot y_j^{t-1} + b_j\right) \tag{2.8}$$

En estas ecuaciones, Φ representa una función de activación no lineal, como la tangente hiperbólica (tanh) o la función sigmoide, que introduce no linealidad al modelo, permitiendo que las RNN aprendan patrones más complejos en los datos secuenciales.

Sin embargo, una limitación significativa de las RNN simples es su memoria a corto plazo, lo que restringe su capacidad para retener información relevante a lo largo de secuencias largas. Este problema se conoce como el desvanecimiento del gradiente. Para abordar esta limitación, se han desarrollado variantes avanzadas como las **LSTM** (*Long Short-Term Memory*) [10] y las **GRU** (*Gated Recurrent Unit*) [11], que emplean mecanismos de puertas para controlar el flujo de información, lo que les permite mantener la información relevante durante secuencias más largas y evitar el problema del desvanecimiento del gradiente.

Celda LSTM

Una celda *Long Short-Term Memory* (LSTM) [5] [8] tiene una estructura compleja, como se muestra en la Figura 2.11. Esta celda almacena información en dos vectores de estado: el estado de la celda (C_t), que captura la memoria a largo plazo, y el estado de salida (Y_t), que gestiona la memoria a corto plazo y genera la salida en cada paso. Una unidad LSTM consta de cuatro componentes principales: una puerta de entrada, una puerta de olvido, una puerta de salida y el contenido de memoria actual. Las puertas logran controlar el flujo de información a través de su

memoria interna, permitiendo almacenar, modificar o eliminar información de manera selectiva en cada paso de tiempo. A continuación, se describe el funcionamiento de cada una de las puertas presentes en una LSTM [5]:

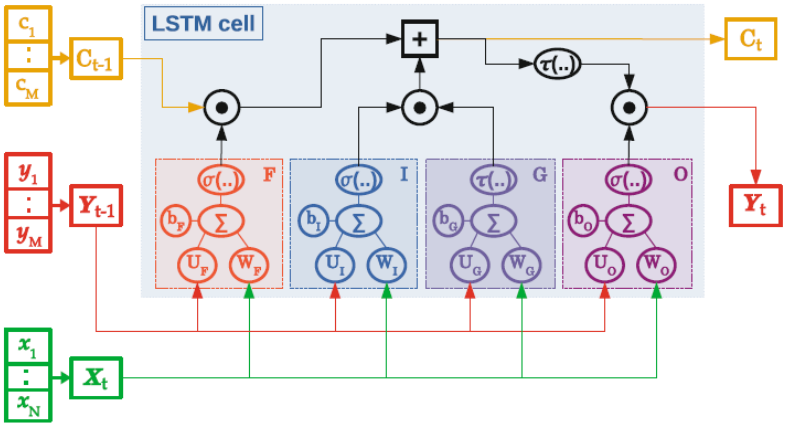

Figura 2.11: Celda LSTM [8]

Puerta de Olvido (*Forget gate*)

La puerta de olvido (F_t) decide qué información del estado anterior (Y_{t-1}) debe ser descartada de la memoria interna C_t. Esta puerta tiene un control crucial sobre el manejo de la memoria, determinando cuánto del estado previo es irrelevante y, por lo tanto, debe ser olvidado. La activación de esta puerta se calcula utilizando la función sigmoide, produciendo un valor entre 0 y 1 para cada componente del estado interno, donde un valor cercano a 0 indica que la información debe olvidarse, y un valor cercano a 1 significa que se debe mantener.

$$F_t = \sigma(W_F \cdot X_t + U_F \cdot Y_{t-1} + b_F) \tag{2.9}$$

Puerta de Entrada (*Input gate*)

La puerta de entrada (I_t) regula la cantidad de nueva información que debe ser almacenada en la memoria interna. Esta puerta es responsable de decidir qué información proveniente de la entrada (X_t) y la salida previa (Y_{t-1}) se considera relevante para el estado actual y debe ser almacenada. La activación de la puerta de entrada también se obtiene utilizando la función sigmoide.

$$I_t = \sigma(W_I \cdot X_t + U_I \cdot Y_{t-1} + b_I) \tag{2.10}$$

Puerta Candidata (*Candidate gate*)

La puerta candidata (G_t) crea una çandidatura"para la actualización del estado interno. Esta puerta genera una nueva versión del estado C_t basada en la entrada actual (X_t) y la salida anterior (Y_{t-1}), filtrada a través de una función tangente hiperbólica (tanh). Este vector contiene la

información que, en conjunto con la puerta de entrada, podrá ser añadida o descartada del estado interno.

$$G_t = \tanh(W_G \cdot X_t + U_G \cdot Y_{t-1} + b_G) \tag{2.11}$$

Puerta de Salida (*Output gate*)

La puerta de salida (O_t) determina qué parte del estado interno actual (C_t) debe ser utilizada para generar la salida de la LSTM (Y_t). La salida de la LSTM se calcula como una combinación del estado interno filtrado por la activación de la puerta de salida. De nuevo, la puerta de salida utiliza una función sigmoide para controlar el flujo de información desde el estado interno hacia la salida.

$$O_t = \sigma(W_O \cdot X_t + U_O \cdot Y_{t-1} + b_O) \tag{2.12}$$

El estado de la celda y la salida se calculan como:

$$C_t = F_t \cdot C_{t-1} + I_t \cdot G_t \tag{2.13}$$

$$Y_t = O_t \cdot \tanh(C_t) \tag{2.14}$$

Celda GRU

La *Gated Recurrent Unit* (GRU, Figura 2.12) [5][8][12] es una variante de la arquitectura RNN que aborda el problema de la memoria a corto plazo y ofrece una estructura más simple en comparación con las LSTM. La GRU combina la puerta de entrada y la puerta de olvido de las LSTM en una única puerta de actualización, lo que resulta en un diseño más simplificado. A diferencia de las LSTM, la GRU no incluye un estado de celda separado. Una unidad GRU consta de tres componentes principales: una puerta de actualización, una puerta de reinicio y el contenido de memoria actual [11]. Estas puertas permiten a la GRU actualizar y utilizar selectivamente la información de pasos de tiempo anteriores, lo que le permite capturar dependencias a largo plazo en las secuencias.

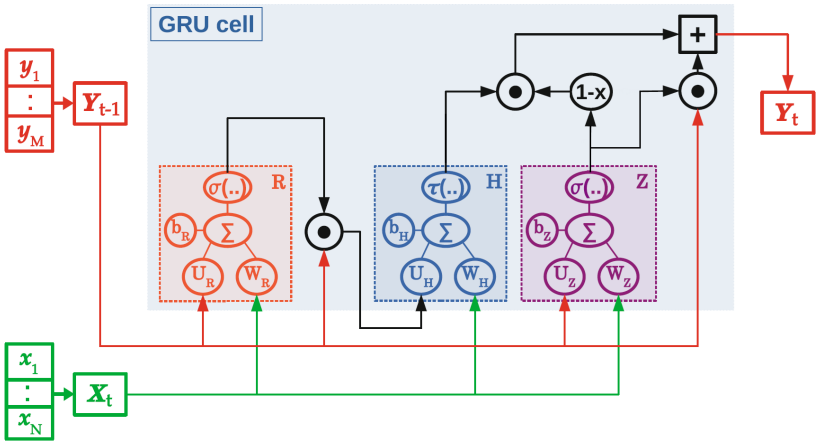

Figura 2.12: Celda GRU

Puerta de Actualización (*Update gate*)

La puerta de actualización (Ecuación 2.15) determina cuánto de la información pasada debe ser retenida y combinada con la entrada actual en un momento específico. Se calcula en función de la concatenación del estado oculto anterior Y_{t-1} y la entrada actual X_t, seguida de una transformación lineal y una función de activación sigmoide:

$$Z_t = \sigma(W_Z \cdot X_t + U_Z \cdot Y_{t-1} + b_Z) \tag{2.15}$$

Puerta de Reinicio (*Reset gate*)

La puerta de reinicio (Ecuación 2.16) decide cuánto de la información pasada debe ser olvidada. Se calcula de manera similar a la puerta de actualización, utilizando la concatenación del estado oculto anterior y la entrada actual:

$$R_t = \sigma(W_R \cdot X_t + U_R \cdot Y_{t-1} + b_R) \tag{2.16}$$

Contenido de Memoria Actual (*Candidate state*)

El contenido de memoria actual (Ecuación 2.17) se calcula en función de la puerta de reinicio y la concatenación del estado oculto transformado anterior y la entrada actual. El resultado se pasa a través de una función de activación tangente hiperbólica para producir la activación candidata:

$$H_t = \tau(W_H \cdot X_t + U_H \cdot (R_t \cdot Y_{t-1}) + b_H) \tag{2.17}$$

Estado de Memoria Final (*Output*)

Finalmente, el estado de memoria final Y_t se determina mediante una combinación del estado oculto anterior y la activación candidata (Ecuación 2.18). La puerta de actualización controla el equilibrio entre el estado oculto anterior y la activación candidata:

$$Y_t = (1 - Z_t) \cdot H_t + Z_t \cdot Y_{t-1} \tag{2.18}$$

Es fácil notar similitudes entre las unidades LSTM y GRU, como se muestra en las figuras anteriores. Una característica prominente compartida por estas unidades es el componente aditivo en su actualización de t a $t + 1$, lo cual está ausente en la unidad recurrente tradicional. La unidad recurrente tradicional reemplaza completamente su activación o contenido con un nuevo valor calculado a partir de la entrada actual y el estado oculto previo. En contraste, tanto la unidad LSTM como la GRU conservan el contenido existente y añaden el nuevo contenido sobre este [13].

2.1.3. Aplicaciones futuras de las ANN

Para los sistemas embebidos futuros, las redes neuronales artificiales (ANN), asociadas con otros métodos de inteligencia artificial, están habilitando nuevas posibilidades en diversos campos. Por ejemplo, en los sistemas de aviónica, las ANN están transformando la gestión de vuelo y la detección de fallos mediante técnicas de aprendizaje profundo para la interpretación de datos sensoriales complejos [14]. En el ámbito automotriz, estas redes están potenciando la conducción autónoma al optimizar sistemas de percepción, predicción y planificación del comportamiento [15].

En sistemas espaciales, las ANN están siendo exploradas para enfrentar desafíos relacionados con la navegación autónoma, la detección de anomalías y el análisis de grandes volúmenes de datos generados por sensores remotos en tiempo real [16]. Además, en sistemas de drones, las ANN permiten optimizar características avanzadas de piloto automático, como el vuelo autónomo en entornos no estructurados, el reconocimiento de patrones en imágenes aéreas y la planificación de trayectorias dinámicas [17].

En particular, la capacidad de las redes neuronales recurrentes (RNN) para manejar relaciones temporales en los datos las hace muy interesantes para aplicaciones específicas en sistemas embebidos. Por ejemplo, son útiles en tareas como predicción de series temporales o procesamiento de señales dependientes del tiempo. Sin embargo, esta capacidad conlleva una mayor complejidad. La naturaleza compleja de las RNN, especialmente con arquitecturas como LSTM o GRU, plantea desafíos significativos para su integración en sistemas embebidos [18]. Estos sistemas suelen tener memoria y potencia de procesamiento limitadas, lo que requiere un análisis cuidadoso para implementar las RNN de manera eficiente.

Para abordar estas limitaciones, se están desarrollando técnicas como la compresión de redes neuronales, la cuantización de pesos y activaciones, y la implementación de arquitecturas de *hardware* especializadas, como los aceleradores basados en FPGA o ASIC [19, 20]. Estas estrategias buscan reducir el consumo energético y mejorar el rendimiento de las redes, haciendo que su uso sea factible en aplicaciones embebidas con restricciones de recursos.

2.2. High-Level Synthesis (HLS)

La Síntesis de Alto Nivel (*High-Level Synthesis*, HLS) constituye un avance significativo en el campo del diseño de sistemas digitales, permitiendo transformar especificaciones funcionales de alto nivel en implementaciones de nivel de transferencia de registros (*Register Transfer Level*, RTL). Este enfoque automatizado optimiza procesos tradicionalmente manuales, no solo reduciendo errores, sino también acelerando el ciclo de desarrollo y simplificando la verificación de diseños complejos.

En el contexto de los flujos de diseño tradicionales, el desarrollo de *hardware* comienza con una especificación funcional, a menudo descrita en lenguajes como ANSI C, C++ o SystemC. Estas especificaciones iniciales son esencialmente modelos de comportamiento que no incluyen detalles específicos de implementación de *hardware*. Su propósito principal es validar el comportamiento deseado antes de proceder con etapas más detalladas del diseño. Una vez aprobados, estos modelos se traducen en una arquitectura óptima que define cómo se implementará dicha funcionalidad, considerando restricciones como el rendimiento, el consumo de energía y el área ocupada en el *hardware*. Posteriormente, esta arquitectura se codifica manualmente en lenguajes RTL, como Verilog o VHDL, lo que implica una alta inversión de tiempo y recursos, así como el riesgo inherente de introducir errores.

Este proceso manual, aunque efectivo en muchos casos, enfrenta limitaciones significativas cuando se aplica a diseños de alta complejidad. A medida que las tecnologías avanzan, el tamaño de los sistemas y la complejidad de las aplicaciones aumentan exponencialmente, lo que dificulta el cumplimiento de los plazos de desarrollo y la eliminación de errores. Aquí es donde HLS marca una diferencia fundamental al automatizar gran parte del flujo de trabajo, proporcionando un camino directo y fiable desde las especificaciones abstractas hasta la implementación detallada en *hardware*.

El flujo de diseño de HLS toma como entrada una funcionalidad de descripción de alto nivel, una biblioteca de componentes RTL y un conjunto de restricción de diseño específicas. Como se ilustra en la Figura 2.13, inicialmente compila la especificación funcional en un modelo formal, como el gráfico de Control y Flujo de Datos (*Control and Data Flow Graph*, CDFG) que describe las dependencias de datos y control entre las operaciones. Luego, asigna recursos de *hardware*, como unidades funcionales, componentes de almacenamiento, buses, etc., organizar la ejecución de las operaciones en ciclos de reloj, vincula las operaciones a unidades funcionales, vincula variables a elementos de almacenamiento, vincula transferencias a buses y genera la arquitectura RTL. Luego, se lleva a cabo la implementación digital, que incluye la síntesis lógica y la ubicación y el rutado. Los pasos mencionados anteriormente suelen estar acoplados entre sí, por ejemplo, cuando se asignan menos recursos de *hardware*, se ahorra área, pero esto a su vez puede aumentar la latencia del diseño y reducir su velocidad. Sin embargo, gracias a HLS, el grado de paralelismo del *hardware* se puede ajustar fácilmente, lo que lleva a implementaciones con diferentes compensaciones entre los recursos de área (por ejemplo, *logic slices* de FPGA ocupadas y LUT) y la latencia.

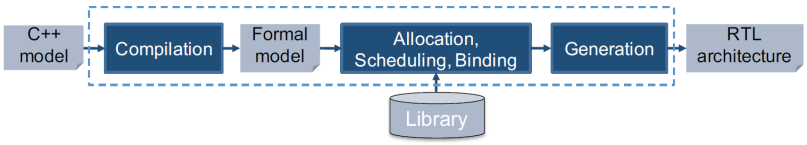

Figura 2.13: Flujo de diseño HLS [21]

El proceso comienza con la entrada de un lenguaje de alto nivel (por ejemplo, C++) con un procedimiento de compilación estándar. Luego, la herramienta generará, en función de las restricciones y los objetivos del circuito, una arquitectura (RTL) lista para ser sintetizada en la plataforma.

2.2.1. Beneficios del HLS y su comparación con los HDL

Uno de los principales beneficios de HLS es la reducción del esfuerzo de diseño y verificación. Al trabajar en un nivel de abstracción más alto, los ingenieros pueden enfocarse exclusivamente en la funcionalidad deseada, sin preocuparse por detalles específicos como la temporización, la jerarquía o las tecnologías utilizadas. Esto no solo simplifica la descripción del diseño, sino que también disminuye el número de líneas de código necesarias, reduciendo así el riesgo de errores. Además, una vez que el modelo funcional ha sido validado, las herramientas de HLS automatizan la generación del RTL, eliminando intervenciones manuales propensas a fallos. No obstante, es importante destacar que este proceso no elimina la necesidad de decisiones de diseño por parte de los ingenieros, quienes continúan desempeñando un papel clave en la definición de niveles de paralelismo, restricciones de arquitectura y optimización de recursos.

Otro aspecto destacable de HLS es su contribución a la reutilización efectiva de bloques de propiedad intelectual (IP). A diferencia de las vistas RTL, que describen interacciones específicas entre señales y dependen de tecnologías y frecuencias de reloj particulares, las especificaciones funcionales de HLS son genéricas y altamente adaptables. Esto permite modificar y verificar IP existentes con mayor facilidad, sin riesgo de romper estructuras como *pipelines* o máquinas de estado. Además, el uso de HLS facilita la adaptación de diseños a nuevas tecnologías, maximizando su utilidad en proyectos futuros.

Desde una perspectiva de optimización de recursos, HLS también permite a los equipos de investigación y desarrollo centrar sus esfuerzos en actividades que realmente añaden valor, como el desarrollo de algoritmos y la optimización arquitectónica. En un entorno competitivo donde factores como el tiempo de comercialización, la superioridad de características, el costo y el consumo energético son críticos, esta eficiencia operativa puede marcar la diferencia entre el éxito y el fracaso de un producto.

Sin embargo, no todo son ventajas en la adopción de HLS. Una de las críticas más comunes es que los diseños generados automáticamente pueden no ser tan eficientes como los creados manualmente por ingenieros experimentados. Esta eficiencia subóptima puede manifestarse en forma de mayor área ocupada o menor rendimiento. Además, aunque HLS simplifica muchos aspectos del diseño, requiere que los desarrolladores comprendan cómo sus decisiones de codificación afectan al *hardware* generado, lo que implica una curva de aprendizaje significativa. Por último, la adopción de HLS suele implicar cambios en las metodologías de diseño y en las habilidades requeridas por los equipos, lo que puede generar resistencia inicial.

En comparación con los lenguajes de descripción de *hardware* (HDL) como Verilog y VHDL, HLS se destaca por su nivel de abstracción. Mientras que HDL describe el comportamiento del *hardware* a nivel de señales y temporización, HLS permite especificar el "qué" en lugar del "cómo", proporcionando una perspectiva más funcional. Esto no solo acelera el desarrollo, sino que también facilita la reutilización y adaptación de componentes. Sin embargo, HDL sigue siendo preferido en situaciones donde se requiere un control fino y una optimización máxima del rendimiento. En términos de rendimiento, describir el diseño en lenguajes HDL, de momento, ofrece un mayor rendimiento, ya que tratamos la problemática de forma específica. No obstante, el propio HLS tiene directivas con propósitos de optimización que pueden llegar a solucionar o reducir los problemas que puedan generar un rendimiento menor. Por tanto, la elección en términos de rendimiento dependerá de cuánto se pueda adaptar el HLS al problema que queramos tratar.

En cuanto a flexibilidad, el *High Level Synthesis* ofrece mejor flexibilidad al cambio, ya que esta herramienta se encarga de generar un RTL en un lenguaje HDL, y por tanto nosotros tratamos de cambiar o corregir el código ya escrito en un lenguaje de propósito general. De esta manera, la tarea de modificar la descripción ya hecha no es un problema, ya que el HLS se encargará después de volverla a generar. Por el contrario, en los lenguajes HDL, habría que pensar en una solución y luego adaptar toda la descripción a ello, lo que puede llegar a ser una labor mucho más tardía.

Por último, el conocimiento de lenguajes HDL suele requerir fundamentos en diseño digital, como la comprensión de distintos tipos de descripciones, lo que puede resultar complejo a pesar de las abstracciones que ofrecen algunas herramientas. Sin embargo, HLS reduce significativamente esta barrera, permitiendo generar código sintetizable en un lenguaje de propósito general con un menor esfuerzo de aprendizaje. Aun así, un conocimiento sólido de diseño digital sigue siendo clave para optimizar el uso de recursos de área y mejorar el comportamiento temporal de los diseños en HLS.

Capítulo 3

Métodos e instrumentos

En este apartado se describen las herramientas empleadas para la implementación de una red neuronal GRU, destacando los componentes principales del entorno de trabajo. Se utilizó una máquina virtual con *Ubuntu 22.04* como sistema operativo, configurada para ejecutar las herramientas necesarias de diseño y desarrollo. La plataforma *RedPitaya Stemlab 125-14 Z7020 Low Noise* fue seleccionada como dispositivo de *hardware* para la implementación y verificación de la GRU.

Para el desarrollo de la red neuronal, se empleó el entorno de desarrollo proporcionado por Xilinx (versión 2022.2), incluyendo *Vivado*, *Vitis HLS* y *Vitis IDE*. Cada una de estas herramientas desempeñó un papel fundamental en el flujo de trabajo:

- Vitis HLS se utilizó para diseñar la IP de la red neuronal a partir de múltiples IP, proporcionando además estimaciones del uso de recursos y permitiendo la co-simulación para verificar la funcionalidad.
- Vivado permitió la integración de la IP generada en el ecosistema de Vitis HLS mediante su incorporación en el *block design*, preparándola para su implementación final en el *hardware*.
- Vitis IDE se empleó para desarrollar el software de comprobación que se ejecuta en la RedPitaya a través del archivo `.elf`.

Cabe mencionar que se presentaron dificultades durante la instalación de Ubuntu en la máquina virtual, así como en la configuración del entorno de desarrollo de Xilinx, las cuales requirieron soluciones específicas. Estas soluciones se encuentran detalladas en el Anexo C, para referencia y soporte adicional.

3.1. RedPitaya

Como se describe en la web oficial [22], Red Pitaya es una plataforma de desarrollo de FPGA de código abierto, digitalizador y multi instrumento definido por software. Tiene similitudes con sistemas como Raspberry Pi, con un sistema operativo basado en Linux y una comunidad de usuarios que desarrollan y comparten sus aplicaciones.Sin embargo, la Red Pitaya se diferencia de estos sistemas en dos aspectos clave:

- ***Hardware* especializado** para la adquisición y generación de señales analógicas.
- **CPU integrada con una FPGA programable**, permitiendo realizar tareas avanzadas de procesamiento digital de señales.

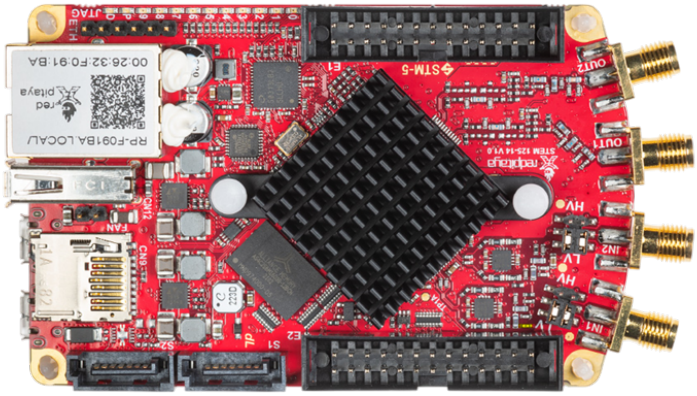

Figura 3.1: RedPitaya

Gracias a sus capacidades *hardware* y software, la Red Pitaya puede sustituir diversos instrumentos de laboratorio. A través de su plataforma web, se puede acceder a aplicaciones preinstaladas como osciloscopio, generador de funciones, analizador de espectros y medidor LCR, entre otras. Dado que la tarjeta no cuenta con una interfaz gráfica ni puertos de video, la visualización de estas aplicaciones se realiza desde un ordenador conectado a la misma red local [23].

Aunque las aplicaciones predeterminadas son funcionales y útiles, la principal ventaja de la Red Pitaya radica en su capacidad de personalización y desarrollo. En este trabajo, se aprovecha esta flexibilidad para hacer uso de su FPGA para implementar una red neuronal artificial. Para realizar la elección de la Redpitaya adecuada para el proyecto se ha visto el catalogo de productos de la Red Pitaya, el cual esta clasificado en función de la naturaleza del objeto, así como, detalles técnicos siguiendo una nomenclatura especifica [24]:

- **STEMlab**: Diseñado para aplicaciones académicas en el ámbito *STEM* (*Science, Technology, Engineering, and Mathematics*).
- **SDRlab**: Enfocado en aplicaciones de *radio definida por software (SDR)*, donde componentes típicos como filtros y moduladores/demoduladores se implementan en software.

- **SIGNALlab**: Destinado a aplicaciones avanzadas que requieren una generación de señales sofisticada, donde el costo es un factor secundario.

Para este trabajo como se ha mencionado en el Apartado 3 se ha empleado la **Red Pitaya STEMlab 125-14 Z7020 LN** uno de los dispositivos más versatiles y populares del catálogo empleado principalmente en el ámbito académico y de investigación.

La STEMlab 125-14-Z7020-LN es una placa STEMlab 125-14 estándar que cuenta con un sistema embebido Linux, un procesador ARM Cortex-A9 de doble núcleo y una FPGA Xilinx Zynq-7020, que cuenta con 3 veces más lógica ,que su antecesora la FPGA Zynq-7010, y compatibilidad con lenguajes de programación incluyendo Python, C, C++ y Matlab [25]. Ademas a esta versión se le añaden más E/S digitales en el conector E2 y alimentación analógica lineal adicional a las fuentes de alimentación analógicas para reducir el ruido de las entradas y salidas de RF y, en consecuencia, aumentar el ENOB.Todas estas características hacen de la STEMlab 125-14 Z7020 LN la opción ideal para la implementación del presente trabajo.

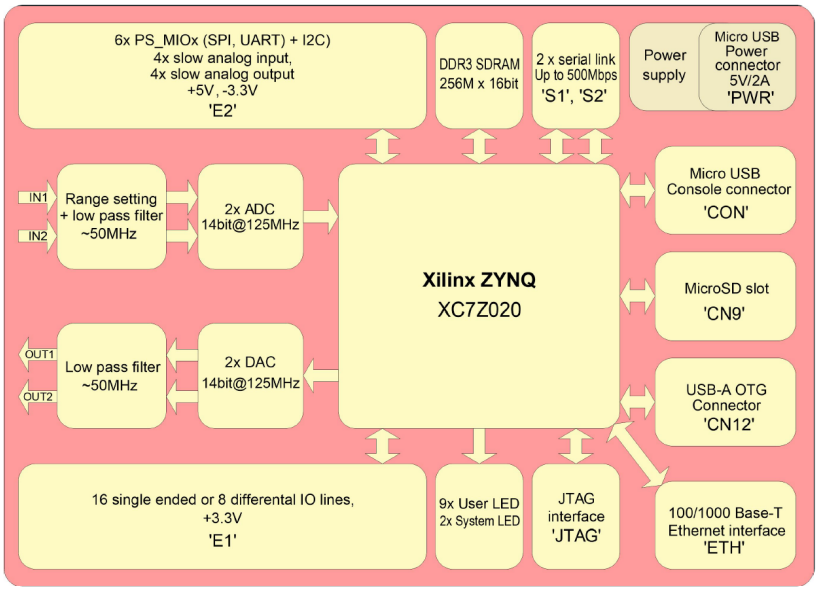

Figura 3.2: Esquema Redpitaya [26]

3.1.1. FPGA: Xilinx ZYNQ XCZ020

Dado que en este proyecto se aprovecha principalmente la FPGA para la implementación de una red neuronal artificial, es fundamental conocer los recursos disponibles en el chip Xilinx Zynq-7020. Este SoC (*System on Chip*) integra un procesador ARM junto con una FPGA programable, lo que permite combinar procesamiento en software y *hardware* de manera eficiente.

La Figura 3.3 muestra un diagrama de bloques simplificado con todas las partes que forman este chip. A primera vista, se distinguen dos partes principales [27]:

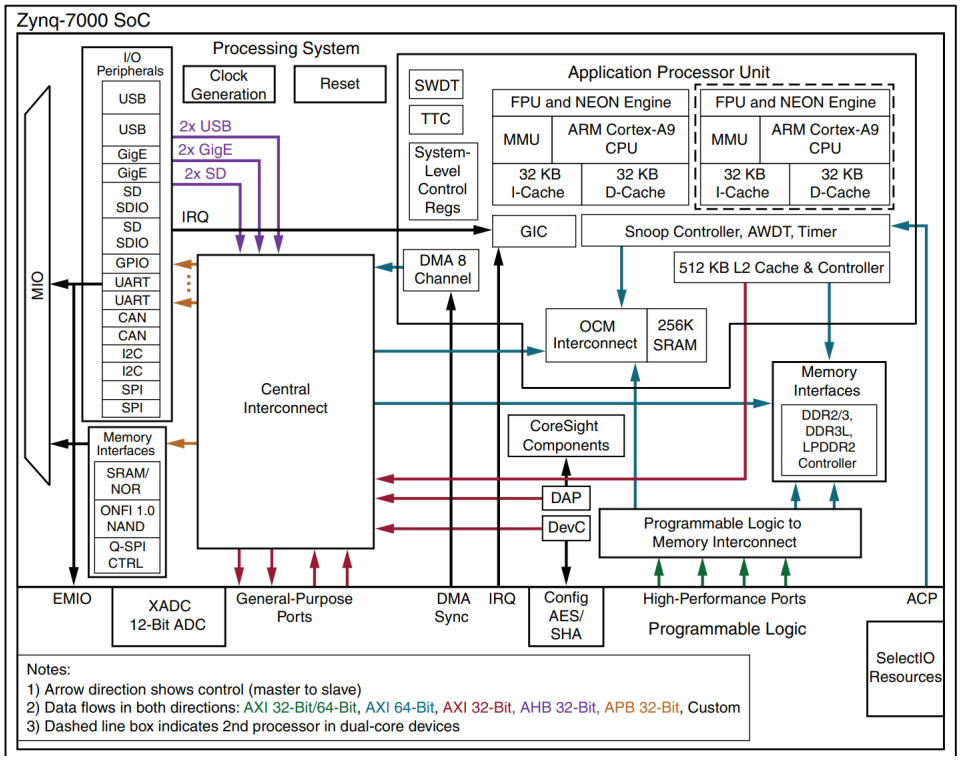

Figura 3.3: Diagrama de bloques Xilinx XC7Z020 [27]

- **Lógica programable (PL):** La parte clásica de FPGA. Contiene todos los bloques necesarios para crear su propio *hardware* descrito mediante el lenguaje de descripción de *hardware* (HDL):

  - **Almacenamiento y procesamiento de señales**: Los FPGA tienen pequeñas memorias integradas que se pueden utilizar para generar bloques de memoria de *hardware* para almacenar datos.

    Xilinx llama a sus propias memorias integradas en FPGA **BlockRAM**. Algunas de sus características son las siguientes:

    - **Operación síncrona**: Cada acceso a la memoria, lectura o escritura, está controlado por el reloj. Se registran todas las entradas, datos, direcciones, habilitaciones de reloj y habilitaciones de escritura. La dirección de entrada siempre está sincronizada, reteniendo los datos hasta la siguiente operación.

      Durante una operación de escritura, la salida de datos puede reflejar los datos almacenados previamente, los datos recientemente escritos o pueden permanecer sin cambios.
    - **Ancho de datos**: Cada Block RAM tiene dos puertos completamente independientes que no comparten nada más que los datos almacenados. Estos puertos se pueden configurar como 32K × 1, 16K × 2, 8K × 4, 4K × 9 (u 8), 2K × 18 (o 16), 1K × 36 (o 32) o 512 × 72 (o 64).
    - **Detección y corrección de errores**: Cada block RAM de 64 bits de ancho puede generar, almacenar y utilizar ocho bits de código Hamming adicionales y realizar

la corrección de errores de un solo bit y la detección de errores de doble bit (ECC) durante el proceso de lectura. La lógica ECC también se puede utilizar al escribir o leer en memorias externas de 64 a 72 bits de ancho.

- **Controlador FIFO**: El controlador FIFO integrado para funcionamiento con reloj único (sincrónico) o reloj dual (asincrónico o multifrecuencia) incrementa las direcciones internas y proporciona cuatro indicadores de protocolo de enlace: lleno, vacío, casi lleno y casi vacío. Los indicadores de casi lleno y casi vacío se pueden programar libremente. De manera similar a la RAM en bloque, el ancho y la profundidad del FIFO son programables, pero los puertos de escritura y lectura siempre tienen el mismo ancho.

También existen bloques **DSP**. Algunos aspectos destacados de la funcionalidad del DSP incluyen:

- Cada segmento DSP consta fundamentalmente de un multiplicador de complemento a dos dedicado de 25 × 18 bits y un acumulador de 48 bits.El multiplicador se puede omitir dinámicamente, y dos entradas de 48 bits pueden alimentar una unidad aritmética de instrucción única o una unidad lógica que puede generar cualquiera de diez funciones lógicas diferentes de los dos operandos.
- Pre-sumador de ahorro de energía para optimizar las aplicaciones de filtros simétricos.
- Funciones avanzadas: canalización opcional, ALU opcional y buses dedicados para conexión en cascada.

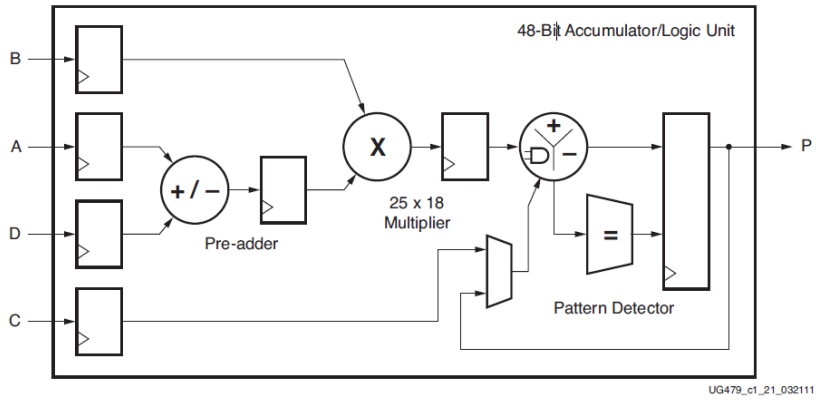

Figura 3.4: DSP SLICE [28]

- **Celdas lógicas del sistema**: unidades lógicas básicas que se utilizan para implementar funciones lógicas. Están formadas por una **tabla de consulta (*Look-up Table*, LUT)** que se pueden configurar como una LUT de 6 entradas (ROM de 64 bits) con una salida, o como dos LUT de 5 entradas (ROM de 32 bits) con salidas separadas pero direcciones o entradas lógicas comunes. Cada salida de LUT se puede registrar opcionalmente en un flip-flop. Cuatro de estas LUT y sus ocho flip-flops, así como los multiplexores y la lógica de acarreo aritmético, forman una *slice*, y dos porciones forman un bloque lógico configurable (CLB).
- **E/S de propósito general**: Los pines de entrada/salida del FPGA se dividen en dos subgrupos: de Alto rango (HR) o Alto rendimiento (HP). Donde los usos típicos de las

HR son comunicación con sensores, buses de datos de baja velocidad y controladores de interfaz. Y para el caso de los HP es la transferencia de datos de alta velocidad, comunicación con la memoria DDR y interfaces de alto rendimiento.

- **Conectividad de alta velocidad**: En esta categoría se incluyen los buses de mayor velocidad, desde los *Gigabit Transceivers Type X* (GTX),*Gigabit Transceivers Type P* (GTP) y *Peripheral Component Interconnect Express* (PCIe), que permiten velocidades de transferencia de decenas de Gigabits por segundo.

Los valores que se reflejan en la Tabla 3.1 resumen de la lógica programable, estos valores serán de vital importancia para la implementación de la red neuronal artificial ya que establecerán sus limites y sus posibles configuraciones para un mayor aprovechamiento de su lógica.

Lógica Programable (PL)	
Logic Cells	85K
LUTs	53200
FF	106400
BRAM	4.9Mb (140)
DSP	220

Tabla 3.1: Contenido lógica programable Z-7020 [29]

- **Sistema de procesamiento (PS)**: Las FPGA solían contener solo CLB pero como a menudo se usaban en conjunto con una CPU *hard* para acelerar ciertas tareas nació el concepto PS. Los bloques de esta parte de la arquitectura Zynq 7000 son los siguientes:

  - **Unidad de procesamiento de aplicaciones (APU)**: Los núcleos principales de la PS encargados de ejecutar el sistema operativo integrado elegido. Las características que se destacan son:
    - Capacidad de operar en modos de procesador único, procesador dual simétrico y procesador dual asimétrico. Para este proyecto se utilizará el modo de procesador único ya que utilizamos el sistema operativo integrado de Linux.
    - Punto flotante de precisión simple y doble.
    - RAM en chip con puerto doble (256 KB), diseñado para acceso de baja latencia desde la CPU. También accesible desde la lógica programable.
  - **Memoria**: La unidad de interfaz de memoria incluye un controlador de memoria dinámica y módulos de interfaz de memoria estática. El controlador de memoria dinámica admite memorias DDR3, DDR3L, DDR2 y LPDDR2. El controlador de memoria DDR multiprotocolo se puede configurar para proporcionar accesos de 16 o 32 bits de ancho a un espacio de direcciones de 1 GB utilizando una configuración de rango único de memorias DRAM de 8, 16 o 32 bits. El controlador de memoria DDR es multipuerto y permite que el sistema de procesamiento y la lógica programable tengan acceso compartido a una memoria común. Como se puede ver en la Figura 3.2 la Redpitaya utiliza una memoria DDR3 de 256M x 16bit.
  - **Periféricos de E/S (IOP)**: La unidad IOP contiene los periféricos de comunicación de datos. Los periféricos IOP se comunican con dispositivos externos a través de un grupo

compartido de hasta 54 pines de E/S multiuso (MIO) dedicados. A cada periférico se le puede asignar uno de varios grupos de pines predefinidos, lo que permite una asignación flexible de varios dispositivos simultáneamente.

- ***Interconnect***: La APU, la unidad de interfaz de memoria y el IOP están todos conectados entre sí y al PL a través de una interconexión ARM AMBA AXI Interconnect. La interconexión no es bloqueante y admite múltiples transacciones maestro-esclavo simultáneas.

  La interconexión AXI está diseñada para optimizar el acceso a la memoria y los periféricos según los requisitos de cada maestro. Los maestros sensibles a la latencia, como la CPU ARM Cortex-A9, tienen rutas de acceso más cortas a la memoria para minimizar la espera. Por otro lado, los maestros con requisitos de alto ancho de banda, como aquellos en la lógica programable (PL), cuentan con conexiones de alto rendimiento a los esclavos con los que necesitan comunicarse.

- **Interfaz PL - PS**: Las dos interfaces de mayor rendimiento entre la PS y la PL para la transferencia de datos son los puertos AXI de alto rendimiento y las interfaces ACP. Los puertos AXI de alto rendimiento se utilizan para la transferencia de datos de alto rendimiento entre la PS y la PL . La coherencia, si es necesaria, se gestiona mediante control de software. Cuando se requiere acceso coherente por *hardware* a la memoria de la CPU, se debe utilizar el puerto ACP.

  - **Puertos AXI de alto rendimiento**: Los puertos AXI de alto rendimiento proporcionan acceso desde la PL a DDR y OCM en la PS. Los cuatro puertos de memoria AXI dedicados desde la PL a la PS se pueden configurar como interfaces de 32 bits o de 64 bits. Como se muestra en Figura 3.5 estas interfaces conectan la PL a la interconexión de memoria a través de un controlador FIFO. Dos de los tres puertos de salida van al controlador de memoria DDR y el tercero va a la memoria en chip (OCM) de dos puertos.

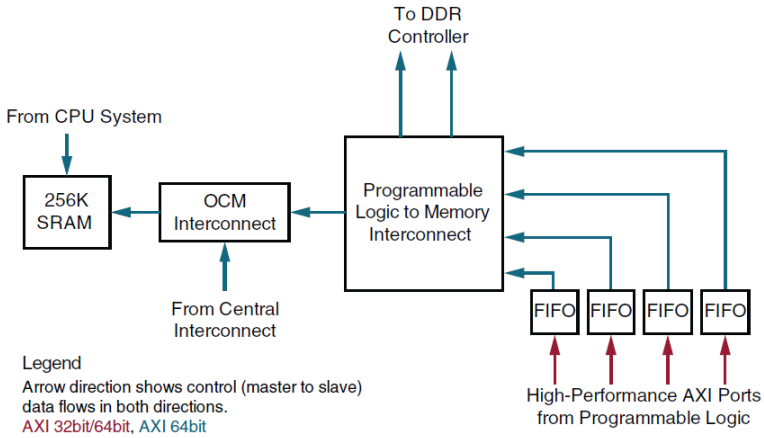

Figura 3.5: Interfaz PL - PS [27]

- **Puerto de coherencia del acelerador (ACP)**: El puerto de coherencia del acelerador (ACP) es una interfaz esclava AXI de 64 bits que proporciona conectividad entre la ALU y una función aceleradora potencial en el PL. El ACP conecta directamente el

PL a la unidad de control de espionaje (SCU) de los procesadores ARMCortex-A9, lo que permite el acceso coherente con la caché a los datos de la CPU en las cachés L1 y L2. El ACP proporciona una ruta de baja latencia entre la PS y un acelerador basado en PL en comparación con un esquema de vaciado y carga de caché heredado.

3.1.2. Sistema Operativo Linux: Ubuntu

Linux es un sistema operativo que presenta diversas características que lo diferencian de otros sistemas disponibles en el mercado. Entre sus principales atributos destacan su naturaleza de software libre, su estructura basada en un núcleo (*kernel*) y un conjunto de programas y bibliotecas que permiten su funcionamiento, así como su capacidad para ser multiusuario y multitarea.

Linux se distribuye bajo la licencia GNU *General Public License*, lo que implica que su código fuente debe estar siempre accesible y cualquier modificación o desarrollo derivado debe mantener esta misma licencia [30].

Existen múltiples distribuciones de Linux, las cuales consisten en el sistema operativo base acompañado de una selección de paquetes específicos. En este trabajo, se ha empleado la distribución Ubuntu debido a su compatibilidad con las herramientas de desarrollo de Xilinx, así como con la plataforma Red Pitaya, la cual también utiliza este mismo sistema operativo y distribución. La elección de Linux, y en particular de Ubuntu, en Red Pitaya responde a las siguientes razones:

- **Compatibilidad con la plataforma Xilinx Zynq SoC**: Red Pitaya se basa en dispositivos de la familia Xilinx Zynq SoC, los cuales integran un procesador ARM junto con lógica FPGA. Linux ofrece soporte nativo para esta arquitectura, lo que facilita el desarrollo y la ejecución de aplicaciones en el sistema [31].
- **Entorno de desarrollo abierto y flexible**: Linux proporciona un entorno de desarrollo robusto y versátil, permitiendo a los usuarios programar en distintos niveles mediante una variedad de interfaces de software, incluyendo HDL, C/C++ y lenguajes de script. Esto resulta especialmente beneficioso para una plataforma de código abierto como Red Pitaya, cuyo objetivo es fomentar la innovación y la personalización por parte de la comunidad [32].
- **Facilidad de integración y despliegue**: Linux permite gestionar eficientemente la carga y sustitución de imágenes FPGA desde el sistema operativo en ejecución, lo que agiliza el desarrollo y las pruebas. Esta característica es fundamental para los desarrolladores que requieren iteraciones rápidas en sus diseños y aplicaciones [31].

3.2. Entorno Xilinx

El desarrollo de sistemas digitales basados en FPGA requiere de herramientas especializadas que permitan el diseño, la simulación y la implementación del *hardware*. En este proyecto, se hace uso del entorno de diseño integrado Vivado Design Suite, lanzado por Xilinx en 2012. Vivado proporciona una suite de herramientas avanzadas para la creación de sistemas digitales, desde el nivel de sistema hasta el nivel de circuito integrado, con un modelo de datos escalable y un entorno de depuración común.

El paquete Vivado ML Enterprise es el software utilizado a lo largo de este proyecto para la generación del *hardware* específico necesario para la implementación de la GRU(Gated Recurrent Unit). El punto de partida de este desarrollo es un proyecto base proporcionado por los tutores basado en el proyecto base de RedPitaya, el cual establece el entorno de trabajo fundamental para la implementación sobre la FPGA. Este proyecto incluye una configuración inicial del *hardware* necesaria para la interacción con la RedPitaya, facilitando la integración de nuevos módulos personalizados.

Además, en el marco de este proyecto se encuentran disponibles diversas IP desarrolladas por el grupo de investigación al que pertenecen los tutores. Estas IP serán adaptadas y utilizadas en el proyecto, destacando especialmente las DMA. Como se detallará en capítulos posteriores, las DMA facilitan la comunicación entre la CPU y la GRU, permitiendo el envío de datos relevantes y la recepción de los resultados obtenidos.

A continuación, se realiza una descripción de cada una de las herramientas que forman parte de este entorno Xilinx y finalmente el flujo seguido a través de ellas que ha hecho que sea posible la implementación de la GRU.

3.2.1. Vitis HLS

La herramienta Vitis HLS [33] sintetiza una función en C o C++ en código RTL para su implementación en la región de lógica programable (PL) de un dispositivo Versal ACAP, Zynq MPSoC o FPGA de Xilinx. Vitis HLS está estrechamente integrado tanto con el Vivado Design Suite para síntesis, emplazamiento y rutado, como con el kit de desarrollo central Vitis para diseño de sistemas heterogéneos y aceleración de aplicaciones.

Vitis HLS se puede utilizar para desarrollar y exportar:

- IPde Vivado para ser integrado en diseños de *hardware* utilizando Vivado Design Suite.
- Núcleos de Vitis para su uso en el flujo de desarrollo de aceleración de aplicaciones de Vitis.

Vitis HLS automatiza gran parte de las modificaciones de código requeridas para implementar y optimizar el código C/C++ en lógica programable, logrando baja latencia y alto rendimiento. La inferencia de directivas necesarias (pragmas) para producir la interfaz adecuada para los argumentos de función y para canalizar bucles y funciones dentro del código es la base de Vitis HLS.

En el flujo de IP de Vivado, Vitis HLS también admite la personalización del código para implementar estándares de interfaz más amplios y lograr los objetivos de diseño. El RTL generado se puede usar como un IP directamente dentro de la herramienta Vivado o Model Composer.

El desarrollo de una función en C++ utilizando Vitis HLS sigue los siguientes pasos:

1. **Arquitectura del algoritmo**: Diseñar el algoritmo en base a los principios de diseño.
2. **C-Simulación**: Verificar el código C/C++ con el banco de pruebas en C/C++. Cuando la simulación se completa correctamente, la consola muestra un mensaje indicando que la simulación a finalizado con éxito. En caso contrario mostrará que ha habido errores y parará la simulación.

3. **C-Síntesis**: Generar el RTL utilizando el HLS.
4. **Co-Simulación**: Este es un paso esencial del flujo en el que se confirma la correcta generación del RTL.
5. **Análisis**: Revisar los informes de síntesis y de co-simulación.

   Cuando se completa la síntesis, Vitis HLS genera un informe de resumen de síntesis y algunos visores los cuales serán utilizados para la estimación de recursos y correcto funcionamiento de la GRU.

   Cuando se completa la co-simulación también se generan una serie de informes, en los que se indica el estado de esta: *PASS / FAIL*, ademas de informar sobre latencias (mínima, máxima, media) y el número de ciclos de reloj que deben transcurrir antes de que una nueva iteración de un bucle pueda comenzar, conocido como *Initiation Interval* (II). También hay una serie de visores con los que se podrá visualizar las formas de onda.

   Estos informes generados y los visores utilizados, entre los diversos disponibles en la herramienta Vitis HLS, son descritos con mayor detalle en el apartado 3.2.1.2.
6. **Iteración**: Repetir los pasos anteriores hasta alcanzar los objetivos de rendimiento.

