Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Diseño de un acelerador de hardware para simulaciones de redes neuronales biológicamente precisas utilizando un sistema multi-FPGA

Informe de Proyecto de Graduación para optar por el título de

Ingeniero en Electrónica con el grado académico de Licenciatura

Jason Kaled Alfaro Badilla

Cartago, 30 de noviembre, 2017

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Proyecto de Graduación
Tribunal Evaluador

Proyecto de Graduación defendido ante el presente Tribunal Evaluador como requisito para optar por el título de Ingeniero en Electrónica con el grado académico de Licenciatura, del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal

Msc. Ing Carlos Salazar García
Profesor Lector

Ing. Daniel Kohkemper
Profesor Lector

Dr. Alfonso Chacón Rodríguez
Profesor Asesor

Los miembros de este Tribunal dan fe de que el presente trabajo de graduación denominado: *Diseño de una arquitectura de comunicación para aceleradores de hardware para simulaciones de una red neuronal artificial*, realizado por el Sr.Jason Kaled Alfaro Badilla ha sido aprobado y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica.

Cartago, 30 de noviembre de 2017

Declaro que el presente Proyecto de Graduación ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de graduación realizado y por el contenido del correspondiente informe final.

Kaleb Alfaro Bad.

Jason Kaled Alfaro Badilla

Cartago, 25 de noviembre de 2017

Céd: 207390325

Resumen

Este proyecto busca implementar un IP sintetizable en FPGA que sea capaz de ejecutar los cálculos de una red neuronal utilizando el modelo Hodgkin-Huxley extendido. La resolución de este modelo es por medio del método de Euler y es descrita su solución numérica desde el lenguaje de programación C; este mismo código fue utilizado para generar la síntesis del IP usando la herramienta de síntesis Vivado de Xilinx. Se adaptó el IP para que sea replicable en múltiples FPGAs de tal forma este IP pueda ejecutar el procesamiento en paralelo entre ellas y mejorar el rendimiento. La plataforma de desarrollo utilizada en este proyecto fue la ZedBoard y se desarrolló el software necesario para el manejo del IP usando las interfaces AXI4-Lite y AXI4-Stream. Hubo una comparación de rendimiento entre una versión *bare-metal* y otra con sistema operativo. Finalmente, se comprobó la diferencia de precisión entre los cálculos del módulo sintetizado por Vivado HLS y el programa de C usado como referencia.

Palabras clave: Hodgkin-Huxley, HLS, Zynq-7000, ZedBoard

Abstract

An FPGA-synthesizable IP was implemented in order to compute a neural network using the extended Hodgkin-Huxley algorithm. An Euler aproximation algorithm written in the programming language C was used as input code, subject to synthesize the IP using the Vivado HLS Synthesis tool from Xilinx. The IP can work in parallel among several devices, so the computation is also parallelized, increasing thus the computational performance. The ZedBoard development board was used to test the generated IP. The software needed to manage the simulation programming initialization and data movement was implemented using different approaches: one using a *bare metal* implementation, and another using an embedded operating system. The interfaces used in the IP were the AXI4-Lite and AXI4-Stream, and a comparation of data transfer speeds was carried on. Also, some comparisons of performance between the implementation of the IP with the operative system and standalone system were carried on. Finally, the computational precision of the system was tested, using the C program as a reference.

Keywords: Hodgkin-Huxley, HLS, Zynq-7000, ZedBoard

a mi querida familia

Agradecimientos

Me encuentro agradecido por la gran oportunidad que me dieron mis profesores Carlos Salazar y Alfonso Chacón para participar en este proyecto. No habría sido posible la finalización del mismo sin el apoyo y guía de los ellos, y valoro toda la confianza que han depositado en mí. Sin olvidar a todos los colaboradores del laboratorio HPC-LAB y del laboratorio hermano DCI-LAB por brindar soporte.

Doy gracias por la gran suerte que tengo de conocer a tan maravillosas personas durante estos cinco años en el TEC y que todavía me siguen acompañando con su amistad. Principalmente a Daniel y Juan Pablo por ayudarme y acompañarme durante todo este tiempo.

Agradezco a todos mis compañeros del colegio de Alajuela por mantener nuestra amistad, servir de apoyo y por contar con ellos en cualquier momento difícil. Gracias a ellos he crecido mucho.

Dedico este logro a mi madre y mis hermanos. Su luchas y sacrificios han sido difíciles para que nosotros salgamos adelante. Gran parte de mi trabajo se los debo a ellos.

Sin menospreciar la compañía de todas mis mascotas felinas, primordialmente Manchii :3

Jason Kaled Alfaro Badilla

Cartago, 4 de diciembre de 2017

Índice general

Índice de figuras

Índice de tablas

Índice de listados de código

Capítulo 1

Introducción

1.1 Justificación

El desarrollo de las investigaciones neurocientíficas para descubrir el funcionamiento del cerebro se ha vuelto cada vez más laborioso, dado que el acceso a este órgano *in vivo* es complejo y riesgoso. Por este motivo, cualquier avance relacionado con el modelado matemático o implementación computacional del cerebro que permita acelerar el desarrollo de estas investigaciones, resulta altamente valorado por el campo de la neurociencia [1].

1.1.1 Iniciativa internacional

El Centro Médico Erasmus (Erasmus Medical Center, o EMC en adelante) ubicado en Rotterdam, Países Bajos, alberga un consorcio multidisciplinario de varias universidades, empresas e institutos, enfocados en el estudio del cerebro, y denominado Erasmus Brain Project; este desarrolla herramientas y metodologías computacionales de avanzada para facilitar la investigación neurocientífica [1].

Uno de sus proyectos en desarrollo es la emulación artificial cerebral, llamado Brain Frame, el cual tiene como meta crear una herramienta genérica y eficiente que imite el comportamiento cerebral en diferentes regiones. Este tipo de simulaciones, por su naturaleza, son de alta demanda computacional y no existe ningún producto especializado en ellas; por ello su modelado está siendo implementado en supercomputadoras que involucran una alta inversión económica, tanto en su adquisición como en su mantenimiento. Además, las supercomputadoras necesitan de mucho espacio físico, ambientes controlados e incurren en un consumo energético elevado; estas características hacen de las supercomputadoras una solución extremadamente cara para muchos investigadores, y que de paso ya empieza a quedarse corta en rendimiento.

Erasmus Brain Project desarrolla una computadora especializada en el modelado de neuronas para disminuir los costos físicos, energéticos y monetarios con el fin de proveer a los laboratorios de una solución eficiente y accesible. Esta computadora apoyaría diferentes

investigaciones como por ejemplo: la validación de hipótesis del funcionamiento cerebral, comprensión de enfermedades neurológicas y sus posibles tratamientos y el desarrollo de implantables cerebrales. Este último, a largo plazo buscaría remplazar secciones dañadas del cerebro por dispositivos electrónicos que las sustituyan. Estas investigaciones han sido tomadas por diferentes laboratorios a nivel internacional y su desarrollo tendría un impacto directo en la población.

De momento Brain Frame está siendo usado para imitar al núcleo olivar inferior, sistema cerebral ubicado en el cerebelo y encargado del control motor, del sentido del ritmo y funciones cognitivas[1]. El sistema olivocerebeloso es uno de los más complejos y demanda una alta capacidad computacional para su modelado, siendo idóneo para una primera iteración de la herramienta. También se encarga de recibir información por los sensores periféricos y procesarla en sus diferentes regiones. El tracto olivocerebeloso, posee patrones estructurales repetitivos y se clasifican en cuatro regiones con sus nombres en inglés: *granulle-cell layer*, *Purkinje-cell layer*, *deep-cerebellar-nuclei* e *inferior-olive nuclei* (ION).

Actualmente, Brain Frame por medio de un sistema computacional basado en metodologías de computación de alto rendimiento (HPC en sus siglas en inglés) logra simular bloques de neuronas del ION bajo el modelo neurocientífico extended-Hodgkin-Huxley (eHH) encontrado en Grujil et al. [2] En la figura 1.1 se muestra un diagrama de las interacciones de una neurona con sus vecinas y sus componentes constituyentes de acuerdo a eHH: Dendrita, Soma y Axón [3] [4].

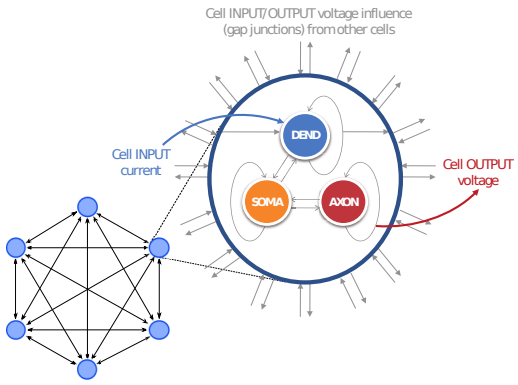

Figura 1.1: Diagrama del modelo computacional del ION. Fuente: [4]

En colaboración con este proyecto, el Laboratorio de Diseño de Circuitos Integrados (DCILAB) de la Escuela de Ingeniería de Electrónica del Instituto Tecnológico de Costa Rica (TEC) se ha incorporado en el diseño de una arquitectura en múltiples FPGAs para simulaciones de redes neuronales; este proyecto se encuentra respaldado como Proyecto de Investigación de la VIE inscrito como “Implementación Multi-FPGA de modelos computacionales artificiales del cerebro”.

1.2 Definición del problema

1.2.1 Generalidades

El modelo eHH describe el comportamiento electroquímico y biológico de cada neurona de manera altamente precisa, pero a un alto costo computacional. De acuerdo al modelo, cada célula neuronal se encuentra compuesta por tres componentes funcionales: de acuerdo a Smaragdos et al. [4]:

- **Dendrita**: que se encarga de recibir las señales eléctricas de estímulo provenientes de sus neuronas vecinas y transferirla al soma.
- **Soma**: que procesa el estímulo recibido y genera una acción de respuesta. Las interacciones generadas entre neuronas se llaman sinapsis, las cuales se observan de manera de pulsos en la respuesta del soma.
- **Axón**: que toma la salida del soma y este la adapta para transmitir la señal a otras neuronas.

El comportamiento en cada paso de simulación de una neurona depende de una serie de requisitos y operaciones. En la tabla 1.1 se observan la cantidad de cálculos realizados por neurona, los cuales se dividen en dos clases: uniones comunicantes y comportamiento de celda. El primero consiste en estímulos ocacionados por neuronas vecinas y el segundo consiste en una serie de operaciones que concluyen con una paso de simulación.

Tabla 1.1: Requisitos de una neurona por paso de simulación [5]. Coma flotante (FP por sus siglas en inglés) se refiere a la forma de representar a los números reales de manera computacional y sus operaciones

Cálculo	**Operaciones FP por neurona**
Uniones comunicantes	475 por conexión
Comportamiento de celda	859
I/O y almacenamiento	**Variables FP por neurona**
Estados de la neurona	19
Vector de conectividad	1 por conexión
Salida de axón	1

Debido a la naturaleza electroquímica del modelo, la mayoría de las operaciones deben ser realizadas en aritmética punto flotante para mantener la precisión muy alta y exacta. En la tabla 1.1 se muestra que para una red neuronal grande, predominan los cálculos aritméticos relacionados con las uniones comunicantes que dependen de neuronas vecinas en la celda.

De acuerdo a estos datos existen cuatro desafíos relevantes:

- El algoritmo computacional es altamente intensivo en manejo de memoria debido al crecimiento de variables locales necesarias para calcular el paso de simulación de una neurona.

- Se encuentra un crecimiento cuadrático de operaciones FP para los pasos de simulación cuando el crecimiento de la red es igual al número de conexiones por neurona. Este factor también se ve afectado por estar asociado al cálculo de funciones exponenciales.

- Cada paso de simulación representa 50μs, esto implica que el tiempo de ejecución máximo para llegar a la ejecución de tiempo real consiste en ese mismo periodo, sin embargo los expertos del EMC consideran que la ejecución de cada paso de simulación en un 1ms es el tiempo suficiente para alcanzar el tiempo real.

- Las redes mayores de 10 mil neuronas son aquellas que se consideran prácticas para estudiar y realizar experimentos neurocientíficos; ello pues el comportamiento dinámico de una red pequeña con una grande puede cambiar. [3] [4] [5]

1.2.2 Investigaciones similares

En el trabajo realizado por Chatzikonstantis et al. [3], se proponen optimizaciones para el flujo computacional de este modelo para simulaciones de la región ION del cerebelo utilizando procesadores Intel Xeon/Xeon Phi. Su propuesta incluyó trabajar con técnicas mixtas de computación de avanzada como computación con memoria compartida con OpenMP y distribuida con MPI, con manejo de variables vectoriales con los aceleradores Phi. Esta investigación obtuvo los mejores resultados para este tipo de sistemas usando las implementaciones con memoria compartida.

De acuerdo a la publicación de Nair et al. [6], ellos elaboraron una exploración de simulaciones eficientes de modelos computacionales neurocientíficos, utilizando los modelos de Hodgkin-Huxley y *Adaptive Exponential Leaky Integrate and Fire* (Integración adaptativa de exponencial con fuga y disparo). La implementación consistió en un uso combinado de múltiples aceleradores gráficos o GPUs comunicados por MPI. Se demostró como el rendimiento se incrementa hasta cinco veces más al utilizar múltiples GPUs que con la implementación con únicamente MPI.

Según Smaragdos et al. [4], el mayor reto de elaborar simulaciones biológicamente representativas consiste en solventar la carga computacional y de comunicación masiva entre los núcleos de procesamiento, aspectos que hacen que una solución convencional en un computador se quede pronto corta de recursos. De acuerdo con Smaragdos et al. [4] algunos métodos alternativos evaluados son:

- Uso GPUs para la ejecución paralela de cómputo de operaciones coma flotante. En redes muy grandes, el cuello de botella recae en la comunicación de las variables. Además términos energéticos, los GPUs son ineficientes.

- Supercomputadoras, multinúcleo. Estas se vuelven inmanejables por su poca portabilidad física, uso inmenso de espacio físico, costo energético y mantenimiento los vuelve en una solución muy costosa.

- Simulaciones analógicas o de señal mixta con circuitos integrados. Son muy eficientes en procesamiento y potencia, pero son difíciles de implementar, y requieren múltiples ajustes.

Por esta razón, una opción más reproducible en ser reproducible y eficiente de acuerdo al costo energético es el diseño de un sistema digital adecuado al problema, implementado en una FPGA, por cumplir tiempo de procesamiento determinista, y consumir mucho menos espacio físico y energía.

La utilización de lógica programable para aceleradores de hardware en aplicaciones de neurociencia ha sido trabajado por otras investigaciones como:

- Upegui et al. [7] describen en su investigación una implementación basada en un modelo simplificado de integración y disparo (IaF por sus siglas en inglés), donde el potencial de membrana de la célula es activado en relación a un umbral definido. En este trabajo, se evalúa una red neuronal de tres capas, cada una compuesta por diez células, en un FPGA Spartan II.

- Shayani et al. [8], construyeron una red de 161 neuronas y 1610 conexiones sinápticas en un sistema FPGA Virtex-5 basado en un modelo IaF cuadrático.

- Los modelos basados en IaF son una simplificación del comportamiento biológico de las neuronas con el fin de alcanzar simulaciones de alta densidad de neuronas interconectadas eficientemente, de acuerdo con Izhikevich[9]. El punto débil de resumir el modelo consiste en perder el control de los procesos químicos y biológicos de las neuronas, porque los investigadores tratan de vincular procesos electroquímicos con los estímulos que ocurren en las neuronas.

- En el proyecto de investigación desarrollado por Moore et al. [10], utilizan una plataforma con múltiples dispositivos de lógica programable. Llamado BlueHive y construido a partir de 64 dispositivos por FPGA, el sistema cuenta con un adaptador PCIe-SATA personalizado. Lograron simulaciones de 64 mil por FPGA, cada una con 64 millones de conexiones sinápticas a sus vecinos, logrando una mejora de 162 veces el rendimiento con respecto a un procesador común.

En resumen, el uso de procesadores de propósito general no es la solución más eficiente para la mayoría de tipos de redes neuronales; existen estudios con aceleradores de hardware de GPUs o FPGAs, pero ninguno ha elaborado una propuesta para el modelo eHH. Se requiere realizar investigación para una solución computacional utilizando múltiples FPGAs para mejorar el rendimiento de cálculo de las simulaciones de tipo SNN basadas en el modelo eHH. La tarea más difícil para programar en FPGAs, es el requisito de

conocimiento del área de diseño y validación de hardware por parte de los programadores para aprovechar el paralelismo de cálculo de la plataforma.

1.2.3 Síntesis

Se requiere profundizar en técnicas de computación de alto rendimiento, para acelerar la simulación de redes neuronales de alta densidad para promover la validación de hipótesis neurocientíficas por medio de simulaciones cerebrales.

