Implementación hardware de una red neuronal Long Short-Term Memory

Trabajo fin de grado
Grado en Ingeniería de Computadores

Jorge López Melchor
Octavio Sales Calvo

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense

Madrid
2019-2020

Sobre TEFLON

Teflon(cc0 1.0(documentación) MIT(código))es una plantilla de LaTeX creada por David Pacios Izquierdo con fecha de Enero de 2018. Con atribuciones de uso CC0.

Esta plantilla fue desarrollada para facilitar la creación de documentación profesional para Trabajos de Fin de Grado o Trabajos de Fin de Máster. La versión usada es la 1.3.

V:1.3 Overleaf V2 with pdfLaTeX, margin 1in, NO-bib

Contacto
Autor: David Pacios Izquiero
Correo: dpacios@ucm.es
ASCII: asciifdi@gmail.com
Despacho 110 - Facultad de Informática

Implementación hardware de una red neuronal Long Short-Term Memory

Trabajo fin de grado
Grado en Ingeniería de Computadores

Autores
Jorge López Melchor
Octavio Sales Calvo

Dirigida por el Doctor
Oscar Garnica Alcázar
Juan Lanchares Dávila

Departamento de Arquitectura de Computadores y Automática

Facultad de Informática

Universidad Complutense

Madrid
2019-2020

Agradecimientos

Deseamos empezar hablando del duro trabajo y sacrificio que ha supuesto este trabajo para todos en pleno confinamiento debido al covid-19, por tanto, queremos empezar dando las gracias a nuestros directores Óscar Garnica Alcázar y Juan Lanchares Dávila por habernos propuesto este proyecto y por invertir su tiempo y dedicación en ayudarnos a desarrollarlo. Gracias a ellos hemos aprendido más sobre el desarrollo en hardware y su implementación, cumpliendo con el objetivo del desarrollo e implementación de una red neuronal LSTM. Gracias a la Universidad Complutense de Madrid y a la Facultad de Informática por darnos la oportunidad de haber estudiado este grado en ingeniería de computadores con el que tanto hemos disfrutado y en especial a todos los profesores que nos han acompañado durante estos años y han contribuido a nuestra formación para afrontar este trabajo. Por último y no menos importante, dedicamos este trabajo a todos nuestros amigos que nos soportaron y ayudaron en los malos momentos y celebraron los buenos, compañeros que nos ayudaron y a nuestras familias que nos han dedicado todo su tiempo, todo su esfuerzo y todos sus recursos con tal de educarnos y formarnos lo mejor posible para afrontar esto y todo lo que quede por venir.

Para todos vosotros os dedicamos este trabajo, muchas gracias.

Resumen

En la actualidad, para poder tener una calidad de vida aceptable los enfermos de diabetes mellitus deben controlar su glucemia, es decir la concentración de glucosa presente en la sangre. El tratamiento para controlar la glucemia se basa en una alimentación equilibrada, actividad física, hábitos generales de higiene, patrones de sueño saludables, medicación, controles periódicos por el endocrinólogo, seguimiento y control diario de los niveles de concentración de glucosa en sangre y suministro diario de insulina. La insulina tarda en hacer efecto, por lo tanto, lo ideal es conocer cómo va a ser la evolución glucémica del paciente para poder suministrar la insulina con tiempo suficiente como para evitar las hiperglucemias, niveles elevados de glucosa en sangre, o las hipoglucemias, bajos niveles de glucosa en sangre. Conocer con antelación cuál va a ser el índice glucémico le permite al paciente controlarlo mejor modificando las cantidades de insulina que se inyecta.

El objetivo principal de este Trabajo de Fin de Grado es el estudio e implementación en hardware de redes neuronales profundas para realizar la predicción de la glucemia a 30 minutos vista. En concreto, se van emplear las redes neuronales LSTM que proporcionan predicciones de mayor precisión frente a las obtenidas con otras técnicas clásicas, sobre todo en lo que a series temporales de datos se refiere. En el estudio de estos modelos se toma como entrada diversos parámetros como son: la glucemia, hidratos de carbono e insulina en sangre. En nuestro dataset, por cada registro de las características, se tiene un histórico desde dos horas atrás hasta la actual. En este trabajo primero explicamos el estudio que se ha realizado en Python para decidir la arquitectura de la red neuronal que se implementará posteriormente en hardware. Esta arquitectura incluye el número capas, el tipo de estas y el número de neuronas que las componen.

A continuación presentamos el diseño e implementación hardware. Primero se explican los aspectos generales del diseño, como los protocolos de comunicación o la representación de los datos utilizados, y posteriormente se detalla el diseño siguiendo una aproximación jerárquica top-down, empezando por la interfaz del sistema y finalizando con los módulos más internos.

En el último capítulo se expone tanto los resultados de sintetizar el diseño hardware sobre una FPGA Xilinx Virtex-6 ML-605 Evaluation Board, como los resultados experimentales de las pruebas realizadas a la red neuronal hardware.