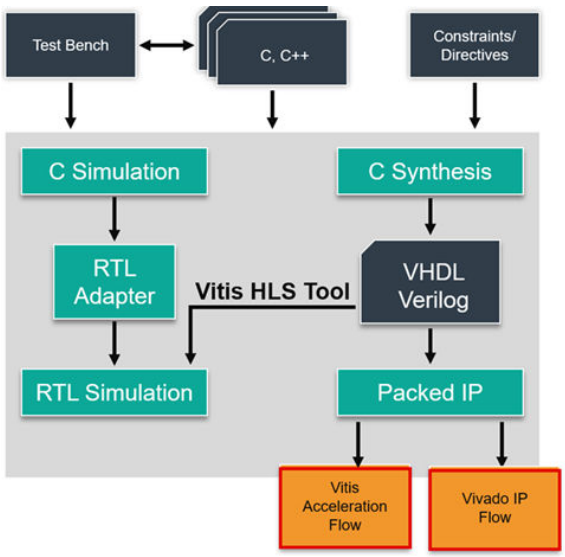

Figura 3.6: Flujo de desarrollo de Vitis HLS [33]

Y por último la parte mas importante de Vitis HLS es la generación de la IP de Vivado o kernels de Vitis según el flujo objetivo, la configuración predeterminada de la herramienta, las restricciones de diseño y cualquier pragma o directiva de optimización que especifiques. Se pueden utilizar directivas de optimización para modificar y controlar la implementación de la lógica interna y los puertos de E/S, anulando los comportamientos predeterminados de la herramienta.

En el diseño HLS con Vitis, hay aspectos fundamentales que influyen en la codificación y síntesis de funciones en C++. A continuación, se describen los más relevantes:

- **Interfaces de *Hardware***: Los argumentos de la función de nivel superior en un diseño HLS de Vitis se sintetizan en interfaces y puertos que agrupan múltiples señales para definir el

protocolo de comunicación entre el diseño HLS y los componentes externos. Vitis HLS define las interfaces automáticamente utilizando estándares de la industria para especificar el protocolo utilizado. Los protocolos de interfaz predeterminados varían dependiendo de si el diseño HLS está orientado a la generación de IP para Vivado o al kernel de Vitis. Las asignaciones predeterminadas de las interfaces se pueden sobrescribir utilizando el pragma o directiva `INTERFACE`.

- **Control de la Ejecución del Diseño HLS**: El modo de ejecución de un diseño HLS se especifica mediante el protocolo de control a nivel de bloque. El diseño HLS puede incluir señales de control para iniciar/detener la ejecución, o puede ser impulsado únicamente cuando los datos están disponibles.

- **Paralelismo a Nivel de Tarea**: Para lograr un alto rendimiento en el *hardware* generado, la herramienta HLS debe inferir paralelismo a partir del código secuencial y explotarlo para obtener un mayor rendimiento. La herramienta Vitis HLS ofrece múltiples tipos de paralelismo a nivel de tarea (TLP), ya sea especificando el pragma `DATAFLOW` o creando paralelismo explícito mediante el objeto `hls::task`.

- **Arquitectura de Memoria**: La arquitectura de memoria está fija en el CPU, pero el desarrollador puede crear su propia arquitectura para optimizar los accesos a memoria al ejecutar aplicaciones en FPGA. En un programa C++, los arreglos son estructuras de datos fundamentales para almacenar o mover datos. En *hardware*, estos arreglos se implementan como memoria o registros después de la síntesis. La memoria puede implementarse como almacenamiento local o memoria global, que suele ser DDR o bancos de memoria HBM. El acceso a la memoria global tiene mayores costos de latencia y puede tomar muchos ciclos, mientras que el acceso a la memoria local suele ser rápido y toma solo uno o más ciclos.

- **Optimización a Nivel Micro**: En los programas C++, es frecuente la necesidad de implementar algoritmos repetitivos que procesen bloques de datos, como en el procesamiento de señales o imágenes. Normalmente, el código fuente C/C++ incluye varios bucles o bucles anidados. Vitis HLS puede desenrollar (`UNROLL`) o canalizar (`PIPELINE`) un bucle o bucles anidados insertando pragmas en los niveles apropiados del código fuente.

  Una vez que el algoritmo está diseñado en base a los principios de diseño e infiriendo paralelismo, todavía se necesita la combinación correcta de pragmas a nivel micro, como `PIPELINE`, `UNROLL`, `ARRAY_PARTITION`, etc.

3.2.1.1. Tipos de datos empleados

En este apartado se describen los tipos de datos utilizados en la implementación de la GRU, haciendo especial énfasis en aquellos de precisión arbitraria. Aunque en la implementación también se han empleado tipos de datos convencionales de C/C++, como `int` y `float`.

Los tipos de datos nativos de C/C++ están restringidos a tamaños fijos de 8, 16, 32 o 64 bits. Sin embargo, en un diseño de *hardware*, los buses RTL permiten longitudes de datos arbitrarias. El uso de tipos estándar de C/C++ en este contexto puede derivar en una implementación ineficiente en términos de recursos y rendimiento.

Para abordar esta limitación, Vitis HLS proporciona tipos de datos de precisión arbitraria (AP), los cuales permiten definir variables con un ancho de bits específico, optimizando así el uso de

recursos en FPGA. La utilización de anchos de bits ajustados reduce el tamaño de los operadores de *hardware* y mejora la velocidad de ejecución, lo que se traduce en un mayor aprovechamiento del área y una mayor frecuencia de operación.

Los tipos de datos AP en Vitis HLS [33] incluyen tanto tipos enteros como tipos de punto fijo:

- Los tipos de datos enteros de precisión arbitraria se definen en el archivo de encabezado `ap_int.h`. Para emplearlos en una función C++, es necesario incluir este archivo y sustituir los tipos de datos estándar por `ap_int<N>` o `ap_uint<N>`, donde `N` representa el número de bits, en un rango de 1 a 1024.
- Los tipos de datos de punto fijo permiten representar valores con componentes enteras y fraccionarias, utilizando el formato `ap_fixed<W,I,[Q,O,N]>`, donde `W` indica la longitud total de la palabra, `I` el número de bits destinados a la parte entera, y `Q`, `O` y `N` especifican el modo de cuantificación, desbordamiento y saturación, respectivamente.

El uso de tipos de punto fijo es especialmente ventajoso en comparación con los de punto flotante, ya que las operaciones aritméticas requieren menos ciclos de reloj y menos recursos lógicos en FPGA. En la implementación de la GRU, se ha optado por tipos de punto fijo con especificaciones precisas para maximizar la eficiencia del *hardware*.

La siguiente tabla resume los parámetros del tipo de dato `ap_fixed`:

ap_fixed<W,I,[Q,O,N]>	
W	Longitud de palabra en bits.
I	Número de bits destinados a la parte entera (a la izquierda del punto binario).
Q	Modo de cuantificación: determina cómo se manejan los valores que exceden la precisión definida.
O	Modo de desbordamiento: define el comportamiento cuando el resultado de una operación supera el rango representable.
N	Número de bits de saturación en los modos de envoltura de desbordamiento.

Tabla 3.2: Resumen de los parámetros del tipo de dato `ap_fixed`.

En la implementación de la GRU, se han utilizado opciones específicas tanto para el modo de cuantificación como para el de desbordamiento:

- Para la cuantificación, se ha seleccionado `AP_RND_CNV`, que implementa el redondeo al más cercano con desempate hacia el par. Este método garantiza que cuando un valor no pueda representarse con precisión, se redondee al valor más cercano. En caso de que el valor a redondear se encuentre justo en la mitad entre dos representaciones posibles, se selecciona el número cuyo bit menos significativo después del redondeo sea `0` (es decir, un número par). Esta estrategia minimiza el sesgo acumulativo en operaciones repetidas y mejora la estabilidad numérica.

- Para el desbordamiento, se ha elegido `AP_SAT`, que aplica saturación, limitando el valor al máximo o mínimo representable en caso de desbordamiento positivo o negativo, respectivamente. Aunque esta opción previene valores incorrectos en caso de desbordamiento, su implementación requiere lógica adicional, lo que puede incrementar el uso de LUTs en hasta un 20 %.

La elección de estos tipos de datos ha sido clave en la optimización del diseño de la GRU, permitiendo reducir el consumo de recursos y mejorar la eficiencia de la implementación en FPGA.

3.2.1.2. Interpretación de Informes y Visores de Vitis HLS

Como se ha mencionado anteriormente, los informes y visores disponibles en Vitis HLS son herramientas fundamentales para evaluar la optimización y el correcto funcionamiento de la red neuronal artificial. A través de ellos, es posible analizar el rendimiento, el uso de recursos y la eficiencia de la implementación en *hardware*. A continuación, se presenta una descripción detallada de cada uno de estos elementos, su funcionalidad y la forma en que deben ser interpretados para garantizar un diseño óptimo.

Síntesis

- **Resumen de síntesis**: Este resumen aparece automáticamente una vez finalizada la síntesis y es el más importante para ver de forma resumida como Vitis ha interpretado el código proporcionado. Los apartados más relevantes y que serán utilizados en la comprobación de la implementación son los que se detallan a continuación.

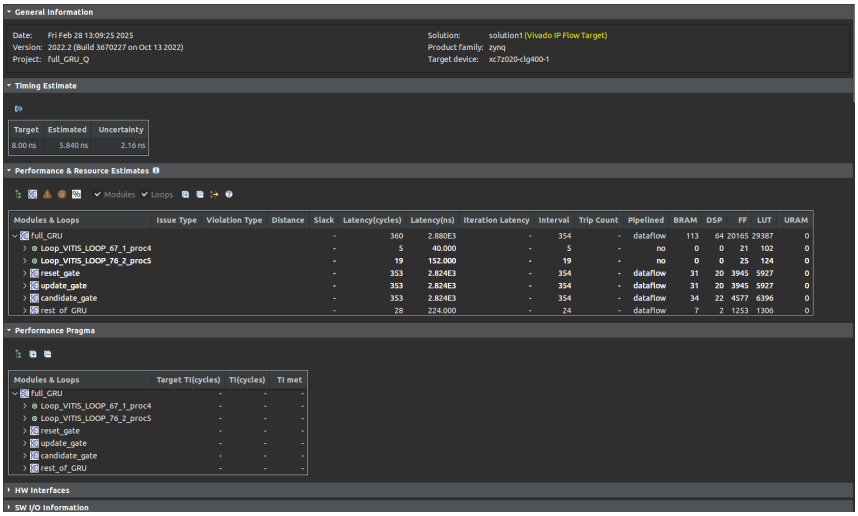

Figura 3.7: Informe resumen síntesis

Los apartados mas relevantes y que serán utilizados en la comprobación de la implementación son los que se detallan a continuación.

- **Estimaciones de rendimiento y recursos**: Es el más importante y está distribuido en formato tabla. Los apartados de esta son:

- *Performance Estimate*: Informa de la latencia y el intervalo de inicio de la función de nivel superior y de cualquier subbloque instanciado en el nivel superior.
- *Slack*: Muestra cualquier problema de tiempo en la implementación
- *Latency*: Muestra la cantidad de ciclos que se necesitan para generar la salida y también se muestra en tiempo (ns). Como veremos en el apartado 4.2 este indicador será relevante a la hora de la implementación de la GRU ya que los datos no pueden tardar mas de x ciclos en ser procesados.
- *Iteration Latency*: Es la latencia de una única iteración de un bucle.
- *Trip Count*: Muestra la cantidad de iteraciones que realiza un bucle específico en el *hardware* implementado.
- *Resource Estimate*: Se indican los recursos estimados necesarios para implementar la función de software en el código RTL. En esta columna se proporcionan estimaciones de BRAM, DSP, FF y LUT. Estas estimaciones como veremos en el apartado 4.2 serán el indicador de si la implementación propuesta es correcta para la FPGA o no realizando una comparación con la Tabla 3.1.

- **Interfaces de *hardware***: Proporciona tablas para las diferentes interfaces de *hardware* generadas durante la síntesis. El tipo de interfaces de *hardware* generadas por la herramienta depende del destino de flujo especificado por la solución, así como de los pragmas o directivas INTERFACE aplicados al código.
- **Información de E/S de SW**: Muestra la relación los argumentos de la función C++ con los nombres de puerto en el código RTL. Útil para la posterior relación de señales en Vivado.
- **Visor de programación**: Muestra cada operación y paso de control de la función, y el ciclo de reloj en el que se ejecuta. Le ayuda a identificar cualquier dependencia de bucle que impida el paralelismo, violaciones de tiempo y dependencias de datos.

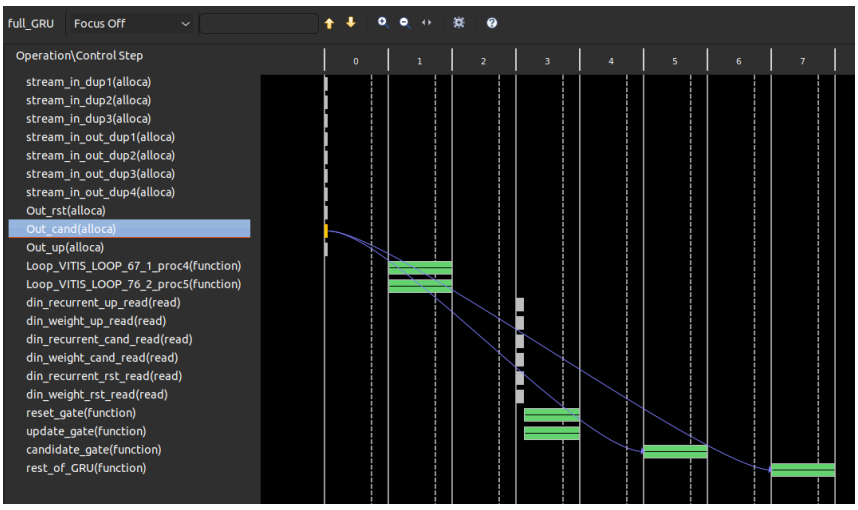

Figura 3.8: Visor de programación

El visor esta distribuido de la siguiente forma:

- El **eje vertical izquierdo** muestra los nombres de las operaciones y los bucles que se implementarán como lógica en la jerarquía RTL. El nombre entre corchetes

indica el nombre de la operación. Las operaciones están en orden topológico. En caso de algún tipo de violación, ya sea limite de recursos o dependencia, el visor muestra información adicional.

- El **eje horizontal superior** muestra los ciclos de reloj en orden consecutivo.
- La **línea discontinua vertical** en cada ciclo de reloj muestra la parte reservada del período de reloj debido a la incertidumbre del reloj.

El visor de programación también muestra las dependencias de datos generales del operador como líneas azules continuas. Como se muestra en la Figura 3.8, al seleccionar una operación, se puede ver flechas azules continuas que resaltan las dependencias específicas del operador, líneas azules que indican dependencias de datos generales del operador y en algunos casos líneas doradas que muestran dependencias de memoria.

Co-Simulación

- **Informe Co-simulación**: Muestra el estado de aprobación o rechazo y las estadísticas medidas sobre latencia e II.

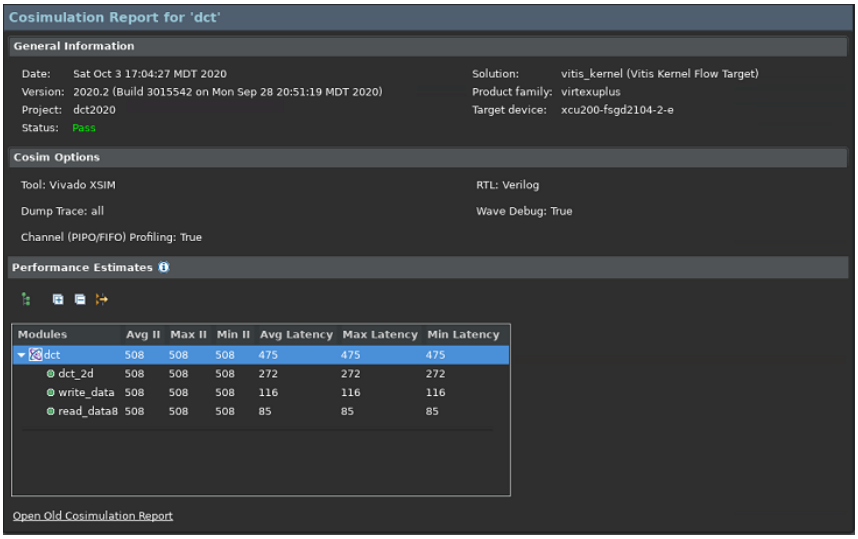

Figura 3.9: Informe Co-Simulación [33]

- **Visor de trazas de la línea de tiempo**: Muestra múltiples iteraciones a través de las distintas subfunciones de una región de flujo de datos, muestra dónde comienzan y terminan las funciones y muestra los datos de co-simulación en tablas debajo de la línea de tiempo.

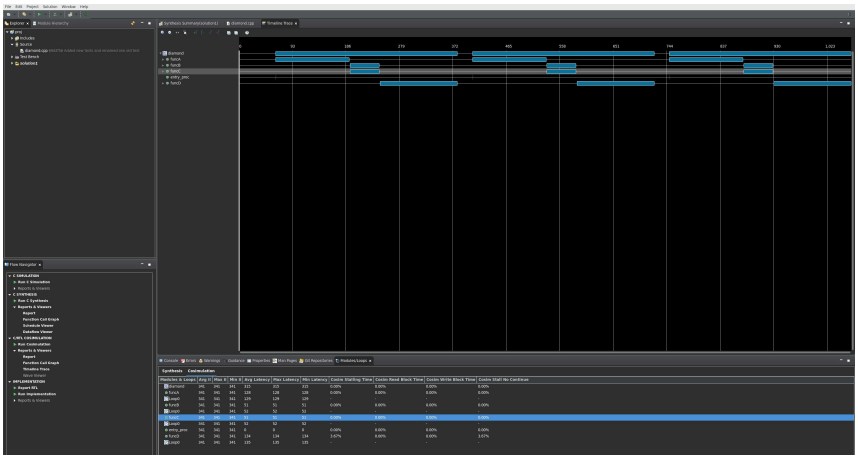

Figura 3.10: Visor de trazas de tiempo[33]

- **Visualización de formas de onda**: Para ver los datos de forma de onda durante la co-simulación RTL, se debe habilitar lo siguiente en el cuadro de diálogo Co-simulación (Figura 3.11):

  - Seleccionar Vivado XSIM como simulador RTL.
  - Habilitar *Dump Trace* con el puerto o con todas las opciones.

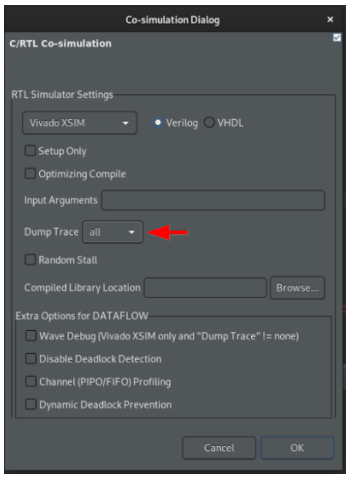

Figura 3.11: Dialogo Co-simulación

La interfaz gráfica de usuario del simulador abre y muestra todos los procesos en el diseño RTL. La visualización de los procesos activos dentro del diseño HLS permite crear perfiles detallados de la actividad y la duración del proceso dentro de cada activación del módulo superior. La visualización ayuda a analizar el rendimiento de cada proceso individual, así como la ejecución simultánea general de procesos independientes.

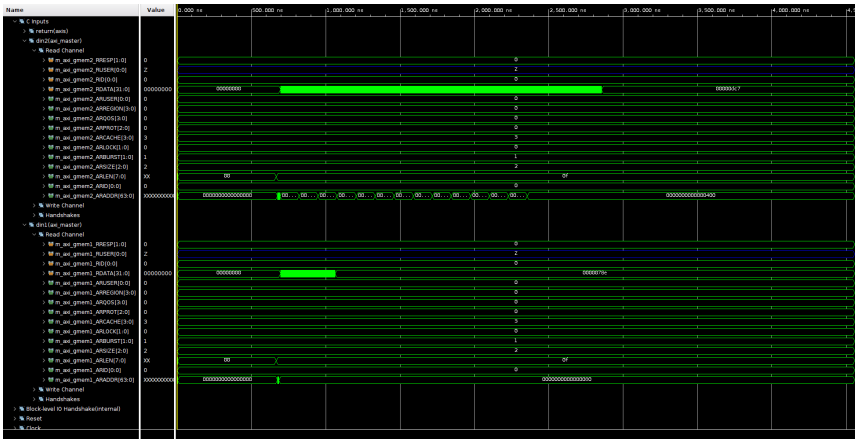

Figura 3.12: Visualizador de forma de ondas

Esta visualización se divide en dos secciones:

- **Resumen del proceso HLS**: Contiene una representación jerárquica del informe de actividad de todos los procesos.
- **Análisis del flujo de datos**: Proporciona información detallada sobre la actividad de las tareas dentro de la región del flujo de datos.

3.2.2. Vivado

Vivado es el entorno de desarrollo integrado (IDE) de Xilinx utilizado para el diseño, síntesis e implementación de *hardware* digital en dispositivos FPGA y SoC. Este entorno permite la creación, simulación, verificación e implementación de diseños mediante un flujo basado en lenguajes de descripción de *hardware* (HDL), como VHDL o Verilog, y un diseño visual a través de la herramienta *Block Design*.

El proceso de diseño para la implementación de la GRU en Vivado sigue una serie de pasos bien definidos. Como se explicó en el apartado 3.2, el proyecto base fue proporcionado, por lo que se utilizó este como punto de partida. En caso de no haber sido suministrado, el primer paso habría sido definir el dispositivo FPGA, las fuentes de diseño y las restricciones. Una vez creado el proyecto, los pasos que siguen son comunes a cualquier flujo de diseño y son los siguientes:

- **Síntesis:** En este paso, el código HDL se traduce a una red de puertas lógicas optimizada para la FPGA seleccionada.
- **Implementación:** Durante la implementación, se asignan los recursos físicos de la FPGA y se optimiza la distribución del diseño para cumplir con las restricciones de tiempo y área.
- **Generación del Bitstream:** Se genera el archivo binario que se carga en la FPGA para programarla y ejecutar el diseño.
- **Exportación del *Hardware*:** Finalmente, se crea el archivo de descripción del *hardware*, que se utiliza en entornos como Vitis para la programación y ejecución del diseño.

3.2.2.1. Integración de IP en Vivado y Block Design

Una de las ventajas de Vivado es su capacidad para integrar y reutilizar módulos de *hardware* predefinidos en forma de *Intellectual Property* (IP). En el contexto del proyecto, las IP diseñadas en Vitis HLS pueden exportarse a Vivado y utilizarse dentro del diseño de *hardware*. Esta integración se realiza a través del **IP Catalog**, que permite acceder a un conjunto de IP predefinidas y a aquellas generadas por el usuario.

Para facilitar la interconexión de múltiples IP y componentes, Vivado proporciona la herramienta **Block Design**, que permite crear el diseño mediante un enfoque visual y esquemático. En este entorno, se pueden agregar IP, configurar sus interfaces y conectar los diferentes bloques mediante buses AXI u otras conexiones personalizadas.

Los pasos generales para la integración de una IP generada en Vitis HLS dentro de Vivado son los siguientes:

1. **Exportación de la IP desde Vitis HLS:** Una vez validada y sintetizada la IP en Vitis HLS, se exporta en un formato compatible con Vivado.
2. **Importación en el IP Catalog de Vivado:** Se agrega la IP generada al IP Catalog de Vivado para que pueda ser utilizada en el diseño.
3. **Creación de un *Block Design*:** Se genera el *Block Design* donde se instancian y conectan las IP necesarias.
4. **Conexión de interfaces:** Se configuran y conectan los puertos de la IP con otros módulos o con la infraestructura del sistema, como el procesador Zynq o buses AXI.

   La metodología de conexión es sencilla, ya que a medida que se mueve el cursor cerca de una interfaz o conector de pin en un bloque IP, el cursor se transforma en un lápiz (Figura 3.13). A continuación, puede hacer clic en una interfaz o conector de pin en un bloque IP, mantener pulsado el botón izquierdo del ratón y, a continuación, dibujar la conexión al bloque de destino.

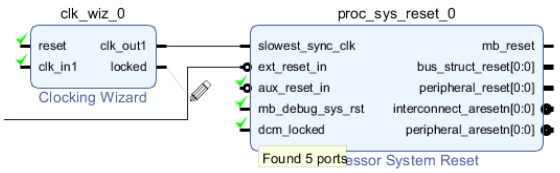

Figura 3.13: Conexión a nivel señal [34]

Existen 2 tipos de conexión:

- **A nivel de señal o de bus:** Esta conexión se muestra como una línea de conexión estrecha. Los buses se tratan de manera idéntica a las señales individuales para fines de conexión.
- **A nivel de interfaz:** Esta conexión se indica mediante un cuadro de conexión más prominente en un símbolo, como se muestra en la Figura 3.14.Cuando las señales se agrupan como una interfaz, puede conectar rápidamente todas las señales y buses de la interfaz con otros pines de interfaz compatibles.

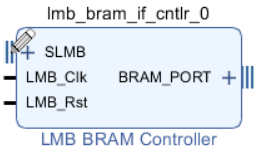

Figura 3.14: Conexión a nivel interfaz [34]

Al realizar conexiones, aparece una marca de verificación verde junto a cualquier conexión de destino compatible, resaltando las conexiones potenciales para la señal o interfaz (Figura 3.13).

Además de poder realizar el conexionado manual, el integrador de IP ofrece una función denominada *Designer Assistance*, que incluye la automatización de bloques y automatización de conexiones, para ayudarlo a armar un subsistema de IP básico mediante la realización de conexiones internas entre diferentes bloques y la realización de conexiones a interfaces externas.

En este proyecto se ha hecho uso de la herramienta automatización de conexiones (Figura 3.15) la cual enumera los puertos e interfaces que admite esta función, junto con una breve descripción de la automatización disponible y las opciones disponibles para cada automatización.

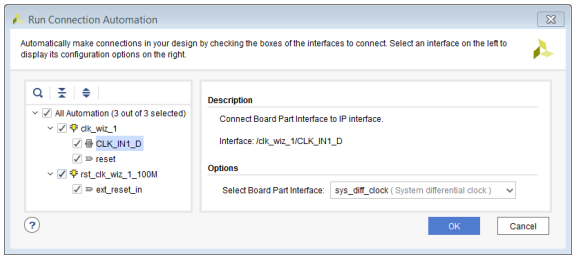

Figura 3.15: Automatización de conexiones [34]

Además realiza una asignación de direcciones de memoria adecuadas que son de vital importancia, ya que sin estas no se podría realizar la comprobación en Vitis IDE porque no se podrían acceder a las IP. Estas direcciones de memoria asignadas por el asistente se pueden observar en el *Address Editor* (Figura 3.16) y *Address Map* (Figura 3.17). Donde la función *Address Editor* muestra el direccionamiento en una vista de tabla agrupada por maestro. Y, la función *Address Map* muestra el direccionamiento en una vista intuitiva centrada en los esclavos y representa gráficamente la disposición de los esclavos AXI en un espacio de direcciones de red.

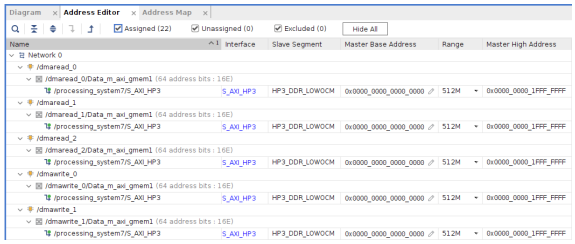

Figura 3.16: Address Editor

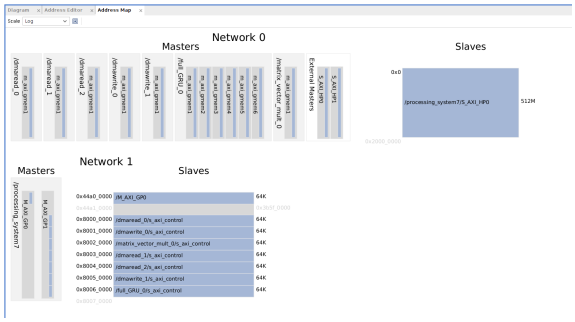

Figura 3.17: Address Map

5. **Generación del diseño y exportación:** Se finaliza la conexión del sistema, se valida el diseño y se genera la descripción del *hardware* para su implementación final.

El uso del *Block Design* y la integración de IP permiten modularizar el diseño, facilitar la reutilización de componentes y mejorar la escalabilidad del sistema, lo cual resulta fundamental en proyectos complejos como la implementación de la GRU en FPGA.

3.2.3. Vitis IDE

La plataforma de software unificada Vitis [35] es una nueva herramienta que combina todos los aspectos del desarrollo de software de Xilinx en un entorno unificado. La plataforma de software Vitis admite tanto el flujo de desarrollo de software integrado Vitis, como el flujo de desarrollo de aceleración de aplicaciones Vitis, para los desarrolladores de software que buscan utilizar lo último en aceleración de software basada en FPGA de Xilinx.

El Vitis IDE está diseñado para su uso en el desarrollo de aplicaciones de software integradas destinadas a procesadores integrados Xilinx. El Vitis IDE funciona con diseños de *hardware* creados con Vivado Design Suite. El Vitis IDE se basa en el estándar de código abierto Eclipse.

3.2.3.1. Flujo de trabajo

La Figura 3.18 muestra el flujo de trabajo de desarrollo de aplicaciones de software integrado para la plataforma de software unificada Vitis.

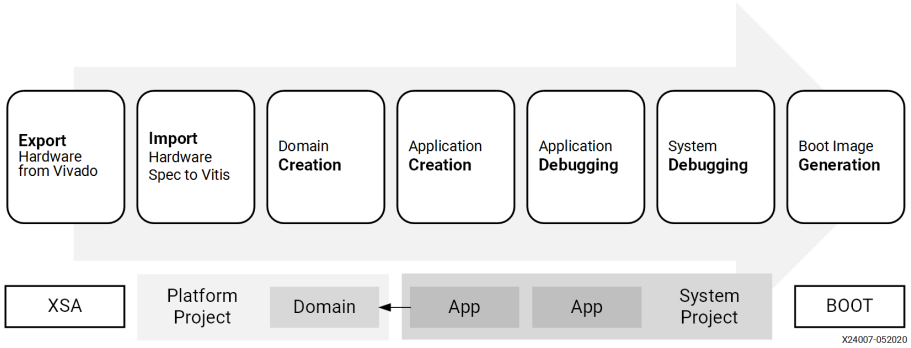

Figura 3.18: Flujo de trabajo Vitis SDK [35]

A continuación, se detallan los pasos a seguir en el flujo de trabajo para garantizar un desarrollo estructurado y eficiente, asegurando la correcta integración entre el *hardware* diseñado en Vivado®Design Suite y el software desarrollado en la plataforma Vitis.

1. Los ingenieros de *hardware* diseñan la lógica y exportan la información requerida por el desarrollo de software desde Vivado®Design Suite a un archivo XSA.
2. Los desarrolladores de software importan XSA a la plataforma de software Vitis mediante la creación de una plataforma. Para unificar la arquitectura del espacio de trabajo de Vitis para todo tipo de aplicaciones, los proyectos de desarrollo de software ahora migran a la arquitectura de la plataforma y la aplicación. Una plataforma incluye especificaciones de *hardware* y configuraciones del entorno de software.
3. Las configuraciones del entorno del software se denominan dominios, que también son parte de una plataforma.
4. Los desarrolladores de software crean aplicaciones basadas en la plataforma y los dominios.
5. Las aplicaciones se pueden depurar en el IDE de Vitis .
6. En un sistema complejo, varias aplicaciones pueden ejecutarse al mismo tiempo y comunicarse entre sí, por lo que también es necesario realizar una verificación a nivel del sistema.
7. Una vez que todo esté listo, Vitis IDE puede ayudar a crear imágenes de arranque que inicializan el sistema y lanzan aplicaciones.

3.2.3.2. Estructura del espacio de trabajo

Hay dos tipos de proyectos en el espacio de trabajo de Vitis, estos se muestran en la Figura 3.19.

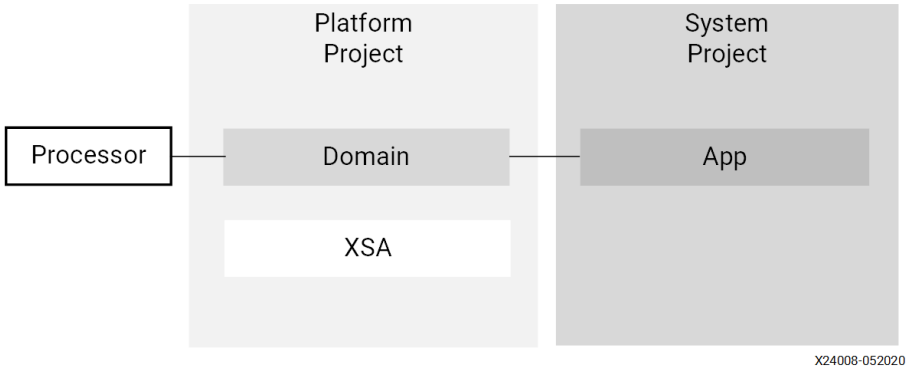

Figura 3.19: Tipos de proyecto en la plataforma Vitis [35]

- **Espacio de trabajo** (*Workspace*): Es una ubicación de directorio que se utiliza para almacenar datos y metadatos del proyecto. Cuando se inicia la plataforma de software Vitis se debe proporcionar una ubicación inicial para este espacio de trabajo.
- **XSA**: Los XSA se exportan desde *Vivado Design Suite*. Contienen las especificaciones de *hardware*, como las propiedades de configuración del procesador, la información de conexión de los periféricos, el mapa de direcciones y el código de inicialización del dispositivo. Debe proporcionar el XSA al crear un proyecto de plataforma.
- **Plataforma**: La plataforma o plataforma de destino es una combinación de componentes de *hardware* (XSA) y componentes de software (dominios/BSP, componentes de arranque como FSBL, etc.). Las plataformas del repositorio no se pueden editar. Las plataformas del espacio de trabajo sí se pueden editar y se denominan *platform project*.
- ***Platform Project***: Un *platform project* proporciona información de *hardware* y un entorno de ejecución de software. Es personalizable; puede agregar dominios y modificar la configuración de dominio. Se puede crear un proyecto de plataforma importando un XSA o una plataforma existente. Se pueden crear varios proyectos de sistema en el mismo *platform project* para que se puedan compartir las configuraciones del entorno de *hardware* y software.
- **Dominio** (*Domain*): Un dominio es un paquete de soporte de placa (BSP) o el sistema operativo (OS) con una colección de controladores de software sobre los cuales se puede crear la aplicación. La imagen de software creada contiene solo las partes de la biblioteca Xilinx que se utilizan en el diseño integrado. Se pueden crear múltiples aplicaciones para ejecutar en el dominio. Un dominio está vinculado a un solo procesador o a un grupo de procesadores isomorfos.
- **Proyecto de sistema**: Un proyecto de sistema agrupa aplicaciones que se ejecutan simultáneamente en un dispositivo. Dos aplicaciones independientes para el mismo procesador no pueden estar juntas en un proyecto de sistema. Dos aplicaciones Linux pueden estar juntas en un proyecto de sistema. Un espacio de trabajo puede contener varios proyectos de sistema.

- **Aplicación** (*APP*): Una aplicación contiene uno o más archivos fuente, junto con los archivos de encabezado necesarios, para permitir la compilación y generación de un archivo de salida binaria (ELF). Un proyecto de sistema puede contener varios proyectos de aplicación. Cada proyecto de software debe tener un dominio correspondiente.

La plataforma Vitis tiene diferentes configuraciones para soportar diferentes casos de uso, que se describen a continuación:

- **Integrada** (*Embedded*): Esta plataforma admite el desarrollo de software integrado para procesadores ARM y procesadores MicroBlaze.
- **Aceleración integrada** (*Embedded Acceleration*): Además del desarrollo de software integrado, este tipo de plataforma también admite la aceleración de aplicaciones. La plataforma proporciona relojes, interfaces de bus y controladores de interrupción para que los utilice el núcleo de aceleración.
- **Aceleración del centro de datos**: En esta plataforma se pueden desarrollar núcleos de aceleración y aplicaciones host x86. El núcleo se controla mediante un bus PCIe.

Capítulo 4

Desarrollo, Implementación y Resultados

En este apartado se estudiarán las distintas estrategias de implementación de la GRU, evaluando su rendimiento y eficiencia. Se realizará una comparación entre las diferentes aproximaciones haciendo uso de diferentes formatos numéricos con el objetivo de determinar la más adecuada según los criterios de precisión y uso de recursos.

Además, se realizarán comentarios sobre aspectos clave de cada una de las funciones que conforman la GRU, destacando su importancia en el diseño y la implementación en *hardware*. Se analizará el impacto de las configuraciones utilizadas y las optimizaciones aplicadas, considerando su integración en un entorno de desarrollo basado en herramientas de Xilinx.

Finalmente, se presentarán los resultados obtenidos y se discutirá su relevancia en el contexto de la implementación en *hardware*.

4.1. Implementación de la GRU

La implementación sigue el esquema denominado *recurrent-bias-after-multiplication* de Matlab [12] (Figura 4.1), una técnica que optimiza la adición de los sesgos (*bias*) durante la multiplicación de la matriz por el vector, mejorando tanto el uso de recursos como el tiempo de computación. Esta optimización se logra al añadir un valor ‘1’ en el primer índice del vector de entrada y colocar los valores de los sesgos en la primera fila de la matriz de pesos. De este modo, los sesgos se suman directamente durante la multiplicación, eliminando la necesidad de una operación de suma adicional. Los datos de entrada, que incluyen los valores de los sesgos y los pesos, se suministran en un formato Q8.8, calculado previamente con TensorFlow. En esta implementación, se utilizan 16 neuronas ocultas y se realizan un total de 150 iteraciones.

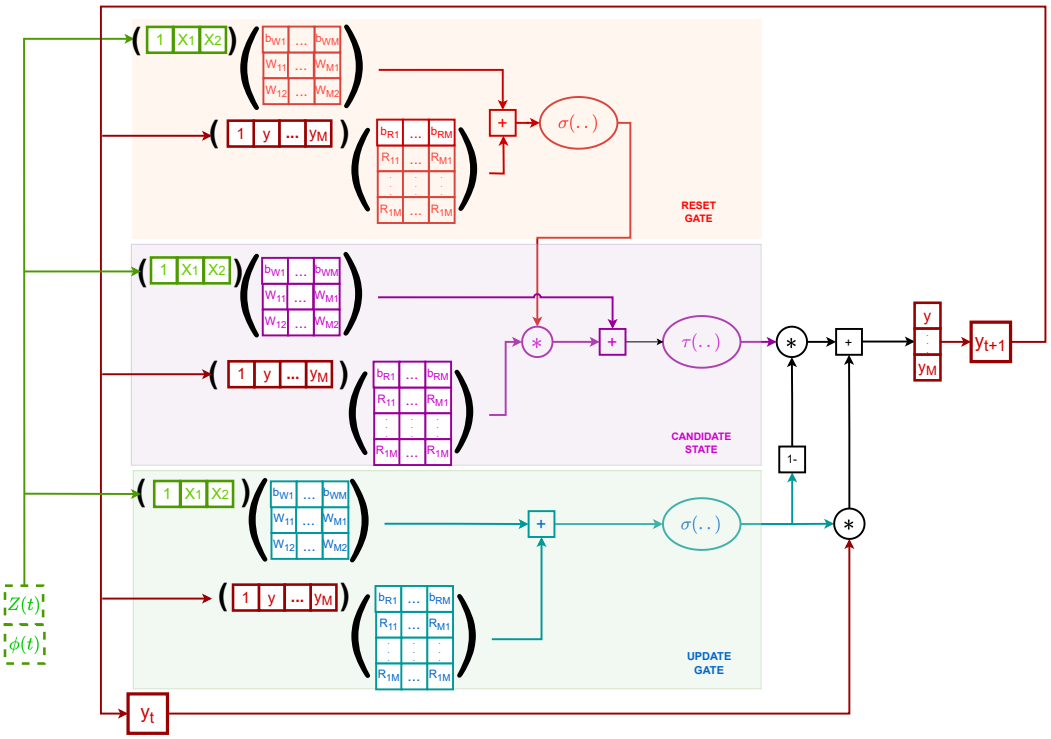

Figura 4.1: Esquema celda GRU proyecto

El esquema anterior proporciona una visión global de la arquitectura y la interacción entre los diferentes componentes de la GRU. En él se observa cómo las entradas, los pesos y los sesgos interactúan en el proceso de la multiplicación de la matriz por el vector, así como la forma en que la optimización de la adición de sesgos se realiza eficientemente. Este esquema muestra de manera abstracta la estructura de la GRU, centrada en la idea general de procesamiento y la optimización durante el cálculo.

En contraste, el esquema detallado de las funciones de la GRU que se muestra a continuación desglosa específicamente las funciones que hacen posible esta implementación de la GRU.

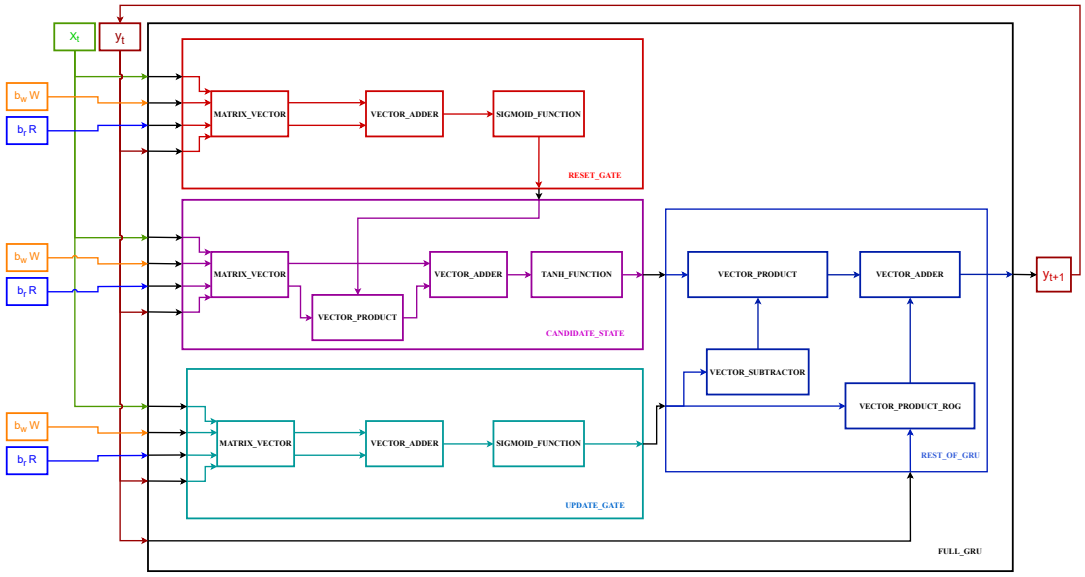

Figura 4.2: Diagrama funciones celda GRU genérico

En resumen, lo que se pretende con estos dos esquemas complementarios es proporcionar una visión global del diseño, y entender cómo se lleva a cabo la computación dentro de la GRU.

4.1.1. Implementaciones y Análisis de sus Componentes

La implementación de la GRU ha sido realizada utilizando distintos tipos de datos, con el objetivo de comparar el rendimiento y la eficiencia de cada uno de ellos en la FPGA, a la vez que se optimizan los recursos y se mantiene la precisión necesaria para la operación. Para ello, se han utilizado tres tipos de datos en particular: float, half y ap_fixed<18,8,AP_RND_CONV,AP_SAT>, cada uno con sus ventajas y limitaciones en el contexto de la implementación de la GRU.

Para llevar a cabo estas implementaciones, se ha hecho uso de una cabecera denominado AXI_VALUE_H (Apéndice A.1), el cual define una estructura común llamada AxiWord. En esta estructura, el campo data se adapta según el tipo de dato especificado. El tipo float y half se seleccionan mediante la inclusión o no del #define bits_32. Cuando se define bits_32, se utiliza el tipo de dato float, que permite representar valores de punto flotante de 32 bits. Si no se define bits_32, el tipo de dato utilizado es half, una representación de menor precisión (16 bits) que permite ahorrar recursos en la FPGA, aunque con la posible pérdida de precisión en algunas operaciones. De esta forma, la selección del tipo de dato se ajusta a los requerimientos específicos de la implementación y la eficiencia en el uso de los recursos de la FPGA. Un ejemplo de implementación de esta estructura en una función se muestra en el Apéndice A.1.

En el caso de ap_fixed<18,8,AP_RND_CONV,AP_SAT>, se está utilizando una representación de 18 bits, donde 10 de esos bits corresponden a la parte fraccionaria y los 8 restantes a la parte entera. Además, se ha especificado que el redondeo se realiza mediante AP_RND_CONV, que es un tipo de redondeo que busca minimizar el error de cuantización y convertir el número al valor más cercano. Por otro lado, la opción AP_SAT asegura que los valores no excedan el rango máximo

permitido, lo que evita desbordamientos o resultados incorrectos debido a la saturación. Este tipo de dato es particularmente relevante ya que, al ser un número de punto fijo, se permite un control más preciso sobre la cantidad de bits para la parte entera y la parte fraccionaria. Dado que el multiplicador DSP de la FPGA es de 25x18 bits, el tipo de dato ap_fixed<18,8,AP_RND_CONV,AP_SAT> garantiza que los resultados de las multiplicaciones no excedan la capacidad del *hardware* sin perder precisión, evitando así la necesidad de utilizar más de un DSP en una operación. Esto resulta en un uso más eficiente de los recursos y en una reducción de los requisitos de *hardware*.

Además de estas implementaciones basadas en el tipo de dato proporcionado por la estructura AxiWord, se llevó a cabo una implementación alternativa en la que se calculó el número de bits necesarios para cada operación con la menor perdida de precisión posible. Todas las explicaciones sobre las funciones y las decisiones tomadas en cuanto a los pragmas se basarán en esta implementación, ya que es la más compleja debido a la variedad de tipos de datos utilizados.

Para continuar con la explicación, se presentará una imagen que muestra el esquema general de las funciones utilizadas en esta implementación junto a su formato de dato. Este esquema refleja la estructura funcional de la GRU, pero con las modificaciones necesarias para adaptarse a la implementación de cuantificación por funciones.

Como se puede observar en la Figura 4.3, las funciones encargadas de los productos y las sumas han tenido que ser implementadas de manera independiente debido a que los tipos de datos utilizados en cada operación no son homogéneos. Sin embargo, a pesar de estas diferencias, la base y lógica de cada una de estas funciones se mantiene consistente con el esquema original.

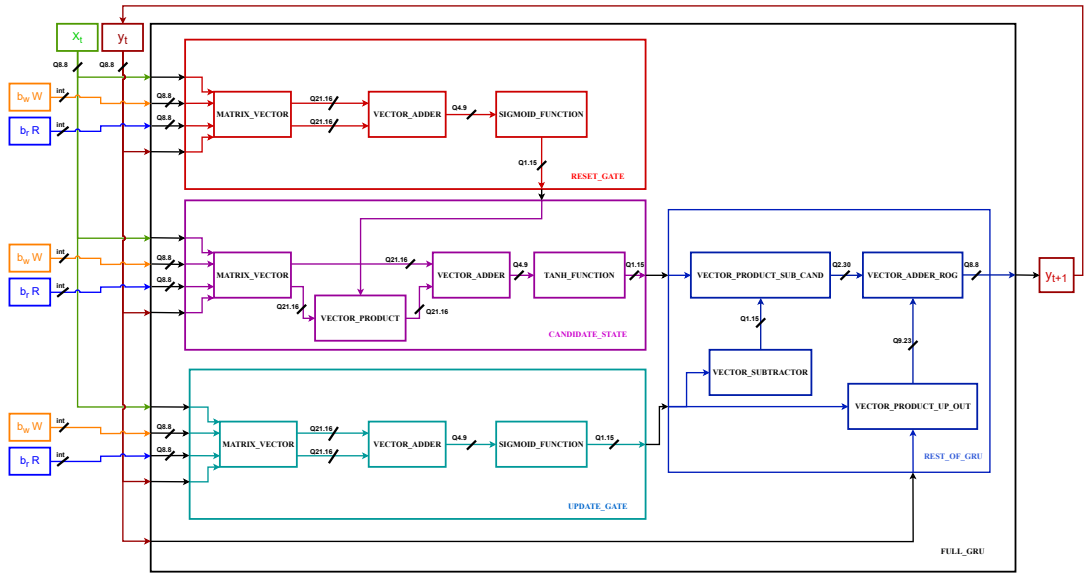

Figura 4.3: Diagrama funciones celda GRU cuantificado

A partir de este esquema, se analizará en detalle cada una de las funciones que componen la implementación de la GRU, explicando los aspectos más relevantes de su diseño, incluyendo las decisiones tomadas en cuanto a los tipos de datos, la estructuración del código y el uso de pragmas para la optimización en *hardware*. Este análisis se realizará de manera jerárquica, abordando cada

módulo en función de su papel dentro del flujo de datos de la GRU, detallando aspectos relevantes de su implementación y optimización en *hardware*.