1.3 Enfoque de la solución

Se abordó el problema utilizando una FPGA como acelerador de hardware, con el fin de aprovechar el paralelismo disponible en ella. El chip que se utilizó es el ZYNQ-7000 (Z-7020), que consiste en un SoC de una arquitectura heterogénea donde se combina hardware programable y un sistema constituido por un microprocesador ARM Cortex A9 dual-core, un controlador de memoria y periféricos empotrados. En el caso de este proyecto, el chip forma parte de una placa demo Zedboard de Digilent Inc, que incluye aparte del SoC varios dispositivos periféricos.

Primero se desarrolló un módulo sintetizable en la FPGA que resuelva los cálculos respectivos del algoritmo discutido anteriormente; este mismo se obtuvo por medio de la metodología de diseño de síntesis de alto nivel (HLS de High Level Synthesis), a partir del código fuente de alto nivel del algoritmo, que se traduce en una descripción de hardware implementable en hardware programable.

Las optimizaciones realizadas son similares a la solución propuesta por Smaragdos [5], quién decide mapear las variables locales de la red neuronal en BRAM, e independiza los procesos de la simulación de acuerdo a los componentes constituyentes de la red neuronal: dendrita, soma y axon; la ilustración de esta arquitectura se muestra en el diagrama de bloques de la figura 1.2 (más adelante, se ofrecerá una descripción más completa de la arquitectura interna de los dispositivos utilizados).

Para poder controlar el módulo anterior en el SoC, se adaptó el mismo a un protocolo de bus de comunicación. Las arquitecturas de bus que se utilizaron son AXI-Lite (datos mapeados en memoria) y AXI-Stream (datos tomados y escritos de memoria por DMA).

1.3.1 Objetivo general

Desarrollar una implementación del algoritmo eHH-ION paralelizable en múltiples FPGAs.

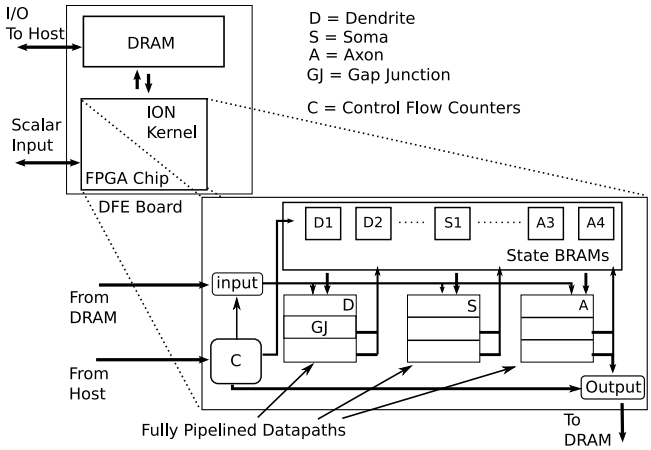

Figura 1.2: Diagrama del modelo computacional propuesto por Smaragdos. Fuente: [5]

1.3.2 Objetivos específicos

1. Generar un módulo del algoritmo eHH-ION sintetizado por Vivado HLS.

2. Proponer una arquitectura de interconexión de bus entre el módulo obtenido en el punto anterior con el procesador ARM Cortex A9, de manera que se maximice la tasa de transferencia de datos.

3. Implementar el software del microprocesador, de manera que se gobierne la FPGA por medio de los protocolos de bus seleccionados anteriormente.

1.3.3 Estructura del documento

El capítulo 2 se presentan las bases teóricas al proyecto como el funcionamiento del algoritmo y de las herramientas utilizadas. En el capítulo 3 se desarrollan las metodologías de manejo de de *IP Cores* construidos por Vivado HLS tanto a nivel *bare-metal* como por sistema operativo. El capítulo 4 se detallará la implementación del IP Core del algoritmo eHH tanto *bare-metal* como por sistema operativo,y los resultados pertinentes obtenidos. Finalmente en, el capítulo 5 se encuentran las conclusiones del proyecto, y se ofrecen recomendaciones para proyectos futuros.

Capítulo 2

Marco teórico

En este capítulo se abordan los temas de teoría de mayor interés para este proyecto. Se hace mención del modelo matemático utilizado para definir el comportamiento de una neurona, específicamente del ION. Se introduce un poco sobre la incursión de FPGAs para la computación de alto rendimiento. Posteriormente, se introduce la plataforma de desarrollo ZedBoard con la cual se elaboró el proyecto. Después, se introduce la metodología de diseño HLS para síntesis de módulos implementables en FPGA. Finalmente, se mencionan el grupo de interfaces que se utilizaron en este proyecto y su principio de funcionamiento

2.1 Modelo del núcleo inferior olivar (ION por inferior-olive nuclei)

El cerebro humano es un sistema complejo: las células nerviosas que lo componen son llamadas comúnmente neuronas y juntas constituyen una conglomeración de interconexiones entre ellas, las interconexiones también se denominan conexiones sinápticas. Han habido grandes avances para la creación de modelos matemáticos suficientemente realistas de estas células que no solamente simulan su comportamiento abstracto sino que revelan también, con gran detalle, su funcionamiento biológico y electroquímico. Las redes neuronales de espiga (SNN por Spike Neuronal Network) son un tipo de red neuronal que incluye un gran nivel de realismo y permite observar características como amplitudes de trenes de impulsos, frecuencias de oscilación y tiempo precisos de llegada de pulsos [11].

El modelo matemático en estudio es el eHH, el cual consiste en uno de los más completos. Este modelo consiste en un conjunto de ecuaciones diferenciales no lineales. Para su resolución se utiliza el método de Euler y se determinó que para su simulación representativa cada cada paso de simulación equivale a $50\mu s$ de actividad cerebral real [11]. En este proyecto, se enfocará el estudio en esta zona, por medio del modelo eHH.

La región olivar inferior juega un rol vital para el funcionamiento del cerebelo; por ejem-

plo, se ha indicado en estudios realizados que en lesiones localizadas en esta zona o en interrupciones de las conexiones de la ION y el cerebelo causan problemas motores como ataxia, nistagmo y distonía [2].

El modelo utilizado en este proyecto fue tomado del trabajo elaborado por Gruijl et al.[2]. En la figura 2.1, se muestra el esquemático del modelo neurocientífico de una neurona, dividido en tres secciones: dendrita, soma y axón. Cada una de ellas muestra las corrientes eléctricas que afectan las variables de estado de ellas; las ecuaciones matemáticas que rigen éstas mismas se definen en las tablas 2.1, 2.2 y 2.3. Cada unidad tiene una corriente de fuga pasiva de acuerdo al modelo, definida como:

$$I_{leak} = G_{leak}(V - V_{leak})$$
$$G_{leak} = 0,016\text{mS/cm}^2$$

Similarmente, cada unidad (dendrita, soma o axón) comparte una interacción con su vecina, cuya corriente se modela de manera pasiva y se define como:

$$I_{inter} = \frac{G_{inter}}{p_{a,b}}(V_a - V_b)$$
$$G_{inter} = 0.13\text{mS/cm}^2$$

Dónde $p_{a,b}$ es la razón de superficie entre las unidades, y sus valores corresponden a:

$$dendrita : soma = 4 : 1$$
$$soma : axon = 20 : 3$$

Para el modelo de los acoples de uniones de comunicación entre las dendritas de las neuronas se utiliza la ecuación:

$$w = 0.8e^{-V^2/100} + 0.2$$

donde V representa la diferencia de potencial entre las membranas dendritas de las celdas conectadas. Para una mayor profundización del modelo se puede revisar el trabajo de Gruijl et al.[2].

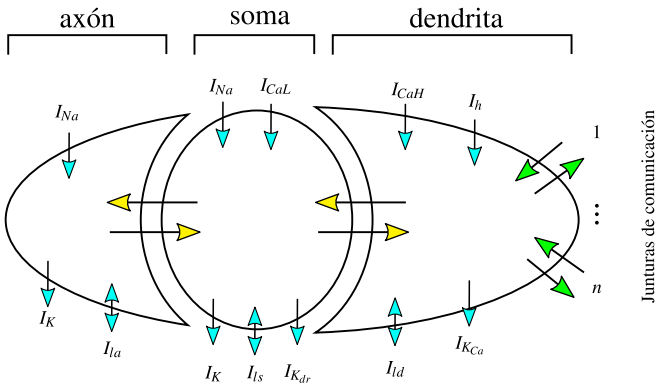

Figura 2.1: Representación esquemática del modelo utilizado para representar una neurona. En la imagen se aprecian las tres unidades principales: axón, soma y dendrita respectivamente. Asimismo se indican las diferentes corrientes que determinan el comportamiento de cada unidad.

Tabla 2.1: Ecuaciones que describen el comportamiento del soma

	Corriente	Activación	Inactivación
Calcio bajo umbral	$I_{CaL} = G_{CaL}k^3l(V - V_{Ca})$ $G_{CaL} = 0.7\text{mS/cm}^2$ (default) $0.55\text{mS/cm}^2 \leq G_{CaL} \leq 0.9\text{mS/cm}^2$ (range)	$k_\infty = \frac{1}{1+e^{-v-61/4.2}}$ $\tau_k = 1$	$l_\infty = \frac{1}{1+e^{-v-61/4.2}}$ $\tau_l = \frac{20e^{V+160/30}}{1+e^{V+84/7.3}} + 35$
Sodio	$I_{Na} = G_{Na}m_\infty^3 l(V - V_{Ca})$ $G_{Na} = 120\text{mS/cm}^2$	$m_\infty = \frac{1}{1+e^{-V-30/5.5}}$	$h_\infty = \frac{1}{1+e^{-V-70/-5.8}}$ $\tau_h = 3e^{(-V-40)/33}$
Potasio, componente tardío	$I_{Kdr} = G_{Kdr}np(V - V_K)$ $G_{Kdr} = 9\text{mS/cm}^2$	$n_\infty = \frac{1}{1+e^{-V-3/10}}$ $\tau_n = 47e^{(-V-50)/900} + 5$	$p_\infty = \frac{1}{1+e^{-V-51/-12}}$ $\tau_n = 47e^{(-V-50)/900} + 5$
Potasio, componente rápido	$I_K = G_K x^4(V - V_K)$ $G_K = 5\text{mS/cm}^2$	$\alpha_x = \frac{0.13V+3.25}{1-e^{-V+25/10}}$ $\beta_x = 1.69e^{-0.0125V-0.4375}$ $x_\infty = \frac{\alpha_x}{\alpha_x+\beta_x}$ $\tau_x = \frac{1}{\alpha_x+\beta_x}$	

Tabla 2.2: Ecuaciones que describen el comportamiento del axón

	Corriente	Activación	Inactivación
Sodio	$I_{Na} = G_{Na}m_\infty^3 h(V - V_{Na})$ $G_{Na} = 240\text{mS/cm}^2$	$m_\infty = \frac{1}{1+e^{-V-30/5.5}}$	$h_\infty = \frac{1}{1+e^{-V-60/-5.8}}$ $\tau_h = 3e^{(-V-40)/33}$
Potasio	$I_K = G_K x^4(V - V_K)$ $G_K = 20\text{mS/cm}^2$	$\alpha_x = \frac{0.13V+3.25}{1-e^{-V+25/10}}$ $\beta_x = 1.69e^{-0.0125V-0.4375}$ $x_\infty = \frac{\alpha_x}{\alpha_x+\beta_x}$ $\tau_x = \frac{1}{\alpha_x+\beta_x}$	

Tabla 2.3: Ecuaciones que describen el comportamiento del dendrita

	Corriente	Activación
Calcio umbral alto	$I_{CaH} = G_{CaH}r^3 l(V - V_{Ca})$ $G_{CaH} = 4.5\text{mS/cm}^2$ $\frac{\partial[Ca^{2+}]}{\partial t} = -3I_{CaH} - 0.075[Ca^{2+}]$	$\alpha_r = \frac{1.7}{1+e^{-V-5/13.9}}$ $\beta_r = \frac{0.02V+1.7}{e^{V+8.5/5}-1}$ $r_\infty = \frac{\alpha_r}{\alpha_r+\beta_r}$ $\tau_r = \frac{5}{\alpha_r+\beta_r}$
Potasio dependiente de calcio	$I_{K_{Ca}} = G_{K_{Ca}}s(V - V_k)$ $G_{K_{Ca}} = 35\text{mS/cm}^2$	$\alpha_s = min(0.00002[Ca^{2+}]\|0.01)$ $\beta_s = 0.015$ $s_\infty = \frac{\alpha_s}{\alpha_s+\beta_s}$ $\tau_s = \frac{1}{\alpha_s+\beta_s}$
Corriente activada por polarización	$I_h = G_h q(V - V_h)$ $G_h = 0.15\text{mS/cm}^2$	$q_\infty = \frac{1}{1+e^{V+80/4}}$ $\tau = \frac{1}{e^{-0.086V-14.6}} + e^{0.07V-1.87}$

2.2 FPGAs para computación de alto rendimiento

En la última década se ha iniciado una tendencia para los problemas del campo de computación paralela: la incorporación de hardware diseñado para solucionar problemas específicos, ya se son más energéticamente eficientes que su equivalente por varios núcleos de procesamiento general. A esta clase de dispositivos se les conoce como aceleradores de hardware. El término general se refiere a todos aquellos sistemas computacionales que cuentan con unidades de coprocesamiento cuyo fin es mejorar el tiempo de cálculo de problemas específicos. Estas unidades son más eficientes que los procesadores de propósito general porque su diseño explota el paralelismo de los algoritmos que resuelve de una manera más óptima en hardware. Algunos aceleradores de hardware más comunes son las unidades de procesamiento gráfico (GPU) y los arreglos de compuertas programables en campo (FPGA por field programmable gate arrays). Los GPUs son optimizados para ejecutar altas cantidades de operaciones coma flotantes en el menor tiempo posible, pero su arquitectura estática obliga al desarrollador adaptar su algoritmo en función del GPU para sacarle el mayor provecho posible. Estas unidades trabajan a una frecuencia de reloj más lenta que los procesadores de propósito general, pero más rápida que las FPGAs [12].

Las FPGAs son plataformas flexibles, que se moldean de acuerdo a las necesidades de la aplicación, porque su funcionamiento interno consiste en un arreglo denso de bloques básicos lógicos programables, los cuales permiten tener resultados similares a un circuito integrado especializado en el problema específico. Las FPGAs modernas proveen principalmente los siguientes bloques programables:

- **CLB**: bloque lógico configurable (configurable logic block) de una FPGA, puede adaptarse en cualquier función booleana usando tablas de busqueda (LUT por look-up tables), puede almacenar datos en registros biestables (FF por Flip-flops) o alguna operación aritmética en sus sumas completas (FA por full-adder)[13].

- **DSP Slices**: unidad optimizada para cálculos ariméticos de multiplicación y suma. Son orientados para utilizarse en problemas de alta demanda computacional de cálculo, por ello son recursos deseables para tener en abundancia para estos fines[13].

- **BRAM**: bloques de RAM incluidos dentro de la FPGA, su importancia funcional es almacenar colecciones de datos utilizados por las unidades de procesamiento[13].

El factor determinante de la FPGA es su posibilidad para paralelizar el computo de problemas, cuya única limitante es el espacio físico disponible en ellas. Normalmente las aplicaciones implementadas en FPGA son más eficientes energéticamente en comparación con los GPUs pero el proceso de diseño en ellas puede llegar a ser más costoso [14].

2.3 ZedBoard ZYNQ System on Chip

ZedBoard es el nombre de la tarjeta de desarrollo utilizada en este proyecto, fabricada por Digilent Inc, y que, integra dentro de ella el SoC ZYNQ XC7Z020-CLG484-1. Entre las especificaciones más relevantes de la tarjeta para este proyecto se incluyen: 512MB RAM DDR3, USB UART, 10/100/1000 Ethernet, USB JTAG, entrada 12V DC y compartimiento para tarjeta SD.

El SoC ZYNQ usado en el proyecto integra un procesador ARM Cortex-A9 doble núcleo y una región de hardware programable; la región llamada sistema de procesamiento (PS en sus siglas en inglés) incluye el procesador, un controlador de memoria externa y conectividad de interfaces de diferentes clases de periféricos; y la otra región nombrada lógica programable (PL en sus siglas en inglés) incluye bloques de lógica configurable, bloques RAM (BRAM en sus siglas en inglés) y bloques DSP [15].

El diagrama interno del SoC ZYNQ se puede observar con más detalle en la figura 2.2, de este se puede diferenciar las dos regiones PS y PL, además de los periféricos I/O disponibles en el mismo; la comunicación entre el PS y el PL se realiza por medio de los puertos de propósito general (General Purpose en inglés), alto rendimiento (High Performance en inglés) o puerto acelerador coherente (Accelerator Coherency Port en inglés) [15].