PALABRAS CLAVE:

- LSTM
- VHDL
- FPGA
- Python
- keras
- Redes Neuronales
- Aprendizaje Automático

Abstract

Nowadays patients diagnosed with Diabetes Mellitus must control their glycaemic levels or quantity of glucose in their blood in order to have an acceptable life quality. Glycaemic control treatment is based on: a balanced diet, personal hygiene measures, physical activity, pharmacological-medication, periodic meetings with and endocrinologist, strict control of glycaemic levels (daily, every 8 hours, or if necessary, every hour), insulin administration.

Taking in account that insulin takes some time to produce its effects, it would be helpful to know or predict an individual glycaemic level curve in order to prevent hyper (excess) or hypo (deficient) glycaemia. A patient who knows his own glycaemic behaviour in advance can take a better control of his disease by administering himself a higher o lower quantity of insulin, just as much as he needs.

The main goal of this study is the analysis and implementation of deep neural network in order to predict glycaemic levels 30 minutes in advance. More specifically, LSTM neural networks will be used in this study as they lead to more precise predictions in comparison with others classic techniques, specially to time series data. In this study, it has taken as an entry diverse parameter such: the blood sugar, carbon hydrates and insulin in blood. In our dataset, for each register of the characteristics, it has a historical record from two hours before up to the present. The research carried out in Python is explained to decide the neural network architecture that it is going to be implemented subsequently in the hardware. This architecture includes the number and the type of layers, the number of neurons that compose them.

Within the hardware design and implementation, we first explain the general aspects of the design, such as the communication protocols or the representation of the data used; and then details of the design in a top-down hierarchical approach are given, starting with the interface with the outer module and ending with the more internal modules.

The last chapter presents both the results of synthesizing the hardware design on a Xilinx Virtex-6 ML-605 Evaluation Board FPGA, and the experimental results of the hardware neural network tests.

KEYWORDS:

- LSTM
- VHDL
- FPGA
- Phython
- keras
- Neural networks
- Machine Learning

Autorización de difusión y uso

Los autores de este proyecto autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar el presente trabajo de investigación, tanto la aplicación como la memoria, únicamente con fines académicos, no comerciales y mencionando expresamente a sus autores. También autorizan a la Biblioteca de la UCM a depositar el trabajo en el Archivo Institucional E-Prints Complutense.

Autores:

- Jorge López Melchor
- Octavio Sales Calvo

Índice general

Índice de Figuras

Índice de Tablas

Parte I

INTRODUCCIÓN

Capítulo 1

Motivaciones

La insulina es una hormona segregada por el páncreas cuya misión es ayudar a las células a absorber la glucosa de la sangre. La diabetes mellitus tipo 1 es una de las dos variedades existentes que se caracteriza porque el páncreas es incapaz de segregar insulina y, por lo tanto, no se reduce la concentración de glucosa de la sangre, lo que puede dar lugar a episodios de hiperglucemia -cantidad excesiva de glucosa en sangre. Las personas que padecen esta enfermedad tienen múltiples problemas de salud que se van agravando a medida que pasa el tiempo: pérdida de visión, cardiopatías, problemas en la piel, fallos renales, etc.

La forma más eficiente de controlar los problemas de salud derivados de esta enfermedad es controlar la concentración de glucosa en sangre mediante inyecciones de insulina, pero este control no es nada sencillo porque en la glucemia influyen muchos factores como pueden ser la ingesta de carbohidratos, el estrés, la calidad del sueño o la actividad física. Un efecto derivado de esta complejidad es que un control deficiente de la glucemia puede conducir a episodios de hipoglucemia -cantidad de glucosa en sangre baja- que pueden acabar incluso en la muerte del paciente.

En la actualidad se está invirtiendo un gran esfuerzo en la implementación de un sistema de control de lazo cerrado, llamado páncreas artificial, que sea capaz de regular la concentración de glucosa en sangre de manera eficaz. Un parte importante de este control es la predicción de la glucemia en un futuro más o menos cercano. Si somos capaces de predecir los niveles de glucosa dentro de dos horas, también seremos capaces de suministrar dosis de insulina lo suficientemente ajustadas como para evitar las hiperglucemias y las hipoglucemias dentro de ese horizonte temporal.

Por otro lado, las redes neuronales están demostrando ser una potente herramienta para solucionar diversos problemas de inteligencia artificial como puede ser la predicción del comportamiento futuro en series temporales. En concreto, las redes de tipo Long Short-Term Memory parecen apropiadas para resolver diversos problemas de series temporales como pueden ser problemas de clasificación y de predicción. Esto nos lleva a pensar que la utilización de este tipo de redes neuronales en la predicción de la concentración de la glucosa en sangre puede ayudar a mejorar la predicción en el páncreas artificial.

Capítulo 2

Objetivos

El objetivo principal es implementar en hardware una red neuronal de tipo Long-Short-Term-Memory (LSTM) capaz de realizar predicciones de la concentración de la glucosa en pacientes con diabetes mellitus en un horizonte temporal de 30 minutos. La consecución de este objetivo principal requiere la realización de las siguientes tareas:

- Estudiar las redes neuronales LSTM, comprender su comportamiento y evaluar su efectividad para la predicción de series temporales.
- Definir y evaluar en Python distintas arquitecturas de redes LSTM y escoger aquella que proporcione predicciones más exactas.
- Describir en VHDL la red neuronal seleccionada en el ítem anterior.
- Simular el diseño para comprobar el correcto funcionamiento del flujo de datos entre los submódulos que lo componen y verificar que la descripción en VHDL coincide con las especificaciones proporcionadas como resultado de la evaluación y selección de la red LSTM realizado en Python.
- Sintetizar e implementar la red neuronal sobre la tarjeta de prototipado Xilinx ML-605 Evaluation Board.