Para facilitar la comprensión del análisis de la implementación, a continuación se presenta un esquema de los puntos que se desarrollarán en los siguientes apartados:

- Función Top (`full_GRU`)
  - **Esquema y Explicación de los Tipos de Datos de las Entradas y Salidas** Se describe el tipo de datos utilizado para las entradas y la salida de la función principal, así como su formato y los requerimientos para su correcta interpretación.
  - **Metodología de Duplicación de Datos de Entrada** Se explica el proceso de duplicación de datos en las entradas, detallando cómo se gestionan y procesan para optimizar el rendimiento de la implementación.
  - **Uso de los Pragma `INTERFACE`, `STREAM` y `DATAFLOW`** Se discute la importancia de los pragmas en el contexto de la implementación, y cómo afectan el rendimiento del sistema. Se justifica por qué es necesario su uso, haciendo hincapié en el pragma `STREAM` y en cómo se puede modificar internamente en Vitis HLS para adaptarlos a las necesidades del proyecto.
- Multiplicación de Matriz por Vector
  - **Esquema y Explicación de los Tipos de Datos de las Entradas y Salidas** Se proporciona un esquema detallado de la multiplicación de la matriz por el vector, explicando los tipos de datos implicados en la operación.
  - **Optimización de la Memoria con `static`** Se analiza la función que utiliza el modificador `static` en las variables que almacenan las matrices, y se proporciona evidencia visual mediante imágenes de cosimulación que muestran los resultados de la latencia máxima y mínima. Además, se presenta la justificación de la prueba doble realizada en el testbench para comprobar la correcta reutilización de las matrices sin duplicación de almacenamiento.
  - **Pragmas Relevantes en la Multiplicación de Matriz por Vector** Se identifican y explican los pragmas internos utilizados en las operaciones de multiplicación. Se profundiza en la importancia del pragma específico en la multiplicación, explicando los efectos que tiene en la eficiencia de la implementación.
- Operaciones de Producto y Suma de Vectores
  - **Esquema y Explicación de los Tipos de Datos de las Entradas y Salidas** Se presenta un esquema detallado de las operaciones de multiplicación y suma de vectores, explicando los tipos de datos de entrada y la salida esperada.
  - **Multiplicación de Vectores** Se describen los diferentes tipos de multiplicación de vectores que se emplean en el sistema, destacando las particularidades en función de los tipos de datos involucrados. Se presta especial atención a la multiplicación de los vectores `out` y `update`, en donde el primer valor del vector `out`, que contiene el número 1, debe ser descartado para evitar interferencias con el cálculo de los sesgos.

- **Suma de Vectores** Se explica la operación de suma de vectores, detallando cómo se realiza la suma de los elementos correspondientes de los vectores de entrada. Además, se explica el porque de la discrepancia del formato teórico de la suma con el real implementado debido a las funciones que le siguen.

- Funciones de Activación

  - **Diseño e Implementación de la Función Sigmoide y Tangente Hiperbólica** Se detallan los procesos de diseño e implementación de las funciones de activación sigmoide y tangente hiperbólica, esenciales en el funcionamiento del GRU.
  - **Estrategia de Almacenamiento y Acceso a Valores Precomputados** Se describe la estrategia de almacenamiento utilizada para los valores precomputados de estas funciones de activación, así como los métodos empleados para acceder a estos valores durante la ejecución.

4.1.1.1. Funciones Top

La función top (Ful_GRU) recibe como entrada los vectores de entrada ,X_t, vectores de salida ,Y_t y los pesos y sesgos utilizados en cada una de las puertas internas. A continuación, se describen las entradas y salidas de la función que se pueden observar en el esquema de la Figura 4.4.

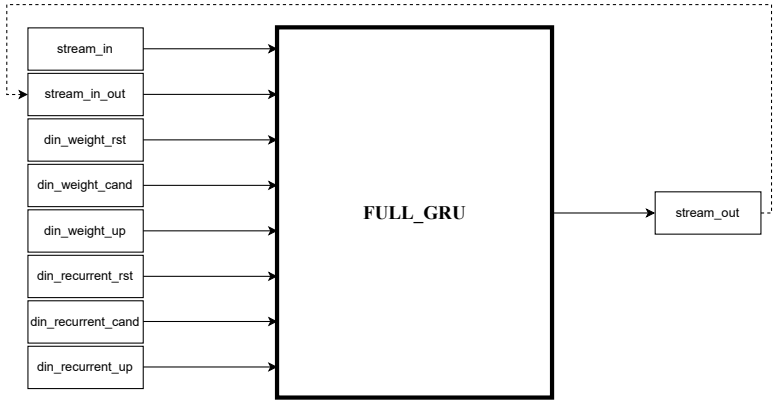

Figura 4.4: Esquema función TOP: FULL_GRU

- **ENTRADAS:**

  - **stream_in:** Representa la nueva información que ingresa a la celda GRU en el instante actual de tiempo. Se encuentra en formato de punto fijo Q8.8.
  - **stream_in_out:** Corresponde al resultado de la celda en el instante de tiempo previo, Y_t, permitiendo que la red mantenga la memoria de estados pasados. También se encuentra en formato Q8.8.
  - **din_weight_X y din_recurrent_X:** Tanto los pesos como los sesgos se almacenan en memoria externa y se leen a través de un bus AXI4. Como se puede observar en la

Figura 4.3, estos parámetros están organizados en bloques correspondientes a cada una de las tres puertas de la GRU (Reset, Update y Candidate). Para simplificar la notación, se emplea la letra X como referencia genérica a cualquiera de estas tres puertas.

En este contexto, los pesos y sesgos (din_weight_X) corresponden a los coeficientes que multiplican la entrada actual de la GRU, mientras que los pesos y sesgos recurrentes (din_recurrent_X) son los coeficientes aplicados al estado oculto anterior. Todos estos valores tienen un formato de 32 bits enteros, que posteriormente son reinterpretados en el formato de punto fijo Q8.8.

- **SALIDAS:**
  - **stream_out:** Esta salida se obtenido a partir de la salida de la función *Rest_of_GRU*. Este vector mantiene el mismo formato que las entradas, Q8.8, para garantizar la coherencia en el flujo de datos dentro del sistema.

Pragma HLS INTERFACE

En la implementación de la función `full_GRU` (Apéndice A.2), es necesario definir las interfaces que permiten la comunicación entre los distintos módulos y las conexiones con la memoria y los flujos de datos. Para lograrlo, se utilizan varios pragmas `HLS INTERFACE`, que configuran los puertos de entrada y salida, así como las interacciones con la memoria externa y las señales de control. A continuación, se detallan las configuraciones específicas de los pragmas utilizados en la función `full_GRU`:

- **Interfaces streaming (AXIS):** Para los flujos de datos de entrada y salida, se emplea la configuración mode=axis que permite una transmisión continua de datos a través de un bus de streaming. Esto es adecuado para manejar las entradas y salidas de tipo `hls::stream<data_in_mvm>` y `hls::stream<data_out_gru>`, que corresponden a los datos de las secuencias de entrada y salida de la GRU. El registro de profundidad configurado es de 1024, permitiendo una alta velocidad de transmisión de datos a través de estas interfaces.
- **Interfaces memoria mapeada (M_AXI):** Las matrices de pesos y estados recurrentes se gestionan utilizando interfaces de memoria mapeada M_AXI. Estas configuraciones permiten acceder a los bloques de memoria externa que contienen los valores de las matrices, utilizando varias interfaces de tipo m_axi.

  Cada una de estas interfaces conecta la función full_GRU con la memoria externa correspondiente para las matrices de reset, candidate y update, utilizando las configuraciones de AXI de acceso a memoria, con diferentes profundidades de memoria según las necesidades de cada matriz debido a las dimensiones de estas.
- **Interfaz de control (S_AXILITE):** Finalmente, el pragma mode=s_axilite se utiliza para configurar una interfaz de tipo AXI-Lite para la señal de retorno, lo que permite la transmisión de señales de control y el retorno de la función.

Es importante destacar que, para poder realizar simulaciones individuales de cada una de las funciones que se describirán en este apartado y a lo largo de los siguientes, sin necesidad de utilizar la función top completa, los pragmas HLS INTERFACE se reescriben dentro de cada función. Esto

se hace para asegurar que todas las interfaces necesarias estén correctamente configuradas en cada módulo, permitiendo su simulación o implementación de forma independiente.

Duplicado de entradas y pragma HLS STREAM

Una de las consideraciones clave en la implementación de la función full_GRU es la necesidad de duplicar los flujos de datos de entrada y salida para asegurar que las operaciones puedan ejecutarse de manera eficiente y paralela dentro de la arquitectura. Este proceso de duplicado, combinado con el uso del pragma HLS STREAM, permite un manejo eficiente de los flujos de datos a través de diferentes módulos y garantiza que la información se procese de manera continua.

El duplicado de flujos de datos se refiere a la creación de múltiples copias de los datos de entrada y salida en el sistema.En el contexto de la full_GRU, el duplicado asegura que las secuencias de entrada y salida puedan ser procesadas de manera paralela en todas las puertas.

El pragma HLS STREAM, que se utiliza en las interfaces de los flujos de datos, es crucial para manejar estos duplicados de manera eficiente. Este pragma permite la transmisión continua de datos a través de un bus de streaming, lo que facilita la conexión de los módulos entre sí y asegura que las operaciones sobre los datos sean procesadas sin intervención externa o esperas innecesarias. Al aplicar HLS STREAM, los datos son enviados y recibidos en tiempo real, asegurando que cada módulo pueda acceder a los datos de entrada o salida de manera inmediata.

Es importante destacar que este pragma es el que hace que la GRU funcione correctamente. Sin este pragma, VITIS HLS crea FIFO de tamaño 2 para la comunicación, lo cual es insuficiente para nuestra implementación, que requiere un tamaño de 17. Este parámetro de tamaño 2 puede cambiarse de forma genérica en VITIS, lo que afectaría a todos los trabajos existentes. Por este motivo, se ha considerado oportuno hacer uso de este pragma `HLS STREAM` para asegurar el funcionamiento adecuado de la GRU sin afectar a trabajos existentes o futuros.

A través de la duplicación y el uso de HLS STREAM, los datos pueden ser gestionados de manera eficiente sin necesidad de copiar manualmente los vectores o de depender de mecanismos de control adicionales. Esto simplifica la implementación y mejora el rendimiento del sistema en general.

Pragma HLS DATAFLOW

El pragma #pragma HLS DATAFLOW juega un papel crucial al permitir que diferentes módulos del diseño se ejecuten en paralelo, lo que es esencial para las redes neuronales y otros sistemas que requieren un alto rendimiento en el procesamiento de grandes volúmenes de datos. En el contexto de la función `full_GRU`, este pragma asegura que las diferentes etapas del proceso, como la activación de las puertas y el cálculo de la salida final, puedan ejecutarse de manera concurrente. A continuación se detalla en qué ayuda este pragma a la implementación de la GRU.

- **Paralelización de Módulos:** La principal ventaja de `HLS DATAFLOW` es la paralelización de las diferentes etapas del proceso. En el caso específico del código de la función `full_GRU`, las tres puertas principales de la GRU: `reset_gate`, `candidate_gate`, y `update_gate`, pueden ejecutarse simultáneamente. Cada una de estas funciones se encarga de operar sobre un conjunto diferente de flujos de datos, lo que permite su ejecución en paralelo sin interferir

entre sí. Esta paralelización es esencial para aprovechar al máximo los recursos del FPGA, maximizando el rendimiento global del sistema.

- **Sincronización de Flujos de Datos:** En este contexto, el pragma `HLS DATAFLOW` garantiza que las salidas de las funciones `reset_gate`, `candidate_gate` y `update_gate` se sincronicen correctamente, manteniendo la coherencia de los datos y evitando cuellos de botella.
- **Optimización de Recursos:** El uso del pragma `HLS DATAFLOW` permite minimizar los tiempos de espera entre operaciones. Sin este pragma, las etapas del procesamiento tendrían que esperar a que las anteriores terminaran para poder comenzar, lo que resultaría en tiempos de inactividad y en un uso ineficiente de los recursos del FPGA. Al permitir que las operaciones se realicen en paralelo, se maximiza el uso de las unidades de procesamiento disponibles, lo que es crucial para sistemas de alto rendimiento como es el caso de la GRU que se implementa en este proyecto.

Cabe mencionar que este pragma también se utiliza en cada una de las funciones llamadas en la implementación de full_GRU por los mismos motivos. Por lo tanto, la explicación de este pragma en la función top, full_GRU, abarca también su aplicación en las funciones llamadas dentro de ella.

Puertas Reset Gate y Update Gate

Las puertas Reset Gate y Update Gate tienen la misma estructura e implementación (Figura 4.5), diferenciándose únicamente en los pesos y sesgos utilizados en cada una de ellas.

Figura 4.5: Esquema funciones puertas: Reset Gate y Update Gate

Ambas puertas reciben como entradas el estado oculto previo y la entrada actual, que son multiplicados por sus respectivos pesos y sumados a su sesgo correspondiente. Posteriormente, se aplica la función sigmoide a la suma ponderada para obtener la salida de la puerta. Todas estas funciones son descritas en apartados anteriores con mas detalle. A continuación se indica indican las entradas y su formato:

- **ENTRADAS:**
  - **stream_in:** Representa la nueva información que ingresa a la celda GRU en el instante actual de tiempo. Se encuentra en formato de punto fijo Q8.8.
  - **stream_in_out:** Corresponde al resultado de la celda en el instante de tiempo previo, Y_t, permitiendo que la red mantenga la memoria de estados pasados. También se encuentra en formato Q8.8.
  - **din_1 y din_2:** Tanto los pesos como los sesgos se almacenan en memoria externa y se leen a través de un bus AXI4. Los pesos y sesgos (din_1) corresponden a los coeficientes que multiplican la entrada actual de la GRU, mientras que los pesos y sesgos recurrentes (din_2) son los coeficientes aplicados al estado oculto anterior. Todos estos valores tienen un formato de 32 bits enteros, que posteriormente son reinterpretados en el formato de punto fijo Q8.8 en las funciones Matriz-Vector.
- **SALIDAS:**
  - **stream_out:** Esta salida se obtenido a partir de la salida de la función *sigmoid_function*, cuyo formato es Q1.15 pero con un rango real de valores entre 0 y 1, debido la expresión de la función sigmoide, cuya representación se puede observar en la Figura 2.6.

Dado que ambas puertas siguen exactamente el mismo proceso matemático, lo ideal habría sido implementar una única función reutilizable para ambas, reduciendo la redundancia y mejorando la eficiencia del código. Sin embargo, durante la implementación en *hardware*, se identificaron problemas al generar instancias separadas de la misma función, lo que impedía su correcto funcionamiento en la síntesis. Para evitar estos inconvenientes, se decidió duplicar el código y definir funciones separadas para cada puerta.

En el apéndice A.2.1 se muestra el código fuente de una de las funciones, indicando que, mediante un simple cambio en el nombre de la función, se instancia la otra puerta sin necesidad de modificar el código interno.

Puerta Candidate State

La puerta Candidate State sigue un proceso similar al de las puertas Reset Gate y Update Gate, diferenciándose principalmente en la función de activación utilizada. En este caso, en lugar de aplicar una sigmoide, se emplea la función tangente hiperbólica (tanh), cuya representación se muestra en la Figura 2.7. Esto permite que los valores de salida se encuentren en el rango [−1,1]. La estructura e implementación de esta puerta se muestra en la Figura 4.6.

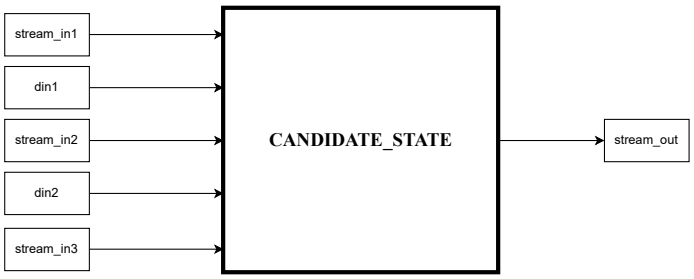

Figura 4.6: Esquema función puerta Candidate State

En el apéndice A.2.2 se presenta el código fuente de esta función, el cual está estructurado de manera similar a las puertas anteriores, aunque con un esquema de procesamiento distinto. Sin embargo, conserva la misma base, incorporando la salida de la función reset gate como nueva entrada, además de una función de activación distinta, tanh. A continuación, se detallan sus entradas y salidas, junto con sus respectivos formatos:

- **ENTRADAS:**
  - **stream_in1:** Representa la nueva información que ingresa a la celda GRU en el instante actual de tiempo, con formato de punto fijo Q8.8.
  - **stream_in2:** Tanto los pesos como los sesgos se almacenan en memoria externa y se leen a través de un bus AXI4. Los pesos y sesgos (din_1) corresponden a los coeficientes que multiplican la entrada actual de la GRU, mientras que los pesos y sesgos recurrentes (din_2) son los coeficientes aplicados al estado oculto anterior. Todos estos valores tienen un formato de 32 bits enteros, que posteriormente son reinterpretados en el formato de punto fijo Q8.8 en las funciones Matriz-Vector.
  - **stream_in3:** Entrada adicional que proporciona la salida de la puerta Reset Gate, cuyo formato es Q1.15.
- **SALIDAS:**
  - **stream_out:** Esta salida se obtenido a partir de la salida de la función *tanh_function*.Esta salida tiene formato Q1.15, con un rango real de [-1,1[, debido a que con 1 bit entero no se puede representar el valor exacto 1, e ha asumido que el error es aceptable, ya que con los 15 bits fraccionales la precisión es de 2^{15}.

Rest of GRU

Las operaciones que no forman parte directamente de las puertas Reset Gate, Update Gate y Candidate State se agrupan en esta función denominada Rest of GRU. Esta función engloba

el conjunto de operaciones necesarias para calcular la salida de la celda GRU, combinando los resultados obtenidos de las puertas previamente descritas. La estructura de esta función se muestra en la Figura 4.7, y su código fuente se encuentra en el apéndice A.2.3. A continuación, se detallan las entradas y salidas asociadas a esta función, explicando cómo se utilizan los valores de las puertas para obtener el resultado final.

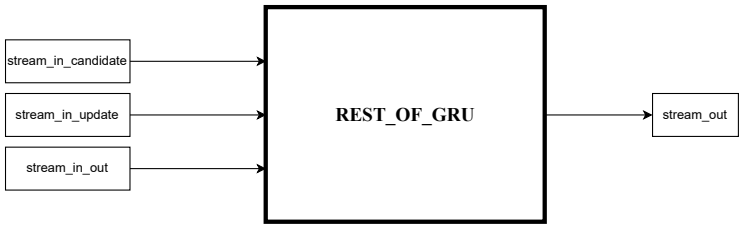

Figura 4.7: Esquema función Rest of GRU

- **ENTRADAS:**
  - **stream_in_candidate:** Representa la salida de la puerta *Candidate State*, con formato de punto fijo Q1.15.
  - **stream_in_update:** Representa la salida de la puerta *Update Gate*, con formato de punto fijo Q1.15. Esta salida será duplicada internamente de igual forma que se explico anteriormente haciendo uso de un bucle y del pragma HLS STREAM.
  - **stream_in_out:** Representa la salida de la etapa anterior, Y_t, con formato Q8.8.
- **SALIDAS:**
  - **stream_out:** Esta salida es la salida de la GRU , Y_{t+1}, por lo que como se va a realimentar a Y_t, su formato debe ser el mismo Q8.8.

4.1.1.2. Multiplicación Matriz-Vector

Esta función cuenta con cuatro entradas y dos salidas, ya que ambas multiplicaciones Matriz-Vector han sido implementadas dentro de la misma función. El esquema correspondiente se muestra en la Figura 4.8. A continuación, se describen las variables que intervienen en esta función:

- **ENTRADAS:** La representación numérica de las entradas tiene un formato de punto fijo Q8.8, lo que significa que se utiliza una representación de 8 bits tanto para la parte entera como para la parte decimal.
  - **stream_in1:** Dato de entrada X_t, provenientes por AXI4-Stream de la DMA de lectura, con formato Q8.8.

- **din1:** Datos de entrada de los pesos y sesgos que forman parte de la primera multiplicación Matriz-Vector encargada de multiplicar la entrada X_t. Estos datos son adquiridos directamente mediante un puerto AXI4 de la DDR de la FPGA. Estos datos de entrada tiene un formato int (32 bits).
- **stream_in2:** Dato de entrada Y_t provenientes por AXI4-Stream de la DMA de lectura, con formato Q8.8.
- **din2:** Datos de entrada de los pesos y sesgos que forman parte de la segunda multiplicación Matriz-Vector encargada de multiplicar la entrada Y_t. Estos datos son adquiridos directamente mediante un puerto AXI4 de la DDR de la FPGA.Estos datos de entrada tiene un formato int (32 bits).

- **SALIDAS:** Al multiplicar dos números de tipo Q8.8, el resultado es un tipo de dato Q16.16, lo que implica que el número de bits de la parte entera y la parte decimal se duplican. En el proceso de acumulación posterior a las multiplicaciones (Apéndice A.2.1.1), se consideran dos casos, correspondientes a las 2 salidas, para asegurar que no haya pérdida de precisión:
  - **stream_out1 (MVM entrada x sesgos y pesos):** El acumulador deberá realizar tres multiplicaciones, lo que provoca que el dato resultante crezca en tamaño según $log_2(3) = 1{,}58$, requiriendo 2 bits adicionales para la parte entera. Por lo tanto, el formato de salida de esta operación sería Q18.16, pero para mantener ambos formatos iguales es Q21.16.
  - **stream_out2 (MVM salida x sesgos y pesos):** En este caso, el acumulador realiza hasta 17 iteraciones, lo que resulta en $log_2(17) = 4{,}0874$, lo que requiere 5 bits adicionales para la parte entera. Como resultado, el formato de salida será Q21.16.

Figura 4.8: Esquema función Matriz-Vector (MVM)

Uno de los aspectos más relevantes de esta implementación es la forma en que se almacenan los datos de las matrices. Aunque los pesos y sesgos se encuentran en formato `int`, la información relevante se encuentra en los 16 bits menos significativos (LSB). Por este motivo, es necesario aplicar una máscara para extraer estos bits y reinterpretarlos en formato Q8.8 antes de almacenarlos.

Dado que estos valores permanecen constantes a lo largo de la ejecución, se han definido como variables `static`, asegurando que se conserven entre llamadas a la función. Para evitar que los datos sean sobrescritos en cada ejecución, se emplea una variable booleana (`first_iteration`), la cual permite que el almacenamiento de los datos en memoria ocurra solo en la primera iteración.

Para verificar este comportamiento, se realizó una simulación ejecutando dos veces el testbench correspondiente. Los resultados obtenidos en la ejecución de la co-simulación (Figura 4.10 y Figura 4.9) muestran que los datos se almacenan únicamente una vez, lo que se refleja en la diferencia de latencia entre la primera y la segunda iteración (MAX vs MIN).

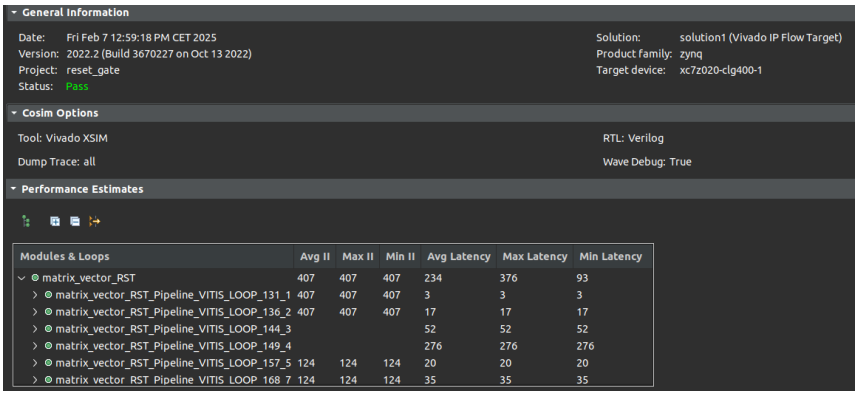

Figura 4.9: Resultado Co-simulación: Tiempos de ejecución

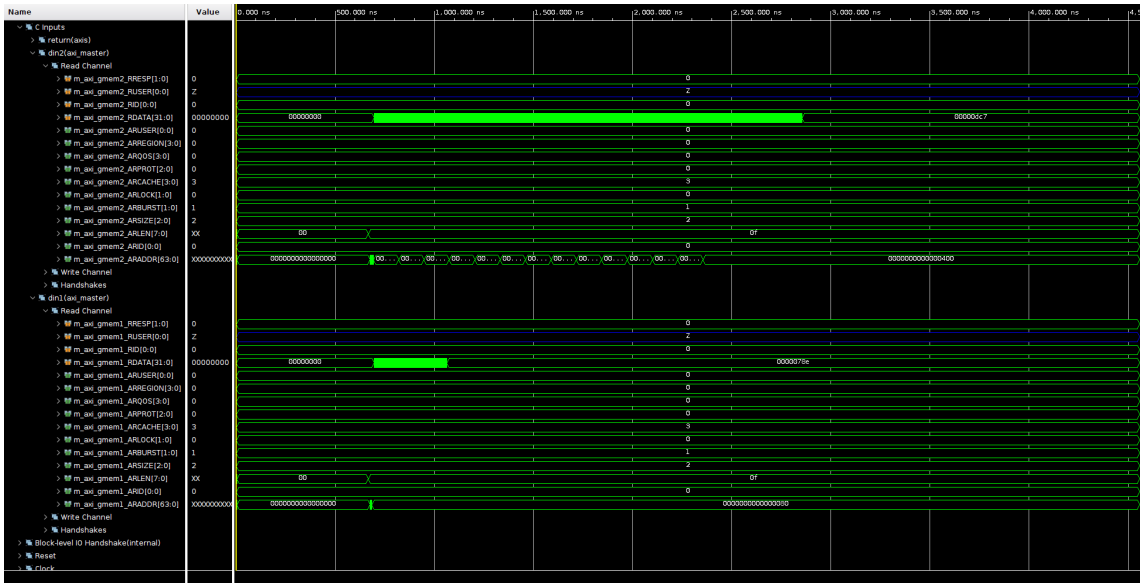

Figura 4.10: Resultado Co-simulación: Wave Viewer

Por ultimo, existe otro aspecto importante a la hora de realizar la implementación de la multiplicación Matriz-Vector, y es el uso la directiva `#pragma HLS PIPELINE` dentro de los bucles de cálculo. Su propósito es optimizar el rendimiento de la función al permitir la ejecución en *pipelining*, reduciendo la latencia y mejorando el aprovechamiento de los recursos de la FPGA.

De forma general en HLS, los bucles sin `#pragma HLS PIPELINE` suelen ejecutarse de manera secuencial, procesando una iteración en cada ciclo de reloj. Esto introduce una latencia significativa en bucles con muchas iteraciones. Al agregar este pragma, se habilita la ejecución parcialmente solapada de las iteraciones del bucle, permitiendo que se inicien nuevas iteraciones antes de que las anteriores finalicen.

Esta directiva mejora el rendimiento de la función al reducir la latencia del bucle externo, ya que se pueden iniciar nuevas iteraciones sin esperar a que la iteración anterior se complete por completo.

El uso de `#pragma HLS PIPELINE` conlleva las siguientes ventajas:

- **Reducción de latencia**: La ejecución en pipeline permite que múltiples iteraciones se solapen, acelerando la ejecución total.
- **Mejor aprovechamiento del *hardware***: Se optimiza el uso de los recursos internos de la FPGA al permitir que distintas operaciones se ejecuten en paralelo.
- **Aumento del rendimiento global**: Especialmente en bucles intensivos en cálculo, como los que involucran la multiplicación Matriz-Vector.

4.1.1.3. Suma y producto de vectores

Las funciones encargadas de realizar la suma de vectores comparten la misma base de código, diferenciándose únicamente en los formatos numéricos de sus datos de entrada y salida. En esta implementación, se han desarrollado dos versiones con representaciones numéricas distintas. El esquema correspondiente se presenta en la Figura 4.11. A continuación, se describen en detalle las variables involucradas en cada una de estas funciones:

- **Vector_adder**(Apéndice A.2.1.2): Esta función se emplea dentro de las puertas de la GRU, por lo que su nomenclatura y formato de datos son uniformes en todas las instancias en las que se utiliza.
  - **Entradas:**
    - **stream_in1:** Dato de entrada correspondiente al primer resultado de la multiplicación Matriz-Vector, con formato Q21.16.
    - **stream_in2:** Dato de entrada correspondiente al segundo resultado de la multiplicación Matriz-Vector, con formato Q21.16. En el caso particular del sumador de vectores de la *Candidate State*, este dato proviene del resultado del segundo producto Matriz-Vector combinado con el resultado de la *Reset Gate*, manteniendo el mismo formato Q21.16.
    - **nwords:** Tamaño del vector a sumar, que en esta implementación es 16.
  - **Salidas:**
    - **stream_out:** Dado que el resultado de una suma debe conservar el mismo formato que el operando con la mayor cantidad de bits en la parte entera y fraccionaria, el formato teórico de salida debería ser Q22.16. Sin embargo, debido a la función de activación que sigue a esta operación, se ha optado por utilizar el formato Q4.12. La justificación de esta elección se detalla en la sección correspondiente a las funciones de activación, pero en términos generales, este formato permite saturar la salida dentro de un rango específico.
- **Vector_adder_ROG** (Apéndice A.2.3.4): Esta variante de la función de suma de vectores se emplea en la función principal *Rest_of_GRU*, que se encarga de procesar las operaciones

restantes fuera de las puertas de la GRU. Como resultado, su configuración numérica es diferente.

- **Entradas:**
  - **stream_in1:** Dato de entrada proveniente de una multiplicación previa, con formato Q2.30.
  - **stream_in2:** Dato de entrada proveniente de otra multiplicación previa, con formato Q9.23.
  - **nwords:** Tamaño del vector a sumar, que en esta implementación es 16.
- **Salidas:**
  - **stream_out:** Considerando que el resultado de la suma debería adoptar el formato con la mayor cantidad de bits en la parte entera y fraccionaria, el formato teórico de salida sería Q10.30. No obstante, dado que estos datos representan la salida final de la GRU y serán realimentados en el sistema, se ha decidido que la salida adopte el mismo formato que la entrada, es decir, Q8.8.

Figura 4.11: Esquema función Vector_adder y Vector_product

Siguiendo la misma estructura de entradas y salidas mostradas en el esquema anterior, la multiplicación de vectores se implementa de manera análoga. A continuación, se detallan las variantes existentes y sus particularidades.

- **Vector_product** (Apéndice A.2.2.1): Esta función se emplea dentro de la puerta *Candidate State*, por lo que su nomenclatura y formato de datos son los indicados a continuación.
  - **Entradas:**
    - **stream_in1:** Dato de entrada correspondiente al segundo resultado de la multiplicación Matriz-Vector, con formato Q21.16.
    - **stream_in2:** Dato de entrada correspondiente al resultado de la puerta *Reset Gate*, con formato Q1.15.
    - **nwords:** Tamaño del vector a multiplicar, que en esta implementación es 16.
  - **Salidas:**

- **stream_out:** Al multiplicar un número en formato Q21.16 por otro en Q1.15, el resultado teórico inicial sería Q22.31, ya que la parte entera aumenta en un bit y la precisión fraccionaria se incrementa. No obstante, dado que el segundo operando tiene un rango entre 0 y 1 (como resultado de la función sigmoide), el valor máximo del producto nunca supera al del primer operando, lo que impide que se requiera un bit adicional en la parte entera. Como consecuencia, el formato correcto del resultado es Q21.31. Para optimizar el uso de recursos y evitar un crecimiento innecesario en la representación numérica, se ha optado por truncarlo a Q21.16, lo que no introduce una pérdida significativa de información. Además, esta decisión permite mantener coherencia en el formato de datos entre las entradas de la siguiente función.

Las próximas dos funciones son multiplicaciones de vectores, al igual que la anterior, pero estas se utilizan dentro de la función *Rest_of_GRU*. Como se describirá a continuación una de ellas tiene una peculiaridad en el tratamiento de una de sus entradas.

- **Vector_product_sub_cand** (Apéndice A.2.3.2): Esta función realiza la multiplicación del resultado de la resta por constante, que será descrita en el apartado siguiente y el resultado de la puerta *Candidate State*.
  - **Entradas:**
    - **stream_in1:** Dato de entrada correspondiente al resultado de la *Candidate State*, con formato Q1.15.
    - **stream_in2:** Dato de entrada correspondiente al resultado de la función resta Constante-Vector, con formato Q1.15.
    - **nwords:** Tamaño del vector a multiplicar, que en esta implementación es 16.
  - **Salidas:**
    - **stream_out:** Al realizar la multiplicación el resultado tiene un nuevo formato determinado por la suma de los bits de la parte entera y la parte fraccionaria de ambos operandos, siendo de este modo el resultado Q2.30.
- **Vector_product_up_out** (Apéndice A.2.3.3): Esta función realiza la multiplicación del resultado de la salida anterior Y_t y el resultado de la puerta *Update Gate*.
  - **Entradas:**
    - **stream_in1:** Dato de entrada correspondiente a la salida anterior , Y_t, con formato Q8.8. Cabe realizar una mención al tratado de esta entrada ya que para poder realizar la implementación *recurrent-bias-after-multiplication* se debe añadir un '1' al vector, consiguiendo de este modo la correcta multiplicación por las matrices. Por lo tanto, como se ha descrito en el apartado de las funciones top, esta entrada se ha duplicado 4 veces, 3 para las puertas y la restante para esta entrada. Por este motivo, el primer valor que llega a esta función debe ser despreciado, eliminando de este modo el '1' correspondiente a la consideración de los sesgos en la multiplicación Matriz-Vector. La solución a esto se puede observar en el apéndice A.2.3.3, haciendo uso de la variable *garbage*.
    - **stream_in2:** Dato de entrada correspondiente al resultado de la *Update Gate*, con formato Q1.15.

- **nwords:** Tamaño del vector a multiplicar, que en esta implementación es 16.

- **Salidas:**
  - **stream_out:** Al realizar la multiplicación el resultado tiene un nuevo formato determinado por la suma de los bits de la parte entera y la parte fraccionaria de ambos operandos, siendo de este modo el resultado Q9.23.

4.1.1.4. Resta constante - vector

Esta función forma parte de la función *Rest_of_GRU* y tiene como objetivo realizar la operación de resta entre una constante y un vector de entrada. Dicha operación es fundamental en el procesamiento interno de la GRU, ya que permite obtener la componente complementaria de ciertos valores clave en la arquitectura.

El esquema correspondiente, mostrado en la Figura 4.12, ilustra las entradas y salidas de la función y su código se puede encontrar en el Apéndice A.2.3.1.

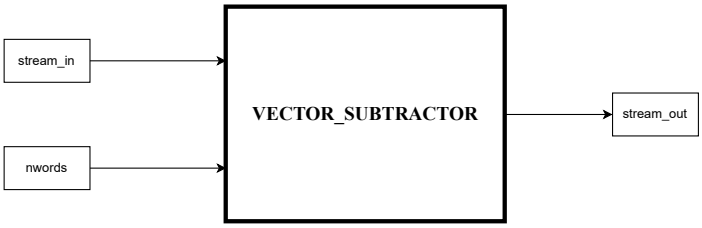

Figura 4.12: Esquema función *Vector_subtractor*

La operación implementada consiste en restar cada elemento del vector de entrada a una constante cuyo valor es 1. No obstante, el formato numérico de la entrada no permitía representar este valor de manera directa, ya que no disponía de suficiente rango en la parte entera. Para solucionar esto, se ha utilizado un formato numérico con un bit adicional en la parte entera `data_uno`, permitiendo representar correctamente el valor de la constante sin pérdida de precisión. A continuación, se detallan los formatos numéricos de las entradas y salidas de la función.

- **ENTRADAS:**
  - **stream_in:** El dato de entrada proviene de la salida de la *Update Gate*, cuyo formato numérico es Q1.15.
  - **nwords:** Indica la dimensión del vector a restar. En este caso, el vector es de tamaño 16 por lo que nwords = 16.
- **SALIDAS:**
  - **stream_out:** La salida de esta operación se obtiene al restar un número en formato Q2.14 (`data_uno`), cuyo valor es siempre 1, y un número en formato Q1.15 (vector de

entrada), cuyo rango es de 0 a 1 debido a la naturaleza de la función sigmoide. Teóricamente, el formato de salida se determina tomando la mayor parte entera y el mayor número de bits fraccionales entre ambos operandos, lo que resultaría en Q2.15. Sin embargo, dado que el resultado de la resta nunca supera 1, la parte entera no se incrementa y un solo bit es suficiente para representarla. Además, 15 bits en la parte fraccionaria garantizan la precisión sin desperdiciar recursos. Por ello, el formato óptimo utilizado para la salida es Q1.15, lo que permite optimizar el uso de *hardware* sin pérdida de información.

4.1.1.5. Funciones de activación

Las funciones de activación juegan un papel clave en la transformación de los valores intermedios y en la regulación de la información dentro de la red, determinando qué datos se conservan o se actualizan en la memoria. En la Figura 4.13 se presenta el esquema general de su implementación. Como se detalla en los Apéndices A.2.1.3 y A.2.2.2, ambas funciones comparten la misma estructura de código, diferenciándose únicamente en la ecuación aplicada. Por ello, la explicación se centrará en una de ellas. A continuación, se describen las variables involucradas en estas funciones:

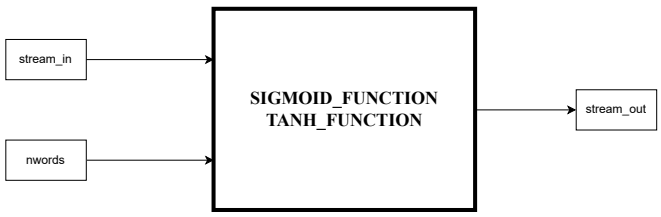

Figura 4.13: Esquema funciones de activación

- **ENTRADAS:**
  - **stream_in:** Para ambas funciones el dato de entrada proviene de la suma de vectores cuyo formato es Q4.12, posteriormente se explicará el porque de este.
  - **nwords:** Indica la dimensión del vector recibido a través de stream_in. En este caso, el vector tiene un tamaño de 16, por lo que nwords = 16. Este valor se implementará mediante un #define en el header del código.
- **SALIDAS:**
  - **stream_out:** La salida de estas funciones sigue el formato Q1.15. En el caso de la sigmoide, este formato es suficiente para representar su rango de valores comprendidos entre 0 y 1. Para la tangente hiperbólica, cuyos valores oscilan entre -1 y 1, el formato Q1.15 también resulta adecuado, salvo por la imposibilidad de representar exactamente el valor 1. No obstante, dado que este valor es poco frecuente, se ha considerado que la precisión proporcionada por 15 bits fraccionales (2^{15}) es suficiente.

Explicación de su implementación

Para la implementación de la función de activación, se han desarrollado tres funciones: una función principal (top) y dos funciones auxiliares. Tanto para la función de activación sigmoide como para la tangente hiperbólica (tanh), el código fuente es el mismo, con la única diferencia en la fórmula utilizada para calcular el valor correspondiente a cada función. En la explicación, se toma como referencia la implementación de la función sigmoide para describir el funcionamiento general del sistema.

init_sigmoid_lut

Esta función se encarga de inicializar los valores de la función sigmoide, los cuales serán almacenados en una memoria interna ROM. Para representar adecuadamente los valores de entrada, se ha seleccionado el formato de punto fijo Q4.9. Esta elección se debe a que el rango de valores para los cuales la función sigmoide se evalúa está acotado entre $[-8, 8]$, por lo que no es necesario utilizar una mayor cantidad de bits enteros.

Además, la función sigmoide se estabiliza en sus extremos, lo que implica que no se requiere una precisión adicional en estos valores. De este modo, el formato Q4.9 proporciona un equilibrio adecuado entre precisión y eficiencia en *hardware*.

```
// Valores de saturación
const float SIGMOID_MIN = -8.0f;
const float SIGMOID_MAX = 8.0f;
```

La resolución con la que se discretiza este rango de valores está determinada por el paso de cuantización (`step_SIG`). Este valor se obtiene dividiendo la amplitud total del rango entre el número de posiciones disponibles en la memoria ROM (`MEM_SIZE_SIG`):

```
const float step_SIG = (SIGMOID_MAX - SIGMOID_MIN)/MEM_SIZE_SIG;
```

Para almacenar los valores en la memoria ROM de manera eficiente, se utiliza un bucle que recorre todas las posibles entradas dentro del rango definido. A partir de cada valor de entrada x, se determina la dirección correspondiente en la memoria ROM. Esto permite que, durante la fase de inferencia, la búsqueda de valores en la ROM no requiera operaciones adicionales, lo que optimiza el acceso.

El proceso de precálculo se muestra a continuación:

```
for (int i = 0; i < MEM_SIZE_SIG; i++) {
    x = SIGMOID_MIN + step_SIG * i;
    float y =  1.0f / (1.0f + exp(-x));
    data_in_Q valor_rom = x;
    memory_index rom_index = valor_rom.range(SIZE_sig_mem-1,0);
    memory[rom_index] = data_out_sig (y);
}
```

En este fragmento de código, podemos observar cómo:

1. Se calcula la entrada x a partir del índice de iteración y el paso de cuantización.

2. Se obtiene el valor de la sigmoide y mediante la ecuación función de activación correspondiente.

3. Se convierte x al tipo `data_in_Q`, cuyo formato es Q4.9, para garantizar la correcta indexación en la ROM.

4. Se convierten los 13 bits de `data_in_Q` utilizando `.range(SIZE_sig_mem-1,0)`, convirtiéndolo al tipo `memory_index` (`ap_uint` de 13 bits). El uso de `.range()` es esencial en esta implementación, ya que, en caso contrario, solo se consideraría la parte entera, lo que limitaría la ROM a solo 2^4 posiciones de memoria. Esto provocaría un muestreo deficiente de la función sigmoide, generando errores significativos en la precisión de los valores almacenados.

5. El resultado y se convierte al formato de salida mediante un cast y se almacena en la memoria ROM en la posición indicada por `rom_index`.

Para facilitar la comprensión del funcionamiento de esta función y su tratamiento de los tipos de datos, se presenta un esquema resumen en la Figura 4.14. Este resumen implementa la función de activación sigmoide, mediante un valor ejemplo de entrada se realizada el proceso de guardado de su salida en la memoria. Este proceso se repite para todos lo valores en el rango indicado ([-8,8]).

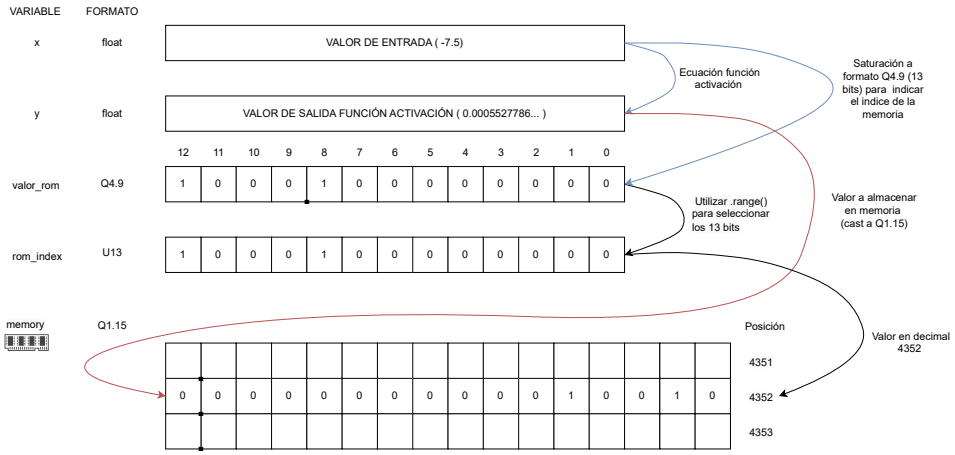

Figura 4.14: Ejemplo almacenamiento en memoria

sigmoid_compute

Esta función es la encargada de realizar la búsqueda del indice de la memoria adecuada para la entrada recibida. Esta función sigue la misma metodología de búsqueda que la función de inicialización de la memoria. Por ello, se ha querido realizar una ilustración de su implementación mediante un ejemplo que se puede observar en la Figura 4.15.

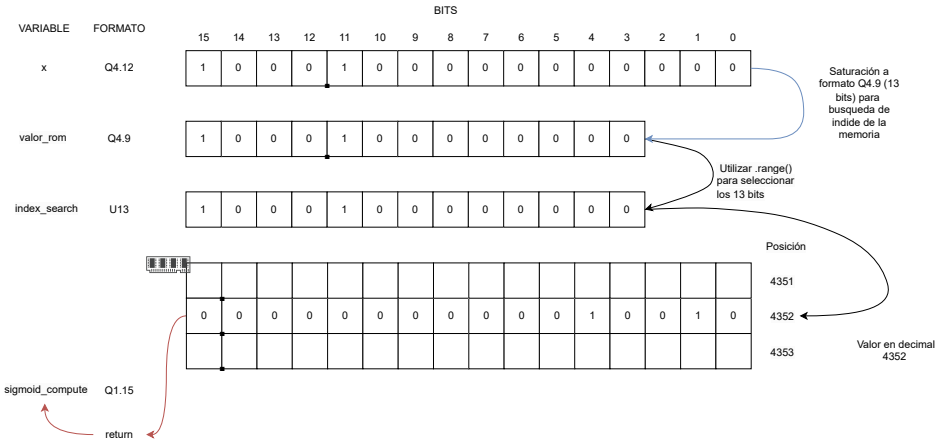

Figura 4.15: Ejemplo búsqueda en memoria

Como se puede observar los pasos que sigue esta función son:

1. Recepción del valor de entrada x en el formato Q4.9 (`data_in_Q`), para garantizar la correcta búsqueda en la memoria.

2. Se convierten los 13 bits de `data_in_Q` utilizando `.range(SIZE_sig_mem-1,0)`, convirtiéndolo al tipo `memory_index` (`ap_uint` de 13 bits). El uso de `.range()` es esencial en esta implementación, ya que, en caso contrario, solo se consideraría la parte entera, lo que limitaría la ROM a solo 2^4 posiciones de memoria. Esto provocaría que solo se pudieran asignar los valores indicados por los 4 bits enteros, es decir, las posiciones de memoria de la 0 a la 16.

   Dado que el dato de entrada está en formato Q4.9, con un total de 13 bits, y la memoria ROM tiene un tamaño de direccionamiento de 13 bits, la saturación se gestiona de manera natural. Esto significa que los 13 bits del dato se utilizan directamente como posición de memoria, garantizando que cada posible entrada dentro del rango definido tenga una dirección única sin necesidad de ajustes adicionales.

3. Esta función devuelve mediante el `return` el valor encontrado en la memoria en la posición indicado por la variable `index_search` (tipo de dato `memory_index`).

sigmoid_function

Esta función actúa como la función principal (TOP) de las tres, encargándose de recibir el dato de entrada y devolver el dato de salida. Primero, se lee el dato de entrada y se realiza una conversión a un formato Q4.9 (tipo de dato data_in_Q), que es el formato adecuado para la búsqueda del índice en la memoria ROM. Tras esta conversión, se invoca a la función sigmoid_compute, la cual, como se explicó anteriormente, busca el valor correspondiente en la memoria ROM. Finalmente, el valor obtenido es enviado a través del streaming de salida.

4.2. Comparación de resultados

En este apartado se presenta un análisis comparativo de las distintas implementaciones de la GRU evaluadas en este trabajo: float, half, ap_fixed<18,8> y la implementación cuantificada paso a paso. El objetivo es determinar cuál de estas configuraciones ofrece el mejor equilibrio entre precisión numérica, eficiencia en el uso de recursos *hardware* y cumplimiento de las restricciones de latencia.

Para ello, se analizarán tres aspectos clave:

- Precisión numérica, evaluando el error en los resultados obtenidos.
- Consumo de recursos, considerando el uso de LUTs, FFs, DSPs y BRAMs en la FPGA.
- Latencia, dado que, como se indicó en apartados anteriores, esta no puede superar los 150 ciclos.