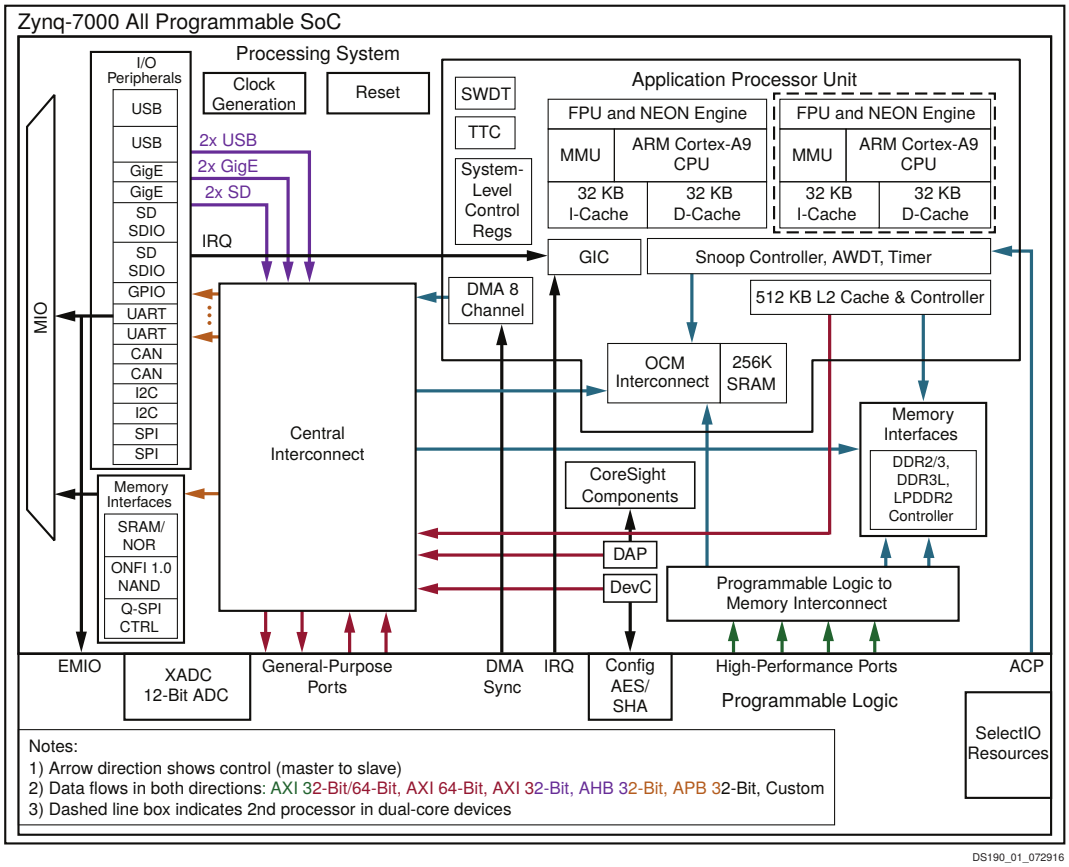

Figura 2.2: Diagrama interno de la arquitectura del SoC ZYNQ. Fuente: [15]

2.4 Vivado HLS

Una de las ventajas de utilizar Vivado HLS es de utilizar una metodología de generación de código RTL (que normalmente se realiza sobre un lenguaje de descripción de hardware, HDL por Hardware Description Language). Mediante el uso de directivas escritas ya sea sobre el código fuente de alto nivel, o en scripts con directivas TCL es posible sintetizar un módulo funcional para incorporar en una FPGA. [16]

Estas directivas, llamadas pragmas en el caso de encotrarse en el código fuente, guían a la herramienta de síntesis para elaborar el flujo de datos, ciclos de ejecución, paralelismo y compatibilidad con la interconexión con alguna arquitectura de bus.

Vivado HLS permite validar el módulo generado por medio de una simulación funcional RTL. Para ello se utiliza un programa de testbench desarrollado al mismo tiempo que la función de alto nivel dirigida para llevarla a hardware. El módulo ya sintetizado puede exportar como *IP-Core* (núcleo con propiedad intelectual , Intelectual property core) para luego ser importado en algún proyecto de Vivado para luego ser sintetizado en una FPGA. [16]

2.5 Especificaciones AMBA®

AMBA son las siglas de Advanced Microcontroller Bus Architecture. Este es un estándar abierto sobre las especificaciones de interconexiones entre bloques funcionales dentro de un SoC. Este facilita el desarrollo de diseños con múltiples procesadores con grandes números de controladores y periféricos. AMBA promueve la reutilización de módulos en los diseños de SoC al estandarizar las interfaces.

Existen diversos acrónimos asociados con AMBA y los dos más populares tratan del bus avanzado de alto rendimiento (AHB por Advanced High-Performance Bus) e la interfaz avanzada extensible (AXI por Advanced eXtensible Interface).

Para el 2010, las especificaciones AMBA 4 fueron introducidas, y así mismo se inicio la adopción de AMBA 4 AXI4. Este protocolo es ampliamente utilizado por los procesadores ARM Cortex-A9 y Cortex-A15. En el caso de AXI4, existen tres interfaces: AXI4, AXI4-Lite y AXI4-Stream.

2.5.1 AXI4

La arquitectura del protocolo AXI4 se encuentra constituida por cinco buses: bus de dirección de lectura, bus de lectura de datos, bus de dirección de escritura, bus de escritura de datos y respuesta de escritura. Los buses de dirección de datos contienen variables de control para describir la naturaleza de los datos por ser transferidos. El bus de escritura transfiere datos desde el maestro al esclavo; este último avisa por el bus de respuesta de

escritura la llegada correcta de los datos. Este protocolo da soporte al envío de datos por ráfagas; lo cual se refiere, la transacción se realiza por el envío de una secuencia de datos en cola. Un ejemplo del funcionamiento de este protocolo se muestra en la figura 2.3. Este protocolo es utilizado en accesos de memoria mapeada de alto rendimiento [17].

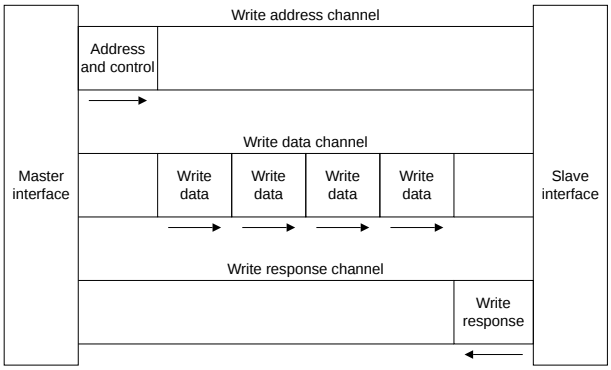

Figura 2.3: Diagrama de la arquitectura del canal AXI4 para escritura de datos por medio de ráfaga. El puerto esclavo da señal de lallegada correcta de datos y permiso para atender más datos. Fuente:[17]

2.5.2 AXI4-Lite

Este protocolo cumple el mismo funcionamiento del AXI4, pero se distingue principalmente por tener deshabilitada la comunicación por ráfaga. Este protocolo es implementado para bajos volumenes de datos utilizando la metodología de mapeo en memoria de dispositivos. [17].

2.5.3 AXI4-Stream

El protocolo AXI4-Stream es usado como una interfaz de conexión entre componentes para movilizar secuencias de datos donde el manejo de direcciones de memoria no esta presente o no es requerido. Cada canal de transminisión está dirigido hacia un único sentido. Esta especificación permite la optimización de rendimiento de aplicaciones adecuando el flujo datos del sistema.

En algunas ocaciones, se desea construir sistemas que combinen los protocolos AXI4-Stream y AXI4. El encargado de manejar estas traducciones trata de un mecanismo de acceso directo a memoria (DMA-engine por Direct Memory Access Engine) quien moviliza datos localizados desde memoria para ser transportados por una secuencia de datos en interfaz AXI4-Stream y/o viceversa [18]. Este mecanismo puede ejemplificarse por el AXI-DMA IP encontrado en la biblioteca de IPs de Xilinx preparado para esta clase de trabajos, el diagrama interno de este IP se muestra en la figura 2.4 del cual puede observarse como se hace la traducción de los protocolos AXI4 y AXI4-Stream.

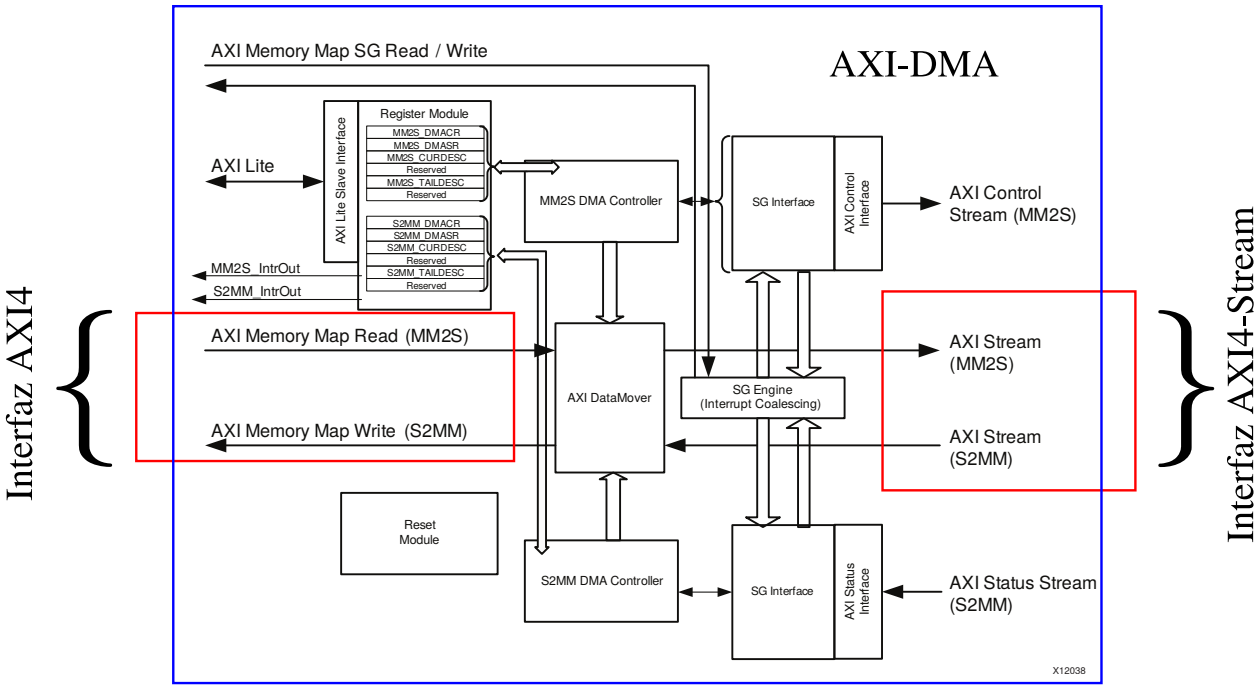

Figura 2.4: Diagrama de arquitectura del AXI-DMA IP encontrado en la biblioteca de IPs de Xilinx. Se puede observar la localización de los protocolos AXI4 y AXI4-Stream, y como el bloque AXI-DataMover realiza la traducción de ambos. El mecanismo anterior se programa/prepara mediante los registros de control localizados en su interfaz AXI4-Lite. Fuente:[19]

Capítulo 3

Desarrollo e integración del ION IP Core dentro de la plataforma de prueba ZedBoard

En este capítulo se aborda el proceso de diseño para elaborar un IP Core a partir del código sintetizado en alto nivel del ION e integrarlo dentro de la plataforma ZedBoard, ya sea para manejarlo desde una implementación *bare-metal* o con un sistema operativo basado en Linux. Finalmente se muestra una evaluación de velocidad de transferencia de datos entre la memoria RAM y la región de PL de la ZYNQ mediante las interfaces AXI4-Lite y AXI4-Stream.

3.1 Generación del IP Core con Vivado HLS

De acuerdo a la documentación de la herramienta, el desarrollo en Vivado HLS se comprende por dos procesos: uno de síntesis de HDL y otro de verificación [16]. El primero consiste en la asignación de configuraciones como: la plataforma de desarrollo, periodo de reloj, directivas de síntesis e interfaz; y el segundo se encarga de comprobar el funcionamiento global del IP sintetizado. El diagrama del proceso de desarrollo de Vivado HLS se muestra en la figura 3.1. Se muestra cómo primero se incorpora un programa de referencia escrito en un lenguaje de alto nivel quien describe la función deseada de sintetizar en HDL. Para verificar este código se realiza la simulación en C. Esta hace la prueba propuesta por el usuario para encontrar fallos dentro del programa de C; en caso de encontrarlos, la prueba da aviso de una inconsistencia funcional del código que se desea sintetizar. De acuerdo con el flujo, se incorporan las bibliotecas de C para síntesis de HLS, con el fin de utilizar tipos de estructuras congruentes a aquellas de hardware; por ejemplo, la biblioteca `ap_fixed.h` permite trabajar con variables de enteros con precisión arbitraria. Otra clase de guía para la síntesis, son las directivas[16]. Las directivas de síntesis se incorporan adrede para señalar variables, bucles y/o funciones para ser asociadas con

clases de interfaces, tipos de memorias y estilos de flujos de datos para el fin de mejorar el rendimiento de cálculo del RTL sintetizado. Después de realizar estos ajustes, se puede generar la síntesis a nivel RTL a partir del código de alto nivel más las directivas y bibliotecas utilizadas. Para asegurarse de no encontrarse errores en la traducción generada por la herramienta, se debe ejecutar la co-simulación del código RTL. La co-simulación adapta la misma prueba realizada desde la simulación C para probar el comportamiento funcional del código RTL. Si la simulación no encuentra anomalías, se puede proceder a finalizar el ciclo de desarrollo al empacar el RTL en un *IP Core* y ser llamado en un proyecto de Vivado[16].

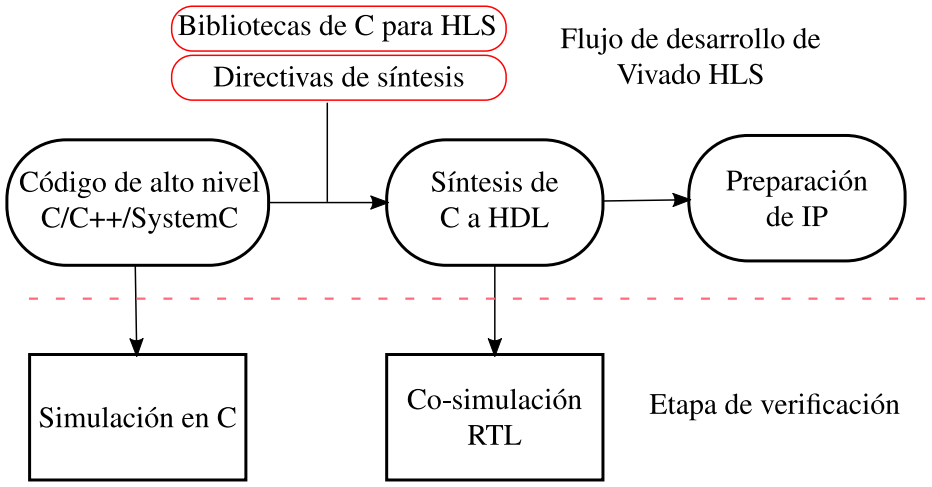

Figura 3.1: Diagrama de flujo del desarrollo de un *IP Core* usando Vivado HLS. El programa escrito en un lenguaje de programación de alto nivel describe un comportamiento que se busca replicar en un módulo de hardware sintetizable. El procedimiento consiste en desarrollar el código de alto nivel y simular su comportamiento hasta tener resultados satisfactorios. Despúes se incorporan bibliotecas de C para HLS y directivas de síntesis para guiar la herramienta de síntesis en una solución apropiada que cumpla especificaciones de uso de hardware y rendimiento computacional. El código generado por la herramienta de síntesis se encuentra escrito en HDL. Este códiga se valida por la cosimulación RTL. Finalmente se empaca el código HDL en un *IP Core* para luego ser incorporado en un diseño de un proyecto de Vivado para ser implementado en FPGA.

3.2 Incorporación de interfaces AXI en un IP Core

Para incorporar el IP Core en un sistema empotrado, se puede usar la interfaz estándar AXI, que es totalmente soportada por el PS de la plataforma Zynq. Hay dos tipos fundamentales usados en este proyecto: el AXI4-Lite y el AXI4-Stream (ya descritos en 2.5). El primero registra las variables de entrada y/o salida del *IP Core* como registros mapeados en memoria. Por ejemplo, el siguiente código ejemplifica como añadir una interfaz AXI4-Lite a una función en C++, que crearía una interfaz hardware como la que se muestra en la figura 3.2.