Parte II

ESTADO DEL ARTE

Capítulo 3

Diabetes Mellitus

La *diabetes mellitus* es un grupo de enfermedades o síndromes metabólicos que se producen cuando el páncreas o bien no puede fabricar suficiente insulina (*diabetes mellitus* tipo 1) o ésta no consigue actuar porque las células no responden a su estímulo (*diabetes mellitus* tipo 2) [1]. La misión de la insulina es permitir que las células absorban la glucosa del torrente sanguíneo, por lo tanto, esta carencia o mal funcionamiento produce hiperglucemia, es decir, una alta concentración de glucosa en sangre. La hiperglucemia puede provocar problemas de salud a medio y largo plazo entre los que destacan: pérdida de visión, afectación renal, afectación en vasos sanguíneos provocando así cardiopatías, enfermedad vascular cerebral o isquémica intestinal [2].

La *diabetes mellitus* presenta un importante impacto sociosanitario por su alta incidencia, las diversas complicaciones que genera y el alto índice de mortalidad. El número de casos y su prevalencia han aumentado en las últimas décadas. La Federación Internacional de Diabetes sitúa la prevalencia mundial de la enfermedad en el 8,3 % y estima que se incrementará hasta alcanzar el 10,1 % en 2035. En España, las estimaciones de prevalencia de diabetes conocida como tipo 2, se sitúan entre el 4,8 % y el 18,7 % [3].

Para poder tener una calidad de vida aceptable los enfermos de *diabetes mellitus* deben controlar su glucemia,es decir, la concentración de glucosa en la sangre. El tratamiento para controlar la glucemia se basa en una alimentación equilibrada, actividad física, hábitos generales de higiene, patrones de sueño saludables, medicación, controles periódicos por el endocrinólogo, seguimiento diario de los niveles de concentración de glucosa en sangre y control de dicha concentración mediante el suministro diario de insulina [4]. La insulina tarda en hacer efecto, por lo tanto, lo ideal es conocer cómo va a ser la evolución glucémica del paciente para poder suministrar la insulina con tiempo suficiente como para evitar las hiperglucemias, que son los niveles elevados de glucosa en sangre. Por otro lado, también será importante tener en cuenta la hipoglucemia, bajos niveles de glucosa en sangre, que puede aparecer por suministrar dosis de insulina cuando en realidad no se necesitaban. Así, conocer con antelación cuál va a ser el índice glucémico del paciente puede evitar que se inyecte insulina por equivocación, produciéndose de esta manera un evento severo de hipoglucemia que puede llevar al paciente a la muerte [5].

El principal problema de este control es la variedad de factores que afectan a la concentración de glucosa como, por ejemplo, el ejercicio físico, las ingestas de carbohidratos, la ansiedad, la calidad del sueño, etc. lo que hace que la predicción no sea sencilla [5].

Para la presente investigación, se van a utilizar redes neuronales profundas para realizar la predicción de la glucemia. En concreto, las redes neuronales LSTM que proporcionan predicciones de mayor precisión frente a las obtenidas con técnicas clásicas, como por ejemplo ARIMA [6].

Capítulo 4

Redes Neuronales

Las redes neuronales artificiales se han vuelto cada vez más populares para resolver problemas computacionalmente complejos en diversas áreas como la medicina, industria, negocios, etc. Una definición comúnmente aceptada en el área de la inteligencia artificial se debe a Robert Hetch-Nielsen [7]: "Sistema de computación que consta de un gran número de elementos simples, muy interconectados, que procesan la información respondiendo dinámicamente frente a unos estímulos externos" [7]. Tal sistema está inspirado en el comportamiento biológico de las neuronas y en cómo se organizan, formando la estructura del cerebro, interconectándose entre ellas en forma representativa de red [8]. En la definición de Robert Hetch-Nielsen se referencia como elemento simple, comúnmente conocido como neurona, a la unidad de procesamiento de información que utiliza modelos matemáticos inspirados en sistemas biológicos adaptados y simulados en computadoras convencionales. La idea general es emular la capacidad de aprendizaje del cerebro humano a través del sistema nervioso.

Como acontecimiento histórico dentro de lo que se denominaría "gestación de las redes neuronales" [9] , en 1951, dos estudiantes graduados en el Departamento de Matemáticas de Princeton, Marvin Minsky y Dean Edmonds, construyeron la primera arquitectura hardware de una red neuronal. El SNARC (Stochastic Neural Analog Reinforcement Computer), como se llamó, era una máquina física que utilizaba 3.000 válvulas de vacío y un mecanismo de piloto automático (obtenido de los desechos de un avión bombardero B-24) para implementar una red con 40 neuronas [10] que simulaba el comportamiento de los ratones al moverse por un laberinto. Tras un error en el diseño electrónico, Minsky vio que podía incluir la emulación de dos o tres ratones más al mismo tiempo en su máquina, y que aprendían de los otros a la hora de encontrar la salida [11].