Estos parámetros permitirán cuantificar el impacto de cada implementación en términos de exactitud, coste computacional y viabilidad temporal. A continuación, se presentan estos datos en formato de tablas, indicando para cada implementación su viabilidad en función de los criterios mencionados.

	Recursos				
	Latency (cycles)*	BRAM	DSP	FF	LUTs
Disponibles (Tabla 3.1)	-	140	220	106400	53200
float	553	91	274	55339	75844
half	596	49	220	43959	38548
ap_fixed<18,8>	415	43	63	28814	41627
Cuantificación de crecimiento natural	390	65	64	17815	29263

Tabla 4.1: Recursos utilizados en las implementaciones

(*) La latencia indicada corresponde a un valor estimado obtenido tras la simulación. Los valores reportados son elevados debido a la carga inicial de las matrices, lo que provoca que la latencia reflejada en el informe de Vitis corresponda a la máxima posible. Sin embargo, en iteraciones sucesivas, la latencia disminuirá drásticamente, ya que las matrices ya habrán sido cargadas en memoria.

	Precisión	
N	Todos los formatos	Formatos inferiores a ap_fixed<13,4> (Ejemplo: ap_fixed<11,4>)
0	0.0003	0.0036
1	0.0017	0.0017
2	0.0018	0.0018
3	0.0002	0.0041
4	0.0015	0.0015
5	0.0008	0.0070
6	0.0019	0.0097
7	0.0013	0.0013
8	0.0015	0.0063
9	0.0015	0.0015
10	0.0016	0.0023
11	0.0014	0.0014
12	0.0008	0.0008
13	0.0003	0.0036
14	0.0004	0.0004
15	0.0009	0.0107

Tabla 4.2: Precisiones de cada implementación

Los datos obtenidos permiten realizar un análisis comparativo de las distintas implementaciones, considerando los criterios previamente definidos de precisión, consumo de recursos y latencia.

En primer lugar, las implementaciones en float y half quedan descartadas debido a que su latencia es significativamente superior al resto de implementaciones. Además, en el caso de float, se excede el número de DSPs disponibles en la FPGA, y en el caso de half, la cantidad de DSPs utilizados es exactamente igual a los disponibles. Dado que en la RedPitaya no solo se implementará la GRU, sino también DMAs y otros módulos esenciales, estas opciones quedan excluidas debido a su alto consumo de recursos.

Por otro lado, las implementaciones en ap_fixed<18,8> y cuantificación de crecimiento natural cumplen con los requisitos de latencia, ya que, como se observa en la Tabla 4.3, tras la primera iteración, la latencia efectiva se reduce a 149 ciclos para ap_fixed<18,8> y 131 ciclos para cuantificación de crecimiento natural, ambas dentro del margen permitido. Como se ha comentado anteriormente, a pesar de que la latencia inicial es mayor, esta disminuye en las iteraciones posteriores, ya que la carga de datos en memoria solo ocurre en la primera ejecución.

En cuanto a la precisión obtenida, los valores presentados en la Tabla 4.2 muestran que todas las implementaciones utilizando los formatos float, half, ap_fixed<18,8> y cuantificación de crecimiento natural tienen una precisión similar. Esto se debe a un cuello de botella originado por las funciones de activación, cuya implementación mediante memorias ROM precalculadas limita la cantidad de valores que se pueden almacenar. En particular, la memoria ROM utilizada para la evaluación de la función de activación, como se detalló en apartados anteriores, opera con 13 bits, lo que restringe la entrada a un formato de precisión fija: ap_fixed<13,4>. Esto significa que todas las implementaciones están limitadas a la precisión definida por este formato.

Sin embargo, cuando la precisión previa a las funciones de activación es inferior, como se observa en la segunda columna de la tabla (precisión con formato ap_fixed<11,4>), se aprecia una degradación significativa en algunos valores debido a la imprecisión en el direccionamiento de la memoria. Esto ocurre porque los 2 bits menos significativos (LSB) están en '0', lo que hace que solo puedan ser seleccionadas 2^{11} direcciones en lugar de 2^{13}. Como resultado, se produce una pérdida de precisión en algunos de los resultados.

Latencias Co-Simulación

	Latency	
	MAX	MIN
float	613	330
half	662	377
ap_fixed<18,8>	439	149
Cuantificación de crecimiento natural	414	131

Tabla 4.3: Resultados Latencia Co-Simulación

Por lo tanto, analizando el consumo de *hardware*, ambas opciones se mantienen dentro de los límites de la FPGA. No obstante, la implementación basada en cuantificación de crecimiento natural presenta ventajas significativas, ya que reduce la latencia final (131 ciclos frente a 149 ciclos de ap_fixed<18,8>) y minimiza el uso de FFs y LUTs, facilitando la integración de otros módulos en el diseño.

Dado que ambas opciones cumplen con los requisitos funcionales y de recursos, pero la cuantificación de crecimiento natural ofrece una menor latencia y un uso más eficiente del *hardware*, se selecciona esta opción como la implementación más adecuada para el sistema final. Con esta elección, se garantiza un equilibrio óptimo entre precisión, latencia y consumo de recursos, permitiendo su integración en la FPGA sin comprometer el rendimiento general del sistema.

4.3. Integración de la IP

La estructura del diseño se presenta en el Block Design (Figura 4.16), el cual se basa en un proyecto preexistente de otra implementación similar a la de este trabajo. En ella se incluía parte de los bloques fundamentales necesarios para la implementación. Sin embargo, se ha llevado a cabo una reestructuración del diseño para adaptarlo a los requerimientos específicos de la unidad recurrente GRU. Esta reorganización ha implicado modificaciones en la interconexión de los bloques, ajustes en la gestión de datos y la implementación de una estrategia de almacenamiento y acceso eficiente para los pesos y sesgos de la red neuronal.

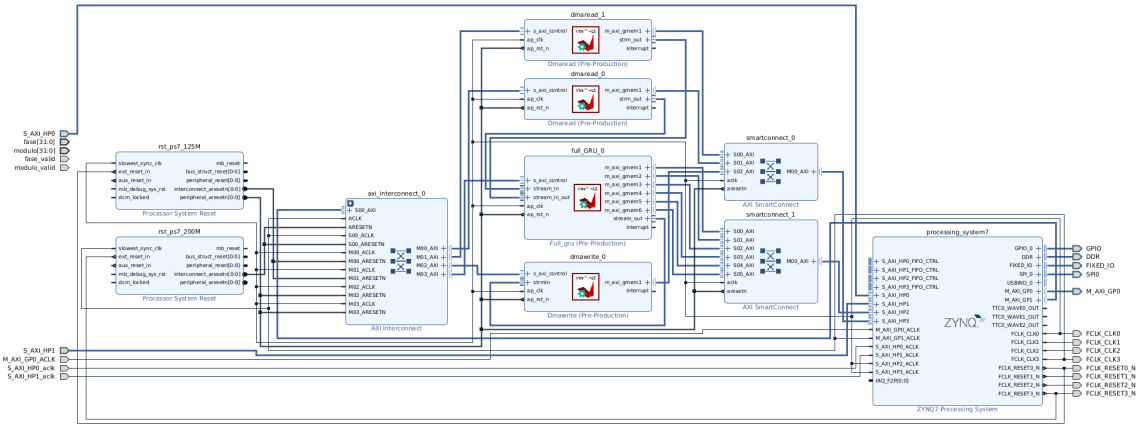

Figura 4.16: Block Design GRU

El sistema se basa en un procesador ARMCortex-A9 embebido en la FPGA, que se comunica con los distintos módulos a través de interfaces AXI. Para la transferencia eficiente de los datos de entrada y salida, se han incorporado dos módulos DMA encargados de la lectura y escritura entre la memoria externa y los módulos de procesamiento. Estas IPs, proporcionadas con el proyecto original, han sido estudiadas para comprender su funcionamiento y modificadas para la implementación correcta de la GRU. En particular, el diseño cuenta con dos módulos de DMA de lectura: uno encargado de leer los valores de entrada y otro de recuperar los valores de salida previos. Estos valores de salida son previamente almacenados en memoria por un módulo DMA de escritura, situado a la salida de la GRU.

El módulo central de la implementación es la IP de la GRU, la cual recibe los datos de entrada mediante una interfaz AXI-Stream y produce la salida correspondiente tras ejecutar las operaciones de multiplicación de matrices y aplicar las funciones de activación. Debido a las restricciones impuestas por la IP SmartConnect, la cual requiere un ancho mínimo de 32 bits para la transferencia de datos, se ha adoptado una estrategia de empaquetamiento para los pesos y sesgos. En particular, estos valores se almacenan en los 16 bits menos significativos (LSB) de una variable de 32 bits.

A nivel de infraestructura del diseño, se han utilizado varios bloques fundamentales para la interconexión y el control del sistema:

- AXI SmartConnect: Este bloque actúa como un interconector avanzado que permite gestionar múltiples interfaces AXI de manera eficiente. Su función principal es la de enrutamiento y conmutación de datos, asegurando que la comunicación entre los módulos sea óptima y sin congestión. Además, debido a sus restricciones en cuanto al tamaño mínimo de transferencia de datos (32 bits), se ha implementado la estrategia previamente mencionada para empaquetar los pesos y sesgos en los 16 bits menos significativos de cada palabra de datos. Dentro de la IP de la GRU, se aplican máscaras para extraer estos valores antes de su utilización en las operaciones matemáticas, como se ilustra en el siguiente fragmento de código correspondiente a la multiplicación matriz-vector:

```
if (!first_iteration) {
    for (int i = 0; i < NWORDS_M1; i++) {
        mascara = *din1++ & 0xFFFF;
        MATRIX_A[i] = *reinterpret_cast<data_in_mvm*>(&mascara);
```

```
        }

        for (int i = 0; i < NWORDS_M2; i++) {
            mascara = *din2++ & 0xFFFF;
            MATRIX_B[i] = *reinterpret_cast<data_in_mvm*>(&mascara);
        }
        first_iteration = true;
    }
```

- AXI Interconnect: Este bloque se encarga de facilitar la comunicación entre el procesador y los periféricos mediante la interfaz AXI. Esta IP está destinada únicamente a transferencias asignadas a memoria, las transferencias AXI4-Stream no son aplicables. A diferencia del SmartConnect, su arquitectura es más sencilla y estática, lo que lo hace adecuado para conectar periféricos esenciales al sistema sin una gestión compleja del tráfico de datos. En este diseño, se ha utilizado para interconectar los módulos importados de Vitis HLS con el procesador mediante el M_AXI_GP1, garantizando la correcta transmisión de información.
- rst_ps7 (Processor System Reset): Este bloque se encarga de generar y distribuir señales de reinicio para los distintos componentes del sistema ZYNQ. Su función es esencial para garantizar que los módulos se inicialicen correctamente y en el orden adecuado. Se han configurado distintas versiones del rst_ps7 para sincronizar el reinicio de los módulos en función de los relojes utilizados en el diseño.

La integración de estos módulos permite una ejecución eficiente del procesamiento en *hardware*, optimizando tanto el uso de memoria como el rendimiento computacional.

Para complementar esta descripción, se incluyen imágenes del editor de direcciones (Figura 4.18) y del mapa de direcciones (Figura 4.17), que ilustran la organización y asignación de los recursos en el diseño del sistema. Las direcciones de las IPs y sus rangos no han sido modificados, sino que han sido asignados automáticamente por la herramienta Vivado.

La correcta configuración de las direcciones es un aspecto crucial, ya que, en implementaciones previas donde se realizaban las conexiones de manera manual, se presentaban casos en los que no se asignaban direcciones a las IPs. Para evitar este problema, se empleó la herramienta *Run Connection Automation* (Apartado 3.2.2.1), la cual proporciona un abanico de posibles conexiones restantes por completar para cada señal. La ejecución de esta herramienta asegura que las interfaces AXI control recibieran su correspondiente dirección. Esto evitaba inconsistencias al exportar el diseño para su validación física, donde en pruebas previas se detectaba la ausencia de ciertos módulos, como si no existieran dentro del sistema.

Figura 4.17: Address Map

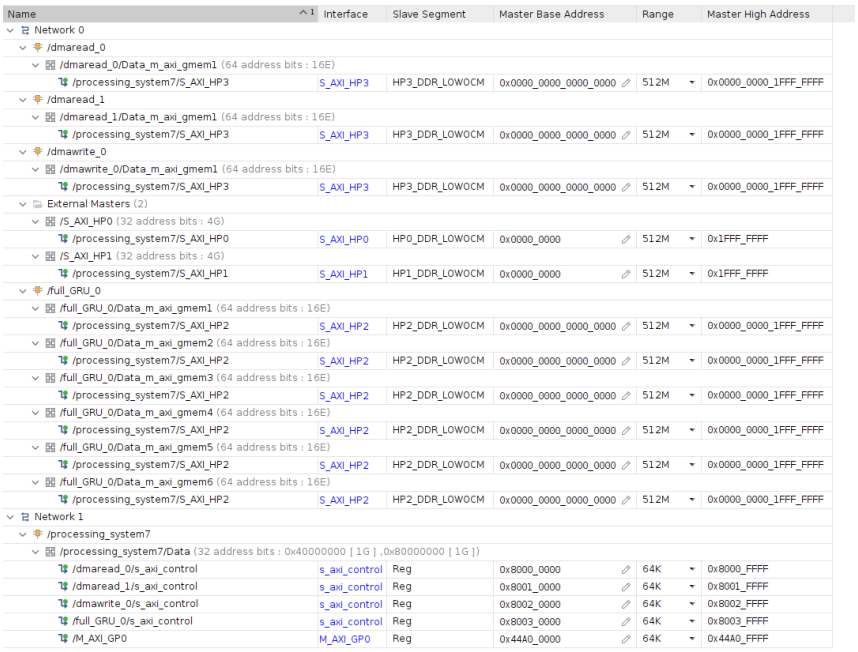

Figura 4.18: Address Editor

Una vez completada la implementación, se procede a validar el diseño haciendo clic en Validate Design (icono ✓) en la parte superior del block design o utilizando el atajo F6. Luego, se genera el bitstream, que representa la configuración física de la FPGA y se utiliza para programar el dispositivo, asegurando que todos los módulos, direcciones y conexiones estén correctamente configurados y optimizados. Finalmente, se exporta el archivo XSA (File → Export → Export Hardware), al exportar este archivo se debe indicar que se incluya el `bitstream` ya que es necesario para su integración en la herramienta de software Vitis IDE.

4.4. Implementación y resultados sobre RedPitaya

En Vitis IDE, se llevará a cabo la validación física del sistema en la plataforma RedPitaya, verificando su funcionamiento en un entorno real y asegurando que todas sus funcionalidades operen correctamente en *hardware*. Para ello, se sigue el flujo de trabajo descrito en los apartados 3.2.3.1 y 3.2.3.2, proporcionando a continuación una explicación detallada con imágenes de cada paso.

1. Para iniciar Vitis IDE, se puede ejecutar el comando `vitis` desde el terminal o acceder desde Vivado en *Tools* → *Launch Vitis IDE*. Una vez abierto, se debe seleccionar el *Workspace*, que será el directorio donde se almacenarán los datos del proyecto.

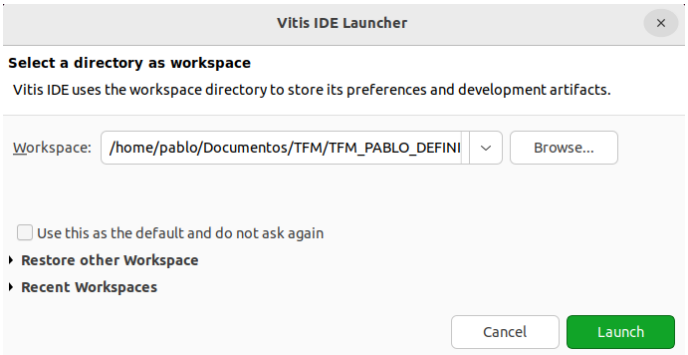

Figura 4.19: Selección del Workspace

2. La pantalla de bienvenida de Vitis IDE es la mostrada en la Figura 4.20. Para realizar la validación en RedPitaya, es necesario generar la plataforma y la aplicación.

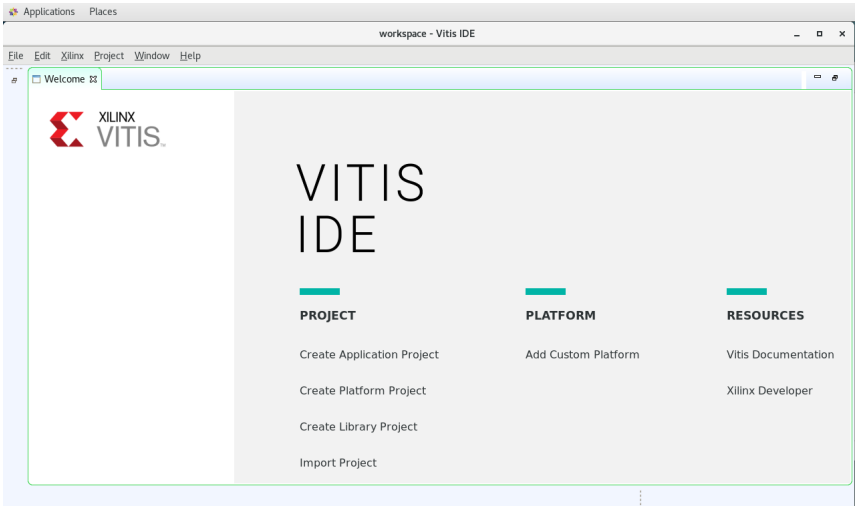

Figura 4.20: Página de bienvenida de Vitis IDE

Existen dos métodos de creación:

- **Create Platform Project + Create Application Project**: La plataforma y la aplicación se generan en procesos separados.

- **Create Application Project**: Permite seleccionar o crear una plataforma dentro del mismo proceso antes de desarrollar la aplicación.

Para este proyecto, se ha optado por la segunda opción, generando la plataforma y la aplicación en un solo proceso. A continuación, se detallan los pasos seguidos para su creación.

3. Al hacer clic en *Create Application Project*, se abrirá una ventana para elegir la plataforma entre las disponibles en el repositorio o crear una nueva. En este caso, se ha creado una nueva plataforma basada en el *hardware* exportado desde Vivado (archivo XSA).

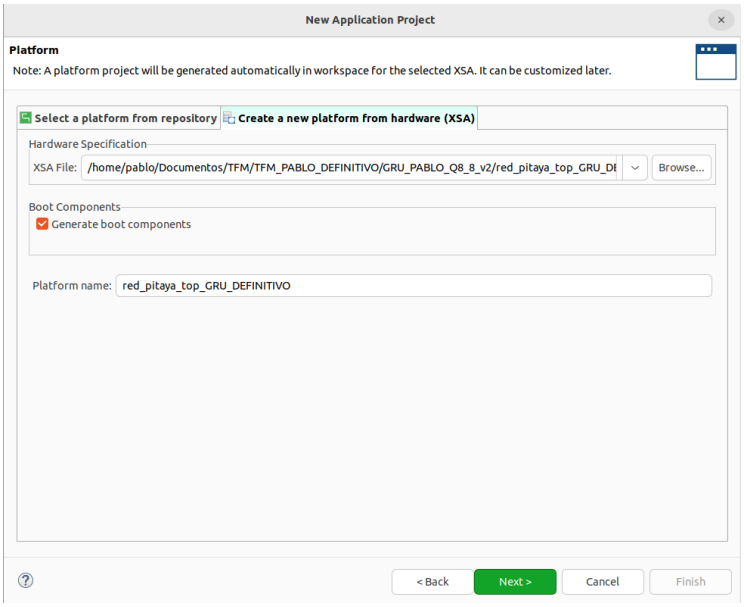

Figura 4.21: Creación de la plataforma con archivo XSA

4. Luego, al crear la aplicación, se debe seleccionar `ps7_cortex9_SMP`, ya que esta opción permite utilizar ambos núcleos en modo SMP, optimizando la distribución de tareas en Linux.

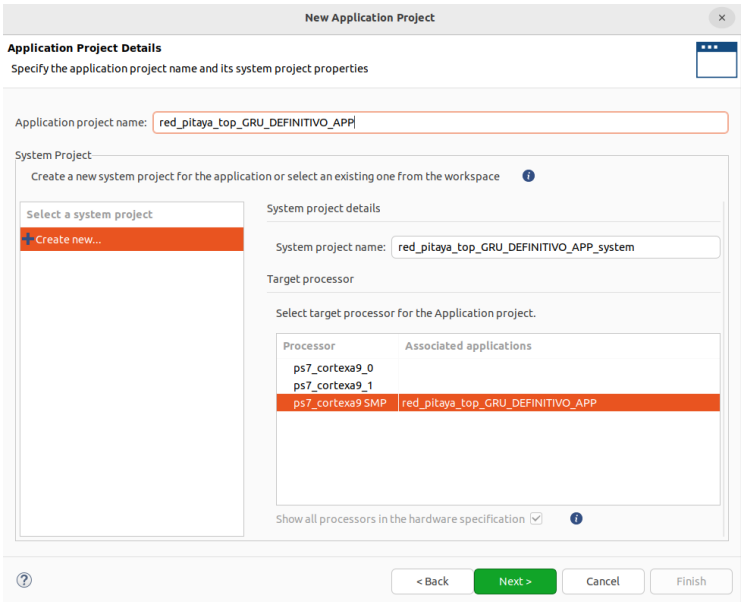

Figura 4.22: Creación de la aplicación

5. Se debe elegir el dominio, manteniendo los valores predeterminados y asegurando que el sistema operativo sea Linux.

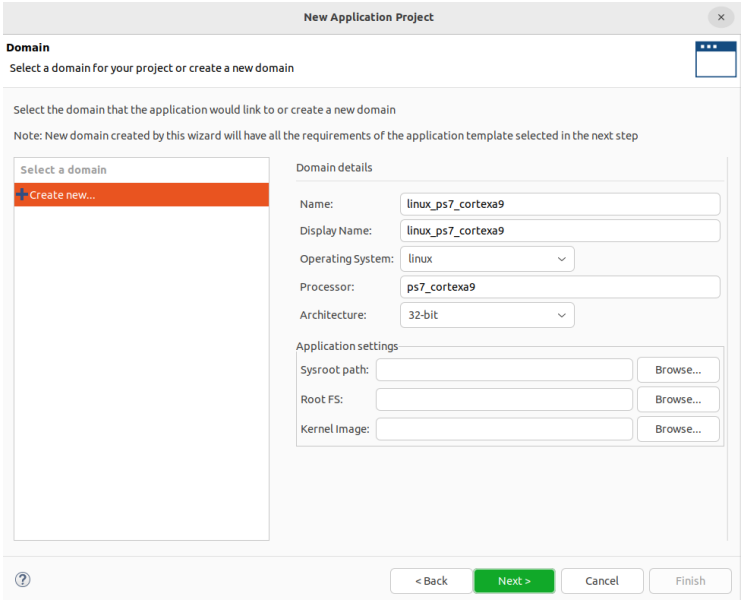

Figura 4.23: Selección del dominio

6. Finalmente, se elige la plantilla de la aplicación, seleccionando *Linux Empty Application*.

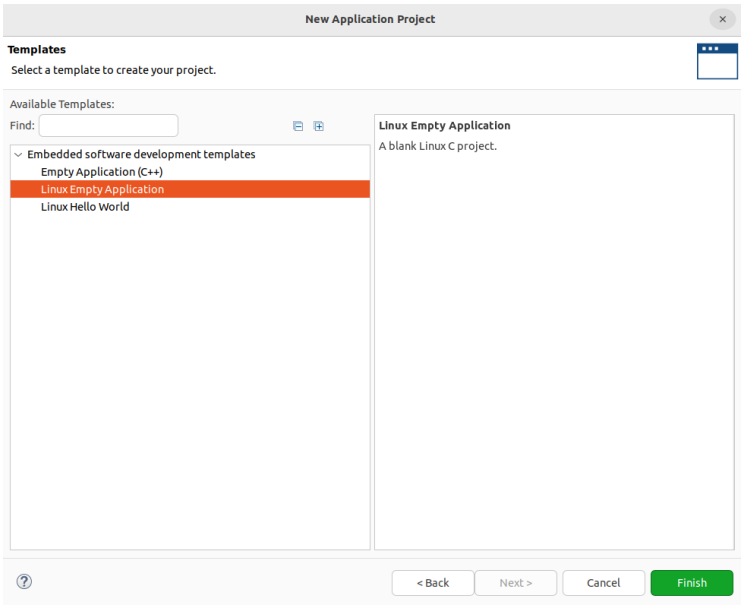

Figura 4.24: Selección de la plantilla

Una vez finalizado el proceso de creación del proyecto, plataforma y aplicación, la pestaña *Explorer* de Vitis IDE tendrá el aspecto mostrado en la Figura 4.25, donde se observan los componentes de la plataforma (ícono verde) y la aplicación (ícono azul).

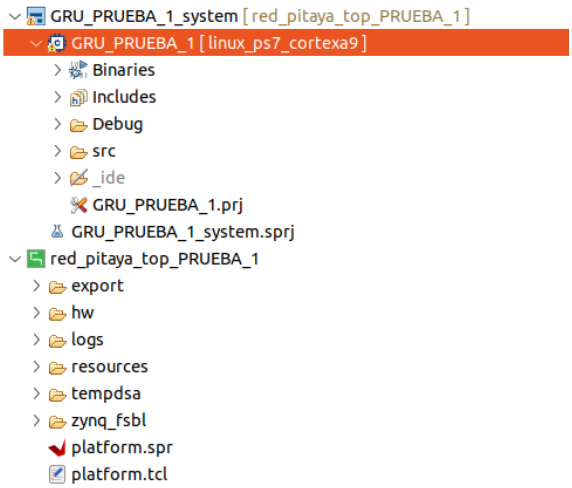

Figura 4.25: Pestaña Explorer

Tras haber generado el proyecto junto a este se han generado varios archivos que serán de vital importancia para la correcta ejecución e implementación sobre la Redpitaya. Estos archivos son:

- `xparameters.h`: Es un archivo generado automáticamente por Vitis que contiene macros con las direcciones base, configuraciones y parámetros de las IPs y periféricos del diseño. Facilita el acceso a los recursos de *hardware* en la aplicación embebida, evitando la necesidad de definir manualmente direcciones de memoria y configuraciones.

- Archivos generados para las IPs: Estos archivos se encuentran en la carpeta drivers y contienen las definiciones y funciones necesarias para interactuar con los periféricos personalizados dentro de la plataforma embebida. Existe una carpeta para cada una de las IPs. En mi caso al ser una aplicación sobre Linux se encuentran los siguientes archivos:
  - <nombre_ip>_hw.h: Contiene las direcciones de registro y macros específicas del *hardware* de la IP, permitiendo el acceso directo a los registros de control y estado.
  - <nombre_ip>_linux.c: Contiene funciones específicas para la integración de la IP en un sistema operativo Linux. Normalmente, incluye implementaciones compatibles con controladores de Linux para facilitar el acceso a la IP desde el espacio de usuario o el kernel.
  - <nombre_ip>_sinit.c: Se encarga de la inicialización estática de la IP en sistemas sin sistema operativo (bare-metal). Define una estructura con la configuración predeterminada de la IP y funciones para su inicialización.
  - <nombre_ip>.h: Define funciones de alto nivel para interactuar con la IP, facilitando su uso en aplicaciones C/C++.
  - <nombre_ip>.c: Implementa las funciones declaradas en el .h, proporcionando una capa de abstracción para la comunicación con la IP.

Para garantizar una correcta compilación del proyecto, es necesario incluir en el *include* de la aplicación los archivos generados en la carpeta *drivers* y los directorios donde se encuentran el archivo `xparameters.h` y las bibliotecas estándares de C. Sin su inclusión, la compilación podría fallar debido a referencias no resueltas a los periféricos o funciones de inicialización. Para realizar esta inclusión en el proyecto se deben seguir los siguientes pasos:

1. Haz clic derecho sobre la aplicación remarcada en la pestaña Explorer (Figura 4.25).
2. Selecciona *C/C++ Build Settings*.
3. En la sección *C/C++ General*, elige *Paths and Symbols* y luego la pestaña *Includes*.
4. Usa el botón *Add* para agregar las rutas de los archivos necesarios. El resultado final debería verse como en la Figura 4.26.

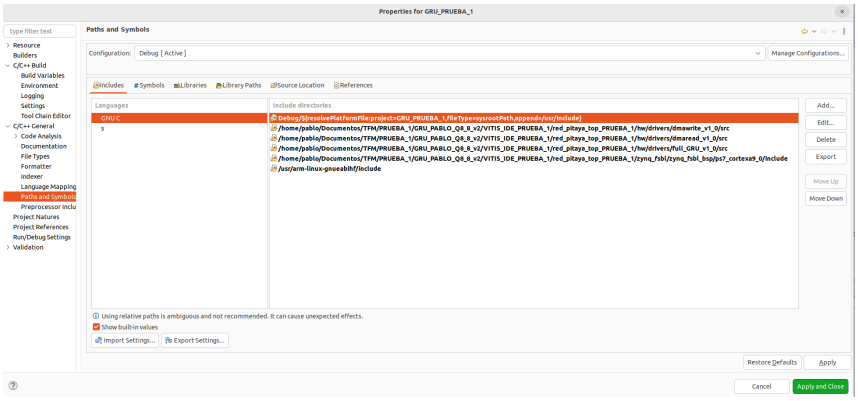

Figura 4.26: Includes añadidos

Para garantizar la correcta vinculación de las funciones matemáticas incluidas en `math.h`, es necesario agregar la biblioteca matemática `libm` en la configuración del enlazador. Esta inclusión se debe a que en el código se hace uso de la función `expf()`, la cual forma parte de esta biblioteca y no está incluida por defecto en la biblioteca estándar de C.

Sin la inclusión de `-lm`, el compilador reconoce la declaración de `expf()` gracias a `math.h`, pero el enlazador no encuentra su implementación, lo que genera un error de referencia no definida: `undefined reference to expf`. Para solucionar este problema, se debe agregar `m` en la sección Libraries (-l) dentro de la configuración de compilación de Vitis, lo que indicará al enlazador que incluya la biblioteca matemática.

Para realizar esta inclusión, sigue los siguientes pasos:

1. Haz clic derecho sobre la aplicación remarcada en la pestaña Explorer (Figura 4.25).
2. En la sección *C/C++ General*, elige *Settings* y luego la carpeta *Libraries*.
3. Usa el icono de la hoja con el símbolo + para agregar la biblioteca `m`. El resultado final debería verse como en la Figura 4.27.

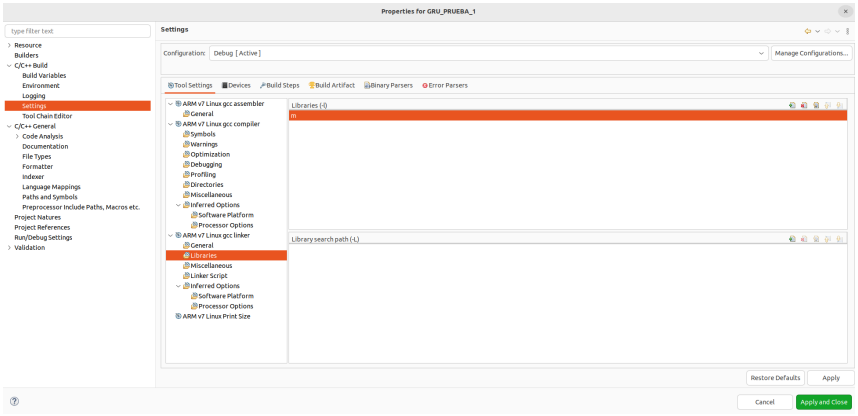

Figura 4.27: Adición de la biblioteca matemática para funciones de `math.h`

4.4.1. Código

En este apartado se detallan las secciones más relevantes del código implementado en la aplicación. Se analizarán los módulos principales, destacando su funcionalidad y su interacción con el *hardware*. Se incluirá una descripción de las funciones clave utilizadas para la comunicación con las IPs, la gestión de datos y la integración con el sistema operativo. Además, se explicará el flujo de ejecución del programa.

4.4.1.1. Gestión de dispositivos

La gestión de dispositivos se ha integrado dentro de la aplicación a través de los archivos `device.c` y `device.h`, disponibles en el Apéndice B.2. Estos archivos, fueron suministrados con

el proyecto y tras realizar un análisis de su funcionamiento, permiten una comunicación simplificada con los periféricos mediante memoria mapeada, facilitando operaciones básicas de apertura, cierre, lectura y escritura sobre los dispositivos sin necesidad de controladores complejos o acceso directo al kernel de Linux. A diferencia de los archivos `<nombre_ip>_sinit.c` y `<nombre_ip>_linux.c`, que están diseñados específicamente para cada IP e incluyen configuraciones avanzadas e integración con controladores del sistema operativo, `device.c` y `device.h` ofrecen una solución genérica y adaptable a distintos periféricos sin requerir modificaciones significativas.

El archivo `device.h` define la estructura de datos ***uio_device***, que encapsula la información del dispositivo, incluyendo su descriptor de archivo (fd), la dirección base mapeada en memoria (mp), el tamaño del mapa de memoria (mapsize), y nombres identificativos (name y alias). Además, declara las funciones necesarias para interactuar con el *hardware*:

- **`open_device()`:** Abre el dispositivo, mapea la memoria y lo prepara para su uso.
- **`close_device()`**: Cierra el dispositivo y libera los recursos asociados.
- **`write_device()` y `write_device_float()`:** Permiten escribir valores enteros y en punto flotante en registros específicos de la IP.
- **`read_device()` y `read_device_float()`:** Recuperan valores enteros o en punto flotante desde registros mapeados en memoria.

El archivo `device.c` implementa estas funciones utilizando llamadas al sistema como `open()`, `mmap64()` y `munmap()` para gestionar la memoria compartida con las IPs.La función `open()` permite abrir el dispositivo, obteniendo un descriptor de archivo que luego se utiliza para realizar operaciones sobre la memoria mapeada, mientras que `munmap()` se emplea para desasociar un área de memoria previamente mapeada, liberando los recursos de manera eficiente. Su diseño optimizado permite acceder rápidamente a los registros de las IPs sin necesidad de controladores específicos en el kernel.

Para garantizar la compatibilidad con direcciones de memoria superiores a 2GB en sistemas de 64 bits, en `device.h` se ha definido la macro `#define __USE_LARGEFILE64`. Esta directiva habilita el uso de funciones especializadas en el manejo de archivos y memoria que operan con desplazamientos de 64 bits, lo cual es crucial en sistemas embebidos donde la memoria mapeada puede abarcar grandes regiones de direcciones.

Dentro de `device.c`, la función `mmap64()` desempeña un papel fundamental en la interacción con los dispositivos de memoria mapeada. Su propósito es asignar un área de memoria de un dispositivo a un espacio accesible desde el usuario, permitiendo así operaciones eficientes sobre registros de *hardware*. Su nomenclatura es la siguiente:

```
extern void *mmap64 (void *__addr, size_t __len, int __prot,
                     int __flags, int __fd, __off64_t __offset);
```

Donde:

- `__addr`: Dirección donde se desea mapear la memoria. Si es NULL, el sistema elegirá la dirección automáticamente.

- `__len`: Tamaño del área de memoria a mapear.
- `__prot`: Permisos de acceso (lectura, escritura, ejecución).
- `__flags`: Indicadores de tipo de mapeo, como compartido o privado.
- `__fd`: Descriptor de archivo del dispositivo a mapear.
- `__offset`: Desplazamiento de 64 bits dentro del archivo o dispositivo.

El uso de `mmap64()` es esencial para proporcionar acceso eficiente y directo a los registros de *hardware* desde el espacio de usuario, evitando la necesidad de controladores adicionales o mecanismos más complejos de comunicación con el kernel.

4.4.1.2. Código principal para la evaluación de GRU

Además de estos archivos, la implementación de la GRU se encuentra en otros dos archivos donde se realiza la lógica de procesamiento neuronal haciendo uso de las funciones definidas en `device.c`. Estos archivos son el `GRU_Q8_8.c` y `GRU_Q8_8.h` que implementan la lógica de prueba y verificación de la GRU en la RedPitaya, gestionando la transferencia de datos entre la CPU y las IPs encargadas de la computación neuronal. Para ello, se emplean funciones de lectura y escritura sobre registros de memoria mapeada, utilizando el framework definido en `device.c` y `device.h`.

Comenzando por el archivo de cabecera, `GRU_Q8_8.h` (Apéndice B.1), en este se definen las estructuras, constantes y funciones necesarias para la implementación del código en `GRU_Q8_8.c`. Esta cabecera contiene las librerías estándar necesarias como `stdio.h`, `stdlib.h`, `stdint.h` y `unistd.h`, además de las cabeceras específicas para la gestión de las IPs de la DMA y la GRU, como `xdmaread.h`, `xdmawrite.h` y `xfull_gru.h`, que como se mencionó anteriormente deben ser agregadas al *include* de la aplicación. También incluye la cabecera `device.h`, que facilita el acceso a la memoria mapeada de los periféricos.

En cuanto a la estructura de datos, `GRU_Q8_8.h` define las dimensiones de las matrices y vectores utilizados en la implementación de la GRU. Se establecen los tamaños de la matriz de pesos de entrada (3x16), la matriz de pesos recurrentes (17x16) y los vectores de entrada y salida (3, 17 y 16 elementos respectivamente). Estos valores son utilizados para la correcta gestión de la memoria y las operaciones de carga y almacenamiento de datos en la FPGA.

Además, en esta cabecera se definen macros para calcular los tamaños totales en memoria de cada matriz y vector, facilitando su asignación y transferencia. También se especifican las direcciones base de las IPs de la DMA y de la GRU mediante macros predefinidas que se encuentran en el archivo `xparameters.h`, asegurando una correcta interacción con el *hardware* de la RedPitaya.

```
    /* Definitions for driver FULL_GRU */
#define XPAR_XFULL_GRU_NUM_INSTANCES 1

/* Definitions for peripheral FULL_GRU_0 */
#define XPAR_FULL_GRU_0_DEVICE_ID 0
#define XPAR_FULL_GRU_0_S_AXI_CONTROL_BASEADDR 0x80030000
#define XPAR_FULL_GRU_0_S_AXI_CONTROL_HIGHADDR 0x8003FFFF
```

Figura 4.28: Fragmento código `xparameters.h`

Por último, se declaran las funciones auxiliares que permiten la inicialización y manipulación de los datos en el código principal. Entre ellas se encuentran `fill_vectors()` para inicializar los vectores de entrada y estado oculto, `fill_matrices_W()` y `fill_matrices_R()` para la carga de pesos desde archivos externos, y `update_values()` para actualizar los valores de entrada en cada iteración. También se incluyen funciones de depuración como `print_vector()` y `print_matrix()`, que permiten visualizar los valores almacenados y verificar la correcta transferencia de datos.

Este conjunto de funciones, junto con las funciones del gestor de dispositivos (`device.c`), facilita la implementación general del sistema, ya que proporcionan las herramientas necesarias para la correcta configuración y manipulación de los datos dentro del flujo de ejecución.

El flujo de ejecución sigue un esquema secuencial estructurado en cinco fases principales: creación de almacenamiento, configuración de IPs, inicio de ejecución (start), espera de finalización (done) y cierre de dispositivos (close). Primero, se inicializan los vectores y matrices, cargando los datos en la memoria asignada. Luego, se configuran las direcciones base y los tamaños de los datos en las IPs de la DMA y la GRU para garantizar la correcta transferencia de información. Una vez configuradas, se inicia la ejecución de las IPs, lo que activa la propagación de datos en la GRU y las transferencias DMA. Posteriormente, el código verifica que cada operación haya finalizado correctamente antes de proceder a la siguiente iteración. Finalmente, tras completar todas las iteraciones, se cierran los dispositivos utilizados para liberar recursos y evitar accesos indebidos a memoria.

Una vez explicada la base de la cabecera y el flujo del código (archivo `GRU_Q8_8.c`, (Apéndice B.1)) se va a realizar una explicación más detallada de su implementación. El código comienza con la declaración de variables que representan los datos de entrada, salida y los pesos de las capas de la GRU. Se definen los vectores de entrada y salida: `V_IN`, que representa la entrada de la red y tiene un tamaño de 3 elementos; `V_IN_OUT`, que almacena el estado oculto y es reutilizado como entrada en cada iteración con 17 elementos; y `V_OUT`, el vector de salida de 16 elementos inicializado en cero. También se declaran las matrices de pesos `W_RESET`, `W_CAND`, `W_UPDATE` para las conexiones de entrada y `R_RESET`, `R_CAND`, `R_UPDATE` para las conexiones recurrentes. Como se comento en apartados anteriores estos valores de vectores y matrices se almacenan en variables enteras de 32 bits (`int`) representando números en formato Q8.8 en sus 16 LSB.

Para inicializar los datos, se implementa la función `fill_vectors()`, que asigna valores predefinidos a los vectores de entrada. `V_IN` se llena con `0x00000100`, lo que equivale a un valor de 1 en formato Q8.8, mientras que `V_IN_OUT` se inicializa en `0x00000000`, correspondiente a un valor de 0 en el mismo formato. Estos valores han sido elegidos estratégicamente para garantizar

un correcto funcionamiento de la red en la primera iteración. Dado que en este punto no existe un estado oculto previo que pueda influir en la operación de la GRU, se establece un vector de entrada con valores inicializados en 1 para asegurar que los pesos y sesgos de la red sean considerados en los cálculos desde el inicio. Al mismo tiempo, el vector de estado oculto previo `V_IN_OUT` se mantiene en 0, ya que no hay información previa que deba ser reutilizada en esta primera iteración.

Las funciones `fill_matrices_W()` y `fill_matrices_R()` se encargan de leer los valores de las matrices de pesos desde archivos de texto externos. Estos valores, almacenados en formato hexadecimal, son leídos fila por fila y convertidos en enteros antes de ser asignados a las matrices utilizadas en la ejecución de la GRU. Para garantizar el correcto funcionamiento del sistema, estos archivos deben estar disponibles en el directorio de la RedPitaya, de manera similar a cómo se transfieren otros elementos esenciales como el bitstream. Si los archivos no están presentes en el sistema o ocurre un error durante su apertura o lectura, el programa imprime un mensaje de error y detiene su ejecución, ya que la ausencia de estos datos impediría la configuración correcta de los pesos de la red. Además, se incluyen funciones auxiliares como `print_vector()` y `print_matrix()` que permiten visualizar los valores cargados, facilitando la depuración y verificación del proceso.

Con el objetivo de evaluar el comportamiento de la implementación *hardware* en la RedPitaya y compararlo con la ejecución en software, se han añadido funciones específicas para la adaptación y validación de los datos. La implementación trabaja con valores representados en formato int (en realidad, Q8.8), por lo que se hace necesario convertirlos a una representación en punto flotante para su correcta interpretación en software. Para ello, se han definido las funciones `int_to_float_vector()` y `int_to_float_matrix()` , que permiten transformar los valores de 16 bits en flotantes dividiéndolos por 256.0. Esta conversión facilita la comparación con los resultados obtenidos en *hardware*.

Adicionalmente, se han implementado funciones para replicar la lógica de la GRU en software. Estas funciones permiten evaluar el comportamiento de la red en un entorno puramente software y contrastarlo con los resultados obtenidos en *hardware*. Se han desarrollado las siguientes funciones:

- `rst_gate_SW()`, que implementa la activación de la puerta de reseteo mediante dos productos matriz-vector y la aplicación de la función sigmoide.
- `cand_state_SW()`, que calcula el estado candidato de la GRU utilizando la activación de la puerta de reseteo y la función tangente hiperbólica.
- `upd_gate_SW()`, que replica la lógica de la puerta de actualización.
- `rest_of_GRU_SW()`, que combina los resultados de las puertas para actualizar el estado oculto de la GRU.

Para verificar la precisión de los cálculos realizados en *hardware*, se ha desarrollado la función `precision()`, que evalúa la diferencia absoluta entre los resultados software y *hardware*, permitiendo cuantificar el error introducido por la implementación en la FPGA. Esta función toma los valores de salida de *hardware* en formato Q8.8, los convierte a flotante y los compara con los valores obtenidos en software.

Este conjunto de funciones permite una validación más completa del sistema, asegurando que los datos procesados en *hardware* sean consistentes con la referencia software. Además, facilitan la depuración y permiten analizar cómo las operaciones aritméticas en *hardware* afectan el comportamiento de la GRU.

En la función principal `main()`, se realiza la inicialización de datos ejecutando `fill_vectors()` para los vectores de entrada y `fill_matrices_W()` y `fill_matrices_R()` para las matrices de pesos. Posteriormente, se imprimen los valores de los vectores y matrices para verificar que los datos se han cargado correctamente antes de escribirlos en la memoria mapeada de las IPs. Para la ejecución en *hardware*, estos datos deben ser transferidos a las direcciones de memoria correspondientes, utilizando la función `write_device()` para la escritura en memoria mapeada y `read_device()` para la verificación de la correcta escritura. Cada conjunto de pesos `W_RESET`, `W_CAND`, `W_UPDATE`, `R_RESET`, `R_CAND`, `R_UPDATE` se almacena en diferentes direcciones de memoria mediante la definición de estructuras de tipo `uio_device` definidas en el archivo `device.h`. Estas direcciones de memoria han sido elegidas mediante el *Device Tree* (DTB - Device Tree Blob). El *Device Tree* es un mecanismo esencial en sistemas embebidos basados en Linux, como Red Pitaya, que permite describir la estructura del *hardware*, incluyendo la memoria disponible, los periféricos y los dispositivos mapeados en memoria. En este proyecto, se emplea el Device Tree proporcionado por Red Pitaya para determinar los rangos de memoria disponibles y garantizar la correcta asignación de los *device* creados en el código.

El fragmento del Device Tree relevante para la memoria en Red Pitaya es el siguiente:

```
reserved-memory {
    #address-cells = <0x01>;
    #size-cells = <0x01>;
    ranges;

    buffer@1000000 {
        phandle = <0x39>;
        reg = <0x1000000 0x200000>;
    };

    labuf@a000000 {
        phandle = <0x38>;
        reg = <0xa000000 0x2000000>;
    };

    linux,cma {
        alignment = <0x2000>;
        compatible = "shared-dma-pool";
        linux,cma-default;
        reusable;
        size = <0x1000000>;
    };
};
```

Este bloque extraído del archivo `draw.dts`, encontrado en la tarjeta SD suministrada por Redpitaya, define las regiones de memoria reservada en el sistema. Los elementos más relevantes son:

- buffer@1000000: Define un bloque de memoria de 0x200000 bytes (2 MB) a partir de la dirección 0x01000000.
- labuf@a000000: Define otro bloque de 0x2000000 bytes (32 MB) a partir de la dirección 0x0A000000.
- linux,cma: Define un área de memoria reutilizable para DMA, con una alineación de 0x2000 y un tamaño de 0x1000000 (16 MB).

Dado que los *device* requieren memoria accesible para la transferencia de datos, es fundamental asegurarse de que las direcciones elegidas estén dentro de un rango válido. En este proyecto, se

utiliza un valor dentro de este rango para mapear las memorias que deben ser accedidas para cargar los valores de los pesos y sesgos y para recibir los valores resultantes, además de mapear las IP que forman parte del proyecto (DMA_READ_0, DMA_READ_1, FULL_GRU, DMA_WRITE_0). Todo esto se ha realizando mediante el acceso a /dev/mem.

Como podemos observar el Device Tree juega un papel fundamental en la gestión de memoria del sistema, permitiendo identificar regiones de memoria adecuadas para mapear los dispositivos. En este caso, se ha aprovechado la información del Device Tree para garantizar que la memoria asignada a los *devices* se asignen dentro de un rango válido, evitando conflictos con otras secciones de memoria reservada.