Listado de código 3.1: Código escrito en el lenguaje de programación C++ que describe la asociación entre los argumentos de la función con la interfaz AXI4-Lite. El vínculo se realiza con el uso de directivas de síntesis, escritas adrede sobre la cabeza de instrucciones de la función.

```
void adder(float A[SIZE], float B[SIZE], float Q[SIZE]){
        #pragma HLS INTERFACE s_axilite port=return
        #pragma HLS INTERFACE s_axilite register port=A
        #pragma HLS INTERFACE s_axilite register port=B
        #pragma HLS INTERFACE s_axilite register port=Q
        ...
}
```

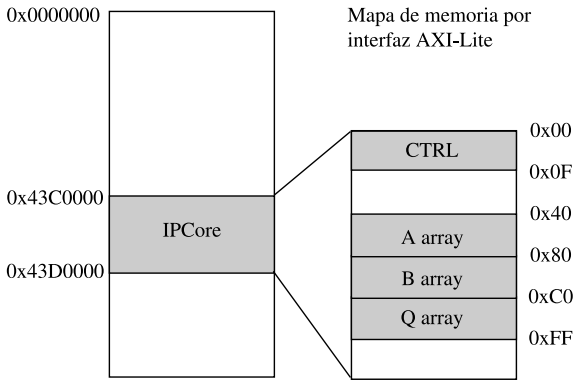

Figura 3.2: Diagrama del mapa de memoria al incorporar el *IP Core* con uso de interfaz AXI4-Lite de acuerdo a las especificaciones escritas en el listado de código 3.1. Las variables `A`, `B` y `Q` son mapeadas en registros de 32 bits cada una. Se incorpora un registro adicional de un byte, `CTRL` para manejar el control del IP Core.

3.2.1 Manejo de la interfaz AXI4-Stream en la plataforma Zynq-7000

Una interfaz AXI4-Stream permite trasegar datos que llegan por ráfaga continua, tal como señales de video, señales muestreadas por convertidores analógicos a digital, datos vectorizados que arriban con una cierta secuencia en bloque, entre otros. Esto puede resultar útil para administrar transferencias de largas cadenas de datos entre una memoria RAM y el IP Core, sin la necesidad de intervención del microprocesador; para lograr este cometido, se debe incorporar al proyecto de Vivado el *IP Core* AXI-Direct Memory Access o `AXI-DMA` de Xilinx, quien se encarga de recolectar los datos de la RAM por medio del bus AXI4 HP/ACP del Zynq 7000. En la figura 3.3 se ilustra el diagrama de conexiones entre el *IP Core* generado por HLS y el *IP Core* `AXI-DMA` para procesar información desde la RAM y regreserla a la misma; los registros de control del `AXI-DMA` se encuentran mapeados en memoria por AXI4-Lite, lo que permita programar las direcciones de memoria para leer y escribir de la RAM.

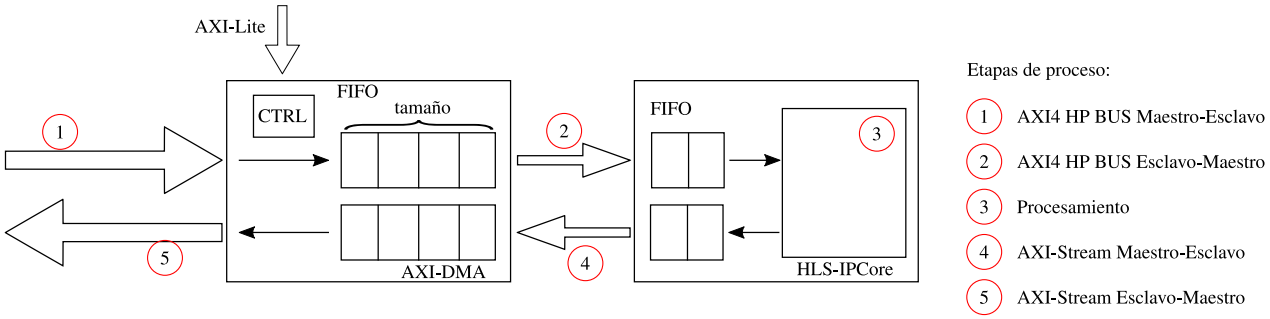

Figura 3.3: Diagrama de bloques sobre la utilización del *IP Core* con interfaz AXI4-Stream y manejado por el `AXI-DMA`. Nótese que el IP Core para procesar datos desde memoria, estos son recibidos y entregados por el `AXI-DMA`.

3.2.2 Configuración del protocolo AXI4-Stream en la síntesis de alto nivel de un núcleo

Para utilizar interfaz AXI4-Stream para un argumento de la función escrita en C/C++, existen dos opciones: la implementación sencilla o la implementación con señales de múltiples canales. La primera trata únicamente de dos señales de control: `TVALID` para asegurar la lectura del dato cuando es válido y `TREADY` para informar cuando el puerto esclavo está listo para recibir otro dato; y un puerto para el paso del dato, `TDATA`.

El `AXI-DMA` utiliza la interfaz de AXI4-Stream con manejo de múltiples canales, que requiere implementar el *IP Core* generado por el HLS con las siguientes señales adicionales: `TDEST`, `TKEEP`, `TSTRB`, `TUSER`, `TLAST` y `TID`. De estas la más relevante es `TLAST` porque señala el envío de un segundo dato, por ejemplo un segundo arreglo. El listado de código 3.2, escrito en C++, ejemplifica la construcción en HLS el protocolo de múltiples canales. Aquí se muestra la abstracción del protocolo con múltiples canales por medio de una estructura de datos de C++; la que se importa de la biblioteca `ap_axi_sdata.h`. Los argumentos `A` y `B` son variables tipo `ap_axis`, estructura modificada por el `template` de C++. De acuerdo con el ejemplo: `ap_axis<32,2,5,6>`, el `template` adapta las variables `data` con un largo de 32 bits, `user` con 2 bits, `id` con 5 bits y `dest` con 6 bits. Dado que se tiene acceso a estas señales de control del protocolo AXI4-Stream para múltiples canales, es posible tomar decisiones a partir de la información brindada, como discriminar información proveniente de diferentes canales con las variables `user`, `id` y `dest` [16]; asimismo guiar los datos resultantes de las operaciones a los canales deseados.

Listado de código 3.2: Código escrito en C++ con la intención de ejemplificar la utilización de la biblioteca `ap_axi_sdata.h` para trabajar con el protocolo AXI4-Stream con múltiples canales. La variable `A` es tipada como `ap_axis`, con la función de abstraer el canal de ingreso de datos con interfaz FIFO. De igual manera, la variable `B` abstrae los resultados de la función por medio de otro canal FIFO. Basicamente todos los datos ingresados por `A` son recuperados como enteros de 32 bits, se les realiza la operación de suma con cinco y el resultado es escrito en el canal de salida de la variable `B`.

```
#include "ap_axi_sdata.h"
void example(ap_axis<32,2,5,6> A[50], ap_axis<32,2,5,6> B[50]){
        //Map ports to Vivado HLS interfaces
        #pragma HLS INTERFACE axis port=A
        #pragma HLS INTERFACE axis port=B
        int i;
        for(i = 0; i < 50; i++){
                B[i].data = A[i].data.to_int() + 5;
                B[i].keep = A[i].keep;
                B[i].strb = A[i].strb;
                B[i].user = A[i].user;
                B[i].last = A[i].last;
                B[i].id = A[i].id;
                B[i].dest = A[i].dest;
        }
}
```

3.2.3 Utilización del AXI-DMA en aplicaciones bare-metal

La herramienta de desarrollo de software de Xilinx (Xilinx SDK por sus siglas en inglés) permite hacer pruebas de los diseños de periféricos implementados en la región PL de FPGA de la ZYNQ sin la necesidad de incorporar un sistema operativo (es decir, estilo *bare metal*)[20]. El SDK se encarga del manejo de proyectos de software sin sistema operativo o aquellos no monolíticos como el freeRTOS, e incorpora bibliotecas de Xilinx para el manejo de *IP Cores*, tal como el `AXI-DMA`. Para usar el `AXI-DMA` se llama la biblioteca `xaxidma.h` dentro del código principal y se utilizan las funciones de `XAxiDma_SimpleTransfer`; un ejemplo de utilización pude verse en el listado 3.3. En este código se muestra que se requiere calcular el tamaño del arreglo que se deseea enviar y recibir, luego se inicializa la configuración del `AXI-DMA`, se invalida la cache de datos de los núcleos ARM para que el arreglo de datos `A` inicializado se asegure escribir en la RAM, se ejecutan las transferencias de los arreglos y finalmente se espera en un bucle hasta que el `AXI-DMA` levanta una bandera indicando la finalización de las transferencias.

Es importante resaltar que en caso de haber introducido en el código del HLS la directiva `#pragma HLS INTERFACE s_axilite port=return` el *IP Core* generado no admitirá

datos por AXI4-Stream hasta que el registro de control sea programado para iniciar el procesamiento. El *IP Core* creado por Vivado HLS incorpora una biblioteca con rutinas para facilitar el uso del mismo en proyectos del SDK; de acuerdo con esta misma, se llama la función `Xexample_Start` de la biblioteca `xexample.h` quien programa el registro de control para empezar a correr el flujo de datos por AXI4-Stream.

Listado de código 3.3: Código escrito en C para ejemplificar la utilización de la biblioteca `xaxidma.h`. El programa realiza una transacción de datos por el IP `AXI-DMA` usando sus dos canales `XAXIDMA_DEVICE_TO_DMA` y `XAXIDMA_DMA_TO_DEVICE`. El arreglo `A` es enviado a través del IP y el resultado es recuperado en el arreglo `B`. La finalización de la transferencia se realiza por método de `polling` o sondeo de las banderas de control del IP `AXI-DMA`. Para actualizar los datos localizados en la región de memoria del arreglo `B`, se limpia la memoria cache de datos.

```
#include "xaxidma.h"
XAxiDma AxiDma;
XAxiDma_Config *CfgPtr;
int A[50];
int B[50];
int dma_size = 50*sizeof(int);
int status;
...
main(){
        ...
        CfgPtr = XAxiDma_LookupConfig(
                XPAR_AXI_DMA_1_DEVICE_ID);
        if(!CfgPtr){
                print(
                "Error looking for AXI_DMA config\n\r");
                return XST_FAILURE;}
        status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);
        if(status != XST_SUCCESS){
                print("Error initializing DMA\n\r");
                return XST_FAILURE;}
        Xil_DCacheFlushRange((unsigned int)A, dma_size);
/*transfer A to the Vivado HLS block*/
        int status = XAxiDma_SimpleTransfer(
                &AxiDma,
                (unsigned int) A,
                dma_size,
                XAXIDMA_DMA_TO_DEVICE);
        if (status != XST_SUCCESS){
                return XST_FAILURE;}
/*get results from the Vivado HLS block*/
        status = XAxiDma_SimpleTransfer(
                &AxiDma,
```

```
                (unsigned int) B,
                dma_size,
                XAXIDMA_DEVICE_TO_DMA);
        if (status != XST_SUCCESS) {
                return XST_FAILURE;}
/* Wait for transfer to be done */
        while (
                (XAxiDma_Busy(&AxiDma, XAXIDMA_DEVICE_TO_DMA)) ||
                (XAxiDma_Busy(&AxiDma, XAXIDMA_DMA_TO_DEVICE)));
        Xil_DCacheFlushRange((unsigned int)B, dma_size);
        ...
```

3.3 Cómo crear una imagen de Linux reducida para ZedBoard para utilizar los IPs sintetizados en el PL

La opción de implementar aceleradores de hardware sobre plataformas *bare-metal* es mucho más eficiente en términos de recursos, pero mucho menos flexible y poco escalable. Es por ellos que se desarrolló una segunda versión de manejo del IP ION, esta vez incorporando al Zynq un sistema operativo reducido basado en Linux. El sistema operativo provee servicios de manera de interfaz para que las aplicaciones hagan uso de los distintos recursos de hardware contengan la arquitectura computacional que se disponga. En el caso de la ZYNQ, la región llamada PS contiene un diseño completo entre un conjunto periféricos comunes, controladores de DMA y memoria RAM DDR3 e interconexiones; por ello, todo estos dispositivos deben tomarse en cuenta en las configuraciones de compilación para hacer la imagen de Linux.

Para lograr utilizar los periféricos alojados en la región de PL o FPGA del ZYNQ se necesita utilizar los puertos de interconexión GP, HP o ACP. Dependiendo del caso, los *IP Cores* creados desde Vivado HLS deben adaptarse por medio de controladores (drivers en inglés) para que sean usados por las aplicaciones que son administradas por el sistema operativo.

En la figura 3.4 se presenta un diagrama del contenido requerido en una tarjeta SD quemada para arrancar una distribución de Linux para la ZedBoard. En ella se presentan dos particiones:

1. **Boot**. Partición con formato FAT32, los archivos que almacena son necesarios para el correcto arranque del sistema operativo. Los archivos encontrados aquí son: `BOOT.BIN`, `devicetree.db` y `uImage` [21].

2. **RootFS**. Aquí Región de tipo EXT4 donde se almacena el sistema de archivos de todos los programas, los directorios y los archivos precompilados para ser utilizados

por el usuario cuando se haya cargado el sistema operativo en el sistema. También se almacerán acá los archivos generados por las aplicaciones [21].

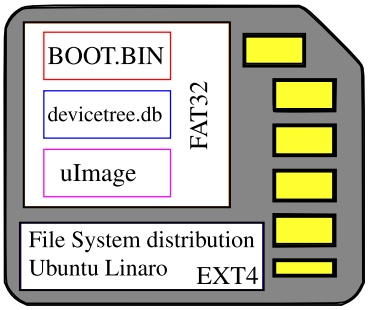

Figura 3.4: Diagrama del contenido requerido en una tarjeta SD, para arrancar una distribución de Linux en la ZedBoard.

3.3.1 Construcción del BOOT.BIN

El `BOOT.BIN` es el programa que se encarga del arranque del sistema y se conforma a su vez por dos rutinas o subprogramas y el *bitstream* que programa a la FPGA [21]. Los subprogramas, `FSBL.elf` (de las siglas en inglés *First Stage Bootloader* arranque de primera etapa) y el `U-Boot.elf` (del inglés *Universal Boot Loader*, o cargador de arranque universal) quien se encarga de correr la imagen del sistema operativo. El *bitstream* es el archivo de configuración de la FPGA que ha generado la herramienta Vivado de síntesis, colocación y ruteo a partir del HDL que describe la funcionalidad del sistema.

El `U-Boot.elf` fue compilado con base en la distribución del `U-Boot` de Digilent, recuperado del repositorio: https://github.com/Digilent/u-boot-digilent. Este mismo se debe adaptar a las configuraciones requeridas por la ZedBoard como mapa de memoria y opciones de arranque por SD. En este caso se desea arrancar con el `device-tree.dtb` tomado desde la SD y el `uImage` también. Se realiza la compilación cruzada por medio del tool-chain de Xilinx para plataforma ARM recuperado en línea desde https://github.com/Micro-Studios/Xilinx-Arm-Tool.

Una vez compilado el `U-Boot`, se abre un proyecto del SDK de Xilinx y se crea un proyecto de software de los ejemplos que ofrece (en este caso, se toma el del `FSBL`). Se realiza la compilación cruzada desde el SDK del `FSBL` y se abre la herramienta de construcción de `BOOT.BIN` del SDK. Se agregan los códigos binarios `U-Boot.elf`, `FSBL.elf` y el `hw-bitstream.hdf` para su creación[20].

3.3.2 Compilación de la imagen de Linux

La distribución del núcleo de Linux de Digilent se recuperó del repositorio en línea https://github.com/DigilentInc/Linux-Digilent-Dev. En él, se encuentra una configuración recomendada para la compilación cruzada de la imagen de Linux, incluyendo los

drivers necesarios para el ZYNQ; el único cambio realizado de la configuración estándar fue la deshabilitación del arranque del sistema del sistema de archivos por archivo `RAMdisk`, porque se planeó mantener el sistema de archivos localizado en la tarjeta SD en la partición `RootFS`. De lo contrario se requeriría construir el archivo `initramfs` o `initrd` y los cambios realizados del sistema de archivos durante el uso de la ZedBoard se perderían al apagar el sistema. Finalmente, por el `Makefile` se crea el archivo `uImage`.

3.3.3 Generación del archivo Device-tree

El `device-tree` es un código que reune información de los periféricos de la plataforma utilizada [21]. Aquí se describen parámetros y configuraciones importantes para los controladores o *drivers* del sistema operativo. Este código es compilado por el compilador de `device-tree` (dtc por sus siglas en inglés). Se utilizó el código preparado para la ZedBoard del repositorio de Digilent,del que se editaron los argumentos de arranque (bootargs de su acrónimo del inglés) que se reemplazaron por como se indica en el listado 3.4. Aquí tenía el fin de arrancar el sistema de archivos desde la segunda partición (`RootFS`) de la tarjeta SD.