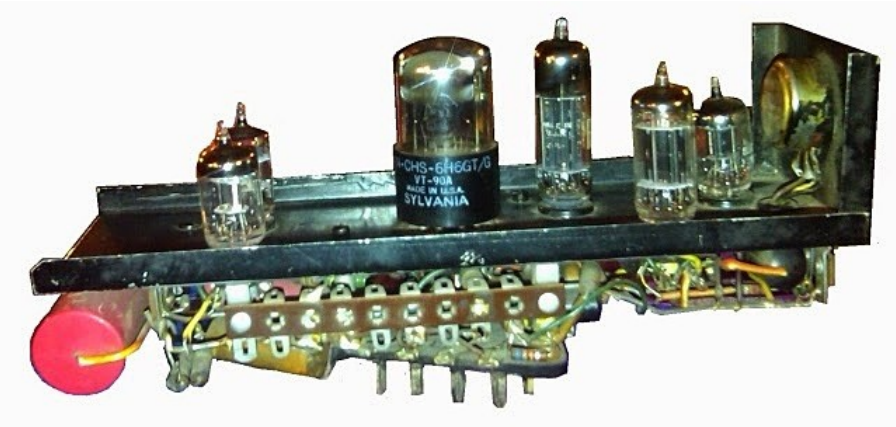

Figura 4.1: Neurona física del SNARC. [9].

4.1. Fundamentos, Desarrollo y Evolución

Una red neuronal biológica es un sistema compuesto por células individuales. Éstas permiten establecer una relación de aprendizaje entre las entradas y salidas de información que genera el sistema nervioso. Sus elementos principales son dendritas, axón, cuerpo y sinapsis [12].

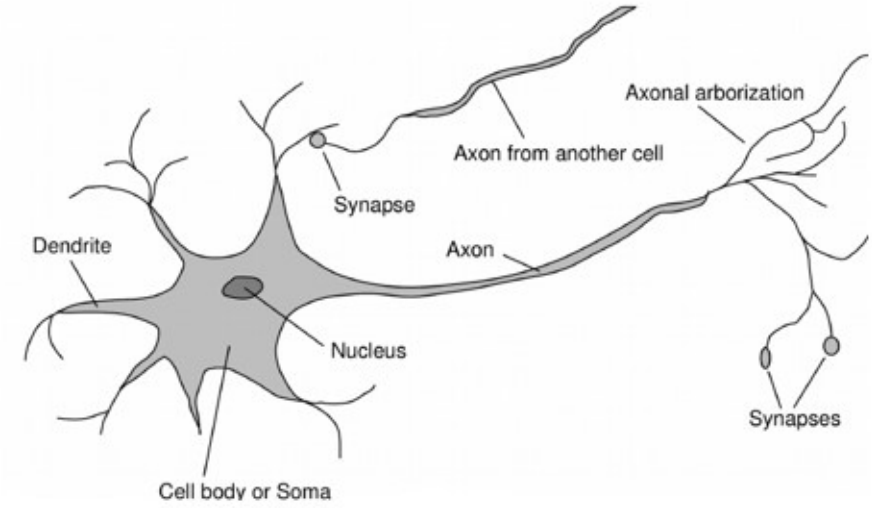

Figura 4.2: Neurona biológica [13].

Como se puede observar en la Figura 4.2:

- Las dendritas (Dentrite) son el principal elemento de recepción de información de las neuronas [12].
- El axón (Axon) es el responsable de transportar y transmitir la información recogida e integrada por las dendritas y el soma a otras neuronas. Sin embargo, no es un

elemento pasivo puesto que hay también procesamiento e integración axónica de la información [12].

- El cuerpo o soma (Cell body) es donde está contenido el núcleo del que parten las dendritas y también el axón [12].
- Las sinapsis (Synapses) son estructuras encargadas del intercambio de información entre dos neuronas [12].

En el área de la inteligencia artificial, una neurona es la unidad mínima de procesamiento de información conectada a su vez a una o múltiples neuronas mediante conexiones sinápticas [8]. Dichas conexiones tienen un peso $W_(i,j)$, que determina la importancia asociada al dato x_i, que viaja desde la salida de una neurona i hasta la entrada de otra j. Según [8], en cada neurona la combinación lineal de las entradas x_i, usando los pesos sinápticos $W_(i,j)$, pasa por una función de activación que devuelve un valor resultado y_j. El correcto funcionamiento de una red neuronal proviene de determinar el peso asociado a todas las conexiones de la red.

En la figura 4.3 se muestra la imagen de una neurona artificial. La expresión $\sum in_j$ corresponde a la fórmula 4.11 por simplicidad en el diseño de la imagen.