Para cada matriz, se abre su *device* correspondiente mediante `open_device()`, se escriben los valores en la memoria mapeada iterando sobre las filas y columnas y, finalmente, se realiza una verificación de los valores guardados.

```
 uio_device MATRIX_W_RST;
 volatile __off64_t address_matrix_W_RST = 0x01000000;
printf("\r\n base address weight reset :--: 0x%x",(unsigned int) address_matrix_W_RST);
open_device(&MATRIX_W_RST, address_matrix_W_RST, SIZE_MW, "/dev/mem",   "MATRIX_W_RST");
printf("\r\n successfully open device MATRIX WEIGTH RESET \n");

    for (int i = 0; i < MATRIX_A_ROWS; i++) {
        for (int j = 0; j < MATRIX_A_COLS; j++) {
            write_device(&MATRIX_W_RST, (i + j * MATRIX_A_ROWS)*4,W_RESET[i][j]);
        }
    }

    // Comprobacion valores guardados
    for (int i = 0; i < MATRIX_A_ROWS; i++) {
        for (int j = 0; j < MATRIX_A_COLS; j++) {
            value = read_device(&MATRIX_W_RST, (i + j * MATRIX_A_ROWS)*4);
            printf("%x ",value);
        }
        printf("\n");
    }
```

Figura 4.29: Ejemplo almacenamiento de valores y comprobación

Existe una diferencia en el almacenamiento de matrices y vectores que radica en su organización en memoria y en la manera en que los datos son accedidos y manipulados. En el caso de las matrices, los valores se almacenan en memoria de forma bidimensional, recorriéndolas fila por fila, lo que implica calcular la dirección de cada elemento mediante la expresión $(i + j \times \text{MATRIX\_B\_ROWS}) \times 4$. Este cálculo asegura que los datos se distribuyan correctamente en la memoria mapeada de acuerdo con su estructura matricial. En contraste, los vectores son estructuras unidimensionales, por lo que sus elementos pueden escribirse secuencialmente en memoria, utilizando simplemente un desplazamiento lineal basado en su índice ($j \times 4$).

Del mismo modo que se crean los *devices* para almacenar las matrices y vectores, también es necesario instanciar los *devices* correspondientes a las IPs de la GRU, incluyendo las DMAs de lectura, la DMA de escritura y la unidad `full_GRU`. La metodología para su creación es idéntica; sin embargo, en este caso, las direcciones base de memoria utilizadas provienen del archivo `xparameters.h`, donde se definen los mapeos de *hardware* generados por la herramienta de sín-

tesis. Para facilitar su acceso y gestión dentro del código, estas direcciones como se ha comentado anteriormente han sido organizadas en el `device.h`, proporcionando una referencia más directa y estructurada.

Una vez que los datos han sido almacenados en memoria, se procede a la configuración de las IPs de DMA y la IP de la GRU. Aunque el proceso de configuración sigue un procedimiento común para todas las IPs, cada una de ellas tiene registros específicos que deben ser configurados de acuerdo con sus necesidades. Estos registros se encuentran en la cabeceras de las IPs añadidas, en la Figura 4.30 se puede observar un ejemplo de una cabecera de la IP `dma_read`.

```
#define XDMAREAD_CONTROL_ADDR_AP_CTRL      0x00
#define XDMAREAD_CONTROL_ADDR_GIE          0x04
#define XDMAREAD_CONTROL_ADDR_IER          0x08
#define XDMAREAD_CONTROL_ADDR_ISR          0x0c
#define XDMAREAD_CONTROL_ADDR_WIDTH_DATA   0x10
#define XDMAREAD_CONTROL_BITS_WIDTH_DATA   32
#define XDMAREAD_CONTROL_ADDR_DIN_DATA     0x18
#define XDMAREAD_CONTROL_BITS_DIN_DATA     64
```

Figura 4.30: Ejemplo fragmento *header* de una IP (dma_read)

Para las DMA de lectura, `DMA_READ_VIN` y `DMA_READ_VIN_OUT`, se establece la dirección base de los vectores de entrada en el registro `XDMAREAD_CONTROL_ADDR_DIN_DATA`, y se define el tamaño del vector con `XDMAREAD_CONTROL_ADDR_WIDTH_DATA`. A continuación se presenta un fragmento de código que ejemplifica este proceso:

```
// DMA READ VECTOR IN configure base address vector in
    printf("\r\n\n configure dma read vin..");
    write_device(&DMA_READ_VIN, XDMAREAD_CONTROL_ADDR_DIN_DATA,(int)address_vector_IN);

    data = read_device(&DMA_READ_VIN,XDMAREAD_CONTROL_ADDR_DIN_DATA);   // DEBUG: Comprobación dirección a leer
    printf("\r\n address DMA READ VIN.. 0x%x",data);

    write_device(&DMA_READ_VIN, XDMAREAD_CONTROL_ADDR_DIN_DATA+0x4, 0x00000000);
    write_device(&DMA_READ_VIN, XDMAREAD_CONTROL_ADDR_WIDTH_DATA, VECTOR_IN_SIZE);

    data = read_device(&DMA_READ_VIN,XDMAREAD_CONTROL_ADDR_WIDTH_DATA); // DEBUG: Comprobación tamaño
    printf("\r\n width dma read VIN.. 0x%x \n",data);
```

En este fragmento de código, se configura la IP de DMA de lectura (DMA_READ_VIN) para que lea un vector de entrada desde la memoria. Para ello, se asigna la dirección base del vector, address_vector_IN, lo cual permite que la IP sepa desde qué lugar de la memoria debe comenzar a leer los datos. Además, se ajustan los registros de la IP para garantizar que los bits más significativos de la dirección, `XDMAREAD_CONTROL_ADDR_DIN_DATA+0x4`, se establezcan correctamente, ya que los datos manejados son de 32 bits. Posteriormente, se configura el tamaño del vector a leer haciendo uso de la macro creada en su cabecera (`GRU_Q8_8.h`) lo cual es necesario para que la IP sepa cuántos elementos transferir.

Una vez configurados tanto la dirección como el tamaño, se realiza una verificación para asegurar que ambos valores se hayan establecido correctamente. Esto se hace leyendo los registros correspondientes y comprobando que los valores escritos coinciden con los esperados. Este proceso es repetido para cada una de las IPs involucradas, con la diferencia de que cada IP tiene sus

propios valores de dirección y tamaño configurados, pero el procedimiento de configuración y verificación es consistente para todas ellas.

En la IP de la GRU, `full_GRU`, se configuran las direcciones base de todas las matrices de pesos en los registros correspondientes, como `XFULL_GRU_CONTROL_ADDR_DIN_WEIGHT_RST_DATA` para los pesos del reset gate. Finalmente, en la DMA de escritura `DMA_WRITE`, se configura la dirección base del vector de salida, completando así el proceso de configuración de las IPs para la transferencia de datos.

Finalmente, se encuentra el proceso de verificación de la GRU, el cual se ejecuta 150 veces, donde se ejecuta el proceso de activación de las IPs. El proceso de activación de las IPs sigue el mismo procedimiento para todas, con la diferencia de que cada una modifica su propio registro de control. Para ilustrar este procedimiento, se detalla la activación de la DMA WRITE. En primer lugar, se lee el registro de control XDMAWRITE_CONTROL_ADDR_AP_CTRL y se aplica una máscara para conservar únicamente el bit 7 (auto_restart). Este bit es mantenido para evitar modificar una posible configuración previa que habilite la reejecución automática de la DMA sin necesidad de volver a escribir en el registro de control en cada ciclo. Luego, se establece el bit 0 (ap_start), lo que inicia la transferencia de datos. Una vez iniciada la operación, se verifica su estado leyendo nuevamente el registro de control y se extrae el valor del registro XDMAWRITE_CONTROL_ADDR_DOUT_DATA, que contiene los datos transferidos a la salida de la GRU. Este valor se imprime para comprobar que la operación se ha realizado correctamente. Siguiendo el mismo procedimiento se realiza la ejecución de la IP FULL GRU, DMA_READ_VIN y la _READ_VIN_OUT.

Una vez activadas las IPs, es necesario monitorizar su estado para asegurar que cada operación se complete antes de continuar con la siguiente. Este proceso es idéntico para todas las IPs y se basa en la lectura de sus respectivos registros de control, verificando el estado hasta que indiquen que la operación ha finalizado correctamente. Para ilustrarlo, se detalla la verificación del estado de la DMA_READ_V_IN.

En primer lugar, se lee el valor del registro XDMAREAD_CONTROL_ADDR_AP_CTRL de la DMA READ V IN, lo que permite conocer su estado actual. Se imprime este valor para depuración y se entra en un bucle de espera donde se sigue leyendo el estado hasta que la IP alcance el estado ap_idle (0x4 en el registro de control). Este estado indica que la DMA ha finalizado la transferencia de datos y está lista para recibir una nueva solicitud.

Este mismo procedimiento se aplica a las demás IPs, utilizando sus respectivas macros de control. Este mecanismo garantiza la sincronización entre las IPs, evitando lecturas o escrituras de datos antes de que la operación anterior haya finalizado.

Cuando la ejecución de la GRU y la transferencia de datos finalizan en cada iteración, se almacenan los resultados en el vector V_OUT, que se obtiene leyendo los valores de VECTOR_OUT en la memoria mapeada. Estos valores son impresos para su análisis y evaluación. Luego, los valores de V_IN y V_IN_OUT son actualizados mediante la función `update_values` para la siguiente iteración, asegurando que la GRU reciba una entrada dinámica en cada ciclo. Finalmente, se imprimen los nuevos valores de los vectores de entrada y salida antes de pasar a la siguiente iteración.

El proceso de actualización de los valores de entrada sigue una estrategia donde V_IN se modifica manualmente en cada iteración, mientras que V_IN_OUT se actualiza con los valores obtenidos en V_OUT. La función `update_values` se encarga de realizar esta actualización de manera estructurada. En primer lugar, asigna valores a los primeros elementos de V_IN y luego el vector

V_IN_OUT se actualiza tomando el primer elemento de V_OUT como referencia y desplazando los valores previos, ya que su primer valor es '1' debido a la implementación *recurrent-bias-after-multiplication*. Una vez actualizados, ambos vectores son escritos nuevamente en la memoria mapeada utilizando la función `write_device`, lo que garantiza que las siguientes iteraciones de la GRU trabajen con los valores más recientes.

Este mecanismo de actualización asegura que la GRU reciba entradas coherentes y dependientes de las iteraciones previas, lo cual es esencial para el correcto funcionamiento de una red recurrente. Además, la impresión de los valores de V_IN y V_IN_OUT permite verificar visualmente su evolución a lo largo de las iteraciones, facilitando la detección de posibles inconsistencias en la propagación de datos.

Al finalizar las 150 iteraciones, se cierran todas las IPs y dispositivos utilizados mediante `close_device()`, asegurando la correcta liberación de recursos. Esto incluye las DMA READ, la IP de la GRU y la DMA WRITE, evitando fugas de memoria o accesos indebidos a dispositivos mapeados. Con esto, el código completa la verificación del funcionamiento de la GRU en la RedPitaya, asegurando la correcta transferencia de datos mediante DMA y la ejecución del modelo neuronal en la FPGA.

4.4.2. Programación de la Red Pitaya

En este apartado se describe el proceso seguido para la programación de la RedPitaya. Cabe destacar que no se ha seguido estrictamente la metodología indicada en la documentación oficial de Red Pitaya debido a las dificultades descritas en el Anexo C, relacionadas con la ejecución de archivos `.elf` en la Red Pitaya.

A pesar de contar con la versión 2.0 del sistema operativo en la SD Card, la metodología de programación corresponde al de la versión 1.04. A continuación, se detallan los pasos realizados:

4.4.2.1. Carga del bitstream en la FPGA

1. **Acceder al archivo bitstream**
   - Abrir una terminal de Linux.
   - Navegar hasta la ubicación del archivo *bitstream* (`.bit`) generado en Vivado.
2. **Transferir el archivo bitstream a la Red Pitaya [36]**
   - Enviar el archivo `.bit` a la Red Pitaya utilizando el comando SCP:

     ```
     scp nombre_archivo.bit root@rp-xxxxxx.local:/root
     ```

     donde `xxxxxx` corresponde a los últimos seis caracteres de la dirección MAC de la Red Pitaya. Alternativamente, también se puede utilizar la dirección IP del dispositivo. Esta IP se puede encontrar accediendo desde la web a la Redpitaya e ir al apartado *System* → *Manager Network*.
3. **Verificar la transferencia y establecer conexión SSH [37]**
   - Iniciar una sesión SSH en la Red Pitaya.

```
ssh root@rp-xxxxxx.local
```

- **Primera conexión**
  La primera vez que se realice la conexión con la Red Pitaya mediante SSH, el sistema solicitará la confirmación para añadir la huella digital (*fingerprint*) del dispositivo al ordenador. Este mensaje aparece de la siguiente manera:

  ```
  The authenticity of host 'rp-xxxxxx.local (XXX.XXX.XXX.XXX)'
  can't be established.
  ECDSA key fingerprint is SHA256:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx.
  Are you sure you want to continue connecting (yes/no)?
  ```

  Para continuar con la conexión, es necesario escribir `yes` y presionar `Enter`. A continuación, el sistema añadirá la huella digital al archivo de *hosts* conocidos y permitirá la conexión con la Red Pitaya.
  Si en el futuro se cambia el sistema operativo de la Red Pitaya o su dirección IP, es posible que aparezca un mensaje de advertencia sobre un cambio en la huella digital. En ese caso, puede ser necesario eliminar la entrada anterior (comando inferior) o se pedirá de nuevo que se añada la huella digital (*fingerprint*).

  ```
  ssh-keygen -R rp-xxxxxx.local
  ```

  Luego, se podrá volver a intentar la conexión SSH.
- **Autentificación con contraseña**
  Cada vez que se intente establecer una conexión SSH o transferir archivos mediante SCP, el sistema solicitará la contraseña del usuario `root` de la Red Pitaya. El mensaje que aparece en el terminal es el siguiente:

  ```
  root@rp-xxxxxx.local's password:
  ```

  Es necesario introducir la contraseña y presionar `Enter` para completar la conexión o la transferencia de archivos. Tanto el usuario como la contraseña es `root`.
- Comprobar que el archivo `.bit` se encuentra en el directorio raíz ejecutando:

  ```
  redpitaya > ls
  ```

4. **Cargar el bitstream en la FPGA**

   - Ejecutar el siguiente comando para cargar el bitstream en el módulo `xdevcfg`:

     ```
     cat nombre_archivo.bit > /dev/xdevcfg
     ```

Una vez completados estos pasos, el *hardware* de la Red Pitaya queda configurado con el diseño correspondiente al proyecto, asegurando la implementación de la arquitectura de la red neuronal.

4.4.2.2. Ejecución del archivo ejecutable

Para comprobar el funcionamiento del *hardware*, se utiliza el archivo `.elf` generado en Vitis IDE. Su carga y ejecución siguen un procedimiento similar al del bitstream:

1. **Transferir el archivo `.elf` a la Red Pitaya**
   - Enviar el archivo ejecutable con SCP:

     ```
     scp nombre_archivo.elf root@rp-xxxxxx.local:/root
     ```

2. **Ejecutar el archivo en la Red Pitaya**
   - Conectarse a la Red Pitaya por SSH (si no se ha hecho previamente).
   - Ejecutar el archivo con:

     ```
     ./nombre_archivo.elf
     ```

Con estos pasos, se finaliza la configuración, ejecución del *hardware* y sofware que realizará comprobación del correcto funcionamiento de la red neuronal sobre la Red Pitaya.

4.4.3. Resultados obtenidos

Antes de presentar los resultados obtenidos, es fundamental describir el procedimiento seguido para garantizar la correcta ejecución del código en la Red Pitaya y la verificación de la GRU. Para ello, en primer lugar, se debe generar el archivo ejecutable `.elf` en Vitis IDE, lo cual se logra compilando el proyecto mediante la opción *Build Project* en el entorno de desarrollo. Una vez completada la compilación sin errores, se obtiene el archivo `.elf`, que posteriormente será transferido a la Red Pitaya junto con el archivo de *bitstream* `.bit` y los archivos de salida en formato `.txt`.

La verificación del funcionamiento de la GRU puede llevarse a cabo tanto desde un terminal convencional conectado a Redpitaya como desde el terminal integrado en Vitis IDE. Para ejecutar la comprobación desde el entorno de Vitis, se deben seguir los siguientes pasos:

1. Abrir Vitis IDE y hacer clic sobre la lupa en la parte superior derecha (atajo `Ctrl+3`).
2. Buscar la opción *Terminal* y seleccionarla.
3. Hacer clic en el icono azul para abrir un terminal.
4. Establecer la conexión con la Red Pitaya. Para ello, se debe ingresar la dirección IPdel dispositivo en el campo *Host* y utilizar las credenciales correspondientes.

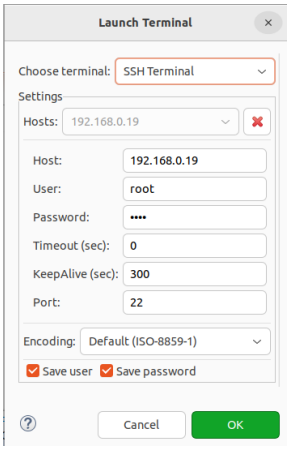

Figura 4.31: Establecer conexión Redpitaya desde terminal Vitis

Una vez establecida la conexión, es necesario verificar que todos los archivos han sido correctamente transferidos a la Red Pitaya. Esta verificación se realiza mediante una conexión SSH al dispositivo y la ejecución del siguiente comando: `ls`. Este comando listará los archivos presentes en el directorio de la Red Pitaya, permitiendo confirmar la correcta transferencia de los ficheros. A continuación, se incluirá una captura de pantalla mostrando el contenido esperado en el sistema de archivos del dispositivo.

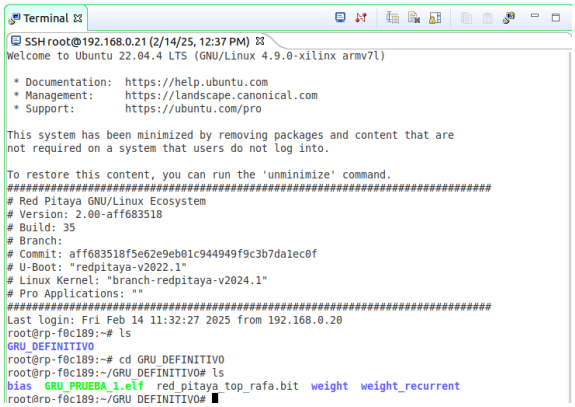

Figura 4.32: Listado archivos en Redpitaya

Finalmente, tras la ejecución del código, se procede al análisis de los datos obtenidos para evaluar el desempeño de la red neuronal implementada en la Red Pitaya. En el Apéndice B.3, se presentan fragmentos representativos de los resultados obtenidos, los cuales incluyen la carga de los vectores de entrada, las matrices de pesos, las matrices recurrentes utilizadas y los valores obtenidos en cada una de las iteraciones.

El análisis de estos datos confirma la correcta inicialización de la GRU en la Red Pitaya, asegurando que las matrices de pesos y las matrices recurrentes han sido correctamente cargadas en el *hardware*. Además, se verifica que los vectores de entrada y salida han sido procesados sin errores, lo que indica un funcionamiento adecuado del modelo implementado.

Para evaluar la precisión de la implementación en *hardware*, se ha realizado una comparación

con los valores obtenidos en la simulación de Vitis HLS. La Tabla 4.2 muestra los valores de precisión obtenidos en la simulación, mientras que los valores medidos en la Red Pitaya se presentan al final de cada iteración en el código.

Los resultados demuestran que la precisión alcanzada en la implementación *hardware* a lo largo de las 150 iteraciones es adecuada y se mantiene en valores similares a los obtenidos en la simulación de Vitis HLS. Esta coherencia valida la correcta implementación del modelo en *hardware* y confirma que el comportamiento de la red neuronal en la Red Pitaya es consistente con las expectativas derivadas de la simulación.

Capítulo 5

Conclusiones y Líneas Futuras

5.1. Conclusiones

La realización de este Trabajo Fin de Máster ha permitido la implementación exitosa de una célula GRU en una FPGA de Xilinx, utilizando herramientas de síntesis de alto nivel para optimizar su diseño y evaluación. Se ha logrado una integración eficiente en el entorno de la RedPitaya, superando desafíos técnicos como la gestión de la transferencia de datos mediante DMA y la optimización del consumo de recursos en la FPGA.

Uno de los objetivos clave de este trabajo era garantizar que la implementación de la GRU se ajustara a los recursos disponibles en la FPGA. En este sentido, la versión final del diseño basada en cuantificación de crecimiento natural ha demostrado ser la opción más eficiente, reduciendo el consumo de LUTs a 29.263 y el de FFs a 17.815, en comparación con otras implementaciones como la de punto flotante, que consumía 75.844 LUTs y 55.339 FFs. Además, el uso de DSPs se ha reducido a 64 en la implementación final, manteniéndose por debajo del límite disponible en la FPGA.

En el contexto del proyecto suministrado, donde la implementación debía integrarse sin comprometer el funcionamiento del sistema ya existente, se ha evaluado el uso total de recursos tras la fusión de ambas implementaciones. Los resultados obtenidos muestran que el diseño completo, incluyendo la GRU y el resto de módulos, ha requerido un total de 31.112 LUTs, 40.068 FFs, 108 DSPs, 52 BRAMs y 3.262 LUTRAM, con la mayoría de estos valores, excepto las LUTs (58 %), por debajo del 50 % de los recursos disponibles en la FPGA. Esta integración ha demostrado ser viable, manteniendo un margen de un 50 % de recursos libres, permitiendo así futuras optimizaciones o ampliaciones del sistema.

Otro aspecto fundamental era la latencia, la cual debía mantenerse por debajo de 150 ciclos de reloj para cumplir con los requisitos del sistema. Los resultados de co-simulación muestran que la implementación en ap_fixed<18,8> alcanzó una latencia mínima de 149 ciclos, mientras que la implementación basada en cuantificación de crecimiento natural logró reducirla aún más, hasta 131 ciclos, cumpliendo así con el objetivo de diseño.

En términos de precisión, se ha verificado que todas las implementaciones mantienen un error inferior al 0.002 en la mayoría de los casos, lo que indica que la conversión a formatos numéricos optimizados no ha afectado significativamente la exactitud de los resultados. La principal limita-

ción en este aspecto proviene de la representación de las funciones de activación mediante ROMs pre-calculadas.

Un aspecto relevante de este trabajo ha sido la adquisición de conocimientos sobre las herramientas y plataformas utilizadas. Antes de iniciar este proyecto, no se contaba con experiencia previa en el ecosistema de Xilinx ni en el uso de RedPitaya, ni el diseño digital con HLS. Sin embargo, a lo largo del desarrollo, se ha adquirido un dominio considerable de estas tecnologías, lo que ha permitido una implementación funcional y optimizada de la GRU. Esta curva de aprendizaje ha sido un reto, pero también un factor clave en el éxito del proyecto, ya que ha facilitado la toma de decisiones de diseño y ha permitido una mejor comprensión del flujo de trabajo en *hardware* reconfigurable.

Finalmente, la validación experimental en la RedPitaya ha demostrado que la implementación en *hardware* es funcional y que los resultados obtenidos son consistentes con los de la simulación en software. La experiencia adquirida en este proceso, junto con la documentación detallada del desarrollo, proporciona una base sólida para futuros trabajos en el ámbito de la implementación de redes neuronales en FPGA.

En conclusión, este trabajo ha logrado cumplir con los objetivos planteados, obteniendo una implementación eficiente en términos de recursos, latencia y precisión. No obstante, se identifican áreas de mejora, como la optimización de la gestión de memoria y la exploración de arquitecturas más compactas para reducir aún más el consumo de *hardware*.

5.2. Líneas Futuras

Como parte del desarrollo futuro de este trabajo, se plantea desarrollar una versión de la GRU utilizando el formato Q8.8, lo que permitirá realizar una comparación directa con una implementación previa en Verilog. Esta comparación será clave para evaluar diferencias en términos de eficiencia computacional, precisión numérica y consumo de recursos, permitiendo extraer conclusiones sobre la idoneidad de cada enfoque en función del contexto de aplicación. La implementación en Q8.8 supone un reto adicional en cuanto a la gestión de la representación numérica en *hardware*, por lo que será necesario analizar en profundidad los efectos de esta cuantización sobre la estabilidad y exactitud de los cálculos.

Además, se pretende optimizar la implementación con el objetivo de reducir la latencia del procesamiento por debajo de los 125 ciclos de reloj. Dado que el sistema opera con un reloj interno de 125 MHz y los datos de entrada pueden llegar a una frecuencia de 1 MHz, garantizar una latencia menor a estos 125 ciclos permitirá procesar cada dato dentro del intervalo de tiempo disponible, asegurando así un funcionamiento en tiempo real sin pérdidas de información. Para alcanzar este objetivo, será necesario explorar técnicas de paralelización y optimización del pipeline, así como realizar un ajuste fino en la gestión de dependencias de datos y en el acceso a memoria, de modo que se minimicen los retardos innecesarios.

Otra posible línea de mejora consiste en modificar la arquitectura de la implementación para que el bucle de iteraciones se ejecute de manera interna en el *hardware*, en lugar de depender de una ejecución secuencial desde el software. Esta estrategia permitiría reducir la sobrecarga de comunicación entre la CPU y la FPGA, eliminando latencias asociadas a la transmisión de datos y mejorando la eficiencia global del sistema. Para ello, será fundamental diseñar un mecanismo de control adecuado dentro del propio *hardware*, que permita gestionar de forma autónoma el flujo de

iteraciones sin comprometer la correcta sincronización con el resto de componentes del sistema.

Estas mejoras no solo contribuirán a la optimización del diseño actual, sino que también abrirán la puerta a nuevas aplicaciones donde la aceleración en *hardware* de redes neuronales recurrentes sea un factor clave. Además, su implementación permitirá ampliar el conocimiento sobre las técnicas avanzadas de optimización en FPGA, aportando experiencia valiosa en la implementación de arquitecturas eficientes para el procesamiento en tiempo real.

Bibliografía

[1] Denis Kleyko et al. “Perceptron Theory Can Predict the Accuracy of Neural Networks”. En: *IEEE Transactions on Neural Networks and Learning Systems* 35.7 (jul. de 2024), págs. 9885-9899. ISSN: 2162-2388. DOI: 10.1109/tnnls.2023.3237381. URL: http://dx.doi.org/10.1109/TNNLS.2023.3237381.

[2] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/.

[3] Frank Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain”. En: *Psychological Review* 65.6 (1958), págs. 386-408. DOI: 10.1037/h0042519. URL: https://www.ling.upenn.edu/courses/cogs501/Rosenblatt1958.pdf.

[4] Shiv Ram Dubey, Satish Kumar Singh y Bidyut Baran Chaudhuri. *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark*. 2022. arXiv: 2109.14545 [cs.LG]. URL: https://arxiv.org/abs/2109.14545.

[5] Farhad Mortezapour Shiri et al. *A Comprehensive Overview and Comparative Analysis on Deep Learning Models: CNN, RNN, LSTM, GRU*. 2024. arXiv: 2305.17473 [cs.LG]. URL: https://arxiv.org/abs/2305.17473.

[6] Iqbal H. Sarker. “Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions”. En: *SN Computer Science* 2.6 (2021), pág. 420. ISSN: 2661-8907. DOI: 10.1007/s42979-021-00815-1. URL: https://doi.org/10.1007/s42979-021-00815-1.

[7] Nikhil B. Gaikwad et al. “Efficient FPGA Implementation of Multilayer Perceptron for Real-Time Human Activity Classification”. En: *IEEE Access* 7 (2019), págs. 26696-26706. DOI: 10.1109/ACCESS.2019.2900084.

[8] J. B. Chaudron y A. Dion. “Evaluation of Gated Recurrent Neural Networks for Embedded Systems Applications”. En: *Computational Intelligence. IJCCI 2021*. Ed. por J. Garibaldi et al. Vol. 1119. Studies in Computational Intelligence. Springer, Cham, 2023. DOI: 10.1007/978-3-031-46221-4_11. URL: https://doi.org/10.1007/978-3-031-46221-4_11.

[9] L. Qin, N. Yu y D. Zhao. “Applying the convolutional neural network deep learning technology to behavioural recognition in intelligent video”. En: *Tehnički vjesnik* 25.2 (2018), págs. 528-535. DOI: 10.17559/TV-20171229024444.

[10] F.A. Gers, J. Schmidhuber y F. Cummins. “Learning to forget: continual prediction with LSTM”. En: *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*. Vol. 2. 1999, 850-855 vol.2. DOI: 10.1049/cp:19991218.

[11] Rahul Dey y Fathi M. Salem. “Gate-variants of Gated Recurrent Unit (GRU) neural networks”. En: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2017, págs. 1597-1600. DOI: 10.1109/MWSCAS.2017.8053243.

[12] MathWorks. *GRULayer*. 2025. URL: https://es.mathworks.com/help/deeplearning/ref/nnet.cnn.layer.grulayer.html (visitado 13-01-2025).

[13] Junyoung Chung et al. “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling”. En: *CoRR* abs/1412.3555 (2014). URL: http://arxiv.org/abs/1412.3555.

[14] A. Kumar et al. “Deep learning for avionics systems: Challenges and opportunities”. En: *Journal of Aerospace Computing* (2023).

[15] D. Levin et al. “Optimizing autonomous driving with neural networks”. En: *Automotive Systems Research* (2022).

[16] S. Ravi et al. “Artificial neural networks in space systems: An overview”. En: *Space Systems Review* (2021).

[17] Y. Zhang et al. “Neural networks for drone autonomy: Applications and challenges”. En: *Unmanned Systems Quarterly* (2023).

[18] J. Shi et al. “Recurrent neural networks in embedded systems: A review”. En: *Embedded Systems Journal* (2020).

[19] S. Han et al. “Deep compression: Compressing deep neural networks with pruning, trained quantization, and Huffman coding”. En: *International Conference on Learning Representations* (2015).

[20] H. Chen et al. “Efficient hardware for RNNs in edge devices”. En: *IEEE Transactions on Neural Networks and Learning Systems* (2022).

[21] Marcello Traiola et al. “Impact of High-Level Synthesis on Reliability of Artificial Neural Network Hardware Accelerators”. En: *IEEE Transactions on Nuclear Science* 71.4 (2024), págs. 845-853. DOI: 10.1109/TNS.2024.3377596.

[22] Red Pitaya Team. *Red Pitaya Documentation - Introduction*. 2025. URL: https://redpitaya.readthedocs.io/en/latest/intro.html.

[23] Radares y Sensores. *¿Qué es la Red Pitaya?* 2018. URL: https://radaresysensores.wordpress.com/2018/05/28/que-es-la-red-pitaya/.

[24] Red Pitaya Team. *A Guide to Red Pitaya Product Naming*. 2025. URL: https://content.redpitaya.com/blog/a-guide-to-red-pitaya-product-naming.

[25] Red Pitaya Team. *Red Pitaya Developer Guide - 125-14 Z20*. 2025. URL: https://redpitaya.readthedocs.io/en/latest/developerGuide/hardware/125-14-Z20/top.html.

[26] Red Pitaya Team. *Red Pitaya Developer Guide - Hardware*. 2025. URL: https://redpitaya.readthedocs.io/en/latest/developerGuide/hardware/hardware.html.

[27] AMD. *XA Zynq-7000 Automotive Family Overview (DS188)*. 2025. URL: https://docs.amd.com/v/u/en-US/ds188-XA-Zynq-7000-Overview.

[28] AMD. *7 Series DSP48E1 Slice User Guide (UG479)*. 2025. URL: https://docs.amd.com/v/u/en-US/ug479_7Series_DSP48E1.

[29] AMD. *Zynq-7000 Product Selection Guide*. 2025. URL: https://docs.amd.com/v/u/en-US/zynq-7000-product-selection-guide.

[30] Pedro Fermé. *Introducción a los Algoritmos*. 2025. URL: https://cs.famaf.unc.edu.ar/~pedro/introalg/intro.pdf.

[31] Red Pitaya Team. *FPGA Applications with Red Pitaya*. 2025. URL: https://redpitaya.com/applications-measurement-tool/fpga/.

[32] RS Components. "RS Components presenta las ventajas de Red Pitaya, la plataforma de prueba y medida de código abierto en electrónica". En: *RedeWeb* (2014). URL: https://www.redeweb.com/articulos/componentes/rs-components-presenta-las-ventajas-de-red-pitaya-la-plataforma-de-prueba-y-medida-de-codigo-abierto-en-electronica-2014/.

[33] Xilinx. *Vitis HLS User Guide (UG1399)*. 2022. URL: https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug1399-vitis-hls.pdf.

[34] AMD. *Vivado IP Subsystems User Guide: Cross-Probing with Address Editor*. 2022. URL: https://docs.amd.com/r/2022.2-English/ug994-vivado-ip-subsystems/Cross-Probing-with-Address-Editor.

[35] AMD. *Getting Started with the Vitis Software Platform (UG1400)*. 2022. URL: https://docs.amd.com/r/2022.2-English/ug1400-vitis-embedded/Getting-Started-with-the-Vitis-Software-Platform.

[36] Red Pitaya Team. *Red Pitaya FPGA Tutorial: Vivado Environment*. 2025. URL: https://redpitaya-knowledge-base.readthedocs.io/en/latest/learn_fpga/3_vivado_env/tutorfpga2.html.

[37] Red Pitaya Team. *Red Pitaya Developer Guide: SSH Access*. 2025. URL: https://redpitaya.readthedocs.io/en/latest/developerGuide/software/console/ssh/ssh.html.

[38] Xilinx Support. *Vivado 2018.3 final processing hangs at "Generating installed device list" on Ubuntu 19.04*. 2023. URL: https://support.xilinx.com/s/question/0D52E00006hpmTmSAI/vivado-20183-final-processing-hangs-at-generating-installed-device-list-on-ubuntu-1904?language=en_US.

[39] Tecmint. *How to Fix "Failed to load module canberra-gtk-module" Error*. 2024. URL: https://www.tecmint.com/failed-to-load-module-canberra-gtk-module/.

[40] Red Pitaya Documentation. *SD Card Quick Start Guide*. 2024. URL: https://redpitaya.readthedocs.io/en/latest/quickStart/SDcard/SDcard.html.

Parte II

Anexos

Apéndice A

Código HLS

Este apéndice presenta los códigos utilizados para realizar las implementaciones descritas en el documento. Los códigos se organizan de forma jerárquica, siguiendo el orden lógico de la arquitectura, para facilitar su comprensión y análisis.

A.1. Ejemplo código GRU implementación definida por la estructura

En este primer apartado se muestra un ejemplo sobre la función vector_adder de como se debe hacer uso de la estructura AXI_VALUE. Este código se a mencionado en el Apartado 4.1.1.

Header AXI_VALUE

```
#ifndef AXI_VALUE_H
#define AXI_VALUE_H

    #define bits_32  // Selector tipo de dato

    // Estructura común para AXI_VALUE
        struct AxiWord {
            #ifdef bits_32
                float data;
            #else
                half data;
            #endif
        };

    typedef AxiWord AXI_VALUE;

#endif // AXI_VALUE_H

```

Codigo C++

```

#include "vector_adder.h"

void vector_adder(hls::stream<AXI_VALUE> &stream_in1,hls::stream<AXI_VALUE> &stream_in2,
                  hls::stream<AXI_VALUE> &stream_out, int nwords){

#pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in1
```

```
#pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in2
#pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out

#pragma HLS INTERFACE s_axilite port=nwords

#pragma HLS INTERFACE mode=s_axilite port=return

    //OUTPUT
        AXI_VALUE C;

    //Lectura del vector A de entrada
        for(int i = 0; i < nwords; i++){
            C.data = stream_in1.read().data + stream_in2.read().data;    // Realiza la suma
            stream_out.write(C);                                         // Devuelve el resultado
        }
}

```

A.2. Código GRU implementación cuantificada

En este apartado se presenta de forma jerárquica el código que implementa la GRU de forma cuantificada. Donde la función *top* del proyecto es nombrado full_GRU y es la mostrada a continuación.

Este código se ha mencionado en el Apartado 4.1.1.1.

Header

```


#include "../Reset_Gate_Q/reset_gate/src/TOP/reset_gate.h"                    // RESET GATE
#include "../Update_Gate_Q/update_gate/src/TOP/update_gate.h"                 // UPDATE GATE
#include "../Candidate_gate_Q/candidate_gate/src/TOP/candidate_gate.h"         // CANDIDATE GATE
#include "../Rest_of_GRU_Q/rest_of_GRU/src/TOP/rest_of_GRU.h"                 // REST OF GRU


// GENERAL TOP
void full_GRU(hls::stream<data_in_mvm> &stream_in,hls::stream<data_in_mvm> &stream_in_out,  // STREAMS ENTRADA
              int *din_weight_rst, int *din_recurrent_rst,                                 // MATRICES RESET
              int *din_weight_cand, int *din_recurrent_cand,                               // MATRICES CANDIDATE
              int *din_weight_up, int *din_recurrent_up,                                   // MATRICES UPDATE
              hls::stream<data_out_gru> &stream_out);                                      // STREAM SALIDA
```

Código C++

```
#include "full_GRU_Q.h"

void full_GRU(hls::stream<data_in_mvm> &stream_in,hls::stream<data_in_mvm> &stream_in_out,  // STREAMS ENTRADA
              int *din_weight_rst, int *din_recurrent_rst,                                 // MATRICES RESET
              int *din_weight_cand, int *din_recurrent_cand,                               // MATRICES CANDIDATE
              int *din_weight_up, int *din_recurrent_up,                                   // MATRICES UPDATE
              hls::stream<data_out_gru> &stream_out){                                      // STREAM SALIDA

    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in_out
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out

    #pragma HLS INTERFACE mode=m_axi bundle=gmem1 depth=256 port=din_weight_rst offset=slave
    #pragma HLS INTERFACE mode=m_axi bundle=gmem2 depth=512 port=din_recurrent_rst offset=slave
```

```
15      #pragma HLS INTERFACE mode=m_axi bundle=gmem3 depth=256 port=din_weight_cand offset=slave
16      #pragma HLS INTERFACE mode=m_axi bundle=gmem4 depth=512 port=din_recurrent_cand offset=slave
17      #pragma HLS INTERFACE mode=m_axi bundle=gmem5 depth=256 port=din_weight_up offset=slave
18      #pragma HLS INTERFACE mode=m_axi bundle=gmem6 depth=512 port=din_recurrent_up offset=slave
19
20      #pragma HLS INTERFACE mode=s_axilite port=return
21
22      #pragma HLS DATAFLOW
23
24      // Streams para duplicar entradas
25          // STREAM IN -> VECTOR
26              hls::stream <data_in_mvm> stream_in_dup1;
27              hls::stream <data_in_mvm> stream_in_dup2;
28              hls::stream <data_in_mvm> stream_in_dup3;
29
30              #pragma HLS STREAM variable=stream_in_dup1 depth=1024
31              #pragma HLS STREAM variable=stream_in_dup2 depth=1024
32              #pragma HLS STREAM variable=stream_in_dup3 depth=1024
33
34          // STREAM IN OUT -> SALIDA
35              hls::stream <data_in_mvm> stream_in_out_dup1;
36              hls::stream <data_in_mvm> stream_in_out_dup2;
37              hls::stream <data_in_mvm> stream_in_out_dup3;
38              hls::stream <data_in_mvm> stream_in_out_dup4;
39
40              #pragma HLS STREAM variable=stream_in_out_dup1 depth=1024
41              #pragma HLS STREAM variable=stream_in_out_dup2 depth=1024
42              #pragma HLS STREAM variable=stream_in_out_dup3 depth=1024
43              #pragma HLS STREAM variable=stream_in_out_dup4 depth=1024
44
45
46      // Split stream_in
47          data_in_mvm temp1;
48          for( int i = 0; i < VECTOR_INDEX1; i++) {
49              temp1 = stream_in.read();
50              stream_in_dup1.write(temp1);
51              stream_in_dup2.write(temp1);
52              stream_in_dup3.write(temp1);
53          }
54
55      // Split stream_in_out
56          data_in_mvm temp2;
57          for( int i = 0; i < VECTOR_INDEX2; i++) {
58              temp2 = stream_in_out.read();
59              stream_in_out_dup1.write(temp2);
60              stream_in_out_dup2.write(temp2);
61              stream_in_out_dup3.write(temp2);
62              stream_in_out_dup4.write(temp2);
63          }
64
65
66      hls::stream <data_out_sig> Out_rst;
67      hls::stream <data_out_tanh> Out_cand;
68      hls::stream <data_out_sig> Out_up;
69
70      #pragma HLS STREAM variable=Out_rst depth=1024
71      #pragma HLS STREAM variable=Out_cand depth=1024
72      #pragma HLS STREAM variable=Out_up depth=1024
73
74      // RESET GATE
75          reset_gate(stream_in_dup1,stream_in_out_dup1,din_weight_rst,din_recurrent_rst,Out_rst);
76
77      //CANDIDATE STATE
78          candidate_gate(stream_in_dup2,stream_in_out_dup2,Out_rst,din_weight_cand,din_recurrent_cand,Out_cand);
79
80      // UPDATE GATE
81          update_gate(stream_in_dup3,stream_in_out_dup3,din_weight_up,din_recurrent_up,Out_up);
```

```
82
83      // REST OF GRU
84          rest_of_GRU(Out_cand,Out_up,stream_in_out_dup4,stream_out);
85
86  }
87
```

A.2.1. Reset Gate y Update Gate

Los códigos que implementan estas funciones TOP se basan en las mismas funciones a diferencia de su nomenclatura en el caso de la función top su nomenclatura es reset_gate y update_gate, respectivamente y en el caso de la multiplicación matriz vector se llama matrix_vector_RST y matrix_vector_UP, respectivamente.

Este código se ha mencionado en el Apartado 4.1.1.1.

Header

```
1   #ifndef RESET_GATE_H
2   #define RESET_GATE_H
3
4   #include "../matrix_vector/matrix_vector_RST.h"
5   #include "../sigmoid/sigmoid_function.h"
6   #include "../adder/vector_adder.h"
7
8   #define NWORDS 16
9
10  // TOP
11  void reset_gate(
12  hls::stream<data_in_mvm> &stream_in1,hls::stream<data_in_mvm> &stream_in2,
13  int *din1, int *din2,hls::stream<data_out_sig> &stream_out);
14
15  #endif // RESET_GATE_H
```

Código C++

```
1   #include "reset_gate.h"
2
3   void reset_gate(
4   hls::stream<data_in_mvm> &stream_in1,hls::stream<data_in_mvm> &stream_in2,
5   int *din1, int *din2,hls::stream<data_out_sig> &stream_out)
6   {
7
8   #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in1
9   #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in2
10  #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out
11
12  #pragma HLS INTERFACE mode=m_axi bundle=gmem1 depth=512 port=din1 offset=slave
13  #pragma HLS INTERFACE mode=m_axi bundle=gmem2 depth=512 port=din2 offset=slave
14
15  #pragma HLS INTERFACE mode=s_axilite port=return
16
17  #pragma HLS DATAFLOW
18
19  hls::stream <data_out_mvm> Out_mv1;
20  hls::stream <data_out_mvm> Out_mv2;
21  hls::stream <data_out_add> Out_vadd;
22
23  #pragma HLS STREAM variable=Out_mv1 depth=1024
24  #pragma HLS STREAM variable=Out_mv2 depth=1024
```

```
25  #pragma HLS STREAM variable=Out_vadd depth=1024
26
27
28  // MULT VECTOR - MATRIX
29      matrix_vector_RST(stream_in1,Out_mv1,din1,stream_in2,Out_mv2,din2);
30  // VECTOR ADDER
31      vector_adder(Out_mv1,Out_mv2,Out_vadd,NWORDS);
32  // SIGMOID FUNCTION
33      sigmoid_function(Out_vadd,stream_out,NWORDS);
34
35  }
```

A.2.1.1. Multiplicación Matriz - Vector

El contenido del siguiente código es igual para las diferentes nomenclaturas/puertas (matrix_vector_RST, matrix_vector_CAND, matrix_vector_UP). A continuación se muestra la cabecera y el código c++ correspondiente a la *reset gate*.

Este código se ha descrito en el Apartado 4.1.1.2.

Header

```
1   #include "string.h"
2   #include "ap_int.h"
3   #include "hls_stream.h"
4   #include "ap_axi_sdata.h"
5   #include "hls_math.h"
6   #include "hls_half.h"
7
8   // MVM 1
9       // MATRIZ
10          #define ROWS_A1  3
11          #define COLS_A1  16
12          #define NWORDS_M1 ROWS_A1*COLS_A1
13      //VECTOR
14          #define VECTOR_INDEX1  3
15      // SALIDA
16          #define NWORDS_V_OUT1 COLS_A1
17
18  // MVM 1
19      // MATRIZ
20          #define ROWS_A2  17
21          #define COLS_A2  16
22          #define NWORDS_M2 ROWS_A2*COLS_A2
23      //VECTOR
24          #define VECTOR_INDEX2  17
25      // SALIDA
26          #define NWORDS_V_OUT2 COLS_A2
27
28  // Tipos de datos
29      // Punto fijo ENTRADA
30          #define Q_IN_MVM 16
31          #define N_IN_MVM 8
32      // Punto fijo SALIDA
33          #define Q_OUT_MVM 37
34          #define N_OUT_MVM 16
35
36  // Formato Q8.8 con el que entraran los datos del vector, pesos y sesgos
37  using data_in_mvm = ap_fixed<Q_IN_MVM,N_IN_MVM,AP_RND_CONV,AP_SAT>;
38
39  // Formato salida Q21.16 ( no tiene perdida de precision en todoo el recorrido del mvm)
40  using data_out_mvm = ap_fixed<Q_OUT_MVM,N_OUT_MVM,AP_RND_CONV,AP_SAT>;
41
```

```
42 void matrix_vector_RST(hls::stream<data_in_mvm> &stream_in1,hls::stream<data_out_mvm> &stream_out1, int *din1,
43                         hls::stream<data_in_mvm> &stream_in2,hls::stream<data_out_mvm> &stream_out2, int *din2);
```

Código c++

```
1  #include "matrix_vector_RST.h"
2
3  void matrix_vector_RST(hls::stream<data_in_mvm> &stream_in1,hls::stream<data_out_mvm> &stream_out1, int *din1,
4                         hls::stream<data_in_mvm> &stream_in2,hls::stream<data_out_mvm> &stream_out2, int *din2){
5
6      #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in1
7      #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out1
8      #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in2
9      #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out2
10
11
12     #pragma HLS INTERFACE mode=m_axi bundle=gmem1 depth=512 port=din1 offset=slave
13     #pragma HLS INTERFACE mode=m_axi bundle=gmem2 depth=512 port=din2 offset=slave
14
15
16     #pragma HLS INTERFACE mode=s_axilite port=return
17
18
19     // MVM 1
20         // INPUTS
21             static data_in_mvm MATRIX_A[NWORDS_M1];
22             data_in_mvm VECTOR_A[VECTOR_INDEX1];
23
24         //OUTPUT
25             data_out_mvm RESULT_A;
26
27     // MVM 2
28         // INPUTS
29             static data_in_mvm MATRIX_B[NWORDS_M2];
30             data_in_mvm VECTOR_B[VECTOR_INDEX2];
31
32         //OUTPUT
33             data_out_mvm RESULT_B;
34
35     // INTERNAL VARIABLES
36         static bool first_iteration = false;
37         short int mascara;
38
39     //Guardar los datos del vector en memoria ( Lectura de los input stream)
40     for(int i = 0; i < VECTOR_INDEX1; i++){
41         VECTOR_A[i] =stream_in1.read();
42     }
43
44     for(int i = 0; i < VECTOR_INDEX2; i++){
45         VECTOR_B[i] =stream_in2.read();
46     }
47
48
49     // Guarda los datos de la matriz en memoria (weight and bias)
50     if(!first_iteration){
51         for(int i = 0; i < NWORDS_M1; i++){
52             mascara = *din1++ & 0xFFFF;
53             MATRIX_A[i]= *reinterpret_cast<data_in_mvm*> (&mascara);
54         }
55
56         for(int i = 0; i < NWORDS_M2; i++){
57             mascara = *din2++ & 0xFFFF;
58             MATRIX_B[i]= *reinterpret_cast<data_in_mvm*> (&mascara);
59         }
60         first_iteration = true;
```

```
61      }
62
63      // Multiplicacion matriz vector
64      for( int i = 0; i < COLS_A1; i++){
65          #pragma HLS PIPELINE
66          RESULT_A = 0;
67          for(int j = 0; j < ROWS_A1; j++){
68              RESULT_A += MATRIX_A[j + i*ROWS_A1] * VECTOR_A[j];
69          }
70          stream_out1.write(RESULT_A);        // Sacar valor de resultado A
71      }
72
73
74      for( int i = 0; i < COLS_A2; i++){
75          #pragma HLS PIPELINE
76          RESULT_B = 0;
77          for(int j = 0; j < ROWS_A2; j++){
78              RESULT_B += MATRIX_B[j + i*ROWS_A2] * VECTOR_B[j];
79          }
80          stream_out2.write(RESULT_B);        // Sacar valor de resultado B
81      }
82
83  }
```

A.2.1.2. Suma de vectores

Este código se ha descrito en el Apartado 4.1.1.3.