Listado de código 3.4: Edición del `bootargs` para el arranque del núcleo de Linux en el archivo `device-tree`. Se incorpora principalmente la localización del *filesystem* en la segunda partición de la tarjeta SD.

```
bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw
earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=0;
```

3.3.4 Instalación del sistema de archivos de Linaro Ubuntu

Como último paso se quema una distribución de Ubuntu en el sistema de archivos. Se tomó la versión de Ubuntu 15.04 precompilada en los repositorios de Linaro, se copió en la partición RootFS de la tarjeta SD. En la figura 3.5, se brinda la información del sistema operativo completo en la ZedBoard ya después de haber arrancado exitosamente.

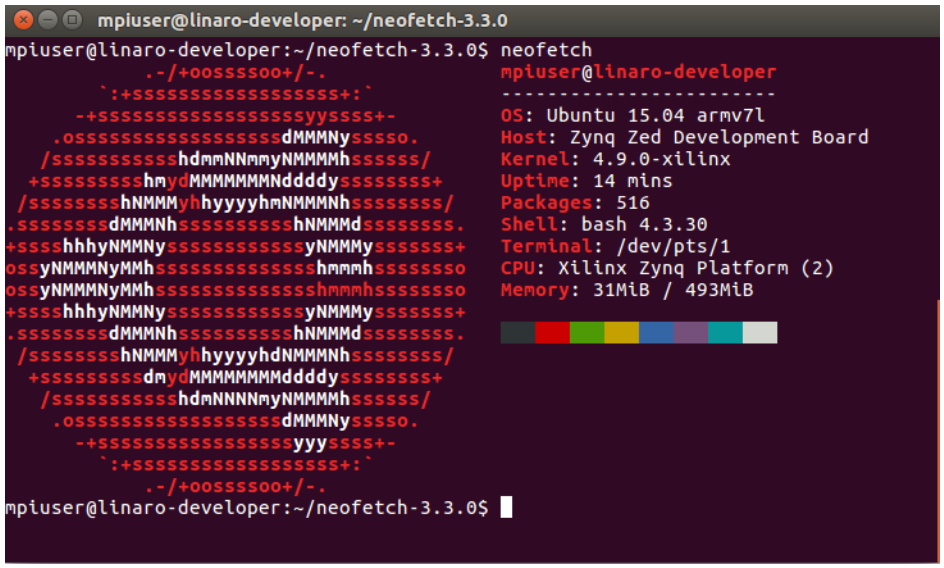

Figura 3.5: Información del sistema operativo recuperado de la ZedBoard. Aquí se ilustra la instalación correcta de la distribución de paquetes de Linux de `Ubuntu 15.04` para arquitectura `armv7l`. El núcleo de Linux corresponde a la versión 4.9.0-xilinx. Además, se indica tener habilitados hasta 493MB de memoria RAM, del que solo se encuentra en uso 31MB cuando el sistema se encuentra desocupado.

3.4 Manejo de IP Cores construidos por HLS desde el sistema operativo Linux

Una vez creada una distribución de Linux adecuada para las necesidades de la plataforma Zynq 7000, se requiere ahora poder llamar los periféricos construidos desde Vivado HLS en las aplicaciones montadas sobre el sistema operativo. Un *IP Cores* es un dispositivo de hardware que se inteconecta por medio de una interfaz al PL. El acceso al mismo debe hacerse por un controlador que hace transparente para el programador los detalles de interconexión. Para interfaces compatibles con AXI4-Lite, no se requiere elaborar controladores nuevos, porque para dispositivos mapeados en memoria se tiene la opción de adaptar uno ya trabajado por la comunidad de Linux. En este caso, se utilizó el controlador Userspace-IO (UIO), conocido como `uio_pdrv_genirq` en el núcleo de Linux.

En el caso particular de aquellos *IP Cores* con interfaz AXI4-Stream, se mantiene el estándar de ser controlados por el `AXI-DMA`. El `AXI-DMA` posee un controlador de Linux mantenido por Xilinx encargado de adaptar el uso del canal de DMA por las APIs del *kernel* conocidas como `DMA-Engine` y `DMA-Mapping`. Estas APIs deben usarse desde el espacio del núcleo de Linux por un controlador; esto último obliga a crear un controlador personalizado a las necesidades del *IP Core*.

3.4.1 Adaptación del controlador UIO para usar IPs con interfaz AXI-Lite

El controlador UIO fue creado por la comunidad de Linux para situaciones donde los periféricos son controlados por uso de registros programables mapeados en memoria. Para que una aplicación tenga acceso directo a estos dispositivos, este controlador simplemente ayuda a mapear direcciones de memoria física o virtual que el usuario desea escribir o leer. En la figura 3.6 se enseña un esquema del funcionamiento de la interfaz y su utilización en la aplicación.

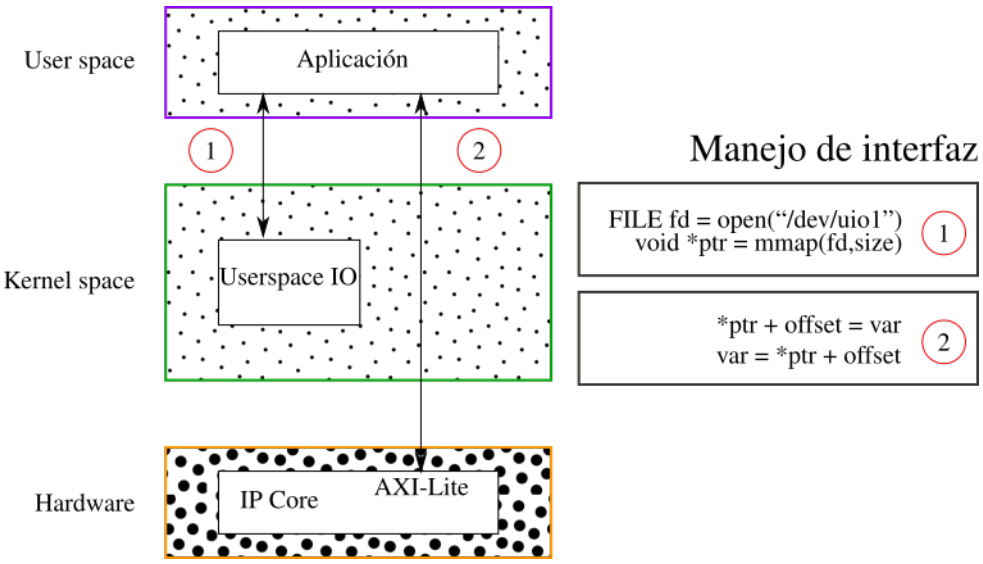

Figura 3.6: Esquema de utilización del controlador UIO para acceder directamente a los registros del *IP Core* por interfaz AXI-Lite. El procedimiento consiste en primero realizar la operación `open` sobre el archivo generado por el *driver* `uio_pdrv_genirq`. Luego, se realiza la solicitud de mapear una región de memoria sobre el espacio de usuario. Esta se obtiene con la operación `mmap`. Ya realizada esta operación, se puede manejar leer o escribir datos sobre las direcciones procedentes el IP.

Para incorporar este controlador al núcleo de Linux se debe habilitar desde las configuraciones de compilación de Linux, para construir su imagen con éste incorporado. Para que este controlador reconozca los registros manejados por AXI-Lite, debe editarse el archivo `device-tree`. Los cambios más importantes son:

1. De acuerdo con el listado de código 3.5, se edita el `bootargs` de la siguiente forma para definir el tipo de compatibilidad, definida en este caso como `generic-uio`.

2. Se incorpora un archivo `pl.dtsi` para definir todos los periféricos construidos en la región de FPGA. Se muestra el listado de código 3.6 en caso de agregar un *IP Core* con interfaz AXI-Lite.

3. Posteriormente, se compila el archivo `device-tree` y se vuelve actualizar el `device-tree.dtb` de la tarjeta SD. Una vez que haya iniciado el sistema operativo, debe exister un

archivo `/dev/uio0` por el que las aplicaciones realizan las operaciones de lectura y escritura.

Listado de código 3.5: Edición del `bootargs` para el arranque del núcleo de Linux en el archivo `device-tree`. Se incorpora el alias para la identificación del *driver* `uio_pdrv_genirq` como `generic-uio`.

```
bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw
earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=0;
uio_pdrv_genirq.of_id=generic-uio";
```

Listado de código 3.6: Edición del bloque `amba_pl` del archivo `pl.dtsi`. Se incorpora la región de memoria correspondiente para el IP `HLS_accel` en las direcciones `0x43c00000` hasta `0x43c40000`. Se define que este dispositivo es compatible con el *driver* `generic-uio`.

```
/ {
        amba_pl: amba_pl {
                #address-cells = <1>;
                #size-cells = <1>;
                compatible = "simple-bus";
                ranges ;
                HLS_IPCore: HLS_accel@43c00000 {
                        compatible = "generic-uio";
                        interrupt-parent = <&intc>;
                        interrupts = <0 29 4>;
                        reg = <0x43c00000 0x40000>;
                        xlnx,s-axi-axilites-addr-width = <0x12>;
                        xlnx,s-axi-axilites-data-width = <0x20>;
                };
        };
};
```

3.4.2 Interfaz de IP Cores con AXI4-Stream por sistema operativo

Para este tipo de interfaz se requiere crear un controlador específico. Para el caso de poseer un solo canal en ambos sentidos de los puertos de entrada y salida del AXI4-Stream, el controlador más sencillo consiste en un `DMA-proxy`, constituido por un controlador de caracteres. Este tipo de controlador simplemente se modela como un archivo en el sistema de archivos del sistema operativo, del que se le pueden definir operaciones como `open`, `read`, `write`, `mmap` y `ioctl` por ejemplo.

Para la aplicación de interés, únicamente se requieren las siguientes opciones:

1. Mapear dos regiones de memoria física en memoria virtual para escribir y leer directamente desde la aplicación los arreglos de dato para enviar y recibir.

2. Dar mando de inicio para transacción del `DMA-engine`.

3. Manejar una bandera de finalización de transacción para saber cuando los datos en memoria son habilitados para lectura.

El controlador se implementa de acuerdo con el diagrama de la figura 3.7. Este controlador se busca manejar por el *filesystem* de Linux, por ello se construye con base en el dispositivo caracter (del inglés `char-device`) para abstraer su utilización con las operaciones de archivo `open`, `mmap`, `ioctl` y `close`. La operación `mmap` se describe en el controlador de manera que retorne una región de memoria dedicado para suplir o recuperar datos de los canales de DMA. El mapeo de los canales de DMA se realiza con la API `DMA-mapping` de Linux[21]. Después, se definió la operación del archivo dispositivo caracter `ioctl` como detonador del proceso de transacción de DMA. El método de transacción por DMA es manejado por la API de Linux `DMA-engine`[21]. La operación `ioctl` mantiene bloqueada la aplicación hasta su finalización.

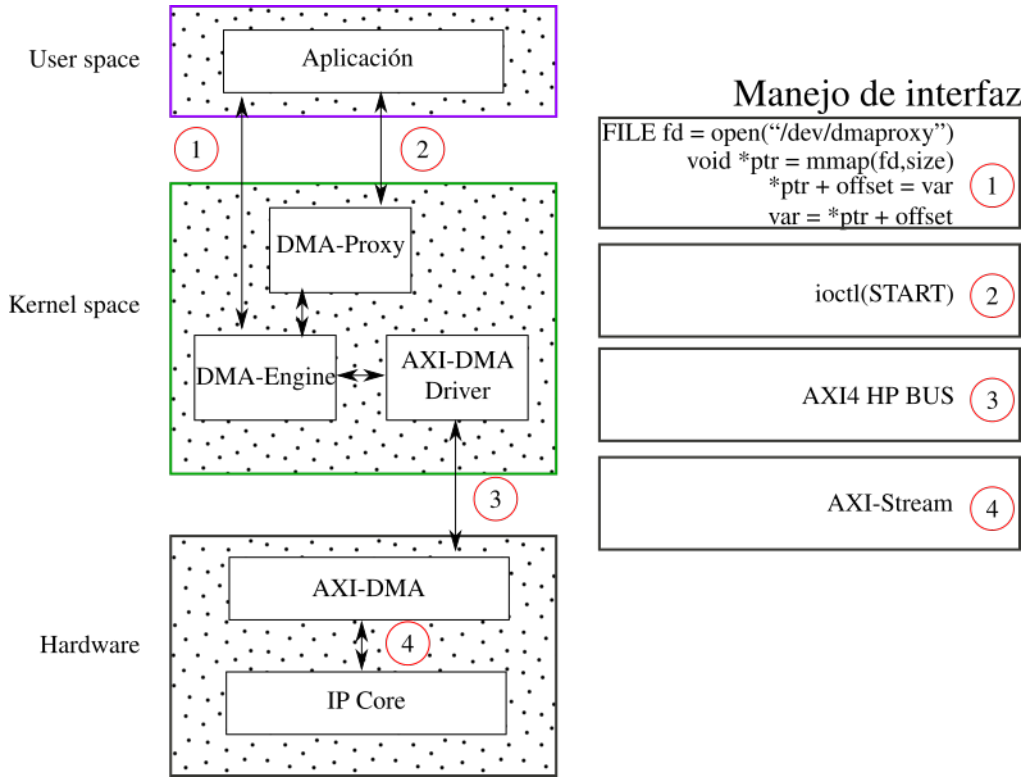

Figura 3.7: Esquema del flujo de datos y control del controlador implementado para manejo de transferencias por DMA para el IP `AXI-DMA`. La aplicación incorpora la utilización del dispositivo caracter al realizar la operación `open`. Luego, realiza el mapeo de canales de DMA en el espacio de usuario con la operación `mmap`. Finalmente, se inicia la transacción por los canales de DMA con la operación `ioctl`.

Este controlador `dmaproxy`, debe construirse por compilación cruzada utilizando el proyecto de la distribución de Linux utilizada, en este caso aquella distribuida por Digilent utilizada anteriormente. Además se requiere agregar en el archivo `device-tree` en el archivo `pl.dtsi` la información de los canales del `AXI-DMA` de manera similar a como se muestra en el listado de código 3.7.

Listado de código 3.7: Definición del archivo `pl.dtsi` para ser incluido en el archivo `device-tree`. Este incorpora las definiciones necesarias para poder utilizar los canales de DMA manejados por el IP `AXI-DMA`. Los registros de control del IP, son manejados por el controlador con alias `xlnx,axi-dma-1.00.a`, el cual se encuentra incorporado en el *kernel* de Linux. Para invocar y utilizar los canales de DMA en el controlador diseñado, se utilizan los nombres `dma0` y `dma1`.

```
/ {
        amba_pl: amba_pl {
                #address-cells = <1>;
                #size-cells = <1>;
                compatible = "simple-bus";
                ranges ;
                axi_dma_0: dma@40400000 {
                        #dma-cells = <1>;
                        clock-names = "s_axi_lite_aclk",
                                "m_axi_sg_aclk",
                                "m_axi_mm2s_aclk",
                                "m_axi_s2mm_aclk";
                        clocks = <&clkc 15>,
                                 <&clkc 15>,
                                 <&clkc 15>,
                                 <&clkc 15>;
                        compatible = "xlnx,axi-dma-1.00.a";
                        interrupt-parent = <&intc>;
                        interrupts = <0 30 4 0 31 4>;
                        xlnx,mm2s-burst-size = <0x100>;
                        xlnx,s2mm-burst-size = <0x100>;
                        reg = <0x40400000 0x10000>;
                        xlnx,addrwidth = <0x20>;
                        dma-channel@40400000 {
                                compatible =
                                        "xlnx,axi-dma-mm2s-channel";
                                dma-channels = <0x1>;
                                interrupts = <0 30 4>;
                                xlnx,datawidth = <0x20>;
                                xlnx,device-id = <0x0>;
                        };
                        dma-channel@40400030 {
                                compatible =
```

```
                                        "xlnx,axi-dma-s2mm-channel";
                                dma-channels = <0x1>;
                                interrupts = <0 31 4>;
                                xlnx,datawidth = <0x20>;
                                xlnx,device-id = <0x0>;
                        };
                };
                axidmatest_0:axidmatest@0{
                        compatible="xlnx,axi-dma-test-1.00.a";
                        dmas = <&axi_dma_0 0
                                &axi_dma_0 1>;
                        dma-names = "dma0","dma1";
                };
        };
};
```

De este archivo `device-tree`, se describe información importante de la configuración utilizada del `AXI-DMA`, por ejemplo: el número de canales de DMA y su dirección, el vector de interrupción de cada canal, el tamaño de bits de los datos y los nombres de los canales de DMA para ser llamadas en controlador. Una vez realizada esta parte, se compila el archivo `device-tree`,y se adjunta en el bloque de arranque de la tarjeta SD. Se instancia el módulo de controlador en el *kernel* y finalmente se crea una aplicación que haga uso del mismo controlador.