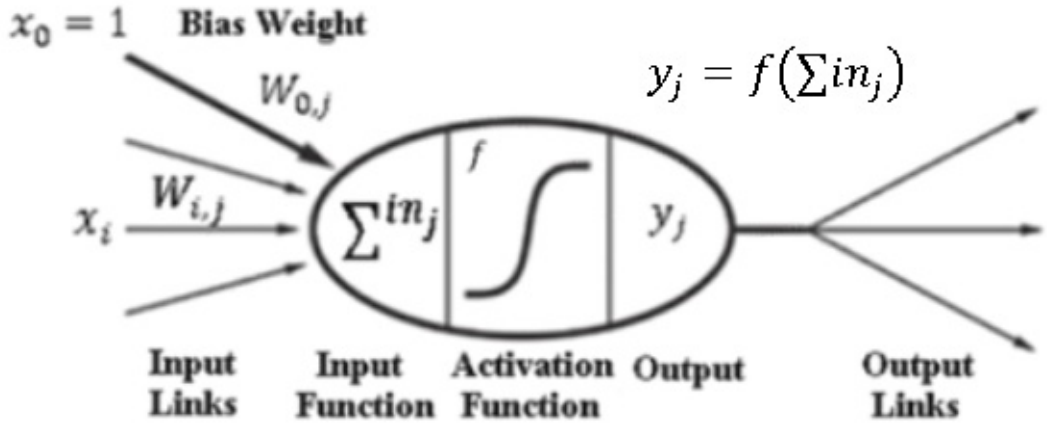

Figura 4.3: Neurona artificial [13].

- El conjunto de entrada x_i, puede venir del exterior o procedentes de otras neuronas [17, 8]. Las entradas y salidas de una neurona pueden ser binarias (digitales) con valores 0,1 o -1,1, continuas (analógicas) que admiten bien un rango de valores entre [-1, 1] u otros valores que luego serán acotados por la función de activación en alguno de los dos rangos anteriores. Cuando se tiene una entrada y una salida analógica se define como redes continuas. Si la entrada es analógica y la salida binaria, es una red híbrida. Por último, si la entrada y la salida son binarias son redes discretas [13, 14].
- Los pesos sinápticos $W_{i,j}$, sirven para guardar el conocimiento adquirido, es decir, el valor del peso que está asociado a la entrada para conseguir una salida deseada que, a su vez, será la entrada de otras neuronas. Se define de esta forma la relación

entre las neuronas como una conexión sináptica que puede tomar valores positivos, negativos o cero. Si el peso ponderado es del mismo signo, actúa como excitador y si por el contrario son distintos, actúa de inhibidor. En caso de que el peso sea cero, la relación entre el par de neuronas no existe. El ajuste de los pesos determinará el paso o no del umbral que es marcado por la función de activación que se detallará más adelante [13, 14].

$$(W_{i,j} \cdot x_i) \tag{4.1}$$

- La función de entrada o regla de propagación, a la cual nos referiremos en este documento como suma pesada, es el sumatorio de entrada x_i por el valor de su peso asociado $W_{i,j}$. El resultado se considera el estímulo de importancia que se asocia a los datos que recibe la neurona por la entrada [13, 14].

$$\left(\sum_{j=1}^{n_j} W_{i,j} \cdot x_i \right) \tag{4.2}$$

- El sesgo del peso sináptico b_j (Bias Weight), es el que decidía en las redes iniciales la activación de la función de salida. Si el resultado de la suma pesada (sumatorio de los pesos por las entradas) era menor que un valor umbral b_j, la salida era 0 y no se activaba la función. En el caso contrario, la salida era 1 y por lo tanto se activaba la función. En redes más complejas con una función de activación, puede ser considerado como un peso sináptico a favor o en contra del valor que se desea obtener antes de aplicar la función de activación. El sesgo de peso sináptico puede interpretarse como un peso sináptico más, pero con la entrada asociada $x_0 = 1$ [13, 14]. La ecuación (4.1) representa la función de salida de una red neuronal con el peso b_j como umbral.

$$f(n) = \begin{cases} 0 & \text{si } w_{i,j} \cdot x_i + b_j <= 0 \\ 1 & \text{si } w_{i,j} \cdot x_i + b_j > 0 \end{cases} \tag{4.3}$$

- La función de activación toma como entrada el resultado de la suma pesada que actúa como una función limitadora o umbral de los datos [13, 14].

$$f\left(in_j\right) \tag{4.4}$$

- La salida y_j, es el estado de activación o la salida de la neurona j [13, 14].

$$y_j = f\left(\sum_{j=1}^{n_j} W_{i,j} \cdot x_i \right) \tag{4.5}$$

Existen diversas funciones de activación [15] donde cada una determina un rango de valores asociado a las respuestas de las neuronas. La elección de la función de activación será determinada por el problema o aplicación a resolver. Fuera del esquema planteado en la 4.3, se muestran otras funciones de activación como:

- La función umbral (threshold function) generalmente se utiliza para establecer criterios de clasificación puesto que su salida es 0 o 1 (verdadero o falso) [15].

$$f(n) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x >= 0 \end{cases} \tag{4.6}$$

- La función signo determina los valores -1, 0 o 1 como respuesta de la neurona [15] .

$$sgn(x) = \begin{cases} -1 & \text{si } x < 0 \\ 0 & \text{si } x = 0 \\ 1 & \text{si } x > 0 \end{cases} \tag{4.7}$$

- La función logística o función de activación sigmoidal es una de las más usadas en las redes neuronales, su salida es continua a valores en el rango [0, 1] e infinitamente diferenciable, y es usada en problemas de aproximación como versión continua de la función umbral [15] .