Header

```
1   #ifndef VECTOR_ADDER_H
2   #define VECTOR_ADDER_H
3
4   #include "string.h"
5   #include "ap_int.h"
6   #include "hls_stream.h"
7   #include "ap_axi_sdata.h"
8   #include "hls_math.h"
9   #include "hls_half.h"
10
11  // Punto fijo ENTRADA
12      #define Q_IN_ADD 37
13      #define N_IN_ADD 16
14      using data_in_add = ap_fixed<Q_IN_ADD,N_IN_ADD,AP_RND_CONV,AP_SAT>;
15  // Punto fijo SALIDA
16      #define Q_OUT_ADD 16
17      #define N_OUT_ADD 4
18      using data_out_add = ap_fixed<Q_OUT_ADD,N_OUT_ADD,AP_RND_CONV,AP_SAT>;
19
20  void vector_adder( hls::stream<data_in_add> &stream_in1,hls::stream<data_in_add> &stream_in2,
21                     hls::stream<data_out_add> &stream_out,int nwords);
22
23  #endif // VECTOR_ADDER_H
```

Código C++

```
1   #include "vector_adder.h"
2
3
4   void vector_adder(hls::stream<data_in_add> &stream_in1,hls::stream<data_in_add> &stream_in2,
5                     hls::stream<data_out_add> &stream_out, int nwords){
6
7       #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in1
```

```
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in2
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out

    #pragma HLS INTERFACE s_axilite port=nwords

    #pragma HLS INTERFACE mode=s_axilite port=return

    //OUTPUT
        data_out_add C;

    for(int i = 0; i < nwords; i++){
        C = stream_in1.read() + stream_in2.read();      // Realiza la suma
        stream_out.write(C);                            // Devuelve el resultado
    }

}
```

A.2.1.3. Función sigmoide

Este código se ha descrito en el Apartado 4.1.1.5.

Header

```
#ifndef SIGMOID_FUNCTION_H
#define SIGMOID_FUNCTION_H

#include "string.h"
#include "ap_int.h"
#include "hls_stream.h"
#include "ap_axi_sdata.h"
#include "hls_math.h"
#include "hls_half.h"

// SIGMOID
    // Punto fijo ENTRADA
        #define Q 16
        #define N 4

// Tamaño memoria
    const unsigned SIZE_sig_mem = 13;
    const unsigned MEM_SIZE_SIG = 1<<SIZE_sig_mem;

// Tipos de datos precalculo
using memory_index = ap_uint<SIZE_sig_mem>; // Indice de la memoria     Formato: SIZE bits
using data_in_sig = ap_fixed<Q,N,AP_RND_CONV,AP_SAT>;
using data_in_Q = ap_fixed<SIZE_sig_mem,4,AP_RND_CONV,AP_SAT>;
// data_in_Q: Es el tipo de dato con el que se buscará el indice
// y con el que se indica donde se guardan los datos en la memoria

using data_out_sig = ap_fixed<Q,1,AP_RND_CONV,AP_SAT>;  // Este es el tipo de dato que estara guardado en la memoria

// Valores de saturacion
    const float SIGMOID_MIN = -8.0f;
    const float SIGMOID_MAX = 8.0f;
// Steps
    const float step_SIG = (SIGMOID_MAX - SIGMOID_MIN)/MEM_SIZE_SIG;

void sigmoid_function(hls::stream<data_in_sig> &stream_in,hls::stream<data_out_sig> &stream_out,int NWORDS);

#endif // SIGMOID_FUNCTION_H

```

Código C++

```
#include "sigmoid_function.h"

// Función para inicializar la tabla de búsqueda
static void init_sigmoid_lut(data_out_sig memory[MEM_SIZE_SIG]) {
    float x;
    for (int i = 0; i < MEM_SIZE_SIG; i++) {
        // Funciona como un acumulador que va aumentando el valor de X el valor step.
            x = SIGMOID_MIN + step_SIG * i;
        // Calculo de la sigmoide
            float y =  1.0f / (1.0f + exp(-x));
        // El dato de entrada lo convierte al tipo de dato de entrada
        // NO ES NECESARIO GASTAR RANGE porque tienen los mismos bits enteros
            data_in_Q valor_rom = x;
        // Obtienes la direccion de memoria con los 13 bits correspondientes al dato de entrada.
        // Es necesario indicar el rango porque sino solo coge la parte entera.
            memory_index rom_index = valor_rom.range(SIZE_sig_mem-1,0);
        // En la direccion de memoria indicada por el dato de entrada se guarda el valor de y pero con el formato deseado
            memory[rom_index] = data_out_sig (y);
    }
}

data_out_sig sigmoid_compute(data_in_Q x){
    static data_out_sig sigmoid_memory[MEM_SIZE_SIG];
    // Inicialización de la memoria
        init_sigmoid_lut(sigmoid_memory);
    // Con el valor de entrada, sus 13 bits seran los que servirán para encontrar el indice de la ROM.
    // Es necesario indicar el rango porque sino solo coge la parte entera
        memory_index index_search = x.range(SIZE_sig_mem-1,0);
    // Devolvemos el valor de la ROM correspondiente a la entrada
        return(sigmoid_memory[index_search]);
}

void sigmoid_function(hls::stream<data_in_sig> &stream_in,hls::stream<data_out_sig> &stream_out,int NWORDS){

    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out

    #pragma HLS INTERFACE s_axilite port=NWORDS

    #pragma HLS INTERFACE mode=s_axilite port=return

    // INPUTS
        data_in_sig vector_in;

    //OUTPUT
        data_out_sig vector_out;

    //Variables internas
        data_in_Q input;

    //Lectura del vector de entrada
        for(int i = 0; i < NWORDS; i++){
            vector_in = stream_in.read();
            // Hacemos una saturación al formato correspondiente de guardado y busqueda de indice  de la ROM
                input = vector_in;
            // Obtención de la salida
                vector_out = sigmoid_compute(input);
            // Enviar los resultados a stream_out
                stream_out.write(vector_out);
        }
}
```

A.2.2. Candidate State

Este código se ha descrito en el Apartado 4.1.1.1.

Header

```
#ifndef CANDIDATE_STATE_H
#define CANDIDATE_STATE_H

#include "../matrix_vector/matrix_vector_CAND.h"
#include "../tanh/tanh_function.h"
#include "../adder/vector_adder.h"
#include "../product/vector_product.h"

#define NWORDS 16

// TOP
void candidate_gate(hls::stream<data_in_mvm> &stream_in1,hls::stream<data_in_mvm> &stream_in2,
hls::stream<data_in2_prod> &stream_in3, int *din1, int *din2,hls::stream<data_out_tanh> &stream_out);

#endif // CANDIDATE_STATE_H
```

Código C++

```

#include "candidate_gate.h"

void candidate_gate(hls::stream<data_in_mvm> &stream_in1,hls::stream<data_in_mvm> &stream_in2,
hls::stream<data_in2_prod> &stream_in3, int *din1, int *din2,hls::stream<data_out_tanh> &stream_out){

    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in1
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in2
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in3
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out

    #pragma HLS INTERFACE mode=m_axi bundle=gmem1 depth=256 port=din1 offset=slave
    #pragma HLS INTERFACE mode=m_axi bundle=gmem2 depth=512 port=din2 offset=slave

    #pragma HLS INTERFACE mode=s_axilite port=return

    #pragma HLS DATAFLOW
    hls::stream <data_out_mvm> Out_mv1;
    hls::stream <data_out_mvm> Out_mv2;
    hls::stream <data_out_prod> Out_vpro;
    hls::stream <data_out_add> Out_vadd;

    #pragma HLS STREAM variable=Out_mv1 depth=1024
    #pragma HLS STREAM variable=Out_mv2 depth=1024
    #pragma HLS STREAM variable=Out_vpro depth=1024
    #pragma HLS STREAM variable=Out_vadd depth=1024


    // MULT VECTOR - MATRIX
        matrix_vector_CAND(stream_in1,Out_mv1,din1,stream_in2,Out_mv2,din2);
    // PRODUCT
        vector_product(Out_mv2,stream_in3,Out_vpro,NWORDS);
    // VECTOR ADDER
        vector_adder(Out_mv1,Out_vpro,Out_vadd,NWORDS);
    // SIGMOID FUNCTION
        tanh_function(Out_vadd,stream_out,NWORDS);

}

```

A.2.2.1. Multiplicación de vectores

Este código se ha descrito en el Apartado 4.1.1.3.

Header

```
#ifndef VECTOR_PRODUCT_H
#define VECTOR_PRODUCT_H

#include "string.h"
#include "ap_int.h"
#include "hls_stream.h"
#include "ap_axi_sdata.h"
#include "hls_math.h"
#include "hls_half.h"

// Tipos de datos

#define Q_IN1_PROD 37
#define N_IN1_PROD 16

#define Q_IN2_PROD 16
#define N_IN2_PROD 1

using data_in1_prod = ap_fixed<Q_IN1_PROD,N_IN1_PROD,AP_RND_CONV,AP_SAT>;    // Formato entrada Q21.16 (Salida mvm)
using data_in2_prod = ap_fixed<Q_IN2_PROD,N_IN2_PROD,AP_RND_CONV,AP_SAT>;    // Formato entrada Q2.14 (Salida reset gate)
using data_out_prod = ap_fixed<Q_IN1_PROD,N_IN1_PROD,AP_RND_CONV,AP_SAT>;    // Formato salida Q21.16  mismo formato
                                                                             // debido a valores por los que multiplica

void vector_product(hls::stream<data_in1_prod> &stream_in1,hls::stream<data_in2_prod> &stream_in2,
hls::stream<data_out_prod> &stream_out,int nwords);

#endif // VECTOR_PRODUCT_H

```

Código c++

```

#include "vector_product.h"

void vector_product(hls::stream<data_in1_prod> &stream_in1,hls::stream<data_in2_prod> &stream_in2,
hls::stream<data_out_prod> &stream_out, int nwords){

    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in1
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in2
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out

    #pragma HLS INTERFACE s_axilite port=nwords

    #pragma HLS INTERFACE mode=s_axilite port=return

    //OUTPUT
        data_out_prod C;

    for(int i = 0; i < nwords; i++){
        C=stream_in1.read() * stream_in2.read();    // Realiza el producto
        stream_out.write(C);                        // Devuelve el resultado
    }
}

```

A.2.2.2. Función tangente hiperbólica

Este código se ha descrito en el Apartado 4.1.1.5.

Header

```
#ifndef TANH_FUNCTION_H
#define TANH_FUNCTION_H

#include "string.h"
#include "ap_int.h"
#include "hls_stream.h"
#include "ap_axi_sdata.h"
#include "hls_math.h"
#include "hls_half.h"

// TANH
    // Punto fijo ENTRADA
        #define Q 16
        #define N 4

// Tamaño memoria
    const unsigned SIZE = 13;
    const unsigned MEM_SIZE = 1<<SIZE;

// Tipos de datos precalculo
using memory_index = ap_uint<SIZE>;     // Indice de la memoria Formato: SIZE bits
using data_in_tanh = ap_fixed<Q,N,AP_RND_CONV,AP_SAT>;
using data_in_tanh_Q = ap_fixed<SIZE,4,AP_RND_CONV,AP_SAT>;
// data_in_tanh_Q: Es el tipo de dato con el que se buscará el indice
// y con el que se indica donde se guardan los datos en la memoria
using data_out_tanh = ap_fixed<Q,1,AP_RND_CONV,AP_SAT>; // Este es el tipo de dato que estara guardado en la memoria

// Valores de saturacion
    const float TANH_MIN = -8.0f;
    const float TANH_MAX = 8.0f;
// Steps
    const float step = (TANH_MAX - TANH_MIN)/MEM_SIZE;

void tanh_function(hls::stream<data_in_tanh> &stream_in,hls::stream<data_out_tanh> &stream_out,int NWORDS_tanh);

#endif // TANH_FUNCTION_H
```

Código C++

```
#include "tanh_function.h"

// Función para inicializar la tabla de búsqueda
static void init_tanh_lut(data_out_tanh memory[MEM_SIZE]) {
    float x;
    for (int i = 0; i < MEM_SIZE; i++) {
        // Funciona como un acumulador que va aumentando el valor de X el valor step.
            x = TANH_MIN + step * i;
        // Calculo de la sigmoide
            float y =  (exp(2*x)-1)/(exp(2*x)+1);
        // El dato de entrada lo convierte al tipo de dato de entrada
        // NO ES NECESARIO GASTAR RANGE porque tienen los mismos bits enteros
            data_in_tanh_Q valor_rom = x;
        // Obtienes la direccion de memoria con los 13 bits correspondientes al dato de entrada.
        // Es necesario indicar el rango porque sino solo coge la parte entera
            memory_index rom_index = valor_rom.range(SIZE-1,0);
        // En la direccion de memoria indicada por el dato de entrada se guarda el valor de y pero con el formato deseado
            memory[rom_index] = data_out_tanh (y);
    }

}

data_out_tanh tanh_compute(data_in_tanh_Q x){
    static data_out_tanh sigmoid_memory[MEM_SIZE];
    // Inicialización de la memoria
        init_tanh_lut(sigmoid_memory);
    // Con el valor de entrada, sus 13 bits seran los que servirán para encontrar el indice de la ROM.
```

```
    // Es necesario indicar el rango porque sino solo coge la parte entera
        memory_index index_search = x.range(SIZE-1,0);
    // Devolvemos el valor de la ROM correspondiente a la entrada
        return(sigmoid_memory[index_search]);
}

void tanh_function(hls::stream<data_in_tanh> &stream_in,hls::stream<data_out_tanh> &stream_out,int NWORDS_tanh){

    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out

    #pragma HLS INTERFACE s_axilite port=NWORDS_tanh

    #pragma HLS INTERFACE mode=s_axilite port=return

    // INPUTS
        data_in_tanh vector_in;

    //OUTPUT
        data_out_tanh vector_out;

    //Variables internas
        data_in_tanh_Q input;


    for(int i = 0; i < NWORDS_tanh; i++){
        vector_in = stream_in.read();
        input = vector_in;                  // Hacemos una saturación al formato correspondiente de guardado
                                            // y busqueda de indice  de la ROM
        vector_out = tanh_compute(input);   // Obtención de la salida
        stream_out.write(vector_out);       // Enviar los resultados a stream_out
    }
}
```

A.2.3. Rest of GRU

Este código se ha descrito en el apartado 4.1.1.1.

Header

```
#ifndef REST_OF_GRU_H
#define REST_OF_GRU_H

#include "../subtractor/vector_subtractor.h"
#include "../adder_roG/vector_adder_roG.h"
#include "../product_out_up/vector_product_out_up.h"
#include "../product_sub_cand/vector_product_sub_cand.h"

#define NWORDS 16
// TOP
void rest_of_GRU(hls::stream<data_in_prod_sub_cand> &stream_in_candidate,hls::stream<data_in_out_sub> &stream_in_update,
hls::stream<data_in_prod_out> &stream_in_out,hls::stream<data_out_gru> &stream_out);

#endif // REST_OF_GRU_H
```

Código C++

```
#include "rest_of_GRU.h"

void rest_of_GRU(hls::stream<data_in_prod_sub_cand> &stream_in_candidate,hls::stream<data_in_out_sub> &stream_in_update,
hls::stream<data_in_prod_out> &stream_in_out,hls::stream<data_out_gru> &stream_out){

    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in_candidate
```

```
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in_update
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in_out
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out

    #pragma HLS INTERFACE mode=s_axilite port=return

    #pragma HLS DATAFLOW

    hls::stream <data_in_out_sub> Out_sub;
    hls::stream <data_out_prod_out_up> Out_m1;
    hls::stream <data_out_prod_sub_cand> Out_m2;


    // Streams duplicados
        hls::stream<data_in_out_sub> stream_update_dup1;
        hls::stream<data_in_out_sub> stream_update_dup2;

    #pragma HLS STREAM variable=Out_sub depth=1024
    #pragma HLS STREAM variable=Out_m1 depth=1024
    #pragma HLS STREAM variable=Out_m2 depth=1024


    #pragma HLS STREAM variable=stream_update_dup1 depth=1024
    #pragma HLS STREAM variable=stream_update_dup2 depth=1024

    // DUPLICATE stream_in_update
    data_in_out_sub temp;
    for( int i = 0; i < NWORDS; i++) {
        temp = stream_in_update.read();
        stream_update_dup1.write(temp);
        stream_update_dup2.write(temp);
    }

    // SUBTRACTOR
        vector_subtractor(stream_update_dup1,Out_sub,NWORDS);
    // PRODUCT SUBSTRACTOR * CANDIDATE
        vector_product_sub_cand(stream_in_candidate,Out_sub,Out_m2,NWORDS);
    // PRODUCT OUTPUT * UPDATE
        vector_product_out_up(stream_in_out,stream_update_dup2,Out_m1,NWORDS);
    // VECTOR ADDER
        vector_adder_roG(Out_m2,Out_m1,stream_out,NWORDS);
}

```

A.2.3.1. Resta Constante - Vector

Este código se ha descrito en el Apartado 4.1.1.4.

Header

```
#ifndef VECTOR_SUBSTRACTOR_H
#define VECTOR_SUBSTRACTOR_H

#include "string.h"
#include "ap_int.h"
#include "hls_stream.h"
#include "ap_axi_sdata.h"
#include "hls_math.h"
#include "hls_half.h"

// Punto fijo ENTRADA
    #define Q_SUB 16
    #define N_SUB 1

```

```
using data_in_out_sub = ap_fixed<Q_SUB,N_SUB,AP_RND_CONV,AP_SAT>;    // Formato dato de entrada restador
using data_uno = ap_fixed<Q_SUB,N_SUB+1,AP_RND_CONV,AP_SAT>;         // Formato dato de salida restador

void vector_subtractor(hls::stream<data_in_out_sub> &stream_in, hls::stream<data_in_out_sub> &stream_out,int nwords);

#endif // VECTOR_SUBSTRACTOR_H

```

Código c++

```
#include "vector_subtractor.h"

void vector_subtractor(hls::stream<data_in_out_sub> &stream_in, hls::stream<data_in_out_sub> &stream_out, int nwords){

    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out

    #pragma HLS INTERFACE s_axilite port=nwords

    #pragma HLS INTERFACE mode=s_axilite port=return

        data_uno UNO = 1.0;

    //OUTPUT
        data_in_out_sub C;

    for(int i = 0; i < nwords; i++){
        C = UNO - stream_in.read();      // Realiza la resta
        stream_out.write(C);             // Devuelve el resultado
    }
}
```

A.2.3.2. Multiplicación vectores Resta - Candidate State

Este código se ha descrito en el Apartado 4.1.1.3.

Header

```
#ifndef VECTOR_PROD_SUB_CAND_H
#define VECTOR_PROD_SUB_CAND_H

#include "string.h"
#include "ap_int.h"
#include "hls_stream.h"
#include "ap_axi_sdata.h"
#include "hls_math.h"
#include "hls_half.h"

// Punto fijo ENTRADA
    #define Q_IN_PROD_SUBCAND 16
    #define N_IN_PROD_SUBCAND 1
// Punto fijo SALIDA
    #define Q_OUT_PROD_SUBCAND 32
    #define N_OUT_PROD_SUBCAND 2

// Formato entrada de datos provinientes de la salida de la resta y candidate
using data_in_prod_sub_cand = ap_fixed<Q_IN_PROD_SUBCAND,N_IN_PROD_SUBCAND,AP_RND_CONV,AP_SAT>;
using data_out_prod_sub_cand = ap_fixed<Q_OUT_PROD_SUBCAND,N_OUT_PROD_SUBCAND,AP_RND_CONV,AP_SAT>;

void vector_product_sub_cand(hls::stream<data_in_prod_sub_cand> &stream_in1,
hls::stream<data_in_prod_sub_cand> &stream_in2, hls::stream<data_out_prod_sub_cand> &stream_out,int nwords);

#endif // VECTOR_PROD_SUB_CAND_H
```

Código C++

```cpp
#include "vector_product_sub_cand.h"

void vector_product_sub_cand(hls::stream<data_in_prod_sub_cand> &stream_in1,
hls::stream<data_in_prod_sub_cand> &stream_in2, hls::stream<data_out_prod_sub_cand> &stream_out, int nwords){

    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in1
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in2
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out

    #pragma HLS INTERFACE s_axilite port=nwords

    #pragma HLS INTERFACE mode=s_axilite port=return

    //OUTPUT
        data_out_prod_sub_cand C;

    for(int i = 0; i < nwords; i++){
        C = stream_in1.read() * stream_in2.read();  // Realiza el producto
        stream_out.write(C);                        // Devuelve el resultado
    }
}
```

A.2.3.3. Multiplicación vectores Salida previa - Update Gate

Este código se ha descrito en el Apartado 4.1.1.3.

Header

```cpp
#ifndef VECTOR_PROD_OUT_UP_H
#define VECTOR_PROD_OUT_UP_H

#include "string.h"
#include "ap_int.h"
#include "hls_stream.h"
#include "ap_axi_sdata.h"
#include "hls_math.h"
#include "hls_half.h"

// Punto fijo
    // Entrada OUT
        #define Q_IN_PROD_OUT 16
        #define N_IN_PROD_OUT 8
    // Entrada UP
        #define Q_IN_PROD_UP 16
        #define N_IN_PROD_UP 1

    //SALIDA = suma de bits
        #define Q_OUT_PROD_OUTPUP 32
        #define N_OUT_PROD_OUTUP 9

// Formato dato de entrada de datos provinientes de la salida
    using data_in_prod_out = ap_fixed<Q_IN_PROD_OUT,N_IN_PROD_OUT,AP_RND_CONV,AP_SAT>;
// Formato dato de entrada proviniente de salida update gate (sigmoid)
    using data_in_prod_up = ap_fixed<Q_IN_PROD_UP,N_IN_PROD_UP,AP_RND_CONV,AP_SAT>;

using data_out_prod_out_up = ap_fixed<Q_OUT_PROD_OUTPUP,N_OUT_PROD_OUTUP,AP_RND_CONV,AP_SAT>;

void vector_product_out_up(hls::stream<data_in_prod_out> &stream_in1,
hls::stream<data_in_prod_up> &stream_in2, hls::stream<data_out_prod_out_up> &stream_out,int nwords);

#endif // VECTOR_PROD_OUT_UP_H

```

Código C++

```
#include "vector_product_out_up.h"

void vector_product_out_up(hls::stream<data_in_prod_out> &stream_in1,
hls::stream<data_in_prod_up> &stream_in2, hls::stream<data_out_prod_out_up> &stream_out, int nwords){

    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in1
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in2
    #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out

    #pragma HLS INTERFACE s_axilite port=nwords

    #pragma HLS INTERFACE mode=s_axilite port=return

    //OUTPUT
        data_out_prod_out_up C;
        data_in_prod_out garbage;

    for (int j = 0; j < 1 ;j++){
        garbage = stream_in1.read();    // Desperdicio del primer valor de v_in_out ya que no forma parte
                                        // del vector salida
    }

    for(int i = 0; i < nwords; i++){
        C=stream_in1.read() * stream_in2.read();    // Realiza el producto
        stream_out.write(C);                        // Devuelve el resultado
    }
}

```

A.2.3.4. Suma de vectores (Salida GRU)

Este código se ha descrito en el Apartado 4.1.1.3.

Header

```
#ifndef VECTOR_ADDER_roG_H
#define VECTOR_ADDER_roG_H

#include "string.h"
#include "ap_int.h"
#include "hls_stream.h"
#include "ap_axi_sdata.h"
#include "hls_math.h"
#include "hls_half.h"

// Punto fijo
    // ENTRADA CAND * SUBSTRACTOR
        #define Q_IN_PROD1 32
        #define N_IN_PROD1 2
    // ENTRADA UPDATE * OUT
        #define Q_IN_PROD2 32
        #define N_IN_PROD2 9
    //SALIDA
        #define Q_OUT_GRU 16
        #define N_OUT_GRU 8
// Formato de entrada proviniente de salida (product_cand_out)
    using data_in_add1 = ap_fixed<Q_IN_PROD1,N_IN_PROD1,AP_RND_CONV,AP_SAT>;
// Formato de entrada proviniente de salida  (product_out_up)
    using data_in_add2 = ap_fixed<Q_IN_PROD2,N_IN_PROD2,AP_RND_CONV,AP_SAT>;
// Formato de salida de la GRU
    using data_out_gru = ap_fixed<Q_OUT_GRU,N_OUT_GRU,AP_RND_CONV,AP_SAT>;^^I

```

```
28 void vector_adder_roG(hls::stream<data_in_add1> &stream_in1,hls::stream<data_in_add2> &stream_in2,
29 hls::stream<data_out_gru> &stream_out,int nwords);
30
31 #endif // VECTOR_ADDER_roG_H
32
```

Código C++

```
1  #include "vector_adder_roG.h"
2
3  void vector_adder_roG(hls::stream<data_in_add1> &stream_in1,hls::stream<data_in_add2> &stream_in2,
4  hls::stream<data_out_gru> &stream_out, int nwords){
5
6      #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in1
7      #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_in2
8      #pragma HLS INTERFACE mode=axis register_mode=both register depth=1024 port=stream_out
9
10     #pragma HLS INTERFACE s_axilite port=nwords
11
12     #pragma HLS INTERFACE mode=s_axilite port=return
13
14     ap_fixed <40,10> sum;
15
16     //OUTPUT
17         data_out_gru C;
18
19     for(int i = 0; i < nwords; i++){
20         sum=stream_in1.read() + stream_in2.read();  // Realiza la suma
21         C = sum;                                    // Cambiar a formato de salida requerido Q8.8
22         stream_out.write(C);                        // Devuelve el resultado
23     }
24 }
25
```

A.2.4. Testbench general de la GRU

Código C++

```
1  #include "../src/full_GRU_Q.h"
2
3  #define NWORDS 16
4
5  float ini = -0.6f;
6  float step_tb =0.07f;
7
8  // Función para inicializar la matriz y el vector
9  void initialize_data_matrix(int *din1, int matrix1[][COLS_A1],
10                             int *din2, int matrix2[][COLS_A1],
11                             int *din3, int matrix3[][COLS_A1],
12                             int rows_a, int cols_a) {
13
14     // Inicializar la matriz
15     for (int i = 0; i < cols_a; i++) {
16         for (int j = 0; j < rows_a; j++) {
17             // Reset
18                 matrix1[j][i] = 15 *i;
19                 din1[i * rows_a + j] = matrix1[j][i];
20             // Candidate
21                 matrix2[j][i] = 6 * i + 3;
22                 din2[i * rows_a + j] = matrix2[j][i];
23             // Update
24                 matrix3[j][i] = 3 * i + 25;
25                 din3[i * rows_a + j] = matrix3[j][i];
```

```
        }
    }

}

void initialize_data_vector(data_in_mvm *vector_in,data_in_mvm *vector_in_out) {

    // Inicializar el vector de entrada IN ( FASE ANGULO)
    for (int i = 0; i < VECTOR_INDEX1; i++) {
        vector_in[i] = (i/16.0)+ 0.275f;
    }

    // Inicializar el vector de entrada que proviene de salida OUT
    for (int i = 0; i < VECTOR_INDEX2; i++) {
           vector_in_out[i] = (i/16.0)- 0.325f;
       }
}

void print_matrix_vector(int matrixA[][COLS_A1],data_in_mvm *vectorA,int matrixB[][COLS_A1],data_in_mvm *vectorB,
data_in_mvm MATRIX_A[][COLS_A1],data_in_mvm MATRIX_B[][COLS_A1]){

    std::cout<< "MATRIZ A: "<<std::endl;
    for(int i = 0; i < ROWS_A1; i++){
        for(int j = 0; j < COLS_A1; j++){
            short int mascara = matrixA[i][j] & 0xFFFF;
            MATRIX_A[i][j] = *reinterpret_cast<data_in_mvm*> (&mascara);
            std::cout<< MATRIX_A[i][j] << " ";
        }
        std::cout<<std::endl;
    }

    std::cout<< "VECTOR A: "<<std::endl;
    for(int i = 0; i < VECTOR_INDEX1; i++){
        std::cout<<vectorA[i]<<" ";
    }
    std::cout<<std::endl;
    std::cout<<std::endl;

    // MVM 2
    std::cout<< "MATRIZ B: "<<std::endl;
    for(int i = 0; i < ROWS_A2; i++){
        for(int j = 0; j < COLS_A2; j++){
            short int mascara = matrixB[i][j] & 0xFFFF;
            MATRIX_B[i][j] = *reinterpret_cast<data_in_mvm*> (&mascara);
            std::cout<< MATRIX_B[i][j] << " ";
        }
        std::cout<<std::endl;
    }

    std::cout<< "VECTOR B: "<<std::endl;
    for(int i = 0; i < VECTOR_INDEX2; i++){
        std::cout<<vectorB[i]<<" ";
    }
    std::cout<<std::endl;
    std::cout<<std::endl;
}

// RESULTADOS SW
void rst_gate_SW(data_in_mvm matrix_weight[][COLS_A1],data_in_mvm matrix_recurrent[][COLS_A2],data_in_mvm *vector_in,
data_in_mvm *vector_in_out,float *out_rst_gate){

    float result_MVM1[NWORDS_V_OUT1]={0};   // Resultado SW  MVM 1
    float result_MVM2[NWORDS_V_OUT2]={0};   // Resultado SW MVM 2
    float result_adder[NWORDS_V_OUT2]={0};  // Resultado SW adder

    std::cout << "/***********************************************/" << std::endl;
    std::cout << "                  SW RESET GATE                  " << std::endl;
```

```
93      std::cout << "/*********************************************/" << std::endl;
94
95      // MATRIX * VECTOR
96      std::cout << "^^IResults MVM1: " << std::endl;
97      for(int i = 0; i < COLS_A1; i++){
98          for(int j = 0; j < ROWS_A1; j++){
99              result_MVM1[i] += float(matrix_weight[j][i]) * float(vector_in[j]);
100         }
101         std::cout << result_MVM1[i] << " " ;
102     }
103     std::cout << std::endl;
104
105     std::cout << " Results MVM2: " << std::endl;
106     for(int i = 0; i < COLS_A2; i++){
107         for(int j = 0; j < ROWS_A2; j++){
108             result_MVM2[i] += float(matrix_recurrent[j][i]) * float(vector_in_out[j]);
109         }
110         std::cout << result_MVM2[i] << " ";
111     }
112     std::cout << std::endl;
113
114     // VECTOR ADDER
115     std::cout << " Results VECTOR ADDER: " << std::endl;
116     for(int i = 0; i < NWORDS; i++){
117         result_adder[i] = result_MVM1[i] + result_MVM2[i];
118         std::cout << result_adder[i] << " ";
119     }
120     std::cout << std::endl;
121
122     // SIGMOID FUNCTION
123     std::cout << " Results SIGMOID: " << std::endl;
124     for(int i = 0; i < NWORDS; i++){
125         out_rst_gate[i] = 1.0f / (1.0f + exp(-result_adder[i]));
126         std::cout << out_rst_gate[i] << " ";
127     }
128     std::cout << std::endl;
129
130     // Resultado obtenido por SW
131     std::cout << " Results RESET GATE: " << std::endl;
132     for(int i = 0; i < NWORDS; i++){
133         std::cout<< out_rst_gate[i] << " ";
134     }
135
136     std::cout << std::endl<< std::endl;
137
138 }
139
140 void cand_state_SW(data_in_mvm matrix_weight[][COLS_A1],data_in_mvm matrix_recurrent[][COLS_A2],float *in_rst_gate,
141 data_in_mvm *vector_in,data_in_mvm *vector_in_out,float *out_cand_state){
142
143     float result_MVM1[NWORDS_V_OUT1]={0};       // Resultado SW  MVM 1
144     float result_MVM2[NWORDS_V_OUT2]={0};       // Resultado SW MVM 2
145     float result_product[NWORDS_V_OUT2]={0};    // Resultado SW product
146     float result_adder[NWORDS_V_OUT2]={0};      // Resultado SW adder
147
148     std::cout << "/*********************************************/" << std::endl;
149     std::cout << "                SW CANDIDATE STATE              " << std::endl;
150     std::cout << "/*********************************************/" << std::endl;
151
152     // MATRIX * VECTOR
153             std::cout << "  Results MVM1: " << std::endl;
154             for(int i = 0; i < COLS_A1; i++){
155                 for(int j = 0; j < ROWS_A1; j++){
156                     result_MVM1[i] += float(matrix_weight[j][i]) * float(vector_in[j]);
157                 }
158                 std::cout << result_MVM1[i] << " ";
159             }
```

```
160             std::cout << std::endl;
161
162             std::cout << "  Results MVM2: " << std::endl;
163             for(int i = 0; i < COLS_A2; i++){
164                 for(int j = 0; j < ROWS_A2; j++){
165                     result_MVM2[i] += float(matrix_recurrent[j][i]) * float(vector_in_out[j]);
166                 }
167                 std::cout << result_MVM2[i] << " ";
168             }
169             std::cout << std::endl;
170
171     // VECTOR PRODUCT
172         std::cout << "  Results PRODUCT: " << std::endl;
173         for(int i = 0; i < NWORDS; i++){
174             result_product[i] = in_rst_gate[i]  * result_MVM2[i];
175             std::cout << result_product[i] << " ";
176         }
177         std::cout << std::endl;
178
179     // VECTOR ADDER
180         std::cout << "  Results ADDER: " << std::endl;
181         for(int i = 0; i < NWORDS; i++){
182             result_adder[i] = result_MVM1[i] + result_product[i];
183             std::cout << result_adder[i] << " " ;
184         }
185         std::cout << std::endl;
186
187     // TANH FUNCTION
188         std::cout << "  Results TANH: " << std::endl;
189         for(int i = 0; i < NWORDS; i++){
190             out_cand_state[i] = (exp(2*result_adder[i])-1)/(exp(2*result_adder[i])+1);
191             std::cout << out_cand_state[i] << " ";
192         }
193         std::cout << std::endl;
194
195         // Resultado obtenido por SW
196         std::cout << "Resultados SW CANDIDATE STATE" << std::endl;
197         for(int i = 0; i < NWORDS; i++){
198             std::cout<< out_cand_state[i] << " ";
199             }
200
201         std::cout << std::endl<< std::endl;
202
203 }
204
205 void upd_gate_SW(data_in_mvm matrix_weight[][COLS_A1],data_in_mvm matrix_recurrent[][COLS_A2],data_in_mvm *vector_in,
206 data_in_mvm *vector_in_out,float *out_rst_gate){
207
208     float result_MVM1[NWORDS_V_OUT1]={0};       // Resultado SW  MVM 1
209     float result_MVM2[NWORDS_V_OUT2]={0};       // Resultado SW MVM 2
210     float result_adder[NWORDS_V_OUT2]={0};      // Resultado SW adder
211
212     std::cout << "/***********************************************/" << std::endl;
213     std::cout << "                  SW UPDATE GATE                 " << std::endl;
214     std::cout << "/***********************************************/" << std::endl;
215
216     // MATRIX * VECTOR
217         std::cout << "  Results MVM1: " << std::endl;
218         for(int i = 0; i < COLS_A1; i++){
219             for(int j = 0; j < ROWS_A1; j++){
220                 result_MVM1[i] += float(matrix_weight[j][i]) * float(vector_in[j]);
221             }
222             std::cout << result_MVM1[i] << " " ;
223         }
224         std::cout << std::endl;
225
226         std::cout << "  Results MVM2: " << std::endl;
```

```
227         for(int i = 0; i < COLS_A2; i++){
228             for(int j = 0; j < ROWS_A2; j++){
229                 result_MVM2[i] += float(matrix_recurrent[j][i]) * float(vector_in_out[j]);
230             }
231             std::cout << result_MVM2[i] << " ";
232         }
233         std::cout << std::endl;
234 
235     // VECTOR ADDER
236         std::cout << "  Results VECTOR ADDER: " << std::endl;
237         for(int i = 0; i < NWORDS; i++){
238             result_adder[i] = result_MVM1[i] + result_MVM2[i];
239             std::cout << result_adder[i] << " ";
240         }
241         std::cout << std::endl;
242 
243     // SIGMOID FUNCTION
244         std::cout << "  Results SIGMOID: " << std::endl;
245         for(int i = 0; i < NWORDS; i++){
246             out_rst_gate[i] = 1.0f / (1.0f + exp(-result_adder[i]));
247             std::cout << out_rst_gate[i] << " ";
248         }
249         std::cout << std::endl;
250 
251         // Resultado obtenido por SW
252         std::cout << "  Results RESET GATE: " << std::endl;
253         for(int i = 0; i < NWORDS; i++){
254             std::cout<< out_rst_gate[i] << " ";
255             }
256 
257         std::cout << std::endl<< std::endl;
258 
259 }
260 
261 void rest_of_GRU_SW(float *vector_in_update,float *vector_in_candidate,data_in_mvm *vector_in_out,
262 float *out_rest_of_GRU){
263 
264     float result_subtractor[NWORDS]={0};    // Resultado SW resta
265     float result_product1[NWORDS]={0};      // Resultado SW producto 1
266     float result_product2[NWORDS]={0};      // Resultado SW producto 2
267 
268     std::cout << "/************************************************/" << std::endl;
269     std::cout << "                  SW REST OF GRU                  " << std::endl;
270     std::cout << "/************************************************/" << std::endl;
271 
272     //Impresión vectores
273         std::cout<< "VECTOR UPDATE: "<<std::endl;
274         for(int i = 0; i < NWORDS; i++){
275             std::cout<<vector_in_update[i]<<" ";
276         }
277         std::cout<<std::endl;
278         std::cout<<std::endl;
279 
280         std::cout<< "VECTOR CANDIDATE: "<<std::endl;
281         for(int i = 0; i < NWORDS; i++){
282             std::cout<<vector_in_candidate[i]<<" ";
283         }
284         std::cout<<std::endl;
285         std::cout<<std::endl;
286 
287         std::cout<< "VECTOR OUT (IN): "<<std::endl;
288         for(int i = 0; i < NWORDS+1; i++){
289             std::cout<<vector_in_out[i]<<" ";
290         }
291         std::cout<<std::endl;
292         std::cout<<std::endl;
293 
```

```
294     // VECTOR SUBTRACTOR
295     std::cout<< "RESULT RESTA: "<<std::endl;
296         for(int i = 0; i < NWORDS; i++){
297             result_subtractor[i] = 1.0f - vector_in_update[i];
298             std::cout<<result_subtractor[i]<<" ";
299         }
300         std::cout<<std::endl;
301 
302     // VECTOR PRODUCT CANDIDATE * ( 1 - UPDATE)
303         std::cout<< "RESULT PRODUCT CANDIDATE * RESTA: "<<std::endl;
304         for(int i = 0; i < NWORDS; i++){
305             result_product1[i] = vector_in_candidate[i]  * result_subtractor[i];
306             std::cout<<result_product1[i]<<" ";
307         }
308         std::cout<<std::endl;
309 
310     // VECTOR PRODUCT  UPDATE * OUT
311         std::cout<< "RESULT PRODUCT  UPDATE * OUT: "<<std::endl;
312         for(int i = 0; i < NWORDS; i++){
313             result_product2[i] = float(vector_in_out[i+1])  * vector_in_update[i];
314             std::cout<<result_product2[i]<<" ";
315         }
316         std::cout<<std::endl;
317 
318     // VECTOR ADDER
319         std::cout<< "RESULT ADDER: "<<std::endl;
320         for(int i = 0; i < NWORDS; i++){
321             out_rest_of_GRU[i] = result_product1[i] + result_product2[i];
322             std::cout<<out_rest_of_GRU[i]<<" ";
323         }
324         std::cout<<std::endl;
325 
326         // Resultado obtenido por SW
327 
328         std::cout << "/************************************************/" << std::endl;
329         std::cout << "                    SW GLOBAL                   " << std::endl;
330         std::cout << "/************************************************/" << std::endl;
331 
332         for(int i = 0; i < NWORDS; i++){
333             std::cout<< out_rest_of_GRU[i] << " ";
334         }
335         std::cout << std::endl;
336         std::cout << std::endl;
337 
338 }
339 
340 void test_full_GRU() {
341 
342     // Inicializar memoria
343         // MVM 1
344             data_in_mvm vector_in[VECTOR_INDEX1]={0};       // Vector de entrada 1
345 
346             // RESET
347                 int din1_1[NWORDS_M1]={0};                  // Memoria para guardar matriz 1
348                 int matrix1_1[ROWS_A1][COLS_A1]={0};        // Matriz para datos 1
349                 data_in_mvm matrix_W_RST[ROWS_A1][COLS_A1]={0};
350 
351             // CANDIDATE
352                 int din1_2[NWORDS_M1]={0};                  // Memoria para guardar matriz 1
353                 int matrix1_2[ROWS_A1][COLS_A1]={0};        // Matriz para datos 1
354                 data_in_mvm matrix_W_CAND[ROWS_A1][COLS_A1]={0};
355             // UPDATE
356                 int din1_3[NWORDS_M1]={0};                  // Memoria para guardar matriz 1
357                 int matrix1_3[ROWS_A1][COLS_A1]={0};        // Matriz para datos 1
358                 data_in_mvm matrix_W_UP[ROWS_A1][COLS_A1]={0};
359 
360         // MVM 2
```

```
361             data_in_mvm vector_in_out[VECTOR_INDEX2]={0};    // Vector de entrada 2
362
363             // RESET
364                 int din2_1[NWORDS_M2]={0};                   // Memoria para guardar matriz 2
365                 int matrix2_1[ROWS_A2][COLS_A2]={0};         // Matriz para datos 2
366                 data_in_mvm matrix_R_RST[ROWS_A2][COLS_A2]={0};
367
368             // CANDIDATE
369                 int din2_2[NWORDS_M2]={0};                   // Memoria para guardar matriz 2
370                 int matrix2_2[ROWS_A2][COLS_A2]={0};         // Matriz para datos 2
371                 data_in_mvm matrix_R_CAND[ROWS_A2][COLS_A2]={0};
372
373             // UPDATE
374                 int din2_3[NWORDS_M2]={0};                   // Memoria para guardar matriz 2
375                 int matrix2_3[ROWS_A2][COLS_A2]={0};         // Matriz para datos 2
376                 data_in_mvm matrix_R_UP[ROWS_A2][COLS_A2]={0};
377
378         //  SALIDAS
379             float out_rst_gate_SW[NWORDS_V_OUT2]={0};        // Resultado SW reset gate
380             float out_upd_gate_SW[NWORDS_V_OUT2]={0};        // Resultado SW update gate
381             float out_cand_state_SW[NWORDS_V_OUT2]={0};      // Resultado SW candidate state
382             float global_SW[NWORDS_V_OUT2]={0};              // Resultado SW global
383
384         // HW
385             data_out_gru result_HW[NWORDS_V_OUT2]={0};       // Resultado HW global
386
387         // Inicializar datos de prueba
388             //Inicialización vectores
389                 initialize_data_vector(vector_in,vector_in_out);
390
391             // Inicialización matrices 1
392                 initialize_data_matrix(din1_1, matrix1_1,
393                                        din1_2, matrix1_2,
394                                        din1_3, matrix1_3,
395                                        ROWS_A1, COLS_A1);
396
397             // Inicialización matrices 2
398                 initialize_data_matrix(din2_1, matrix2_1,
399                                        din2_2, matrix2_2,
400                                        din2_3, matrix2_3,
401                                        ROWS_A2, COLS_A2);
402
403
404         //Impresión del vector y de la matriz
405             std::cout << "/***********************************************/" << std::endl;
406             std::cout << "                 INPUT DATA RESET GATE           " << std::endl;
407             std::cout << "/***********************************************/" << std::endl;
408             print_matrix_vector(matrix1_1,vector_in,matrix2_1,vector_in_out,matrix_W_RST,matrix_R_RST);
409
410             std::cout << "/***********************************************/" << std::endl;
411             std::cout << "              INPUT DATA CANDIDATE STATE         " << std::endl;
412             std::cout << "/***********************************************/" << std::endl;
413             print_matrix_vector(matrix1_2,vector_in,matrix2_2,vector_in_out,matrix_W_CAND,matrix_R_CAND);
414
415             std::cout << "/***********************************************/" << std::endl;
416             std::cout << "                 INPUT DATA UPDATE GATE          " << std::endl;
417             std::cout << "/***********************************************/" << std::endl;
418             print_matrix_vector(matrix1_3,vector_in,matrix2_3,vector_in_out,matrix_W_UP,matrix_R_UP);
419
420
421     // Crear flujos de entrada y salida
422         hls::stream<data_in_mvm> stream_in1;
423         hls::stream<data_in_mvm> stream_in2;
424         hls::stream<data_out_gru> stream_out;
425
426     // Llenar el flujo de entrada con el vector
427         for (int i = 0; i < VECTOR_INDEX1; i++) {
```

```
428             stream_in1.write(vector_in[i]);
429         }
430
431         for (int i = 0; i < VECTOR_INDEX2; i++) {
432             stream_in2.write(vector_in_out[i]);
433         }
434
435     // SOFTWARE
436         rst_gate_SW(matrix_W_RST,matrix_R_RST,vector_in,vector_in_out,out_rst_gate_SW);
437         cand_state_SW(matrix_W_CAND,matrix_R_CAND,out_rst_gate_SW,vector_in,vector_in_out,out_cand_state_SW);
438         upd_gate_SW(matrix_W_UP,matrix_R_UP,vector_in,vector_in_out,out_upd_gate_SW);
439         rest_of_GRU_SW(out_upd_gate_SW,out_cand_state_SW,vector_in_out,global_SW);
440
441     // DUT
442         full_GRU(stream_in1,stream_in2,          // STREAMS ENTRADA
443                  din1_1,din2_1,                  // MATRICES RESET
444                  din1_2,din2_2,                  // MATRICES CANDIDATE
445                  din1_3,din2_3,                  // MATRICES UPDATE
446                  stream_out);                    // STREAM SALIDA
447
448     // Leer los resultados del flujo de salida
449         for (int i = 0; i < NWORDS; i++) {
450             result_HW[i] = stream_out.read();
451         }
452
453     // Imprimir los resultados
454         std::cout << "Resultados HW" << std::endl;
455         for (int i = 0; i < NWORDS; i++) {
456             std::cout << result_HW[i] << " ";
457         }
458         std::cout << std::endl;
459         std::cout << std::endl;
460
461     // Comparar los resultados HW vs SW
462         std::cout << "Comparacion SW vs HW " << std::endl;
463         for (int i = 0; i < NWORDS; i++){
464             float precision =abs(float(result_HW[i]) - global_SW[i]);
465             std::cout<< "Precision " << "C_HW[" << i << "] y C_SW[" << i << "] es:" << precision<<std::endl;
466         }
467
468 }
469
470
471 // Función principal
472 int main() {
473     test_full_GRU();
474     test_full_GRU();
475     std::cout<< "Done test"<<std::endl;
476     std::cout<<std::endl;
477
478     return 0;
479 }
480
```