3.5 Comparación de velocidad de transferencia de datos entre las interfaces AXI4-Lite y AXI4-Stream

Se estudió la capacidad de transmisión de datos por medio de las interfaces AXI4-Lite y AXI4-Stream. Para ello se elaboró el siguiente experimento: se inició un proyecto de diseño en Vivado del cual se instanció un bloque de memoria RAM controlada por interfaz AXI4-Lite (`AXI-BRAM-Controller`) con capacidad de almacenamiento de 8kB, asimismo se incluyó el IP `AXI-DMA` con interfaz al puerto `AXI4-HP` del PS; con el fin de medir el tiempo para completar la transacción de 8kB por cada dispositivo, se utilizó el IP `AXI-Timer`. Se realizaron 80 mediciones de muestra para comprender el comportamiento de la variabilidad de las transacciones. Como resulta de interés conocer el tiempo de conducir los datos desde la memoria RAM del sistema hasta los bloques de RAM del PL en ambos casos, para cada transacción se invalidó la cache de datos del microprocesador, con el fin de asegurarse que los datos se encuentren alojados en la RAM externa del ZYNQ. Los resultados de esta comparación se muestran en las figuras 3.8 y 3.9.

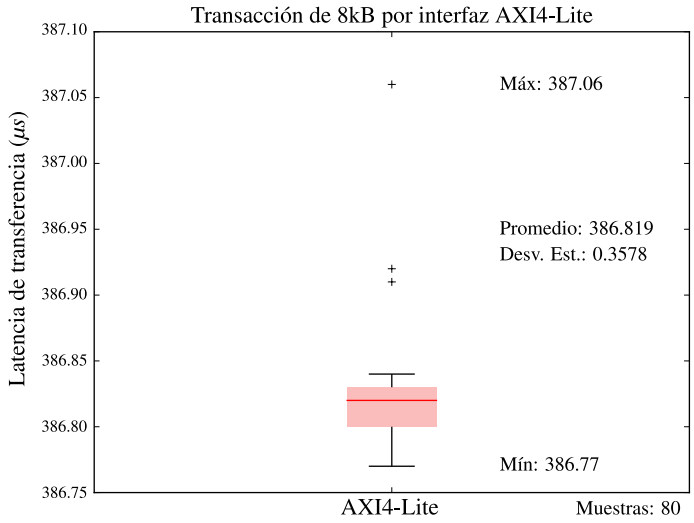

Figura 3.8: Diagrama de caja de las 80 muestras de lapsos de transacción de 8kB para un bloque de BRAM con interfaz AXI4-Lite. En promedio, las transacciones de 8kB tardan $386,819\mu$s por AXI4-Lite.

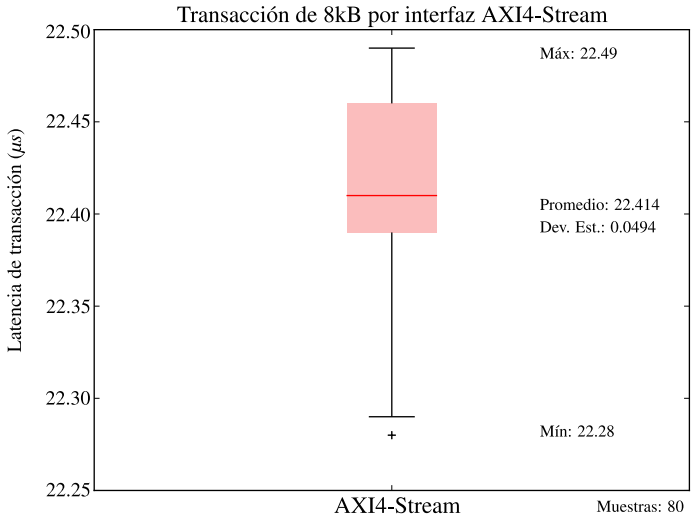

Figura 3.9: Diagrama de caja de las 80 muestras de tiempos de transacción de 8kB por interfaz AXI4-Stream manejado por el `AXI-DMA`. En promedio, las transacciones de 8kB tardan $22,414\mu$s

.

De acuerdo a la figura 3.8, el periodo de transacción promedio para el AXI4-Lite consistió en $386,819\mu s$ con una desviación estándar de $0,03578\mu s$, lo cual permite aproximar este periodo con $386,82 \pm 0,04\mu s$. Para determinar un aproximado de la velocidad del bus se realiza el cálculo:

$$v = \frac{8kB}{386,82\mu s} = 20,681MB/s$$

De forma similar para usando la figura 3.9, el periodo de transacción promedio para el `AXI-DMA` es $22,414\mu s$ con desviación estándar de $0,0494\mu s$, entonces su aproximación es $22,40 \pm 0,05\mu s$ para el periodo de transacción. La velocidad del bus se calcula:

$$v = \frac{8kB}{22,40\mu s} = 357,1MB/s$$

De esta forma se demuestra que la interfaz AXI4-Stream es por lo menos 17 veces más veloz que la de AXI4-Lite. Además, se concluye tener una métrica de que tan rápido son las transacciones por DMA en la Zynq-7000 con respecto a un acceso via núcleo a la memoria mapeada.

Capítulo 4

Implementación del IP Core para cálculo de una red neuronal basado del modelo eHH, a ejecutar sobre múltiples núcleos de procesamiento

Es necesario que el IP Core de la ION pueda replicarse en múltiples ZedBoards, con el fin de paralelizar el cálculo de la red neuronal entre varias FPGAs. La administración del flujo de datos a través de los pasos de simulación lo administrarán los microprocesadores ARM ubicados en el SoC ZYNQ, que utilizarán la red local de Ethernet para movilizar las dependencias de variables necesarias. Como primer paso se buscará señalar las principales dependencias de datos entre el cálculo de cada neurona con respecto a las demás para definir las entradas y salidas del *IP Core* para implementar en Vivado HLS.

4.1 Estudio del algoritmo para paralelizar en distintos núcleos de procesamiento

El modelo eHH está conformado por un sistema de ecuaciones diferenciales no lineales, las cuales son expresadas detalladamente en 2.1. Se utiliza el método de Euler como solución numérica para describir dicho modelo en un algoritmo computalizable. La implementación de dicho algoritmo se ha tomado del proyecto de graduación del Ing. Marco Acuña, en el desarrollo de su informe se utiliza una implementación del modelo escrito en el lenguaje de programación C [22]. Los resultados obtenidos del ejecutable de dicho código se utilizan como referencia para validación funcional del sistema. El código de alto nivel estructura el cálculo de la red neuronal como la función `ComputeNetwork`, cuyo funcionamiento de esta función se explica por medio del diagrama de bloques que se muestra en la figura 4.1; de esta figura se observa que la unidad de cálculo requiere: los valores de la matriz de conectividad que modelan las conexiones físicas entre las neuronas, las corrientes de

estímulo individual para cada neurona llamadas `IApp` y las 19 variables de estado que requiere cada celda neurona para calcular el siguiente estado neuronal y la salida de tensión de cada una.

El principal objetivo a considerar consiste en poder retomar el programa original y adaptarlo para replicarlo en múltiples núcleos, de manera que los resultados sean congruentes con respecto a la referencia de pruebas. Para ello, se procedió a evaluar la dependencia de datos entre las neuronas, de donde se encontró que existe una operación llamada tt V_dend con dependencia total de los valores de sus vecinas.

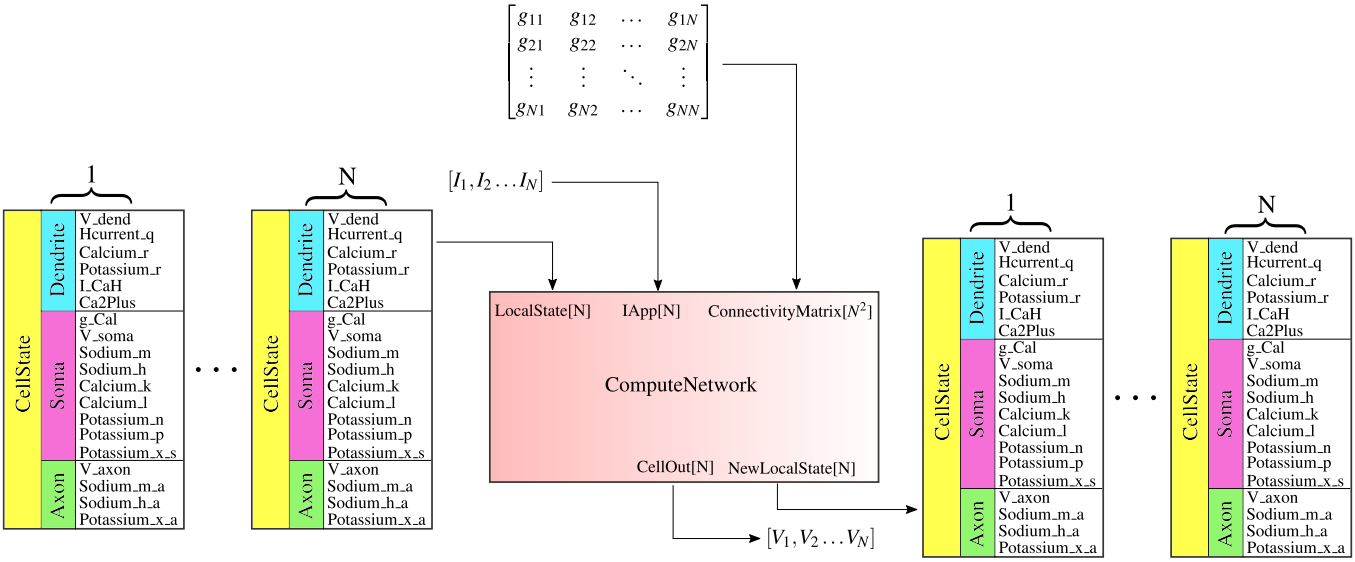

Figura 4.1: Diagrama de bloques de la función de cálculo de la red neuronal de tamaño `N`, basada del modelo neuronal eHH. Para calcular la respuesta de un paso de simulación la red requiere: la matriz de conectividad o `ConnectivityMatrix` que asocia las conexiones sinápticas entre las neuronas, las corrientes de estímulo `IApp` y las variables de estados de todas las neuronas ordenadas como una estructura de datos `CellState`. Las estados de celda son agrupados en un arreglo llamado `LocalState`. Después del cálculo se recuperan los nuevos valores de variables de estado de las neuronas en el arreglo `NewLocalState`. Además, se retorna los valores de tensión de salida de axón de cada neurona.

En la figura 4.2 se ilustra la función de cálculo de la corriente producida por las uniones de comunicación de la neurona, bloque encontrado en el compartimiento de la dendrita. En ella se aprecia el arreglo `NeighVdend`, este es conformado por las tensiones de dendrita de todas las neuronas vecinas. Como cada neurona requiere hacer lectura de este arreglo, existe la dependencia de datos para el cálculo de la corriente I_C. De acuerdo al diseño requerido, se encontró la necesidad de ajustar las entradas y salidas del bloque funcional `ComputeNetwork` de la figura 4.1 para que el mismo bloque pueda recibir las tensiones de dendrita de las neuronas vecinas y asimismo cada neurona retire estos valores como salida.

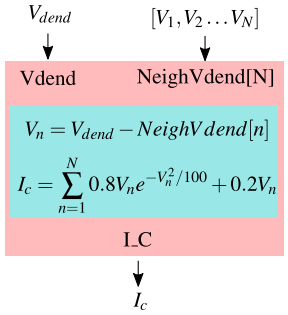

Figura 4.2: Representación de alto nivel del bloque que calcula la corriente I_C. Para realizar el cálculo de esta corriente, se requiere leer el valor de tensión de dendrita de la neurona y un arreglo de las tensiones de dendrita de todas las neuronas vecinas.

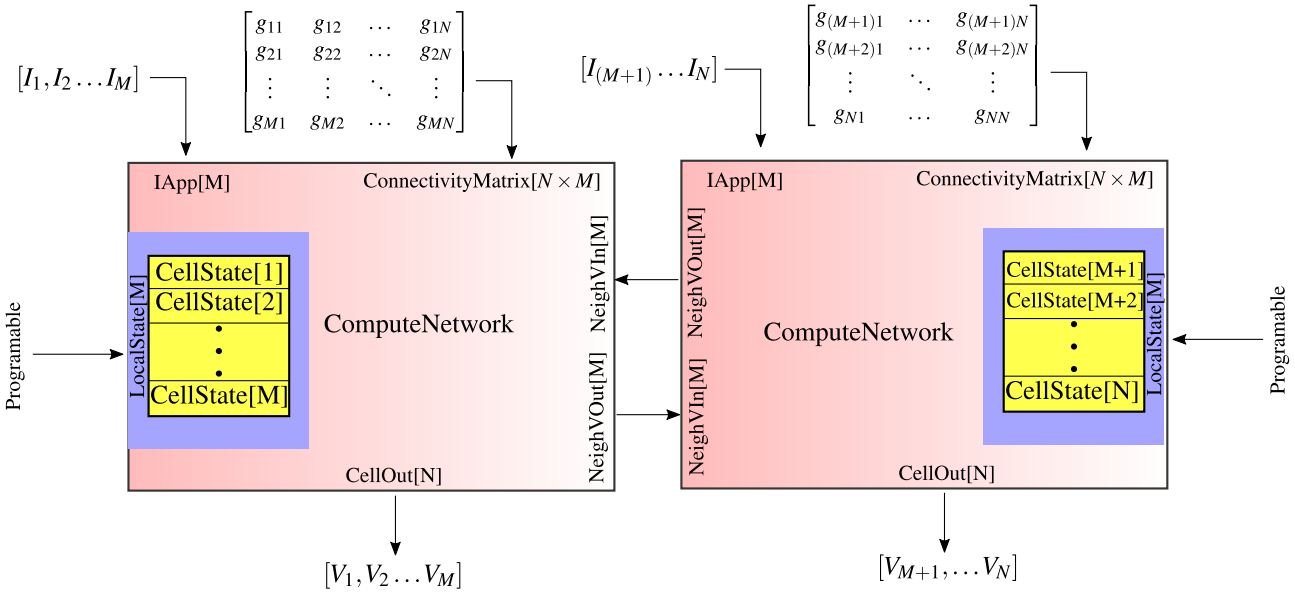

Figura 4.3: Diagrama de bloques de la función de cálculo de la red neuronal distribuida en dos módulos. Cada módulo intercambia las tensiones de dendrita vecinas contenidas por su módulo vecino. La distribución de neuronas entre los módulos se procede al programar los parámetros `M` y `N`. El primer parámetro indica la cantidad de neuronas manejadas en el módulo y el segundo la cantidad global de neuronas en toda la red.

4.2 Implementación de interfaces para el IP Core en Vivado HLS

De acuerdo a lo concluido en la sección 4.1, se hizo una edición de interfaz de la implementación de `ComputeNetwork`, de tal forma sea más similar al de la figura 4.3. Los argumentos de la función después del cambio se muestra en el listado de código 4.1.

Listado de código 4.1: Encabezado del código para síntesis por Vivado HLS. Se indican los argumentos manejados por la función principal `ComputeNetwork`.

```
void ComputeNetwork(
        cellState local_state[MAX_TIME_MUX],
        mod_prec neighVdendIn[MAX_NEIGH_SIZE]
        mod_prec iAppin[MAX_TIME_MUX],
        int N_Size,
        int Mux_Factor,
        mod_prec Connectivity_Matrix[CONN_MATRIX_SIZE],
        mod_prec cellOut[MAX_TIME_MUX],
        mod_prec neighVdendOut[MAX_TIME_MUX]){
                ...
        }
```

Para escoger las interfaces más convenientes para las entradas y salidas del *IP Core* se utilizó como criterio la frecuencia de uso de cada puerto, porque los argumentos `N_Size`, `Mux_Factor`, `Connectivity_Matrix` y `local_state` son necesarios unicamente en el inicio de la simulación. Los demás puertos deben cambiarse por cada paso de simulación, por lo cual lo hacen preferibles manejarlos por una interfaz FIFO para enviar y recibir los datos por ráfagas. En la figura 4.4 se expone la implementación propuesta para definir las interfaces por Vivado HLS.