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.8}$$

Figura 4.4: Gráfica de la función de activación sigmoidal.

- La función tangente hiperbólica es importante por sus propiedades analíticas y es usada en problemas de aproximación. Es la versión continua de la función signo a valores en [-1, 1] e infinitamente diferenciable [15].

$$tanh(x) = \frac{\sinh(x)}{\cosh(x)} \tag{4.9}$$

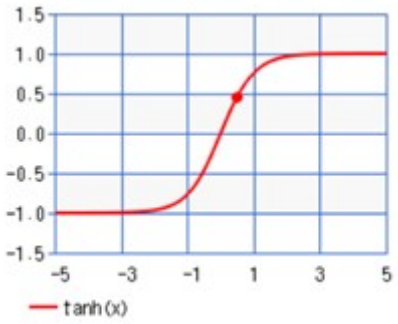

Figura 4.5: Gráfica de la función de activación tangente hiperbólica.

- La función lineal, al ser una recta, no limita la respuesta de la neurona y es aplicada en problemas de aproximación lineal [15].

$$f(x) = (mx + b) \tag{4.10}$$

Capas de una red

En la figura 4.6 se puede observar una red neuronal organizada por las capas que se detallan a continuación [16]:

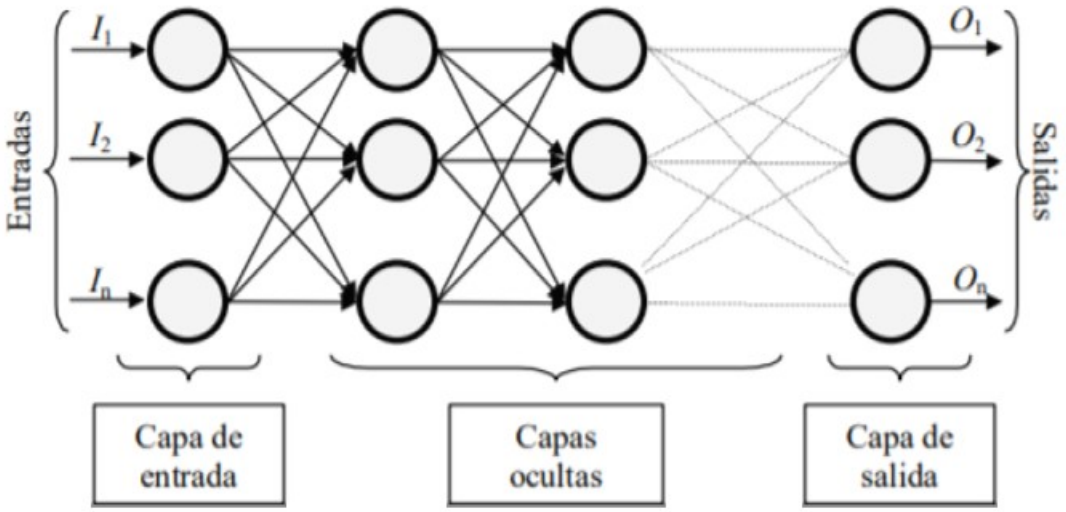

Figura 4.6: Capas de una red neuronal que catalogan a una red compuesta por neuronas simples totalmente conectadas [16].

- Capa de entrada: está compuesta por neuronas de entrada que reciben información del exterior referidas anteriormente como señales de entrada [16].
- Capa de salida: está compuesta por neuronas de salida que reciben los cálculos de la capa intermedia, dando como resultado en la salida el valor predictivo generado

en la red neuronal. Este valor predictivo, puede ser de distinto tipo dependiendo de sus características (finalidad, clasificación, etc.).

La función de salida de esta capa y de la capa de entrada normalmente proporciona el resultado de la función de activación, pero también puede proporcionar el mismo valor de entrada gracias a la función identidad (4.11). Todo ello depende de la especificación de la red [14].

$$f(x) = (x) \tag{4.11}$$

- Capas ocultas: constituyen la capa intermedia de la red compuesta por neuronas ocultas no conectadas directamente con la entrada o con la salida. El número de niveles ocultos es variable y depende de la complejidad del problema a resolver. Las neuronas de las capas ocultas pueden estar interconectadas de distintas maneras, lo que determina junto con su número, las distintas topologías de redes neuronales [16].

4.2. Redes Neuronales Recurrentes

Hasta el momento se ha explicado cómo una red estructurada en capas compuesta por una o varias neuronas, transmitían la información en un único sentido: los datos llegaban por la capa de entrada, se transformaban en la capa oculta y se devolvía un resultado por la capa de salida [17].

Las redes neuronales recurrentes (RNN) son capaces de reconocer y predecir secuencias de datos a lo largo del tiempo. Esto quiere decir que las neuronas que componen dicha red o parte de ella, además de tener una salida que propaga los datos hacia la siguiente neurona, o hacia afuera en el caso de la capa de salida, tienen otra salida que será una de sus entradas en el siguiente paso de tiempo [17].