Apéndice B

Código software

B.1. GRU Q8_8

Header

```

#ifndef SRC_GRU_Q8_8
#define SRC_GRU_Q8_8

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <time.h>
#include <math.h>

#include "xdmaread_hw.h"
#include "xdmaread.h"
#include "xdmawrite_hw.h"
#include "xdmawrite.h"
#include "xfull_gru_hw.h"
#include "xfull_gru.h"

#include "xparameters.h"
#include "xil_types.h"
#include "xstatus.h"
#include "xil_io.h"
#include "device.h"

// Iteraciones
#define ITERATION 150

// Tamaños matrices y vectores
    // MVM_1
        #define MATRIX_A_ROWS 3
        #define MATRIX_A_COLS 16
        #define VECTOR_IN_SIZE 3
    // MVM_2
        #define MATRIX_B_ROWS 17
        #define MATRIX_B_COLS 16
        #define VECTOR_IN_OUT_SIZE 17
    // VECTOR OUT
        #define VECTOR_OUT_SIZE 16


// Tamaño total de las matrices y vectores
```

```
42      #define SIZE_MW (MATRIX_A_ROWS * MATRIX_A_COLS * sizeof(int))
43      #define SIZE_MR (MATRIX_B_ROWS * MATRIX_B_COLS * sizeof(int))
44      #define SIZE_V_IN (VECTOR_IN_SIZE * sizeof(int))
45      #define SIZE_V_IN_OUT (VECTOR_IN_OUT_SIZE * sizeof(int))
46      #define SIZE_V_OUT (VECTOR_OUT_SIZE * sizeof(int))
47
48  // IP's
49      // DMA's READ
50          #define DMAR_BASE_ADDR_0 XPAR_DMAREAD_0_S_AXI_CONTROL_BASEADDR
51          #define DMAR_BASE_ADDR_1 XPAR_DMAREAD_1_S_AXI_CONTROL_BASEADDR
52      // FULL GRU
53          #define FULL_GRU_ADDR XPAR_FULL_GRU_0_S_AXI_CONTROL_BASEADDR
54      // DMA's WRITE
55          #define DMAW_BASE_ADDR XPAR_DMAWRITE_0_S_AXI_CONTROL_BASEADDR
56
57  // TOP
58      int main();
59
60  // SUB FUNCTIONS
61      void fill_vectors(int VECTOR_IN[],int SIZE_IN,
62                        int VECTOR_IN_OUT[],int SIZE_IN_OUT);
63
64      int fill_matrices_W(const char *filename_rst[], int matrix_reset[][MATRIX_A_COLS],
65                          const char *filename_cand[], int matrix_cand[][MATRIX_A_COLS],
66                          const char *filename_up[], int matrix_update[][MATRIX_A_COLS],
67                          int rows, int cols);
68
69      int fill_matrices_R(const char *filename_rst[], int matrix_reset[][MATRIX_B_COLS],
70                          const char *filename_cand[], int matrix_cand[][MATRIX_B_COLS],
71                          const char *filename_up[], int matrix_update[][MATRIX_B_COLS],
72                          int rows, int cols);
73
74      void print_vector(int VECTOR[],int SIZE);
75
76      void print_matrix(int matrix_W[][MATRIX_A_COLS],int rows_W, int cols_W,
77                        int matrix_R[][MATRIX_B_COLS], int rows_R, int cols_R);
78
79
80  void update_values(uio_device V_IN,int VECTOR_IN[],uio_device V_IN_OUT,int VECTOR_IN_OUT[],
81                     int SIZE_IN,int SIZE_IN_OUT, int VECTOR_OUT[],int i);
82
83  ///////////////// FUNCIONES SOFTWARE GRU Y COMPARACIÓN ///////////////
84
85  void int_to_float_vector( int VECTOR_IN[],float VECTOR_OUT[],int SIZE);
86
87
88  void int_to_float_matrix(int matrix_W_IN[][MATRIX_A_COLS],int rows_W, int cols_W,
89                           int matrix_R_IN[][MATRIX_B_COLS], int rows_R, int cols_R,
90                           float matrix_W_OUT[][MATRIX_A_COLS],
91                           float matrix_R_OUT[][MATRIX_B_COLS]);
92
93  void rst_gate_SW(float matrix_weight[][MATRIX_A_COLS],float matrix_recurrent[][MATRIX_B_COLS],
94                   float *vector_in,float *vector_in_out,float *out_rst_gate);
95
96  void cand_state_SW(float matrix_weight[][MATRIX_A_COLS],float matrix_recurrent[][MATRIX_B_COLS],
97                     float *in_rst_gate,float *vector_in,float *vector_in_out,float *out_cand_state);
98
99  void upd_gate_SW(float matrix_weight[][MATRIX_A_COLS],float matrix_recurrent[][MATRIX_B_COLS],
100                  float *vector_in,float *vector_in_out,float *out_rst_gate);
101
102 void rest_of_GRU_SW(float *vector_in_update,float *vector_in_candidate,float *vector_in_out,float *out_rest_of_GRU);
103
104 void print_vector_SW(float VECTOR[],int SIZE);
105
106 void precision(int VECTOR_HW[],float VECTOR_SW[],int SIZE);
107
108 #endif /* SRC_GRU_Q8_8 */
```

Código C++

```
#include "GRU_Q8_8.h"

////////////////////////////////////// VARIABLES HW ////////////////////////////////////////////////
    // INPUT
        // VECTOR IN 1
            int V_IN[VECTOR_IN_SIZE];
        // VECTOR IN 2(OUT)
            int V_IN_OUT[VECTOR_IN_OUT_SIZE];

        //MVM 1
            int W_RESET[MATRIX_A_ROWS][MATRIX_A_COLS];
            int W_CAND[MATRIX_A_ROWS][MATRIX_A_COLS];
            int W_UPDATE[MATRIX_A_ROWS][MATRIX_A_COLS];

        //MVM 2
            int R_RESET[MATRIX_B_ROWS][MATRIX_B_COLS];
            int R_CAND[MATRIX_B_ROWS][MATRIX_B_COLS];
            int R_UPDATE[MATRIX_B_ROWS][MATRIX_B_COLS];

    // OUTPUT
        int V_OUT[VECTOR_OUT_SIZE] = {0};

    // COMPROBACIÓN
        int value;

                    ////////////////// FUNCTIONS ////////////////////

                                ///////// FILL  /////////

void fill_vectors(int VECTOR_IN[],int SIZE_IN,
int VECTOR_IN_OUT[],int SIZE_IN_OUT){

    // Vector in
    for ( int i = 0; i < SIZE_IN; i++){
        VECTOR_IN[i]  = 0x00000100;
    }

    // Vector in_out
    for ( int i = 0; i < SIZE_IN_OUT; i++){
        VECTOR_IN_OUT[i]  = 0x00000000;
    }

    printf( "Vectors loading done" );

}


int fill_matrices_W(const char *filename_rst[], int matrix_reset[][MATRIX_A_COLS],
                    const char *filename_cand[],int matrix_cand[][MATRIX_A_COLS],
                    const char *filename_up[],  int matrix_update[][MATRIX_A_COLS],
                    int rows, int cols) {

    for (int i = 0; i < rows; i++) {

        // Archivo de "reset"
            FILE *file_rst = fopen(filename_rst[i], "r");

        if (file_rst == NULL) {
            perror("Error al abrir el archivo de reset");
            return -1;
        }

        for (int j = 0; j < cols; j++) {
            unsigned int temp_data; // Leer como unsigned int
            if (fscanf(file_rst, "%x", &temp_data) != 1) {
```

```
66                  fprintf(stderr, "Error al leer el valor en el archivo de reset %s\n", filename_rst[i]);
67                  fclose(file_rst);
68                  return -1;
69              }
70              matrix_reset[i][j]  = (int) temp_data;
71          }
72
73          fclose(file_rst);
74
75          // Archivo de "candidate"
76              FILE *file_cand = fopen(filename_cand[i], "r");
77
78          if (file_cand == NULL) {
79              perror("Error al abrir el archivo de candidate");
80              return -1;
81          }
82
83          for (int j = 0; j < cols; j++) {
84              unsigned int temp_data;
85              if (fscanf(file_cand, "%x", &temp_data) != 1) {
86                  fprintf(stderr, "Error al leer el valor en el archivo de candidate %s\n", filename_cand[i]);
87                  fclose(file_cand);
88                  return -1;
89              }
90              matrix_cand[i][j]  = (int) temp_data;
91          }
92
93          fclose(file_cand);
94
95          // Archivo de "update"
96              FILE *file_up = fopen(filename_up[i], "r");
97
98          if (file_up == NULL) {
99              perror("Error al abrir el archivo de update");
100             return -1;
101         }
102
103         for (int j = 0; j < cols; j++) {
104             unsigned int temp_data;
105             if (fscanf(file_up, "%x", &temp_data) != 1) {
106                 fprintf(stderr, "Error al leer el valor en el archivo de update %s\n", filename_up[i]);
107                 fclose(file_up);
108                 return -1;
109             }
110             matrix_update[i][j]  = (int) temp_data;
111         }
112
113         fclose(file_up);
114     }
115     return 0;
116 }
117
118 int fill_matrices_R(const char *filename_rst[], int matrix_reset[][MATRIX_B_COLS],
119                     const char *filename_cand[], int matrix_cand[][MATRIX_B_COLS],
120                     const char *filename_up[],  int matrix_update[][MATRIX_B_COLS],
121                     int rows, int cols) {
122
123     for (int i = 0; i < rows; i++) {
124
125         // Archivo de "reset"
126             FILE *file_rst = fopen(filename_rst[i], "r");
127
128         if (file_rst == NULL) {
129             perror("Error al abrir el archivo de reset");
130             return -1;
131         }
132
```

```
133         for (int j = 0; j < cols; j++) {
134             unsigned int temp_data; // Leer como unsigned int
135             if (fscanf(file_rst, "%x", &temp_data) != 1) {
136                 fprintf(stderr, "Error al leer el valor en el archivo de reset %s\n", filename_rst[i]);
137                 fclose(file_rst);
138                 return -1;
139             }
140             matrix_reset[i][j]  = (int) temp_data;
141         }
142         fclose(file_rst);
143 
144         // Archivo de "candidate"
145             FILE *file_cand = fopen(filename_cand[i], "r");
146 
147         if (file_cand == NULL) {
148             perror("Error al abrir el archivo de candidate");
149             return -1;
150         }
151 
152         for (int j = 0; j < cols; j++) {
153             unsigned int temp_data;
154             if (fscanf(file_cand, "%x", &temp_data) != 1) {
155                 fprintf(stderr, "Error al leer el valor en el archivo de candidate %s\n", filename_cand[i]);
156                 fclose(file_cand);
157                 return -1;
158             }
159             matrix_cand[i][j]  = (int) temp_data;
160         }
161         fclose(file_cand);
162 
163         // Archivo de "update"
164             FILE *file_up = fopen(filename_up[i], "r");
165 
166         if (file_up == NULL) {
167             perror("Error al abrir el archivo de update");
168             return -1;
169         }
170 
171         for (int j = 0; j < cols; j++) {
172             unsigned int temp_data;
173             if (fscanf(file_up, "%x", &temp_data) != 1) {
174                 fprintf(stderr, "Error al leer el valor en el archivo de update %s\n", filename_up[i]);
175                 fclose(file_up);
176                 return -1;
177             }
178             matrix_update[i][j]  = (int) temp_data;
179         }
180         fclose(file_up);
181     }
182     return 0;
183 }
184 
185 
186                                 //////// PRINTS //////////
187 
188 void print_vector( int VECTOR[],int SIZE){
189 
190     for (int i = 0; i < SIZE; i++) {
191         int16_t data_16 = VECTOR[i] & 0xFFFF;
192         float data = data_16 / 256.0f;
193         printf("%f ",data);
194     }
195     printf( "\n" );
196 
197 }
198 
199 void print_matrix(int matrix_W[][MATRIX_A_COLS],int rows_W, int cols_W,
```

```
                    int matrix_R[][MATRIX_B_COLS], int rows_R, int cols_R) {

    // Matriz de pesos (W) 3x16
        printf( " MATRIZ W \n " );
        for (int i = 0; i < rows_W; i++) {
            for (int j = 0; j < cols_W; j++) {
                int16_t data_16 = matrix_W[i][j] & 0xFFFF;
                float data = data_16 / 256.0f;
                printf("%f ",data);
            }
            printf( "\n" );
        }

    // Matriz de pesos recurrentes (R) 17x16
        printf( " MATRIZ R \n " );
        for (int i = 0; i < rows_R; i++) {
            for (int j = 0; j < cols_R; j++) {
                int16_t data_16 = matrix_R[i][j] & 0xFFFF;
                float data = data_16 / 256.0f;
                printf("%f ",data);
            }
            printf( "\n" );
        }
}

void update_values(uio_device V_IN,int VECTOR_IN[],uio_device V_IN_OUT,
                   int VECTOR_IN_OUT[],int SIZE_IN,int SIZE_IN_OUT, int VECTOR_OUT[],int i){

    // Actualizacion Vector entrada (manual)
        VECTOR_IN[0] = 0x100 + i * 0x1;
        VECTOR_IN[1] = 0x029A - i *0x2;
        VECTOR_IN[2] = 0xFC9A + i * 0x2;

    for (int j = 0; j < SIZE_IN; j++) {
        write_device(&V_IN, j*4,VECTOR_IN[j]);
    }

    // Actualizacion vector salida
        VECTOR_IN_OUT[0] = 0X100;

    for (int i = 1; i < SIZE_IN_OUT; i++) {
        VECTOR_IN_OUT[i] = VECTOR_OUT[i - 1];
    }

    for (int j = 0; j < SIZE_IN_OUT; j++) {
        write_device(&V_IN_OUT, j*4,VECTOR_IN_OUT[j]);
    }
}



////////////////////////////////////// VARIABLES SW ////////////////////////////////////////////////
// INPUT
    // VECTOR IN 1
        float V_IN_SW[VECTOR_IN_SIZE];
    // VECTOR IN 2(OUT)
        float V_IN_OUT_SW[VECTOR_IN_OUT_SIZE];

    //MVM 1
        float W_RESET_SW[MATRIX_A_ROWS][MATRIX_A_COLS];
        float W_CAND_SW[MATRIX_A_ROWS][MATRIX_A_COLS];
        float W_UPDATE_SW[MATRIX_A_ROWS][MATRIX_A_COLS];

    //MVM 2
        float R_RESET_SW[MATRIX_B_ROWS][MATRIX_B_COLS];
        float R_CAND_SW[MATRIX_B_ROWS][MATRIX_B_COLS];
        float R_UPDATE_SW[MATRIX_B_ROWS][MATRIX_B_COLS];
```

```
267
268 // OUTPUT
269     float V_OUT_SW[VECTOR_OUT_SIZE] = {0};
270
271 // INTERNS
272     float OUT_RST_GATE[VECTOR_OUT_SIZE];
273     float OUT_CAND_STATE[VECTOR_OUT_SIZE];
274     float OUT_UP_GATE[VECTOR_OUT_SIZE];
275
276                     ////////////////// FUNCTIONS ////////////////////
277
278                                 ////// SW GRU ///////
279 void rst_gate_SW(float matrix_weight[][MATRIX_A_COLS],float matrix_recurrent[][MATRIX_B_COLS],
280                 float *vector_in,float *vector_in_out,float *out_rst_gate){
281
282     float result_MVM1[VECTOR_OUT_SIZE]={0};         // Resultado SW  MVM 1
283 float result_MVM2[VECTOR_OUT_SIZE]={0};             // Resultado SW MVM 2
284     float result_adder[VECTOR_OUT_SIZE]={0};        // Resultado SW adder
285
286     printf( "\n \n");
287     printf( "/************************************************/");
288     printf( "                    SW RESET GATE                 ");
289     printf( "/************************************************/");
290     printf( "\n \n");
291
292     // MATRIX * VECTOR
293         printf( "   Results MVM1: " );
294         for(int i = 0; i < MATRIX_A_COLS; i++){
295             for(int j = 0; j < MATRIX_A_ROWS; j++){
296                 result_MVM1[i] += matrix_weight[j][i] * vector_in[j];
297             }
298             printf( "%f ", result_MVM1[i]);
299         }
300         printf( "\n \n");
301
302         printf( "   Results MVM2: " );
303         for(int i = 0; i < MATRIX_B_COLS; i++){
304             for(int j = 0; j < MATRIX_B_ROWS; j++){
305                 result_MVM2[i] += matrix_recurrent[j][i] * vector_in_out[j];
306             }
307             printf( "%f ", result_MVM2[i]);
308         }
309         printf( "\n \n");
310
311     // VECTOR ADDER
312         printf( "   Results VECTOR ADDER: " );
313         for(int i = 0; i < VECTOR_OUT_SIZE; i++){
314             result_adder[i] = result_MVM1[i] + result_MVM2[i];
315             printf( "%f ", result_adder[i]);
316         }
317         printf( "\n \n");
318
319     // SIGMOID FUNCTION
320         printf( "   Results SIGMOID: " );
321         for(int i = 0; i < VECTOR_OUT_SIZE; i++){
322             out_rst_gate[i] = 1.0f / (1.0f + expf(-result_adder[i]));
323             printf( "%f ", out_rst_gate[i]);
324         }
325         printf( "\n \n ");
326
327 }
328
329 void cand_state_SW(float matrix_weight[][MATRIX_A_COLS],float matrix_recurrent[][MATRIX_B_COLS],
330                   float *in_rst_gate,float *vector_in,float *vector_in_out,float *out_cand_state){
331
332     float result_MVM1[VECTOR_OUT_SIZE]={0};             // Resultado SW  MVM 1
333     float result_MVM2[VECTOR_OUT_SIZE]={0};             // Resultado SW MVM 2
```

```
334     float result_product[VECTOR_OUT_SIZE]={0};          // Resultado SW product
335     float result_adder[VECTOR_OUT_SIZE]={0};            // Resultado SW adder
336 
337     printf( "/***********************************************/");
338     printf( "              SW CANDIDATE STATE                ");
339     printf( "/***********************************************/");
340     printf( "\n \n");
341 
342     // MATRIX * VECTOR
343         printf( "   Results MVM1: " );
344         for(int i = 0; i < MATRIX_A_COLS; i++){
345             for(int j = 0; j < MATRIX_A_ROWS; j++){
346                 result_MVM1[i] += matrix_weight[j][i] * vector_in[j];
347             }
348             printf( "%f ", result_MVM1[i]);
349         }
350         printf( "\n \n");
351 
352     printf( "   Results MVM2: " );
353     for(int i = 0; i < MATRIX_B_COLS; i++){
354         for(int j = 0; j < MATRIX_B_ROWS; j++){
355             result_MVM2[i] += matrix_recurrent[j][i] * vector_in_out[j];
356         }
357         printf( "%f ", result_MVM2[i]);
358     }
359     printf( "\n \n");
360 
361     // VECTOR PRODUCT
362         printf( "   Results PRODUCT: " );
363         for(int i = 0; i < VECTOR_OUT_SIZE; i++){
364             result_product[i] = in_rst_gate[i]  * result_MVM2[i];
365             printf( "%f ", result_product[i]);
366         }
367         printf( "\n \n");
368 
369     // VECTOR ADDER
370         printf( "   Results ADDER: " );
371         for(int i = 0; i < VECTOR_OUT_SIZE; i++){
372             result_adder[i] = result_MVM1[i] + result_product[i];
373             printf( "%f ", result_adder[i]);
374         }
375         printf( "\n \n");
376 
377     // TANH FUNCTION
378         printf( "   Results TANH: " );
379         for(int i = 0; i < VECTOR_OUT_SIZE; i++){
380             out_cand_state[i] = (expf(2*result_adder[i])-1)/(expf(2*result_adder[i])+1);
381             printf( "%f ", out_cand_state[i]);
382         }
383         printf( "\n \n");
384 
385 }
386 
387 void upd_gate_SW(float matrix_weight[][MATRIX_A_COLS],float matrix_recurrent[][MATRIX_B_COLS],
388                  float *vector_in,float *vector_in_out,float *out_rst_gate){
389 
390     float result_MVM1[VECTOR_OUT_SIZE]={0};         // Resultado SW  MVM 1
391     float result_MVM2[VECTOR_OUT_SIZE]={0};         // Resultado SW MVM 2
392     float result_adder[VECTOR_OUT_SIZE]={0};        // Resultado SW adder
393 
394     printf( "/***********************************************/");
395     printf( "                   SW UPDATE GATE                ");
396     printf( "/***********************************************/");
397     printf( "\n \n");
398 
399     // MATRIX * VECTOR
400         printf( "   Results MVM1: " );
```

```
401         for(int i = 0; i < MATRIX_A_COLS; i++){
402             for(int j = 0; j < MATRIX_A_ROWS; j++){
403                 result_MVM1[i] += matrix_weight[j][i] * vector_in[j];
404             }
405             printf( "%f ", result_MVM1[i]);
406         }
407         printf( "\n \n");
408 
409         printf( "   Results MVM2: " );
410         for(int i = 0; i < MATRIX_B_COLS; i++){
411             for(int j = 0; j < MATRIX_B_ROWS; j++){
412                 result_MVM2[i] += matrix_recurrent[j][i] * vector_in_out[j];
413             }
414             printf( "%f ", result_MVM2[i]);
415         }
416         printf( "\n \n");
417 
418     // VECTOR ADDER
419         printf( "   Results VECTOR ADDER: " );
420         for(int i = 0; i < VECTOR_OUT_SIZE; i++){
421             result_adder[i] = result_MVM1[i] + result_MVM2[i];
422             printf( "%f ", result_adder[i]);
423         }
424         printf( "\n \n");
425 
426     // SIGMOID FUNCTION
427         printf( "   Results SIGMOID: " );
428         for(int i = 0; i < VECTOR_OUT_SIZE; i++){
429             out_rst_gate[i] = 1.0f / (1.0f + expf(-result_adder[i]));
430             printf( "%f ", out_rst_gate[i]);
431         }
432         printf( "\n \n");
433 
434 }
435 
436 void rest_of_GRU_SW(float *vector_in_update,float *vector_in_candidate,float *vector_in_out,float *out_rest_of_GRU){
437 
438     float result_subtractor[VECTOR_OUT_SIZE]={0};    // Resultado SW resta
439     float result_product1[VECTOR_OUT_SIZE]={0};    // Resultado SW producto 1
440     float result_product2[VECTOR_OUT_SIZE]={0};    // Resultado SW producto 2
441 
442     printf( "/************************************************/");
443     printf( "                  SW REST OF GRU                  ");
444     printf( "/************************************************/");
445     printf( "\n \n");
446 
447     //Impresión vectores
448         printf("VECTOR UPDATE: ");
449         for(int i = 0; i < VECTOR_OUT_SIZE; i++){
450             printf( "%f ",vector_in_update[i] );
451         }
452         printf( "\n \n");
453 
454         printf( "VECTOR CANDIDATE: ");
455         for(int i = 0; i < VECTOR_OUT_SIZE; i++){
456             printf( "%f ",vector_in_candidate[i]);
457         }
458         printf( "\n \n");
459 
460         printf ("VECTOR OUT (IN): ");
461         for(int i = 0; i < VECTOR_OUT_SIZE+1; i++){
462             printf( "%f ",vector_in_out[i]);
463         }
464         printf( "\n \n");
465 
466     // VECTOR SUBTRACTOR
467         printf ("RESULT RESTA: ");
```

```
468         for(int i = 0; i < VECTOR_OUT_SIZE; i++){
469             result_subtractor[i] = 1.0f - vector_in_update[i];
470             printf( "%f ",result_subtractor[i]);
471         }
472         printf( "\n \n");
473 
474     // VECTOR PRODUCT CANDIDATE * ( 1 - UPDATE)
475         printf("RESULT PRODUCT CANDIDATE * RESTA: ");
476         for(int i = 0; i < VECTOR_OUT_SIZE; i++){
477             result_product1[i] = vector_in_candidate[i]  * result_subtractor[i];
478             printf( "%f ",result_product1[i]);
479         }
480         printf( "\n \n");
481 
482     // VECTOR PRODUCT  UPDATE * OUT
483         printf ("RESULT PRODUCT  UPDATE * OUT: ");
484         for(int i = 0; i < VECTOR_OUT_SIZE; i++){
485             result_product2[i] = vector_in_out[i+1]  * vector_in_update[i];
486             printf( "%f ",result_product2[i]);
487         }
488         printf( "\n \n");
489 
490     // VECTOR ADDER
491         printf ("RESULT ADDER: ");
492         for(int i = 0; i < VECTOR_OUT_SIZE; i++){
493             out_rest_of_GRU[i] = result_product1[i] + result_product2[i];
494             printf( "%f ",out_rest_of_GRU[i]);
495         }
496         printf( "\n \n");
497 }
498 
499                                 ////// HW TO SW ADAPTATION ///////
500 void int_to_float_vector( int VECTOR_IN[],float VECTOR_OUT[],int SIZE){
501 
502     for (int i = 0; i < SIZE; i++) {
503         int16_t data_16 = VECTOR_IN[i] & 0xFFFF;
504         VECTOR_OUT[i] = data_16 / 256.0f;
505     }
506 }
507 
508 void int_to_float_matrix(int matrix_W_IN[][MATRIX_A_COLS],int rows_W, int cols_W,
509                          int matrix_R_IN[][MATRIX_B_COLS], int rows_R, int cols_R,
510                          float matrix_W_OUT[][MATRIX_A_COLS],
511                          float matrix_R_OUT[][MATRIX_B_COLS]) {
512 
513     // Matriz de pesos (W) 3x16
514     for (int i = 0; i < rows_W; i++) {
515         for (int j = 0; j < cols_W; j++) {
516             int16_t data_16 = matrix_W_IN[i][j] & 0xFFFF;
517             matrix_W_OUT[i][j] = data_16 / 256.0f;
518         }
519     }
520 
521     // Matriz de pesos recurrentes (R) 17x16
522     for (int i = 0; i < rows_R; i++) {
523         for (int j = 0; j < cols_R; j++) {
524             int16_t data_16 = matrix_R_IN[i][j] & 0xFFFF;
525             matrix_R_OUT[i][j]  = data_16 / 256.0f;
526         }
527     }
528 }
529 
530                                 ////// PRINT AND PRECISION ///////
531 void print_vector_SW( float VECTOR[],int SIZE){
532 
533     for (int i = 0; i < SIZE; i++) {
534         printf("%f ",VECTOR[i]);
```

```
535     }
536     printf( "\n" );
537 
538 }
539 
540 void precision(int VECTOR_HW[],float VECTOR_SW[],int SIZE){
541 
542     for (int i = 0; i < SIZE; i++) {
543         int16_t data_16 = VECTOR_HW[i] & 0xFFFF;
544         float data_out_HW = data_16 / 256.0f;
545         float precision = fabsf(data_out_HW - VECTOR_SW[i]);
546         printf("%f ",precision);
547     }
548     printf( "\n" );
549 
550 }
551 
552 
553 
554 
555 ////////////////////////////////////// PRINCIPAL ////////////////////////////////////////////
556 int main() {
557 
558 ////////////////////////////////////// INICIALIZACIÓN ////////////////////////////////////////////
559 
560 // Fill vectors and matrices
561     // Vectors
562         fill_vectors(V_IN, VECTOR_IN_SIZE,V_IN_OUT,VECTOR_IN_OUT_SIZE);
563 
564     // Matrices
565         const char *fileNames_W_RESET[MATRIX_A_ROWS] = { "bias/weight/biasW_rst.txt", "weight/RST/weightW1_rst.txt",
566                                                          "weight/RST/weightW2_rst.txt" };
567         const char *fileNames_W_CAND[MATRIX_A_ROWS] = { "bias/weight/biasW_cand.txt", "weight/CAND/weightW1_cand.txt",
568                                                         "weight/CAND/weightW2_cand.txt" };
569         const char *fileNames_W_UPDATE[MATRIX_A_ROWS] = { "bias/weight/biasW_up.txt", "weight/UP/weightW1_up.txt",
570                                                           "weight/UP/weightW2_up.txt" };
571 
572     if (fill_matrices_W(fileNames_W_RESET,W_RESET,fileNames_W_CAND,W_CAND,fileNames_W_UPDATE,W_UPDATE,MATRIX_A_ROWS,
573                         MATRIX_A_COLS) != 0) {
574         printf("Error al procesar los archivos WEIGHTS \n");
575         return -1;
576     }
577 
578     printf("Weight matrices loading done \n");
579 
580     const char *fileNames_WR_RESET[MATRIX_B_ROWS] = {
581     "bias/weight_recurrent/biasR_rst.txt",   "weight_recurrent/RST/weightR1_rst.txt",
582     "weight_recurrent/RST/weightR2_rst.txt", "weight_recurrent/RST/weightR3_rst.txt",
583     "weight_recurrent/RST/weightR4_rst.txt", "weight_recurrent/RST/weightR5_rst.txt",
584     "weight_recurrent/RST/weightR6_rst.txt", "weight_recurrent/RST/weightR7_rst.txt",
585     "weight_recurrent/RST/weightR8_rst.txt", "weight_recurrent/RST/weightR9_rst.txt",
586     "weight_recurrent/RST/weightR10_rst.txt", "weight_recurrent/RST/weightR11_rst.txt",
587     "weight_recurrent/RST/weightR12_rst.txt", "weight_recurrent/RST/weightR13_rst.txt",
588     "weight_recurrent/RST/weightR14_rst.txt", "weight_recurrent/RST/weightR15_rst.txt",
589     "weight_recurrent/RST/weightR16_rst.txt"
590     };
591 
592     const char *fileNames_WR_CAND[MATRIX_B_ROWS] = {
593     "bias/weight_recurrent/biasR_cand.txt",   "weight_recurrent/CAND/weightR1_cand.txt",
594     "weight_recurrent/CAND/weightR2_cand.txt", "weight_recurrent/CAND/weightR3_cand.txt",
595     "weight_recurrent/CAND/weightR4_cand.txt", "weight_recurrent/CAND/weightR5_cand.txt",
596     "weight_recurrent/CAND/weightR6_cand.txt", "weight_recurrent/CAND/weightR7_cand.txt",
597     "weight_recurrent/CAND/weightR8_cand.txt", "weight_recurrent/CAND/weightR9_cand.txt",
598     "weight_recurrent/CAND/weightR10_cand.txt", "weight_recurrent/CAND/weightR11_cand.txt",
599     "weight_recurrent/CAND/weightR12_cand.txt", "weight_recurrent/CAND/weightR13_cand.txt",
600     "weight_recurrent/CAND/weightR14_cand.txt", "weight_recurrent/CAND/weightR15_cand.txt",
601     "weight_recurrent/CAND/weightR16_cand.txt"
```

```
602     };
603
604     const char *fileNames_WR_UPDATE[MATRIX_B_ROWS] = {
605     "bias/weight_recurrent/biasR_up.txt", "weight_recurrent/UP/weightR1_up.txt",
606     "weight_recurrent/UP/weightR2_up.txt", "weight_recurrent/UP/weightR3_up.txt",
607     "weight_recurrent/UP/weightR4_up.txt", "weight_recurrent/UP/weightR5_up.txt",
608     "weight_recurrent/UP/weightR6_up.txt", "weight_recurrent/UP/weightR7_up.txt",
609     "weight_recurrent/UP/weightR8_up.txt", "weight_recurrent/UP/weightR9_up.txt",
610     "weight_recurrent/UP/weightR10_up.txt", "weight_recurrent/UP/weightR11_up.txt",
611     "weight_recurrent/UP/weightR12_up.txt", "weight_recurrent/UP/weightR13_up.txt",
612     "weight_recurrent/UP/weightR14_up.txt", "weight_recurrent/UP/weightR15_up.txt",
613     "weight_recurrent/UP/weightR16_up.txt"
614     };
615
616     if (fill_matrices_R(fileNames_WR_RESET,R_RESET,fileNames_WR_CAND,R_CAND,fileNames_WR_UPDATE,R_UPDATE,MATRIX_B_ROWS,
617                         MATRIX_B_COLS) != 0) {
618         printf("Error al procesar los archivos WEIGHTS RECURRENT \n");
619         return -1;
620     }
621     printf("Weight recurrent matrices loading done \n");
622
623 ////////////////////////////////////// IMPRESIONES ////////////////////////////////////////////////
624 // Print vectors and matrices
625     printf( "Vector in: \n" );
626     print_vector(V_IN,VECTOR_IN_SIZE);
627
628     printf( "Vector in (out): \n" );
629     print_vector(V_IN_OUT,VECTOR_IN_OUT_SIZE);
630
631     printf( "Matrices RESET GATE: \n" );
632     print_matrix(W_RESET, MATRIX_A_ROWS, MATRIX_A_COLS,
633     R_RESET, MATRIX_B_ROWS, MATRIX_B_COLS);
634
635     printf( "Matrices CANDIDATE STATE: \n" );
636     print_matrix(W_CAND, MATRIX_A_ROWS, MATRIX_A_COLS,
637     R_CAND, MATRIX_B_ROWS, MATRIX_B_COLS);
638
639     printf( "Matrices UPDATE GATE: \n" );
640     print_matrix(W_UPDATE, MATRIX_A_ROWS, MATRIX_A_COLS,
641     R_UPDATE, MATRIX_B_ROWS, MATRIX_B_COLS);
642
643 // Print output vector
644     printf( "Output vector: \n" );
645     print_vector(V_OUT,VECTOR_OUT_SIZE);
646
647 ////////////////////////////////////// ALMACENAMIENTO ////////////////////////////////////////////////
648
649 //*********************************************************************************************/
650 uio_device MATRIX_W_RST;
651     volatile __off64_t address_matrix_W_RST = 0x01000000;
652     printf("\r\n base address weight reset :--: 0x%x",(unsigned int) address_matrix_W_RST);
653     open_device(&MATRIX_W_RST, address_matrix_W_RST, SIZE_MW, "/dev/mem",^^I"MATRIX_W_RST");
654     printf("\r\n successfully open device MATRIX WEIGTH RESET \n");
655
656     for (int i = 0; i < MATRIX_A_ROWS; i++) {
657         for (int j = 0; j < MATRIX_A_COLS; j++) {
658             write_device(&MATRIX_W_RST, (i + j * MATRIX_A_ROWS)*4,W_RESET[i][j]);
659         }
660     }
661
662     // Comprobacion valores guardados
663         for (int i = 0; i < MATRIX_A_ROWS; i++) {
664             for (int j = 0; j < MATRIX_A_COLS; j++) {
665                 value = read_device(&MATRIX_W_RST, (i + j * MATRIX_A_ROWS)*4);
666                 printf("%x ",value);
667             }
668         printf("\n");
```

```
669         }
670
671 //*********************************************************************************************/
672 uio_device MATRIX_R_RST;
673     volatile __off64_t address_matrix_R_RST = 0x01005000;
674     printf("\r\n base address weight recurrent reset :--: 0x%x",(unsigned int) address_matrix_R_RST);
675     open_device(&MATRIX_R_RST, address_matrix_R_RST, SIZE_MR, "/dev/mem",^^I"MATRIX_R_RST");
676     printf("\r\n successfully open device MATRIX WEIGTH RECURRENT RESET \n");
677
678     for (int i = 0; i < MATRIX_B_ROWS; i++) {
679         for (int j = 0; j < MATRIX_B_COLS; j++) {
680             write_device(&MATRIX_R_RST, (i + j * MATRIX_B_ROWS)*4,R_RESET[i][j]);
681         }
682     }
683
684     // Comprobacion valores guardados
685         for (int i = 0; i < MATRIX_B_ROWS; i++) {
686             for (int j = 0; j < MATRIX_B_COLS; j++) {
687                 value = read_device(&MATRIX_R_RST, (i + j * MATRIX_B_ROWS)*4);
688                 printf("%x ",value);
689             }
690             printf("\n");
691         }
692
693 //*********************************************************************************************/
694 uio_device MATRIX_W_CAND;
695     volatile __off64_t address_matrix_W_CAND = 0x01010000;
696     printf("\r\n base address weight candidate :--: 0x%x",(unsigned int) address_matrix_W_CAND);
697     open_device(&MATRIX_W_CAND, address_matrix_W_CAND, SIZE_MW, "/dev/mem",^^I"MATRIX_W_CAND");
698     printf("\r\n successfully open device MATRIX WEIGTH CANDIDATE \n");
699
700     for (int i = 0; i < MATRIX_A_ROWS; i++) {
701         for (int j = 0; j < MATRIX_A_COLS; j++) {
702             write_device(&MATRIX_W_CAND, (i + j * MATRIX_A_ROWS)*4,W_CAND[i][j]);
703         }
704     }
705
706     // Comprobacion valores guardados
707         for (int i = 0; i < MATRIX_A_ROWS; i++) {
708             for (int j = 0; j < MATRIX_A_COLS; j++) {
709                 value = read_device(&MATRIX_W_CAND, (i + j * MATRIX_A_ROWS)*4);
710                 printf("%x ",value);
711             }
712             printf( "\n" );
713         }
714
715 //*********************************************************************************************/
716 uio_device MATRIX_R_CAND;
717     volatile __off64_t address_matrix_R_CAND = 0x01015000;
718     printf("\r\n base address weight recurrent candidate :--: 0x%x",(unsigned int) address_matrix_R_CAND);
719     open_device(&MATRIX_R_CAND, address_matrix_R_CAND, SIZE_MR, "/dev/mem",^^I"MATRIX_R_CAND");
720     printf("\r\n successfully open device MATRIX WEIGTH RECURRENT CANDIDATE \n");
721
722     for (int i = 0; i < MATRIX_B_ROWS; i++) {
723         for (int j = 0; j < MATRIX_B_COLS; j++) {
724             write_device(&MATRIX_R_CAND, (i + j * MATRIX_B_ROWS)*4,R_CAND[i][j]);
725         }
726     }
727
728     // Comprobacion valores guardados
729     for (int i = 0; i < MATRIX_B_ROWS; i++) {
730         for (int j = 0; j < MATRIX_B_COLS; j++) {
731             value = read_device(&MATRIX_R_CAND, (i + j * MATRIX_B_ROWS)*4);
732             printf("%x ",value);
733         }
734         printf( "\n" );
735     }
```

```
736
737 //*********************************************************************************************/
738 uio_device MATRIX_W_UP;
739     volatile __off64_t address_matrix_W_UP = 0x01020000;
740     printf("\r\n base address weight UPDATE:--: 0x%x",(unsigned int) address_matrix_W_UP);
741     open_device(&MATRIX_W_UP, address_matrix_W_UP, SIZE_MW, "/dev/mem",^^I"MATRIX_W_UP");
742     printf("\r\n successfully open device MATRIX WEIGTH UPDATE \n");
743
744     for (int i = 0; i < MATRIX_A_ROWS; i++) {
745         for (int j = 0; j < MATRIX_A_COLS; j++) {
746             write_device(&MATRIX_W_UP, (i + j * MATRIX_A_ROWS)*4,W_CAND[i][j]);
747         }
748     }
749
750     // Comprobacion valores guardados
751     for (int i = 0; i < MATRIX_A_ROWS; i++) {
752         for (int j = 0; j < MATRIX_A_COLS; j++) {
753             value = read_device(&MATRIX_W_UP, (i + j * MATRIX_A_ROWS)*4);
754             printf("%x ",value);
755         }
756         printf( "\n" );
757     }
758
759 //*********************************************************************************************/
760 uio_device MATRIX_R_UP;
761     volatile __off64_t address_matrix_R_UP = 0x01025000;
762     printf("\r\n base address weight recurent UPDATE :--: 0x%x",(unsigned int) address_matrix_R_UP);
763     open_device(&MATRIX_R_UP, address_matrix_R_UP, SIZE_MR, "/dev/mem",^^I"MATRIX_R_UP");
764     printf("\r\n successfully open device MATRIX WEIGTH RECURRENT UPDATE \n");
765
766     for (int i = 0; i < MATRIX_B_ROWS; i++) {
767         for (int j = 0; j < MATRIX_B_COLS; j++) {
768             write_device(&MATRIX_R_UP, (i + j * MATRIX_B_ROWS)*4,R_CAND[i][j]);
769         }
770     }
771
772     // Comprobacion valores guardados
773         for (int i = 0; i < MATRIX_B_ROWS; i++) {
774             for (int j = 0; j < MATRIX_B_COLS; j++) {
775                 value = read_device(&MATRIX_R_UP, (i + j * MATRIX_B_ROWS)*4);
776                 printf("%x ",value);
777             }
778             printf( "\n" );
779         }
780
781 //*********************************************************************************************/
782 uio_device VECTOR_IN;
783     volatile __off64_t address_vector_IN = 0x01030000;
784     printf("\r\n base address vector in :--: 0x%x",(unsigned int) address_vector_IN);
785     open_device(&VECTOR_IN, address_vector_IN, SIZE_V_IN, "/dev/mem",^^I"VECTOR_IN");
786     printf("\r\n successfully open device VECTOR INPUT \n");
787
788     for (int j = 0; j < VECTOR_IN_SIZE; j++) {
789         write_device(&VECTOR_IN, j*4,V_IN[j]);
790     }
791
792     // Comprobacion valores guardados
793         for (int j = 0; j < VECTOR_IN_SIZE; j++) {
794             value = read_device(&VECTOR_IN,j*4);
795             printf("%x ",value);
796         }
797         printf( "\n" );
798
799 //*********************************************************************************************/
800 uio_device VECTOR_IN_OUT;
801     volatile __off64_t address_VECTOR_IN_OUT = 0x01035000;
802     printf("\r\n base address vector in (out) :--: 0x%x",(unsigned int) address_VECTOR_IN_OUT);
```

```
803     open_device(&VECTOR_IN_OUT, address_VECTOR_IN_OUT, SIZE_V_IN_OUT, "/dev/mem",^^I"VECTOR_IN_OUT");
804     printf("\r\n successfully open device VECTOR INPUT OUT \n");
805 
806     for (int j = 0; j < VECTOR_IN_OUT_SIZE; j++) {
807         write_device(&VECTOR_IN_OUT, j*4,V_IN_OUT[j]);
808     }
809 
810     // Comprobacion valores guardados
811         for (int j = 0; j < VECTOR_IN_OUT_SIZE; j++) {
812             value = read_device(&VECTOR_IN_OUT, j*4);
813             printf("%x ",value);
814         }
815         printf( "\n" );
816 
817 //*****************************************************************************************/
818 uio_device VECTOR_OUT;
819     volatile __off64_t address_vector_out = 0x01040000;
820     printf("\r\n base address vector out :--: 0x%x",(unsigned int) address_vector_out);
821     open_device(&VECTOR_OUT, address_vector_out, SIZE_V_OUT, "/dev/mem",^^I"VECTOR_OUT");
822     printf("\r\n successfully open device VECTOR_OUT \n");
823 //*****************************************************************************************/
824 
825 //DMA write
826     uio_device DMA_WRITE;
827         volatile __off64_t address_dma_write = DMAW_BASE_ADDR;
828         printf("\r\n base address IP write dma :--: 0x%x",(unsigned int) address_dma_write);
829 
830         open_device(&DMA_WRITE, address_dma_write,0xffff, "/dev/mem","DMA_WRITE");
831         printf("\r\n successfully open device DMA WRITE \n");
832 
833 // FULL GRU
834     uio_device full_GRU;
835         volatile __off64_t address_full_gru = FULL_GRU_ADDR;
836         printf("\r\n\n base address IP full GRU :--: 0x%x",(unsigned int) address_full_gru);
837 
838         open_device(&full_GRU, address_full_gru,0xffff, "/dev/mem","full_GRU");
839         printf("\r\n successfully open device FULL GRU \n");
840 
841 
842 //DMA READ  VECTOR IN
843     uio_device DMA_READ_VIN;
844         volatile __off64_t address_dma_read_vin = DMAR_BASE_ADDR_0;
845         printf("\r\n\n base address IP read dma vector in :--: 0x%x",(unsigned int) address_dma_read_vin);
846 
847         open_device(&DMA_READ_VIN, address_dma_read_vin,0xffff, "/dev/mem","DMA_READ_VIN");
848         printf("\r\n successfully open device DMA READ vector in \n");
849 
850 //DMA READ  VECTOR IN OUT
851     uio_device DMA_READ_VIN_OUT;
852         volatile __off64_t address_dma_read_vin_out = DMAR_BASE_ADDR_1;
853         printf("\r\n\n base address IP read dma vector in (out) :--: 0x%x",(unsigned int) address_dma_read_vin_out);
854 
855         open_device(&DMA_READ_VIN_OUT, address_dma_read_vin_out,0xffff, "/dev/mem","DMA_READ_VIN_OUT");
856         printf("\r\n successfully open device DMA READ vector in (out) \n");
857 
858 int data;
859 
860 // DMA READ VECTOR IN configure base address vector in
861     printf("\r\n\n configure dma read vin..");
862     // Escritura de la direccion del objeto a leer
863         write_device(&DMA_READ_VIN, XDMAREAD_CONTROL_ADDR_DIN_DATA,(int)address_vector_IN);
864     // Lectura de la direccion del objeto a leer
865         data = read_device(&DMA_READ_VIN,XDMAREAD_CONTROL_ADDR_DIN_DATA);
866         printf("\r\n address DMA READ VIN.. 0x%x",data);
867     // Como los datos que se pasan son de 16 bits (int) los MSB (63-31) = '0'
868         write_device(&DMA_READ_VIN, XDMAREAD_CONTROL_ADDR_DIN_DATA+0x4, 0x00000000);
869         write_device(&DMA_READ_VIN, XDMAREAD_CONTROL_ADDR_WIDTH_DATA, VECTOR_IN_SIZE);
```

```
870     // DEBUG: Lectura del tamaño del objeto a leer ('0x3' = 3 decimal)
871         data = read_device(&DMA_READ_VIN,XDMAREAD_CONTROL_ADDR_WIDTH_DATA);
872         printf("\r\n width dma read VIN.. 0x%x \n",data);
873
874 // DMA READ VECTOR IN OUT configure base address vector in out
875     printf("\r\n\n configure dma read vin_out..");
876     // Escritura de la direccion del objeto a leer
877         write_device(&DMA_READ_VIN_OUT, XDMAREAD_CONTROL_ADDR_DIN_DATA,(int)address_VECTOR_IN_OUT);
878     // Lectura de la direccion del objeto a leer
879         data = read_device(&DMA_READ_VIN_OUT,XDMAREAD_CONTROL_ADDR_DIN_DATA);
880         printf("\r\n address DMA READ VIN (OUT).. 0x%x",data);
881     // Como los datos que se pasan son de 16 bits (int) los MSB (63-31) = '0'
882         write_device(&DMA_READ_VIN_OUT, XDMAREAD_CONTROL_ADDR_DIN_DATA+0x4, 0x00000000);
883         write_device(&DMA_READ_VIN_OUT, XDMAREAD_CONTROL_ADDR_WIDTH_DATA, VECTOR_IN_OUT_SIZE);
884     // DEBUG: Lectura del tamaño del objeto a leer ('0x11' = 17 decimal)
885         data = read_device(&DMA_READ_VIN_OUT,XDMAREAD_CONTROL_ADDR_WIDTH_DATA);
886         printf("\r\n width dma read VIN (OUT).. 0x%x \n",data);
887
888 // FULL GRU configure set base address all matrix
889     printf("\r\n\n configure full GRU");
890     // RESET GATE
891         // WEIGHT
892             // Escritura de la dirección base del objeto a escribir
893                 write_device(&full_GRU, XFULL_GRU_CONTROL_ADDR_DIN_WEIGHT_RST_DATA,(int)address_matrix_W_RST);
894             // Lectura de la direccion del objeto a leer
895                 data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_DIN_WEIGHT_RST_DATA);
896             // Como los datos que se pasan son de 16 bits (int) los MSB (63-31) = '0'
897                 write_device(&full_GRU, (XFULL_GRU_CONTROL_ADDR_DIN_WEIGHT_RST_DATA+0x4),0x00000000);
898                 printf("\r\n address matrix weight reset.. 0x%x",data);
899
900         // WEIGHT RECURRENT
901             // Escritura de la dirección base del objeto a escribir
902                 write_device(&full_GRU, XFULL_GRU_CONTROL_ADDR_DIN_RECURRENT_RST_DATA,(int)address_matrix_R_RST);
903             // Lectura de la direccion del objeto a leer
904                 data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_DIN_RECURRENT_RST_DATA);
905             // Como los datos que se pasan son de 16 bits (int) los MSB (63-31) = '0'
906                 write_device(&full_GRU, (XFULL_GRU_CONTROL_ADDR_DIN_RECURRENT_RST_DATA+0x4),0x00000000);
907                 printf("\r\n address matrix weight recurrent reset.. 0x%x",data);
908
909     // CANDIDATE STATE
910         // WEIGHT
911             // Escritura de la dirección base del objeto a escribir
912                 write_device(&full_GRU, XFULL_GRU_CONTROL_ADDR_DIN_WEIGHT_CAND_DATA,(int)address_matrix_W_CAND);
913             // Lectura de la direccion del objeto a leer
914                 data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_DIN_WEIGHT_CAND_DATA);
915             // Como los datos que se pasan son de 16 bits (int) los MSB (63-31) = '0'
916                 write_device(&full_GRU, (XFULL_GRU_CONTROL_ADDR_DIN_WEIGHT_CAND_DATA+0x4),0x00000000);
917                 printf("\r\n address matrix weight candidate.. 0x%x",data);
918
919         // WEIGHT RECURRENT
920             // Escritura de la dirección base del objeto a escribir
921                 write_device(&full_GRU, XFULL_GRU_CONTROL_ADDR_DIN_RECURRENT_CAND_DATA,(int)address_matrix_R_CAND);
922             // Lectura de la direccion del objeto a leer
923                 data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_DIN_RECURRENT_CAND_DATA);
924             // Como los datos que se pasan son de 16 bits (int) los MSB (63-31) = '0'
925                 write_device(&full_GRU, (XFULL_GRU_CONTROL_ADDR_DIN_RECURRENT_CAND_DATA+0x4),0x00000000);
926                 printf("\r\n address matrix weight recurrent candidate.. 0x%x",data);
927
928     // UPDATE GATE
929         // WEIGHT
930             // Escritura de la dirección base del objeto a escribir
931                 write_device(&full_GRU, XFULL_GRU_CONTROL_ADDR_DIN_WEIGHT_UP_DATA,(int)address_matrix_W_UP);
932             // Lectura de la direccion del objeto a leer
933                 data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_DIN_WEIGHT_UP_DATA);
934             // Como los datos que se pasan son de 16 bits (int) los MSB (63-31) = '0'
935                 write_device(&full_GRU, (XFULL_GRU_CONTROL_ADDR_DIN_WEIGHT_UP_DATA+0x4),0x00000000);
936                 printf("\r\n address matrix weight update.. 0x%x",data);
```

```
937
938         // WEIGHT RECURRENT
939             // Escritura de la dirección base del objeto a escribir
940                 write_device(&full_GRU, XFULL_GRU_CONTROL_ADDR_DIN_RECURRENT_UP_DATA,(int)address_matrix_R_UP);
941             // Lectura de la direccion del objeto a leer
942                 data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_DIN_RECURRENT_UP_DATA);
943             // Como los datos que se pasan son de 16 bits (int) los MSB (63-31) = '0'
944                 write_device(&full_GRU, (XFULL_GRU_CONTROL_ADDR_DIN_RECURRENT_UP_DATA+0x4),0x00000000);
945                 printf("\r\n address matrix weight recurrent update.. 0x%x",data);
946
947 //DMA WRITE configure set base address vector OUT
948     printf("\r\n\n configure dma write");
949     // Escritura de la dirección base del objeto a escribir
950         write_device(&DMA_WRITE, XDMAWRITE_CONTROL_ADDR_DOUT_DATA,(int)address_vector_out);
951     // Lectura de la direccion del objeto a leer
952         data = read_device(&DMA_WRITE,XDMAWRITE_CONTROL_ADDR_DOUT_DATA);
953         printf("\r\n address DMA WRITE.. 0x%x",data);
954     // Como los datos que se pasan son de 16 bits (int) los MSB (63-31) = '0'
955         write_device(&DMA_WRITE, (XDMAWRITE_CONTROL_ADDR_DOUT_DATA+0x4),0x00000000);
956         write_device(&DMA_WRITE, XDMAWRITE_CONTROL_ADDR_WIDTH_DATA,VECTOR_OUT_SIZE);
957     // DEBUG: Lectura del tamaño del objeto a escribir ('0x10' = 16 decimal)
958         data = read_device(&DMA_WRITE,XDMAWRITE_CONTROL_ADDR_WIDTH_DATA);
959         printf("\r\n width dma write.. 0x%x",data);
960
961 // ITERACIONES
962     for(int i = 0; i < ITERATION; i++){
963
964     printf( "\n \n");
965     printf( "/************************************************/");
966     printf( "                    ITERATION Nº %d                ",i+1);
967     printf( "/************************************************/");
968     printf( "\n \n");
969
970     printf( "/************************************************/");
971     printf( "                      SOFTWARE                    ");
972     printf( "/************************************************/");
973     printf( "\n \n");
974
975     ////////////////////////////////////////// SW //////////////////////////////////////////
976
977     //Carga de vectores y matrices en variables de SW
978         // VECTORES
979             int_to_float_vector(V_IN,V_IN_SW,VECTOR_IN_OUT_SIZE);
980             int_to_float_vector(V_IN_OUT,V_IN_OUT_SW,VECTOR_IN_OUT_SIZE);
981
982         // MATRICES
983             // RESET
984                 int_to_float_matrix(W_RESET, MATRIX_A_ROWS, MATRIX_A_COLS,
985                                     R_RESET, MATRIX_B_ROWS, MATRIX_B_COLS,
986                                     W_RESET_SW,
987                                     R_RESET_SW);
988             // CAND
989                 int_to_float_matrix(W_CAND, MATRIX_A_ROWS, MATRIX_A_COLS,
990                                     R_CAND, MATRIX_B_ROWS, MATRIX_B_COLS,
991                                     W_CAND_SW,
992                                     R_CAND_SW);
993             // UP
994                 int_to_float_matrix(W_UPDATE, MATRIX_A_ROWS, MATRIX_A_COLS,
995                                     R_UPDATE, MATRIX_B_ROWS, MATRIX_B_COLS,
996                                     W_UPDATE_SW,
997                                     R_UPDATE_SW);
998
999
1000    // Resultados GRU SW
1001        rst_gate_SW(W_RESET_SW,R_RESET_SW,V_IN_SW,V_IN_OUT_SW,OUT_RST_GATE);
1002
1003        cand_state_SW(W_CAND_SW,R_CAND_SW,OUT_RST_GATE,V_IN_SW,V_IN_OUT_SW,OUT_CAND_STATE);
```

```
1004
1005        upd_gate_SW(W_UPDATE_SW,R_UPDATE_SW,V_IN_SW,V_IN_OUT_SW,OUT_UP_GATE);
1006
1007        rest_of_GRU_SW(OUT_UP_GATE,OUT_CAND_STATE,V_IN_OUT_SW,V_OUT_SW);
1008
1009
1010    ////////////////////////////////////// HW ////////////////////////////////////////////////
1011
1012    printf( "\n \n");
1013    printf( "/************************************************/");
1014    printf( "                      HARDWARE                    ");
1015    printf( "/************************************************/");
1016    printf( "\n \n");
1017
1018    //start DMA WRITE
1019        printf("\r\n\n start dma write");
1020        data = (read_device(&DMA_WRITE,XDMAWRITE_CONTROL_ADDR_AP_CTRL)) & 0X80;
1021        write_device(&DMA_WRITE, XDMAWRITE_CONTROL_ADDR_AP_CTRL, (data | 0x01));
1022        // DEBUG: Estado de la DMA WRITE
1023            data= read_device(&DMA_WRITE,XDMAWRITE_CONTROL_ADDR_AP_CTRL);
1024            printf("\r\n state control dma write.. 0x%x",data);
1025        // DEBUG: valor transferido a V OUT desde la full GRU
1026            data = read_device(&DMA_WRITE,XDMAWRITE_CONTROL_ADDR_DOUT_DATA);
1027            printf("\r\n read value of dma write from 0x%x",data);
1028
1029
1030    //start FULL GRU
1031        printf("\r\n\n start full GRU");
1032        // Reset de los bits de control (excepto bits reservados  bit 8)
1033            data = (read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_AP_CTRL)) & 0X80;
1034        // Escribes en la direccion de control un '1' para iniciar (ap_start)
1035            write_device(&full_GRU, XFULL_GRU_CONTROL_ADDR_AP_CTRL, (data | 0x01));
1036        // DEBUG: Estado de la full GRU
1037            data= read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_AP_CTRL);
1038            printf("\r\n state control full GRU.. 0x%x",data);
1039
1040        // DEBUG : VALOR LEIDO DE (MATRICES)
1041            // RESET
1042                // DEBUG: valor leido en la weight full GRU
1043                    data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_DIN_WEIGHT_RST_DATA);
1044                    printf("\r\n read value of weight reset full GRU from 0x%x",data);
1045                // DEBUG: valor leido en la weight recurrent full GRU
1046                    data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_DIN_RECURRENT_RST_DATA);
1047                    printf("\r\n read value of weight recurrent reset full GRU from 0x%x",data);
1048
1049            // CANDIDATE
1050                // DEBUG: valor leido en la weight full GRU
1051                    data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_DIN_WEIGHT_CAND_DATA);
1052                    printf("\r\n read value of weight candidate full GRU from 0x%x..",data);
1053                // DEBUG: valor leido en la weight recurrent full GRU
1054                    data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_DIN_RECURRENT_CAND_DATA);
1055                    printf("\r\n read value of weight recurrent candidate full GRU from 0x%x",data);
1056
1057            // UPDATE
1058                // DEBUG: valor leido en la weight full GRU
1059                    data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_DIN_WEIGHT_UP_DATA);
1060                    printf("\r\n read value of weight update full GRU from 0x%x",data);
1061                // DEBUG: valor leido en la weight recurrent full GRU
1062                    data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_DIN_RECURRENT_UP_DATA);
1063                    printf("\r\n read value of weight recurrent update full GRU from 0x%x",data);
1064
1065
1066    //start DMA READ V IN
1067        printf("\r\n\n start dma read v in");
1068        // Reset de los bits de control (excepto bits reservados  bit 8)
1069            data = (read_device(&DMA_READ_VIN,XDMAREAD_CONTROL_ADDR_AP_CTRL)) & 0X80;
1070        // Escribes en la direccion de control un '1' para iniciar (ap_start)
```

```
1071            write_device(&DMA_READ_VIN, XDMAREAD_CONTROL_ADDR_AP_CTRL, (data | 0x01));
1072
1073        // DEBUG: Estado de la DMA READ
1074            data= read_device(&DMA_READ_VIN,XDMAREAD_CONTROL_ADDR_AP_CTRL);
1075            printf("\r\n state control dma read vin.. 0x%x",data);
1076
1077        // DEBUG: valor leido de V IN en la DMA
1078            data = read_device(&DMA_READ_VIN,XDMAREAD_CONTROL_ADDR_DIN_DATA);
1079            printf("\r\n read value of dma read vin from 0x%x",data);
1080
1081    //start DMA READ V IN (OUT)
1082        printf("\r\n\n start dma read v in (out)");
1083        // Reset de los bits de control (excepto bits reservados  bit 8)
1084            data = (read_device(&DMA_READ_VIN_OUT,XDMAREAD_CONTROL_ADDR_AP_CTRL)) & 0X80;
1085        // Escribes en la direccion de control un '1' para iniciar (ap_start)
1086            write_device(&DMA_READ_VIN_OUT, XDMAREAD_CONTROL_ADDR_AP_CTRL, (data | 0x01));
1087        // DEBUG: Estado de la DMA READ
1088            data= read_device(&DMA_READ_VIN_OUT,XDMAREAD_CONTROL_ADDR_AP_CTRL);
1089            printf("\r\n state control dma read vin (out).. 0x%x",data);
1090        // DEBUG: valor leido de V IN (OUT) en la DMA
1091            data = read_device(&DMA_READ_VIN_OUT,XDMAREAD_CONTROL_ADDR_DIN_DATA);
1092            printf("\r\n read value of dma read vin (out) from 0x%x",data);
1093
1094
1095    // DONE DMA READ V IN
1096        // Lectura del estado de control DMA READ
1097            data = read_device(&DMA_READ_VIN,XDMAREAD_CONTROL_ADDR_AP_CTRL);
1098            printf("\r\n\n waiting finish DMA read v in..");
1099        // DEBUG: Estado de la DMA READ V IN
1100            printf("\r\n state control dma read v in.. 0x%x",data);
1101        // '0x4' = ap_idle, la DMA esta lista para una nueva transmisión
1102            while (data != 4) {
1103                data = read_device(&DMA_READ_VIN,XDMAREAD_CONTROL_ADDR_AP_CTRL);
1104                printf("\r\n waiting dma read v in.. 0x%x",data);
1105            }
1106
1107    // DONE DMA READ V IN (OUT)
1108        // Lectura del estado de control DMA READ
1109            data = read_device(&DMA_READ_VIN_OUT,XDMAREAD_CONTROL_ADDR_AP_CTRL);
1110            printf("\r\n\n waiting finish DMA read v in (out)..");
1111        // DEBUG: Estado de la DMA READ V IN (OUT)
1112            printf("\r\n state control dma read v in (out).. 0x%x",data);
1113        // '0x4' = ap_idle, la DMA esta lista para una nueva transmisión
1114            while (data != 4) {
1115                data = read_device(&DMA_READ_VIN_OUT,XDMAREAD_CONTROL_ADDR_AP_CTRL);
1116                printf("\r\n waiting dma read v in (out).. 0x%x",data);
1117            }
1118
1119    // DONE FULL GRU
1120        // Lectura del estado de control FULL GRU
1121            data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_AP_CTRL);
1122            printf("\r\n\n waiting full GRU..");
1123        // DEBUG: Estado de la FULL GRU
1124            printf("\r\n state control full GRU.. 0x%x",data);
1125        // '0x4' = ap_idle, la FULL GRU esta lista para una nueva transmisión
1126            while (data != 4) {
1127                data = read_device(&full_GRU,XFULL_GRU_CONTROL_ADDR_AP_CTRL);
1128                printf("\r\n waiting full GRU.. 0x%x",data);
1129            }
1130
1131    // DONE DMA WRITE
1132        printf("\r\n\n waiting finish DMA WRITE..");
1133        data = read_device(&DMA_WRITE,XDMAWRITE_CONTROL_ADDR_AP_CTRL);
1134        // DEBUG: Estado de la DMA WRITE
1135            printf("\r\n state control dma write.. 0x%x",data);
1136        // '0x4' = ap_idle, la DMA esta lista para una nueva transmisión
1137            while (data != 4) {
```

```
1138                data = read_device(&DMA_WRITE,XDMAWRITE_CONTROL_ADDR_AP_CTRL);
1139                printf("\r\n waiting dma write.. 0x%x",data);
1140            }
1141
1142    // RESULTADOS
1143        for (int i = 0; i < VECTOR_OUT_SIZE; i++) {
1144            value = read_device(&VECTOR_OUT,i*4);
1145            V_OUT[i] = value;
1146        }
1147
1148    printf( "\n \n");
1149    printf( "/************************************************/");
1150    printf( "              SUMMARY ITERATION Nº %d             ",i+1);
1151    printf( "/************************************************/");
1152    printf( "\n \n");
1153
1154    printf("\n HARDWARE \n\n");
1155
1156    printf(" INPUT V_IN: \n");
1157    print_vector(V_IN, VECTOR_IN_SIZE);
1158
1159    printf("\n INPUT V_IN_OUT: \n");
1160    print_vector(V_IN_OUT, VECTOR_IN_OUT_SIZE);
1161
1162    printf("\n OUTPUT V_OUT: \n");
1163    print_vector(V_OUT, VECTOR_OUT_SIZE);
1164
1165
1166    printf("\n\n SOFTWARE \n \n");
1167
1168    printf(" OUTPUT V_OUT: \n");
1169    print_vector_SW(V_OUT_SW, VECTOR_OUT_SIZE);
1170
1171    printf("\n PRECISION \n");
1172    precision(V_OUT,V_OUT_SW,VECTOR_OUT_SIZE);
1173
1174    printf( "\n \n");
1175    printf( "/************************************************/");
1176    printf( "          END SUMMARY ITERATION Nº %d             ",i+1);
1177    printf( "/************************************************/");
1178    printf( "\n \n");
1179
1180    // Actualización valores de V_IN y V_IN_OUT
1181    update_values(VECTOR_IN,V_IN,VECTOR_IN_OUT,V_IN_OUT,VECTOR_IN_SIZE,VECTOR_IN_OUT_SIZE,V_OUT,i);
1182
1183    if( i < (ITERATION-1) ){
1184
1185        printf( "\n VALORES ENTRADA SIGUIENTE ITERACION (ITERACIÓN %d) \n", i+2);
1186        // Print vectors
1187            printf( "\n Vector in iteration %d: \n", i+2);
1188            print_vector(V_IN,VECTOR_IN_SIZE);
1189
1190        // Comprobacion valores guardados
1191            for (int j = 0; j < VECTOR_IN_SIZE; j++) {
1192                value = read_device(&VECTOR_IN,j*4);
1193                printf("%x ",value);
1194            }
1195            printf( "\n" );
1196
1197            printf( "Vector in (out) iteration %d: \n", i+2);
1198            print_vector(V_IN_OUT,VECTOR_IN_OUT_SIZE);
1199
1200        // Comprobacion valores guardados
1201            printf( "\n Comprobación valores guardados de V_IN_OUT para iteración %d: \n", i+2);
1202            for (int j = 0; j < VECTOR_IN_OUT_SIZE; j++) {
1203                value = read_device(&VECTOR_IN_OUT, j*4);
1204                printf("%x ",value);
```

```
1205                }
1206                printf( "\n" );
1207        }
1208    }
1209
1210    // CLOSE DEVICES
1211        close_device(&DMA_READ_VIN);
1212        close_device(&DMA_READ_VIN_OUT);
1213        close_device(&full_GRU);
1214        close_device(&DMA_WRITE);
1215
1216    printf("\n ITERACIONES COMPLETADAS!! \n ");
1217
1218    return 0;
1219
1220    }
1221
```