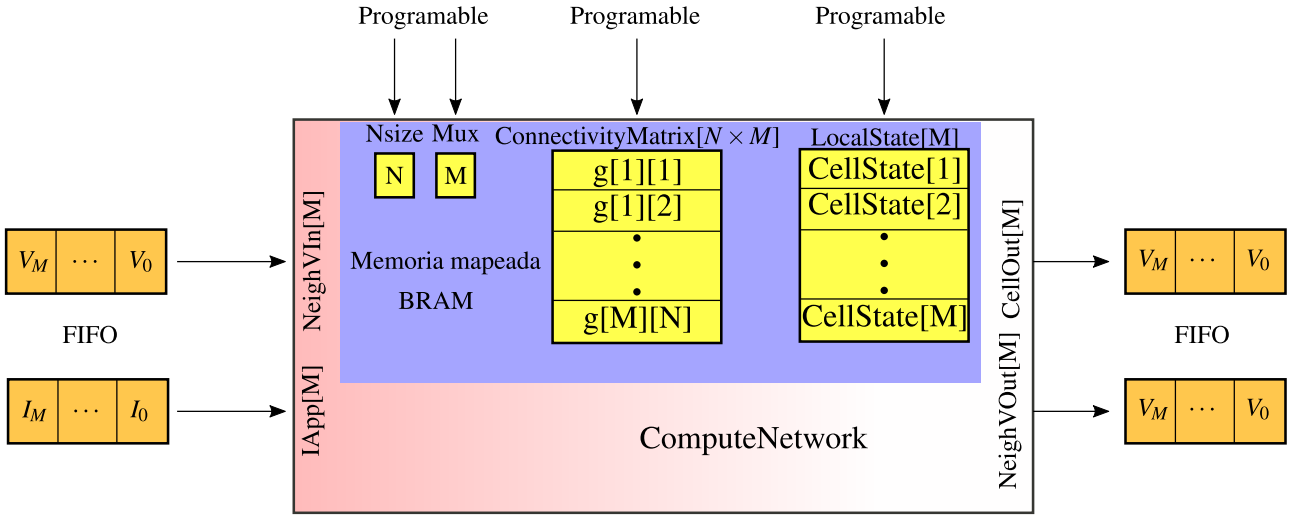

Figura 4.4: Diagrama de bloques de la interfaz propuesta para el *IP Core* de cálculo de la red neuronal. Las variables de estado y matriz de conectividad son mapeados en memoria con interfaz AXI4-Lite. Las variables de `IApp`, `NeighVIn`, `CellOut` y `NeighVOut` son manejados por una interfaz FIFO (AXI4-Stream). Las dimensiones de la red neuronal son programadas por los parámetros `N` y `M`.

La implementación de las interfaces programables se realizarán por AXI4-Lite y las interfaces de FIFO se ajustarán al protocolo AXI4-Stream. Como referencia se utilizó el diseño del multiplicador de matrices encontrado en la documentación de Xilinx. De éste se tomó el esquema para desempacar y empacar los datos de interfaz AXI4-Stream con las señales de múlti canal como se estudió en la sección 3.2.1. Como último ajuste, se

ajustó que los datos de `Iappin` y `neighVdendI` sean enviados desde el mismo bus serial, al igual que las señales de `cellOut` y `neighVdendOut` con el fin de no tener que implementar dos módulos de `AXI-DMA` en el PL de la ZYNQ. Finalmente, la interfaz final del bloque implementado se muestra en el listado de código 4.2

Listado de código 4.2: Encabezado de la función a sintetizar por Vivado HLS. En ella se incluyen los argumentos de la función, así como las directivas para definir el manejo de interfaz para cada variable. Las variables `IApp` y `neighVdendIn` son agrupadas en una sola interfaz AXI4-Stream llamada `INPUT_INPUT_STREAM`. También para las variables de `CellOut` y neighVdendOut se agrupan en una salida de interfaz AXI4-Stream nombrada `OUTPUT_STREAM`. El resto de variables son manejadas por interfaz AXI4-Lite.

```
#include <ap_axi_sdata.h>
typedef ap_axiu<32,4,5,5> AXI_VAL;
void HLS_accel(
        cellState local_state0[MAX_TIME_MUX],
        AXI_VAL INPUT_STREAM[MAX_TIME_MUX +MAX_NEIGH_SIZE],
        int N_Size,
        int Mux_Factor,
        mod_prec Connectivity_Matrix[CONN_MATRIX_SIZE],
        AXI_VAL OUTPUT_STREAM[MAX_TIME_MUX+MAX_NEIGH_SIZE])
{
#pragma HLS INTERFACE s_axilite register port=local_state0
#pragma HLS INTERFACE axis       port=INPUT_STREAM
#pragma HLS INTERFACE s_axilite register port=N_Size
#pragma HLS INTERFACE s_axilite register port=Mux_Factor
#pragma HLS INTERFACE s_axilite register port=Connectivity_Matrix
#pragma HLS INTERFACE axis       port=OUTPUT_STREAM
#pragma HLS INTERFACE s_axilite register port=return
}
```

Los resultados de cosimulación no mestraron alteración apreciable con los del modelo de referencia. El reporte de síntesis de el *IP Core* generado se muestra en la figura 4.5, obtenido para los valores de `MAX_TIME_MUX`= 2000, `MAX_NEIGH_SIZE` = 6000 y `CONN_MATRIX_SIZE`= 6000 × 6000. Si se intenta incrementar a `MAX_NEIGH_SIZE` arriba de 6000 los bloques de BRAM llegan a agotarse drásticamente, de forma que implementar este *IP Core* en conjunto con los demás de interconexión en el proyecto de Vivado se vuelve inviable en la plataforma Zynq utilizada.

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	371
FIFO	-	-	-	-
Instance	178	155	35019	51243
Memory	60	-	0	0
Multiplexer	-	-	-	457
Register	-	-	731	-
Total	238	155	35750	52071
Available	280	220	106400	53200
Utilization (%)	85	70	33	97

Figura 4.5: Reporte de utilización de recursos para la implementación del IP Core propuesto. El IP fue sintetizado para los valores de `MAX_TIME_MUX`= 2000, `MAX_NEIGH_SIZE`= 6000. Se aprecia que la utilización de LUT corresponde al 97% y de BRAM 85%. Fuente: Recuperado de Vivado.

4.3 Desarrollo final de la implementación en la placa de desarrollo

Se creó un proyecto completo en Vivado para la plataforma Zedboard, con una meta de máxima frecuencia de reloj en el PL de 100MHz para el bloque IP de ION. Se instanció el módulo en el entorno de desarrollo de diseño bloques de Vivado tal como se muestra en la figura 4.6 . Obsérvense las interfaces de AXI4-Lite y AXI4-Stream. Al *IP Core* ION se le configuró un acceso al bus AXI4 HP, con ruta directa a la interfaz de memomria DDR3; este bus se conecta por el `AXI-DMA`, tal como se muestra en la figura 4.7.

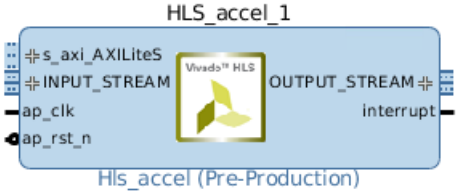

Figura 4.6: Módulo `ComputeNetwork` visto desde el entorno de desarrollo de Vivado. Nótesen los puertos de AXI4-Stream `INPUT_STREAM` y `OUTPUT_STREAM`. También, del puerto mapeado en memoria `s_axi_AXILitesS`. Fuente: Tomado del it block design de Vivado

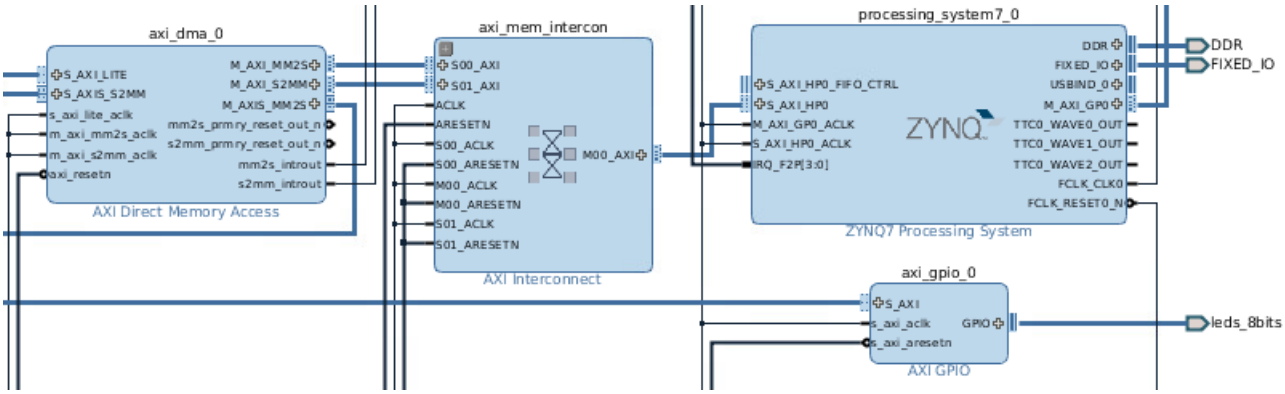

Figura 4.7: Interconexión con el PS y `AXI-DMA` por el puerto `AXI4 HP BUS`. Nótese que el IP `AXI-DMA` (localizado a la izquierda) es interconectado el bloque `AXI_Interconnect` hacia el bus `AXI4 HP BUS`. Fuente: Tomado del *block design* de Vivado

4.4 Pruebas y mediciones

Una vez obtenido ya el *bitstream* del sistema completo, y para. Para manejar el módulo propuesto, se siguieron las instrucciones descritas en el capítulo 3. Se evaluó el comportamiento de este módulo utilizando dos métricas: velocidad de ejecución por paso de simulación y precisión de resultados por porcentaje de error.

4.4.1 Comparación velocidad de cálculo entre implementación *bare-metal* y con sistema operativo

Se buscó encontrar el efecto que implica utilizar las capas de abstracción de un sistema computacional manejado con sistema operativo y uno *bare-metal* sobre la misma plataforma Zedboard. Estas implementaciones se construyeron siguiendo las instrucciones mostradas en el capítulo 3. Los tiempos de medición fueron obtenidos con el IP AXI-Timer. Los resultados de esta comparación se muestran en la figura 4.8. De esta gráfica se observa que el tiempo de ejecución de la versión implementada en sistema operativo pierde en el peor de las casos un 6%. Sin embargo, por tener acceso a los paquetes de software de Linux, se gana en flexibilidad.

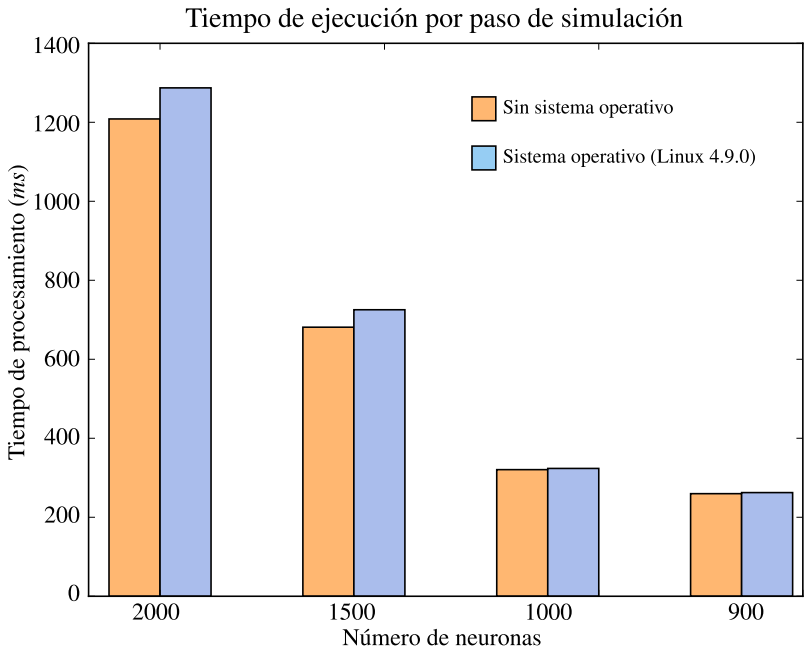

Figura 4.8: Comparación del tiempo de ejecución por paso de simulación entre los métodos de manejo del IP generado, *bare-metal* (sin sistema operativo) y con sistema operativo. Nótese el leve costo en tiempo al usar sistema operativo, pero siempre debajo de un tanto 6%.

4.4.2 Mejora de rendimiento de cálculo incrementando el factor de multiplexación

Para comprobar la mejora de rendimiento del cálculo de los pasos de simulación utilizando la división de procesamiento entre varios módulos, se realizó la simulación al configurar el IP para diferentes valores de tamaño de red (`NSIZE`) y ajustando el factor de multiplexación para representar la mejora con dos, tres y cuatro núcleos de procesamiento. Esto puede hacerse ya que se implementó en el software la simulación de la comunicación de los datos de `neighVdendI` y `neighVdendO` para que simplemente se evaluara el tiempo del rendimiento con uno solo sin tener que comunicar varios módulos de procesamiento físicos. Los resultados que se muestran en la figura 4.9 se obtuvieron usando la implementación con sistema operativo y en ella se observa que al incrementar el factor de multiplexación (número de módulos de procesamiento) mejora el rendimiento de cálculo con respecto a la realizada por un solo módulo. Por ejemplo, en la simulación de 1200 neuronas, si se utiliza el factor de multiplexación de cuatro, el módulo ejecuta los cálculos de 300 neuronas, lo que le toma cuatro veces menos tiempo de realizar.

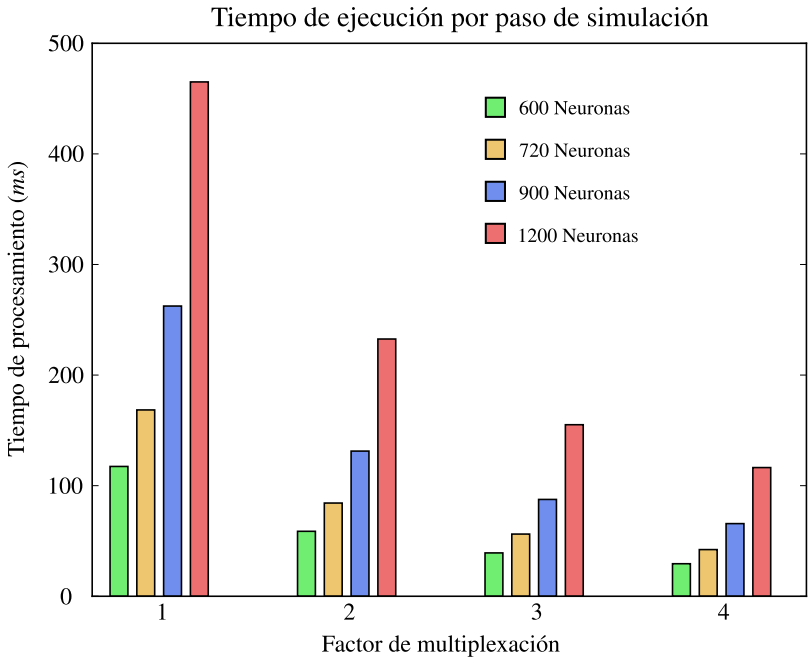

Figura 4.9: Comparación de los tiempos de ejecución por paso de simulación al incrementar el factor de multiplexación del IP para diferentes valores de tamaño de red neuronal, utilizando el IP manejado con sistema operativo.

Se estudió el comportamiento del tiempo de ejeción por paso de simulación en función del número de neuronas y del factor de multiplexación. Primero, se encontraron curvas de mejor ajuste para el número de neuronas contra el tiempo de procesamiento, mateniendo el factor de multiplexación constante. Los resultados se muestran en la figura 4.10. En ella se observa como la tendencia de crecimiento del tiempo de ejecución en función del número de neuronas incremente de manera cuadrática. Ya que, el polinomio cuadrático es el que da mejores resultados de aproximación. Después, se analizó la tendencia de decrecimiento del tiempo de procesamiento al incrementar el factor de multiplexación. De acuerdo a los resultados, mostrados en la figura 4.11, se encontró que la curva de mejor ajuste es la función racional $1/x$. Por ello, se comprueba que el factor de multiplexación es inversamente proporcional con respecto al tiempo de procesamiento.

Para explicar este comportamiento se refiere a la tabla 1.1. En ella se recupera que la cantidad de operaciones de coma flotante de una red de neuronal de tamaño `N` se comporta de acuerdo con la ecuación 4.1. El término cuadrado de la ecuación se debe por las operaciones relacionadas con las uniones comunicantes de las neuronas. También, el segundo término se relaciona con las demás operaciones relacionadas con el comportamiento de la neurona. Al incluir al factor de multiplexación en la ecuación, se obtiene el resultado 4.2. En ella se observa la relación cuadrática en cantidad de operaciones coma flotante, asimismo como el factor de multiplexación es inversamente proporcional. Lo cual concuerda con los resultados encontrados en la figuras 4.10 y 4.11.