Este tipo de redes se fundamentan en bucles temporales, divididas en pasos de tiempo, compuestas por neuronas que reciben en un paso de tiempo actual (t), por una de sus entradas, la salida de la propia neurona en el paso de tiempo anterior (t-1) [18].

Para entender su funcionamiento podemos considerar una red multicapa simple con una sola neurona (A) dentro de la capa oculta, que recibe como entradas: $h_(t-1)$ correspondiente a la salida de la red en el paso de tiempo anterior, y x_t correspondiente al dato de entrada en el paso de tiempo actual [18]. Como salidas de dicha neurona, propaga h_t hacia afuera y hacia sí misma en forma de realimentación, tal y como se puede ver en la 4.7.

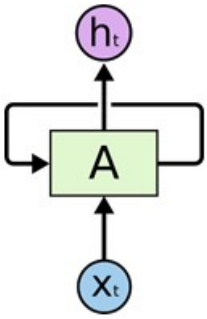

Figura 4.7: Neurona recurrente que muestra la arquitectura de una neurona perteneciente a una RNN básica [18].

En la figura 4.8 se muestra la serie temporal de A que recibe como entradas $x_(t)$ y $h_(t-1)$, comentadas anteriormente en cada paso de tiempo t. La salida h_t, es la salida de la red en cada paso de tiempo t [18].

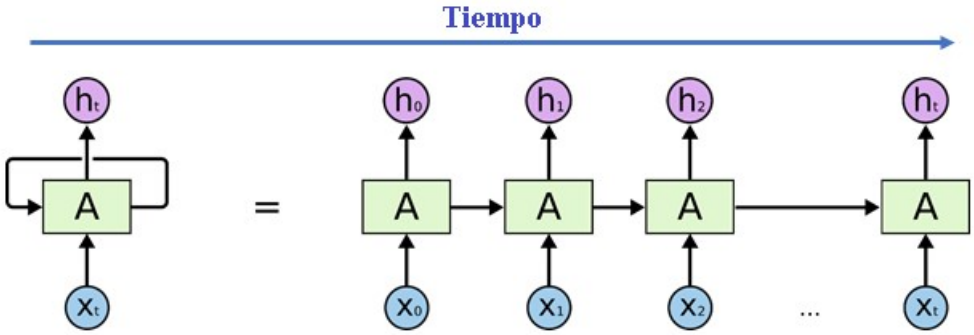

Figura 4.8: Neurona oculta recurrente en el tiempo que representa una serie temporal [18].

El bucle formado entre la entrada y la salida permite recordar la información a lo largo del tiempo. Es precisamente ese bucle el que realiza la acción de memoria para este tipo de redes. En la medida que añadimos capas compuestas por varias neuronas formando una arquitectura de red, se incrementa su capacidad de modelado, así como su capacidad de reconocer mayores secuencias con menor error en cada paso de tiempo [18].

En el entrenamiento surgen dos problemas a la hora de propagar los datos.Por un lado,la influencia del dato propagado se debilita a medida que avanza el algoritmo, debido a la influencia de la señal de error en tiempos anteriores [18]. Este fenómeno fue descrito en 1997 por Hochreiter & Schmidhuber como vanishing gradient [19, 20]. Por otro, el problema es la asignación de una importancia exageradamente alta a los pesos, dando como resultado actualizaciones muy grandes de los mismos. Estos son definidos como gradientes explosivos (Exploding Gradients) [18].

El gradiente de una función escalar multivariable en un punto es un vector que representa la dirección en la cual el cambio producido en los valores de la función es más acentuado en dicho punto y se calcula como la derivada parcial de la función con respecto a sus entradas en dicho punto. Para el caso particular de una función que depende de una sola variable, el gradiente se puede ver como la pendiente de dicha función en un punto. El gradiente también indica el cambio a realizar en los valores de todos los pesos con respecto al cambio en el error producido.

En una red neuronal, cuanto más alto es el gradiente, más pronunciada es la pendiente y más rápido puede aprender un modelo [17]. El proceso de aprendizaje se detiene cuando el gradiente en un punto es 0. En ese caso, la recta es paralela al eje de abscisas.

Las regiones en las que el error es relativamente bajo se las considera como mínimos locales, y a la región con el mínimo más pequeño, es decir el error más bajo, se la considera un mínimo global. En el caso contrario, a las regiones con el error más elevado se las considera máximos locales y a la región con el error más grande se la considera un máximo global [21].

En la figura 4.9 se detallan algunos de los puntos mencionados anteriormente además de un punto denominado como candidato a solución. Éste punto se encuentra en un conjunto de mínimos locales, puesto que encontrar un conjunto de pesos suficientemente buenos es más deseable y manejable a la hora de recibir datos nuevos que encontrar un conjunto global de pesos óptimos [21].