B.2. DEVICE

Este código esta descrito en el Apartado 4.4.1.1.

Header

```
1   #ifndef SRC_DEVICE_H_
2   #define SRC_DEVICE_H_
3
4   #include <sys/mman.h>
5   #include <stdio.h>
6   #include <unistd.h>
7   #include <fcntl.h>
8
9   #define __USE_LARGEFILE64
10
11  typedef struct {
12      int     fd;         // file descriptor
13      void    *mp;        // memory map pointer
14      int     mapsize;    // <MAP_SIZE>
15      char    *name;      // device node name
16      char    *alias;     // device alias name
17
18  } uio_device;
19
20
21  int  open_device(uio_device *dev, __off64_t baseaddr, int mapsize, char *name, char *alias);
22  int  close_device(uio_device *dev);
23
24  float write_device_float(uio_device *dev, int addr, float value);
25  void  write_device(uio_device *dev, int addr, int value);
26  int   read_device(uio_device *dev,int addr);
27  float read_device_float(uio_device *dev,int addr);
28  float write_device_float(uio_device *dev, int addr, float value);
29
30  #endif /* SRC_DEVICE_H_ */
31
```

Código C++

```
1   #include "device.h"
2   #include <err.h>
3
4   extern void *mmap64 (void *__addr, size_t __len, int __prot,
```

```
                    int __flags, int __fd, __off64_t __offset);

int open_device(uio_device *dev, __off64_t baseaddr, int mapsize, char *name, char *alias){

    dev->name = name;
    dev->mapsize = mapsize;
    dev->alias = alias;

    dev->fd = open(dev->name, O_RDWR | O_SYNC);
    if (dev->fd <= 0) {
        errx(1, "Error open device: %s", dev->name);
    }

    dev->mp = mmap64(NULL, (size_t)dev->mapsize, PROT_READ | PROT_WRITE, MAP_SHARED,dev->fd , baseaddr);
    if (dev->mp == MAP_FAILED) {
        errx(1, "Error mmaping device: %s", dev->alias);
    }

    return 1;
}

int close_device(uio_device *dev){

    if (dev->mp != MAP_FAILED) munmap(dev->mp, (size_t)dev->mapsize);
    if (dev->fd > 0) close(dev->fd);

    return 1;
}

void write_device(uio_device *dev, int addr, int value){

    if (dev->fd == 0) return;
    if (dev->mp == MAP_FAILED) return;

    *((unsigned *) (dev->mp + addr)) = value;
}

float write_device_float(uio_device *dev, int addr, float value){

    if (dev->fd == 0) return;
    if (dev->mp == MAP_FAILED) return;

    *((float *) (dev->mp + addr)) = value;
}

int read_device(uio_device *dev,int addr){

    if (dev->fd == 0) return 0;
    if (dev->mp == MAP_FAILED) {
        printf("\r\n map failed ");
        return 0;
    }

    return (*((unsigned *) (dev->mp + addr)));
}

float read_device_float(uio_device *dev,int addr){

    if (dev->fd == 0) return 0;
    if (dev->mp == MAP_FAILED) {
        printf("\r\n map failed ");
        return 0;
    }

    return (*((float *) (dev->mp + addr)));
}

```

B.3. Fragmento Resultados sobre Redpitaya

Este fragmento de código esta descrito en el apartado 4.4.3.

```
Vectors loading done
Weight matrices loading done
Weight recurrent matrices loading done
Vector in:
1.000000 1.000000 1.000000
Vector in (out):
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
Matrices RESET GATE:
 MATRIZ W
 0.500000 -0.500000 -0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000 0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
 MATRIZ R
 0.500000 -0.500000 -0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000 0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
Matrices CANDIDATE STATE:
 MATRIZ W
 0.500000 -0.500000 -0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000 0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
 MATRIZ R
 0.500000 -0.500000 -0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000 0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
Matrices UPDATE GATE:
 MATRIZ W
 0.500000 -0.500000 -0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000 0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
 MATRIZ R
 0.500000 -0.500000 -0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000 0.500000 0.500000 -0.500000 0.500000 -0.500000 0.500000
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
-2.796875 0.757812 -0.929688 1.625000 -1.597656 0.468750 -0.085938 1.515625 -2.003906 2.703125 0.312500 -1.812500 1.949219 0.035156 -2.472656 0.234375
Output vector:
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000

 base address weight reset :--: 0x1000000
 successfully open device MATRIX WEIGTH RESET
80 ff80 ff80 80 ff80 80 ff80 80 ff80 80 80 80 80 ff80 80 ff80 80
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
```

```
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c

 base address weight recurrent reset :--: 0x1005000
 successfully open device MATRIX WEIGTH RECURRENT RESET
80 ff80 ff80 80 ff80 80 ff80 80 ff80 80 80 80 ff80 80 ff80 80
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c

 base address weight candidate :--: 0x1010000
 successfully open device MATRIX WEIGTH CANDIDATE
80 ff80 ff80 80 ff80 80 ff80 80 ff80 80 80 80 ff80 80 ff80 80
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c

 base address weight recurrent candidate :--: 0x1015000
 successfully open device MATRIX WEIGTH RECURRENT CANDIDATE
80 ff80 ff80 80 ff80 80 ff80 80 ff80 80 80 80 ff80 80 ff80 80
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c

 base address weight UPDATE:--: 0x1020000
 successfully open device MATRIX WEIGTH UPDATE
80 ff80 ff80 80 ff80 80 ff80 80 ff80 80 80 80 ff80 80 ff80 80
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c

 base address weight recurent UPDATE :--: 0x1025000
 successfully open device MATRIX WEIGTH RECURRENT UPDATE
80 ff80 ff80 80 ff80 80 ff80 80 ff80 80 80 80 ff80 80 ff80 80
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c
fd34 c2 ff12 1a0 fe67 78 ffea 184 fdff 2b4 50 fe30 1f3 9 fd87 3c

 base address vector in :--: 0x1030000
 successfully open device VECTOR INPUT
100 100 100

 base address vector in (out) :--: 0x1035000
 successfully open device VECTOR INPUT OUT
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

 base address vector out :--: 0x1040000
 successfully open device VECTOR_OUT

 base address IP write dma :--: 0x80020000
 successfully open device DMA WRITE


 base address IP full GRU :--: 0x80030000
 successfully open device FULL GRU
```

```
 base address IP read dma vector in :--: 0x80000000
 successfully open device DMA READ vector in


 base address IP read dma vector in (out) :--: 0x80010000
 successfully open device DMA READ vector in (out)


 configure dma read vin..
 address DMA READ VIN.. 0x1030000
 width dma read VIN.. 0x3


 configure dma read vin_out..
 address DMA READ VIN (OUT).. 0x1035000
 width dma read VIN (OUT).. 0x11


 configure full GRU
 address matrix weight reset.. 0x1000000
 address matrix weight recurrent reset.. 0x1005000
 address matrix weight candidate.. 0x1010000
 address matrix weight recurrent candidate.. 0x1015000
 address matrix weight update.. 0x1020000
 address matrix weight recurrent update.. 0x1025000

 configure dma write
 address DMA WRITE.. 0x1040000
 width dma write.. 0x10

/************************************************/                  ITERATION Nº 1               /************************************************/

/************************************************/                    SOFTWARE                   /************************************************/


/************************************************/                  SW RESET GATE                /************************************************/

Results MVM1:
-5.093750 1.015625 -2.359375 3.750000 -3.695312 1.437500 -0.671875 3.531250 -4.507812 5.906250 1.125000 -3.125000 3.398438 0.570312 -5.445312 0.968750

Results MVM2:
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000

Results VECTOR ADDER:
-5.093750 1.015625 -2.359375 3.750000 -3.695312 1.437500 -0.671875 3.531250 -4.507812 5.906250 1.125000 -3.125000 3.398438 0.570312 -5.445312 0.968750

Results SIGMOID:
0.006098 0.734120 0.086323 0.977023 0.024238 0.808067 0.338077 0.971564 0.010902 0.997285 0.754915 0.042088 0.967656 0.638835 0.004298 0.724870

 /************************************************/             SW CANDIDATE STATE               /************************************************/

Results MVM1:
-5.093750 1.015625 -2.359375 3.750000 -3.695312 1.437500 -0.671875 3.531250 -4.507812 5.906250 1.125000 -3.125000 3.398438 0.570312 -5.445312 0.968750

Results MVM2:
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000

Results PRODUCT:
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000

Results ADDER:
-5.093750 1.015625 -2.359375 3.750000 -3.695312 1.437500 -0.671875 3.531250 -4.507812 5.906250 1.125000 -3.125000 3.398438 0.570312 -5.445312 0.968750

Results TANH:
-0.999925 0.768079 -0.982305 0.998894 -0.998767 0.893193 -0.586212 0.998288 -0.999757 0.999985 0.809301 -0.996147 0.997768 0.515589 -0.999963 0.748154

/************************************************/                  SW UPDATE GATE               /************************************************/

Results MVM1:
-5.093750 1.015625 -2.359375 3.750000 -3.695312 1.437500 -0.671875 3.531250 -4.507812 5.906250 1.125000 -3.125000 3.398438 0.570312 -5.445312 0.968750

Results MVM2:
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000

Results VECTOR ADDER:
-5.093750 1.015625 -2.359375 3.750000 -3.695312 1.437500 -0.671875 3.531250 -4.507812 5.906250 1.125000 -3.125000 3.398438 0.570312 -5.445312 0.968750

Results SIGMOID:
0.006098 0.734120 0.086323 0.977023 0.024238 0.808067 0.338077 0.971564 0.010902 0.997285 0.754915 0.042088 0.967656 0.638835 0.004298 0.724870

/************************************************/                  SW REST OF GRU               /************************************************/

VECTOR UPDATE:
0.006098 0.734120 0.086323 0.977023 0.024238 0.808067 0.338077 0.971564 0.010902 0.997285 0.754915 0.042088 0.967656 0.638835 0.004298 0.724870

VECTOR CANDIDATE:
-0.999925 0.768079 -0.982305 0.998894 -0.998767 0.893193 -0.586212 0.998288 -0.999757 0.999985 0.809301 -0.996147 0.997768 0.515589 -0.999963 0.748154
```

```
VECTOR OUT (IN):
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000

RESULT RESTA:
0.993902 0.265880 0.913677 0.022977 0.975762 0.191933 0.661923 0.028436 0.989098 0.002715 0.245085 0.957912 0.032344 0.361165 0.995702 0.275130

RESULT PRODUCT CANDIDATE * RESTA:
-0.993828 0.204217 -0.897509 0.022952 -0.974559 0.171433 -0.388027 0.028387 -0.988857 0.002715 0.198348 -0.954221 0.032272 0.186212 -0.995665 0.205839

RESULT PRODUCT  UPDATE * OUT:
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000

RESULT ADDER:
-0.993828 0.204217 -0.897509 0.022952 -0.974559 0.171433 -0.388027 0.028387 -0.988857 0.002715 0.198348 -0.954221 0.032272 0.186212 -0.995665 0.205839


/***********************************************/                    HARDWARE                    /***********************************************/


 start dma write
 state control dma write.. 0x1
 read value of dma write from 0x1040000

 start full GRU
 state control full GRU.. 0x1
 read value of weight reset full GRU from 0x1000000
 read value of weight recurrent reset full GRU from 0x1005000
 read value of weight candidate full GRU from 0x1010000..
 read value of weight recurrent candidate full GRU from 0x1015000
 read value of weight update full GRU from 0x1020000
 read value of weight recurrent update full GRU from 0x1025000

 start dma read v in
 state control dma read vin.. 0x1
 read value of dma read vin from 0x1030000

 start dma read v in (out)
 state control dma read vin (out).. 0x1
 read value of dma read vin (out) from 0x1035000

 waiting finish DMA read v in..
 state control dma read v in.. 0xe
 waiting dma read v in.. 0x4

 waiting finish DMA read v in (out)..
 state control dma read v in (out).. 0xe
 waiting dma read v in (out).. 0x4

 waiting full GRU..
 state control full GRU.. 0xe
 waiting full GRU.. 0x4

 waiting finish DMA WRITE..
 state control dma write.. 0xe
 waiting dma write.. 0x4

/***********************************************/             SUMMARY ITERATION Nº 1             /***********************************************/


 HARDWARE

 INPUT V_IN:
1.000000 1.000000 1.000000

 INPUT V_IN_OUT:
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000

 OUTPUT V_OUT:
-0.992188 0.203125 -0.898438 0.023438 -0.972656 0.171875 -0.386719 0.027344 -0.988281 0.003906 0.199219 -0.953125 0.031250 0.187500 -0.996094 0.207031


 SOFTWARE

 OUTPUT V_OUT:
-0.993828 0.204217 -0.897509 0.022952 -0.974559 0.171433 -0.388027 0.028387 -0.988857 0.002715 0.198348 -0.954221 0.032272 0.186212 -0.995665 0.205839

 PRECISION
0.001640 0.001092 0.000928 0.000486 0.001903 0.000442 0.001308 0.001044 0.000576 0.001191 0.000871 0.001096 0.001022 0.001288 0.000429 0.001192


/***********************************************/          END SUMMARY ITERATION Nº 1             /***********************************************/


 VALORES ENTRADA SIGUIENTE ITERACION (ITERACIÓN 2)

 Vector in iteration 2:
1.000000 2.601562 -3.398438
100 29a fc9a
Vector in (out) iteration 2:
```

```
1.000000 -0.992188 0.203125 -0.898438 0.023438 -0.972656 0.171875 -0.386719 0.027344 -0.988281 0.003906 0.199219 -0.953125 0.031250 0.187500 -0.996094 0.207031

 Comprobación valores guardados de V_IN_OUT para iteración 2:
100 ff02 34 ff1a 6 ff07 2c ff9d 7 ff03 1 33 ff0c 8 30 ff01 35


/***********************************************/                  ITERATION Nº 2               /***********************************************/

/***********************************************/                    SOFTWARE                   /***********************************************/


/***********************************************/                  SW RESET GATE                /***********************************************/

Results MVM1:
2.728760 -1.103882 0.240845 -0.794922 0.773132 0.126465 -0.431519 -0.707764 1.096863 -1.654053 0.250977 1.944336 -2.053284 0.471985 1.470398 0.313232

Results MVM2:
14.855835 -4.389709 4.271912 -7.840820 7.700470 -1.906006 -0.058899 -7.279419 9.785675 -13.374634 -1.104004 9.803223 -10.504974 0.319550 12.191681 -0.703003

Results VECTOR ADDER:
17.584595 -5.493591 4.512756 -8.635742 8.473602 -1.779541 -0.490417 -7.987183 10.882538 -15.028687 -0.853027 11.747559 -12.558258 0.791534 13.662079 -0.389771

Results SIGMOID:
1.000000 0.004096 0.989151 0.000178 0.999791 0.144360 0.379795 0.000340 0.999981 0.000000 0.298798 0.999992 0.000004 0.688161 0.999999 0.403773

 /***********************************************/             SW CANDIDATE STATE               /***********************************************/

Results MVM1:
2.728760 -1.103882 0.240845 -0.794922 0.773132 0.126465 -0.431519 -0.707764 1.096863 -1.654053 0.250977 1.944336 -2.053284 0.471985 1.470398 0.313232

Results MVM2:
14.855835 -4.389709 4.271912 -7.840820 7.700470 -1.906006 -0.058899 -7.279419 9.785675 -13.374634 -1.104004 9.803223 -10.504974 0.319550 12.191681 -0.703003

Results PRODUCT:
14.855835 -0.017981 4.225564 -0.001393 7.698862 -0.275151 -0.022370 -0.002473 9.785491 -0.000004 -0.329874 9.803145 -0.000037 0.219901 12.191667 -0.283853

Results ADDER:
17.584595 -1.121863 4.466409 -0.796314 8.471994 -0.148686 -0.453888 -0.710236 10.882354 -1.654057 -0.078898 11.747481 -2.053321 0.691886 13.662065 0.029379

Results TANH:
1.000000 -0.808216 0.999736 -0.661971 1.000000 -0.147600 -0.425090 -0.610825 1.000000 -0.929412 -0.078735 1.000000 -0.967607 0.599192 1.000000 0.029371

/***********************************************/                  SW UPDATE GATE               /***********************************************/

Results MVM1:
2.728760 -1.103882 0.240845 -0.794922 0.773132 0.126465 -0.431519 -0.707764 1.096863 -1.654053 0.250977 1.944336 -2.053284 0.471985 1.470398 0.313232

Results MVM2:
14.855835 -4.389709 4.271912 -7.840820 7.700470 -1.906006 -0.058899 -7.279419 9.785675 -13.374634 -1.104004 9.803223 -10.504974 0.319550 12.191681 -0.703003

Results VECTOR ADDER:
17.584595 -5.493591 4.512756 -8.635742 8.473602 -1.779541 -0.490417 -7.987183 10.882538 -15.028687 -0.853027 11.747559 -12.558258 0.791534 13.662079 -0.389771

Results SIGMOID:
1.000000 0.004096 0.989151 0.000178 0.999791 0.144360 0.379795 0.000340 0.999981 0.000000 0.298798 0.999992 0.000004 0.688161 0.999999 0.403773

/***********************************************/                  SW REST OF GRU               /***********************************************/

VECTOR UPDATE:
1.000000 0.004096 0.989151 0.000178 0.999791 0.144360 0.379795 0.000340 0.999981 0.000000 0.298798 0.999992 0.000004 0.688161 0.999999 0.403773

VECTOR CANDIDATE:
1.000000 -0.808216 0.999736 -0.661971 1.000000 -0.147600 -0.425090 -0.610825 1.000000 -0.929412 -0.078735 1.000000 -0.967607 0.599192 1.000000 0.029371

VECTOR OUT (IN):
1.000000 -0.992188 0.203125 -0.898438 0.023438 -0.972656 0.171875 -0.386719 0.027344 -0.988281 0.003906 0.199219 -0.953125 0.031250 0.187500 -0.996094 0.207031

RESULT RESTA:
0.000000 0.995904 0.010849 0.999822 0.000209 0.855640 0.620205 0.999660 0.000019 1.000000 0.701202 0.000008 0.999996 0.311839 0.000001 0.596227

RESULT PRODUCT CANDIDATE * RESTA:
0.000000 -0.804905 0.010846 -0.661854 0.000209 -0.126292 -0.263643 -0.610618 0.000019 -0.929412 -0.055209 0.000008 -0.967604 0.186852 0.000001 0.017512

RESULT PRODUCT  UPDATE * OUT:
-0.992188 0.000832 -0.888690 0.000004 -0.972453 0.024812 -0.146874 0.000009 -0.988263 0.000000 0.059526 -0.953117 0.000000 0.129030 -0.996093 0.083594

RESULT ADDER:
-0.992188 -0.804073 -0.877844 -0.661849 -0.972244 -0.101480 -0.410517 -0.610608 -0.988244 -0.929412 0.004317 -0.953110 -0.967604 0.315882 -0.996091 0.101105


/***********************************************/                    HARDWARE                   /***********************************************/


 start dma write
 state control dma write.. 0x1
 read value of dma write from 0x1040000

 start full GRU
```

```
 state control full GRU.. 0x1
 read value of weight reset full GRU from 0x1000000
 read value of weight recurrent reset full GRU from 0x1005000
 read value of weight candidate full GRU from 0x1010000..
 read value of weight recurrent candidate full GRU from 0x1015000
 read value of weight update full GRU from 0x1020000
 read value of weight recurrent update full GRU from 0x1025000

 start dma read v in
 state control dma read vin.. 0x1
 read value of dma read vin from 0x1030000

 start dma read v in (out)
 state control dma read vin (out).. 0x1
 read value of dma read vin (out) from 0x1035000

 waiting finish DMA read v in..
 state control dma read v in.. 0xe
 waiting dma read v in.. 0x4

 waiting finish DMA read v in (out)..
 state control dma read v in (out).. 0xe
 waiting dma read v in (out).. 0x4

 waiting full GRU..
 state control full GRU.. 0xe
 waiting full GRU.. 0x4

 waiting finish DMA WRITE..
 state control dma write.. 0xe
 waiting dma write.. 0x4

/************************************************/              SUMMARY ITERATION Nº 2             /************************************************/


 HARDWARE

 INPUT V_IN:
1.000000 2.601562 -3.398438

 INPUT V_IN_OUT:
1.000000 -0.992188 0.203125 -0.898438 0.023438 -0.972656 0.171875 -0.386719 0.027344 -0.988281 0.003906 0.199219 -0.953125 0.031250 0.187500 -0.996094 0.207031

 OUTPUT V_OUT:
-0.992188 -0.804688 -0.878906 -0.660156 -0.972656 -0.101562 -0.410156 -0.609375 -0.988281 -0.929688 0.003906 -0.953125 -0.968750 0.316406 -0.996094 0.101562


 SOFTWARE

 OUTPUT V_OUT:
-0.992188 -0.804073 -0.877844 -0.661849 -0.972244 -0.101480 -0.410517 -0.610608 -0.988244 -0.929412 0.004317 -0.953110 -0.967604 0.315882 -0.996091 0.101105

 PRECISION
0.000000 0.000614 0.001063 0.001693 0.000412 0.000082 0.000360 0.001233 0.000037 0.000276 0.000411 0.000015 0.001146 0.000524 0.000002 0.000457


/************************************************/          END SUMMARY ITERATION Nº 2             /************************************************/
```

Apéndice C

Errores instalación Ubuntu, entorno Xilinx y Redpitaya

En esta sección del anexo se van a explicar los diferentes problemas, y sus soluciones, encontrados tras la instalación de cada uno de los programas para poder llevar a cabo este proyecto.

C.1. Errores Ubuntu

La instalación de Ubuntu fue a través de una maquina virtual, en especifico la Virtual Oracle VM VirtualBox. Tras realizar la instalación e intentar abrir el terminal, este no se abría de ninguna forma, ni con el *shortcut*: Ctrl + Alt + T ni haciendo uso del acceso directo. Por lo tanto, la solución aplicada para poder utilizar el terminal consiste en cambiar el idioma al español. De este modo, el terminal funciona correctamente.

Otro de los errores de la maquina virtual/Ubuntu se hizo notar una vez ya instalados los programas de Xilinx ya que la ejecución del mínimo programa tardaba mucho tiempo. Esto es debido a la aparición del icono de la tortuga verde en la esquina inferior derecha. Para solucionarlas se debe deshabilitar la integridad de memoria en el aislamiento de núcleo. Para llevar a cabo esta tarea, en mi caso en Windows 11, se deben seguir los siguientes pasos:

Configuración → Privacidad y seguridad → Seguridad de Windows → Abrir seguridad de Windows → Seguridad del dispositivo → Detalles del aislamiento del núcleo → Integridad de memoria

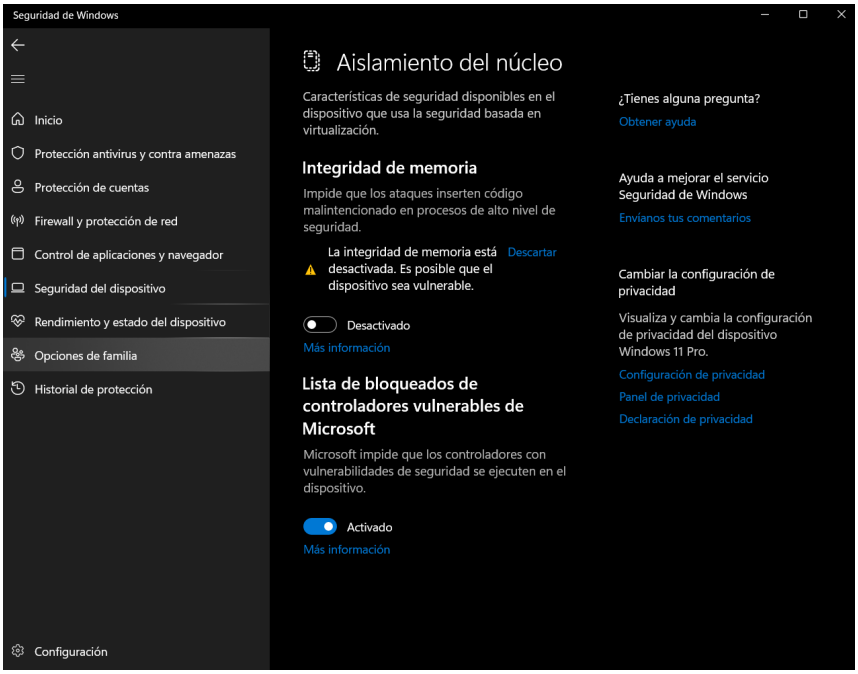

Figura C.1: Integridad de memoria

C.2. Errores entorno Xilinx

Los fallos en la instalación del entorno de Xilinx son debidos a la inexistencia de librerías previa a la instalación. Estos errores aparecen en los siguientes pasos de la instalación de Vivado ML Enterprise:

- Instalación de librerías previa a la instalación de Vivado ya que en caso contrario se queda en generating installed device list [38]. Las librerías en cuestión a instalar son las siguientes:

  ```
  sudo apt install libtinfo5
  sudo apt install libncurses5
  ```

- Instalación de librería para ”Failed to Load Module Canberra-GTK-Module”[39]. Los pasos a seguir para solventar este fallo son:

  - Actualizar el índice de paquetes local.

    ```
    sudo apt update
    ```

  - Luego instale la biblioteca Libcanberra que proporciona todos los módulos necesarios.

    ```
    sudo apt install libcanberra-gtk-module libcanberra-gtk3-module -y
    ```

El siguiente fallo es la instalación del paquete *Vivado ML Enterprise*, debido a que en caso de querer hacer uso de *Vitis IDE*, como ocurre en este proyecto, este paquete no lo instala. Como ya se había instalado este paquete se ha tenido que realizar una extensión de este para poder instalar

Vitis IDE, parte encargada del software. En caso de no haber sido instalado aun, debería instalarse el *Vitis Unified Software Platform* ya que este si que incorpora todo el entorno Vitis, incluido Vitis IDE además de Vivado. Por lo tanto, en caso de haberse equivocado y no querer desinstalarlo y volver a instalar existe la posibilidad de instalarlo por separado siguiendo estos pasos:

1. Abrir Vivado.
2. Help → Add Design tools or devices.
3. Identificarse con la cuenta de AMD.
4. Seleccionar el contenido de *Vitis Unified Software Platform* tal cual se muestra en la Figura C.2.

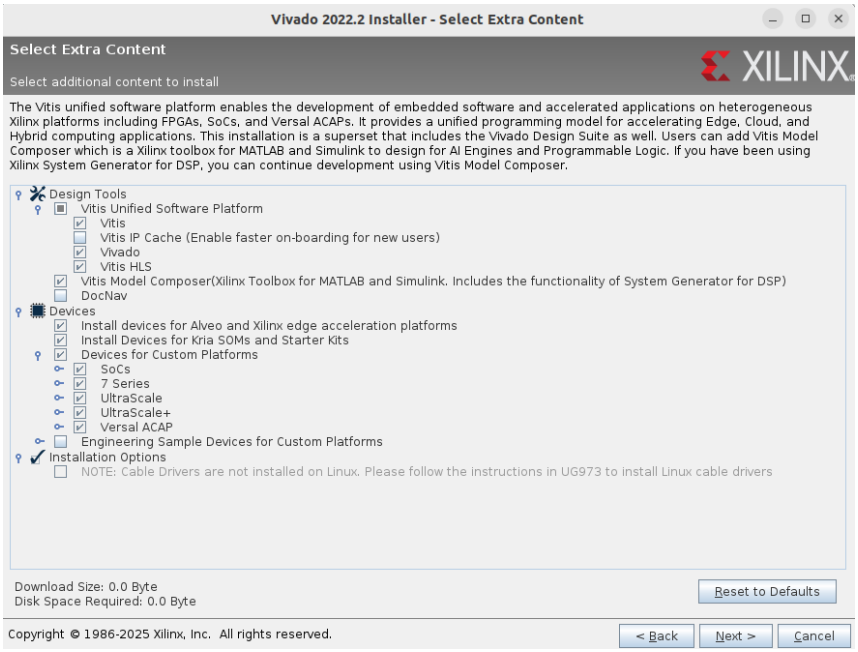

Figura C.2: Añadir Vitis IDE

C.3. Errores Redpitaya

El fallo de la Redpitaya ocurrió mientras me encontraba familiarizando con el entorno de Xilinx, realizando a partir del proyecto base una simple prueba, utilizando las dma de escritura y la de lectura. El conexionado que se realizado es el que se muestran en la siguiente Figura C.3. Donde el propósito era leer los datos de la matriz A de MM del procesador, pasar los datos por streaming a la DMA de escritura y que esta escribiese los datos a través de MM al procesador en una matriz nueva C.

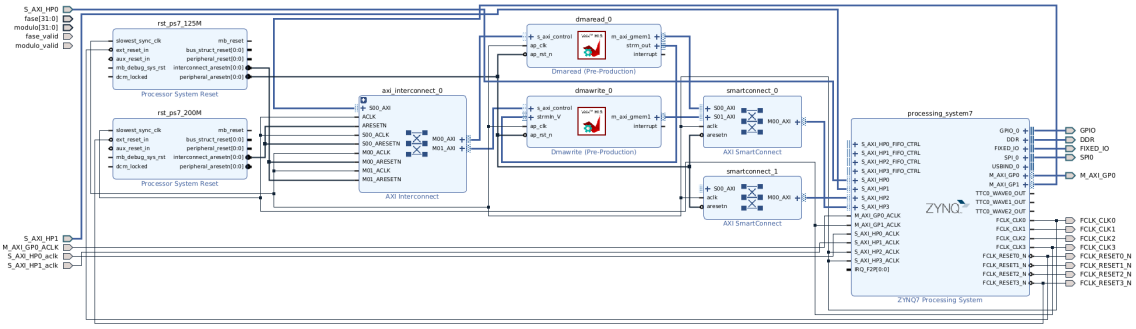

Figura C.3: Block design DMA - DMA

Previamente a la comprobación de este se realizo una prueba de software con el típico *Hello World* el cual se puede seleccionar dentro del abanico de plantillas que suministra Vitis. Se genera el correspondiente archivo `.elf` y se envia y compila sobre la RedPitaya dando un error de librerías GLIB. Este error al parecer es debido a incompatibilidad entre la versiones de Ubuntu instalada (mía) y la de la Redpitaya con el OS 1.04. Por lo tanto la solución fue cambiar la versión del OS de la RedPitaya a la v2.0 mediante la SD Card [40]. Tras haber realizado este cambio se comprobó el correcto funcionamiento.

Por lo tanto, tras esa primera toma de contacto se realizo la verdadera prueba DMA - DMA, en la cual también se obtuvieron fallos. Tras haber probado diferentes tamaños de matrices, el error era diferente dependiendo del tamaño de las filas de la matriz. Cuando las filas eran números impares, toda la matriz era:

dato correcto - dato incorrecto - dato correcto - dato incorrecto

dato correcto - dato incorrecto - dato correcto - dato incorrecto

dato correcto - dato incorrecto - dato correcto - dato incorrecto

Y en el caso de ser par el numero de filas, los datos a la salida son correctos alternando filas.

datos fila correctos

datos fila incorrectos

datos fila correctos

datos fila incorrectos

Tras realizar el debug correspondiente y comprobado cada uno de los posibles fallos, no había ninguno relacionado con el hardware y software implementado. El fallo se encontraba la Redpitaya y es que esta hacia que el resultado no fuera correcto por la parte del hardware al utilizar las dma. Por lo tanto, la solución a esto fue la modificación de parte del OS de la Redpitaya. Las modificaciones consisten en lo siguiente:

- Se ha copiado desde la imagen sd card (STEMlab_125-14-Z7020_OS_1.04-10_stable) los ficheros: `u-boot.src`, `boot.bin` y `devicetree.dtb` a la nueva imagen de la sd card (RedPitaya_OS_2.04-35_stable).
- Además de añadir estos ficheros, como la metodología de la versión 2.0 el grabado del bits-

tream es largo, se ha añadido el fichero `uImage` de la versión 1.04, el bitstream se graba con la opción:

```
cat namebitstream.bit > /dev/xdevcfg
```