Esta explicación es cierta si existe una relación lineal entre la cantidad de operaciones de

coma flotante y el tiempo de ejecución. Si esto es cierto, esto implicaría que las operaciones internas del IP generado, no se encuentran paralelizadas o al menos en tiempo promedio no lo están. Además, se descarta la relación del retardo por la interconexión entre el IP y la memoria. Porque los cantidad de datos totales incrementa de manera lineal con respecto a la cantidad de neuronas en simulación.

$$\text{FP} = (475N_{total} + 859)N_{local} = 475N^2 + 859N \Leftrightarrow N_{total} = N_{local} \tag{4.1}$$

$$\text{FP} = (475N_{total} + 859)\frac{N_{total}}{M} = \frac{475N^2 + 859N}{M} \tag{4.2}$$

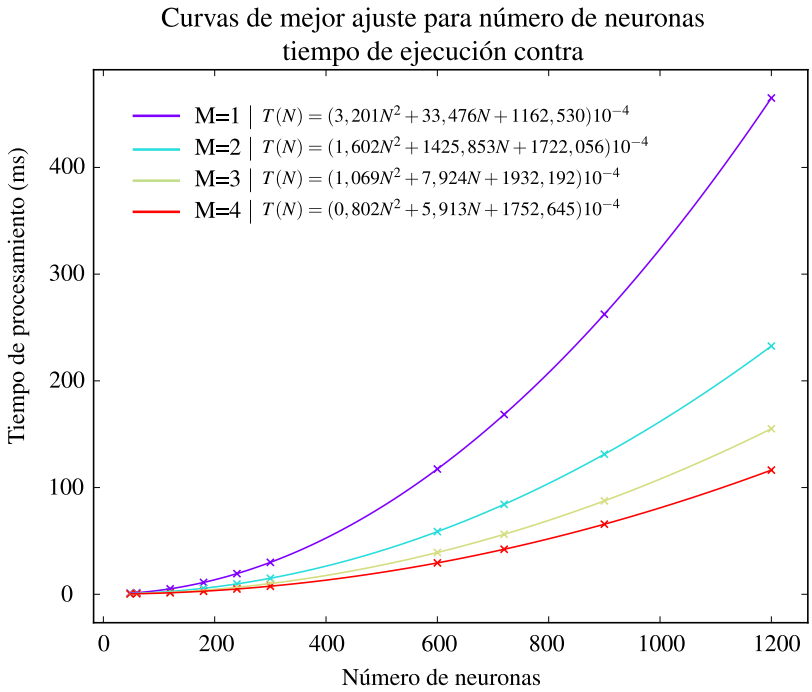

Figura 4.10: Observación de la tendencia de crecimiento del tiempo de ejecución por medio de aproximación de curvas de mejor ajuste. El comportamiento del tiempo de ejecución crece de manera cuadrática de acuerdo con el crecimiento de la cantidad de neuronas.

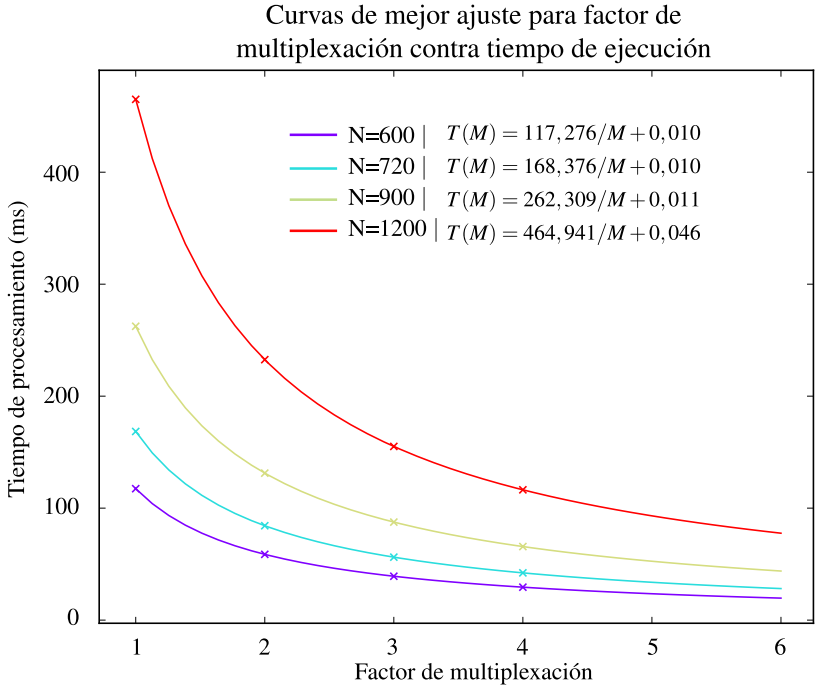

Figura 4.11: Observación de la tendencia de decrecimiento del tiempo de ejecución por medio de aproximación de curvas de mejor ajuste. El tiempo de ejecución es inversamenta proporcional con respecto al factor de multiplexación de la simulación.

4.4.3 Congruencia de los resultados con respecto al modelo ideal

Como último objetivo por cumplir en el proyecto, se desea verificar la precisión de los resultados en la salida del módulo generado con respecto a los resultados del modelo dorado de de alto nivel. En la comprobación inicial, la multiplexación en la ejecución de simulación no afecta directamente en la precisión de los datos.

Sin embargo, al realizarse las pruebas se encontró un problema significativo. La precisión del IP generado por Vivado HLS se ve modificada de acuerdo a la versión que se utilice. El código fuente para la generación del IP fue al inicio fue implementado en la versión de Vivado 2016.1 con la cual se obtuvo los resultados de precisión satisfactorios mostrados en la figura 4.12, dónde el porcetaje de error máximo en toda la simulación de 40 mil pasos es de 1% aproximadamente. Pero, al migrar este mismo código en la versión de Vivado 2016.4, se encuentró distorsión en la precisión de los cálculos ya que el porcentaje de error incrementó considerablemente como se observa en la figura 4.13. Se deja claro que este código no presentó cambio alguno dentro del él, simplemente se cambió de versión de la herramienta se presentó este defecto.

Los motivos por los que se puede explicar este error son: algún cambio interno de los IPs de cálculo numérico de coma flotante que utilice Vivado HLS, cambio en la formulación del proceso de cálculo de la herramienta de síntesis de Vivado HLS de tal forma haga una variable intermedia incrementar y/o disminuir considerablemente de tal forma afecte el

cálculo global, o una pulga en el Vivado HLS en la elaboración de síntesis. Dado que ya el tiempo de desarrollo del proyecto tocaba a su fin, no fue posible sin embargo verificar estas hipótesis. Se vuelve necesario consultar sobre las mismas al mismo fabricante de la herramienta.

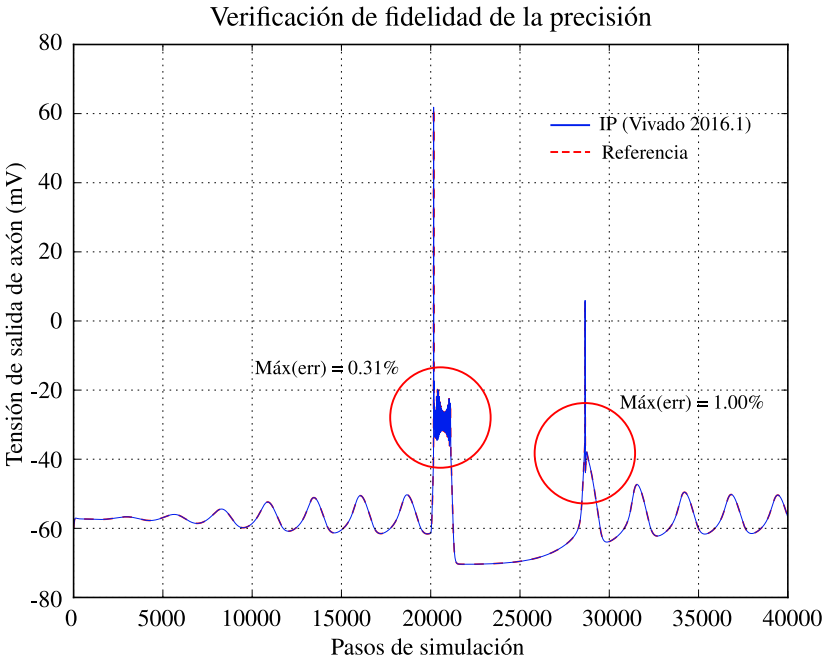

Figura 4.12: Comparación de la tensión de salida del axón de una neurona entre la referencia del programa de alto nivel y el IP sintetizado por Vivado HLS 2016.1. Los errores máximos son señalados dentro de los círculos rojos. En el peor de los casos el porcentaje de error fue de $1,00\%$.

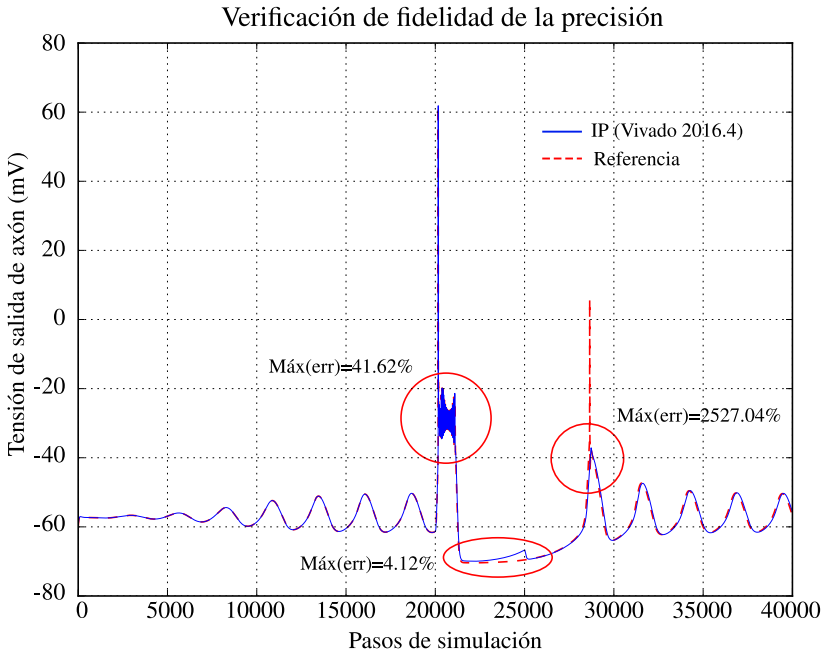

Figura 4.13: Comparación de la tensión de salida del axón de una neurona entre la referencia del programa de alto nivel y el IP sintetizado por Vivado HLS 2016.4. Los errores máximos son señalados dentro de los círculos rojos. En el peor de los casos el porcentaje de error fue de 2527% y $41,62\%$ en el segundo peor caso.

Capítulo 5

Conclusiones

Se elaboró una métrica de comparación para la velocidad de transferencia de datos entre la memoria y los dispositivos encontrados en el PL del SoC Zynq-7000 utilizando interfaces AXI4-Lite y AXI4-Stream. Se encontró que esta última interfaz manejada por el IP `AXI-DMA` es aproximadamente 17 veces más veloz.

Se estudió rendimiento de procesamiento al ejecutar las simulaciones con el IP Core generado por Vivado HLS. Primero se compararon los resultados del tiempo de ejecución utilizando una implementación del sistema con sistema operativo y otra *bare-metal*. Se determinó que al incluir una capa de proceasmiento más abstracta (como un sistema operativo) no afecta de manera significativa el tiempo de procemiento. En el peor de los casos el rendimiento decae hasta un 6% con respecto a la implementación por *bare-metal*.

Se determinó que la relación entre el tiempo de ejecución de la simulación y la cantidad global de neuronas, es cuadrático. Asimismo, la relación del tiempo de procesamiento y el factor de multiplexación es inversamente proporcional. Porque el efecto que tienen estos parámetros en la cantidad de operaciones coma flotante de la simulación crece de manera cuadrática en función con la cantidad de neuronas total de la simulación. El factor de multiplexación disminuye la cantidad de operaciones coma flotante por un factor de $1/x$.

Se comprobó la precisión de las tensiones de axón resultantes de la simulación durante varios pasos de simulación. Se encontró que para el IP generado con la herramienta de Vivado HLS 2016.1, el error máximo obtenido por sus resultados es de 1%. Sin embargo, al sintetizar el IP con la herramienta de Vivado HLS 2016.4, con el mismo código fuente, se encontraron inconsistencias con los resultados recuperados. En el peor de los casos, se encontró un error del 2527%.

Recomendaciones

- Evaluar posibles optimizaciones por Vivado HLS del algoritmo para mejorar el rendimiento de cálculo de operaciones coma flotante.

- Evaluar la migración del código a una versión de Vivado HLS más reciente como Vivado HLS 2017 y de paso, tratar de corregir el error que se tiene actualmente en la red.

- Evaluar la implementación RTL desde algún lenguaje de descripción de hardware para conservar la portabilidad del código en diferentes clases de FPGAs y software de síntesis.

Bibliografía

[1] *Framework for high-detail and real-time brain simulations*, BrainFrame, Erasmus Brain Project. Recuperado el 28 de Mayo de 2017 de: http://erasmusbrainproject.com/index.php/themes/brainframe

[2] J. De Gruijl, P. Bazzigalu, M. de Jeu, C. De Zeeuw *Climbing Fiber Burst Size and Olivary Sub-threshold Oscillations in a Network Setting*, PLoS Comput Biol 8(12): e1002814 2012.

[3] G. Chatzikonstantis, D. Rodopoulos, C. Strydis, C. De Zeeuw & D. Soudris, *Optimizing Extended Hodgkin-Huxley Neuron Model Simulations for a Xeon/Xeon Phi Node*, IEEE Transactions on Parallel and Distributed Systems. 2016.

[4] G. Smaragdos, S. Isaza, M. Van Eijk, I. Sourdis & C. Strydes, *FPGA based Biophysically-Meaningful Modeling of Olivocerebellar Neurons*, FPGA'14. Feb 26-28 2014, USA.

[5] G. Smaragdos, G. Chatzikonstantis, S. Nomikou, D. Rodopoulos, I. Sourdis, D. Soudris, C. De Zeeuw & C. Strydis, *Performance Analysis of Accelerated Biophysically-Meaningful Neuron Simulations*, IEEE International Symposium on Performance Analysis of Systems and Software 2016.

[6] M. Nair, S. Surya, R. S. Kumar, B. Nair and S. Diwakar, "Efficient simulations of spiking neurons on parallel and distributed platforms: Towards large-scale modeling in computational neuroscience," 2015 IEEE Recent Advances in Intelligent Computational Systems (RAICS), Trivandrum, 2015, pp. 262-267.

[7] A. Upegui, C. A. Pena-Reyes y E. Sanchez, *An fpga platform for on-line topology exploration of spiking neural networks*, Microprocessors and microsystems, vol. 29, n. o 5, págs. 211-223, 2005

[8] H. Shayani, P. J. Bentley y A. M. Tyrrell, *Hardware implementation of a bio-plausible neuron model for evolution and growth of spiking neural networks on fpga*, en Adaptive Hardware and Systems, 2008. AHS'08. NASA/ESA Conference on, IEEE, 2008, págs. 236-243.

[9] E. M. Izhikevich, "Which model to use for cortical spiking neurons?", IEEE transactions on neural networks, vol. 15, n. o 5, págs. 1063-1070, 2004.

[10] S. W. Moore, P. J. Fox, S. J. Marsh, A. T. Markettos, A. Mujumdar. (2012). *Bluehive - a field-programable custom computing machine for extreme-scale real-time neural network simulation*, IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines, 133–140.

[11] M. van Eijk, C. Galuzzi, A. Zjajo, G. Smaragdos, C. Strydis and R. van Leuken, *ESL design of customizable real-time neuron networks*, 2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) Proceedings, Lausanne, 2014, pp. 671-674.

[12] S. Asano, T. Maruyama and Y. Yamaguchi, *Performance comparison of FPGA, GPU and CPU in image processing*, 2009 International Conference on Field Programmable Logic and Applications, Prague, 2009, pp. 126-131.

[13] *7 Series FPGAs Data Sheet: Overview*, Xilinx, 2017.

[14] R. Tessier, K. Pocek and A. DeHon, "Reconfigurable Computing Architectures," in Proceedings of the IEEE, vol. 103, no. 3, pp. 332-354, March 2015.

[15] *Zynq-7000 All Programmable SoC Data Sheet*, Xilinx, 2017.

[16] *Vivado Design Suite User Guide High-Level Synthesis*, Xilinx, 2016.

[17] *AMBA AXI and ACE Protocol Specification*, ARM, 2011.

[18] *AMBA 4 AXI4-Stream Protocol*, ARM, 2010.

[19] *AXI Reference Guide*, Xilinx, 2012.

[20] *Zynq-7000 All Programmable SoC: Embedded Design Tutorial A Hands-On Guide to Effective Embedded System Design*, Xilinx, 2016.

[21] P. Barry, P. Crowley. (2012). *Modern Embedded Computing*, 1st Edition, Morgan Kaufmann.

[22] M. Acuña. (2015). *Extensión de las capacidades de comunicación y memoria para sistema de modelado de redes neuronales del olivo inferior*. Escuela de ingeniería de Electrónica. Instituto Tecnológico de Costa Rica, Cartago.