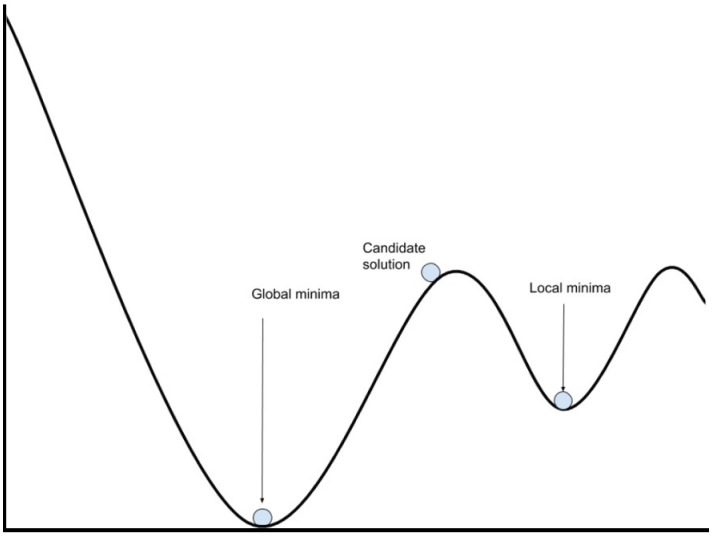

Figura 4.9: Gráfica del gradiente en una RNN donde se muestran diferentes puntos de error [21].

En resumen, cuando se realiza el desacoplamiento temporal de la red, se calcula el error de propagación hacia el principio de la secuencia temporal mediante el cálculo de las pérdidas del gradiente con respecto a los pesos. Los gradientes tienden a ser cada vez más pequeños a medida que nos movemos hacia el final la red. Esto significa que las neuronas del principio, empezando por t_0, aprenden muy lentamente en comparación con las neuronas que se encuentran al final de la de la serie temporal en $t_n - 1$, donde n

representa al número de pasos de tiempo. En redes dónde se pretende tener la memoria necesaria para pronósticos de series temporales es un problema [18].

4.3. Long Short-Term Memory

Una red Long Short-Term Memory (LSTM), es una red neuronal recurrente compuesta por una o varias celdas LSTM que soluciona los problemas de dependencia a largo plazo. Su principal característica es que puede eliminar o agregar información al estado de dicha celda, a la cual nos referiremos en esta sección como celda de estado, utilizando unas estructuras llamadas puertas. [22].

Para entender la estructura de una celda LSTM se puede observar en la figura 4.10 junto con la figura 4.11, una visión generalizada de la misma con su leyenda, antes de comenzar con la lectura del flujo de los datos que entran a la celda, su transformación dentro de la celda y la propagación de los nuevos datos de salida hacia la siguiente celda LSTM o capa de salida.

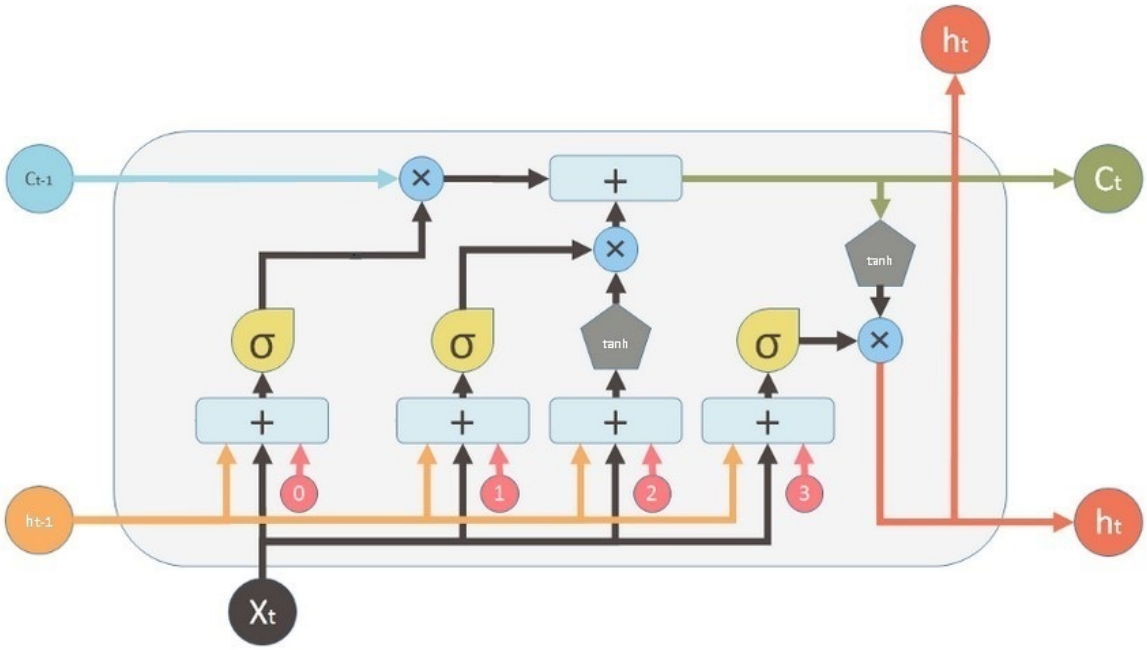

Figura 4.10: Estructura de una celda LSTM que representa una celda LSTM en el tiempo t [23].

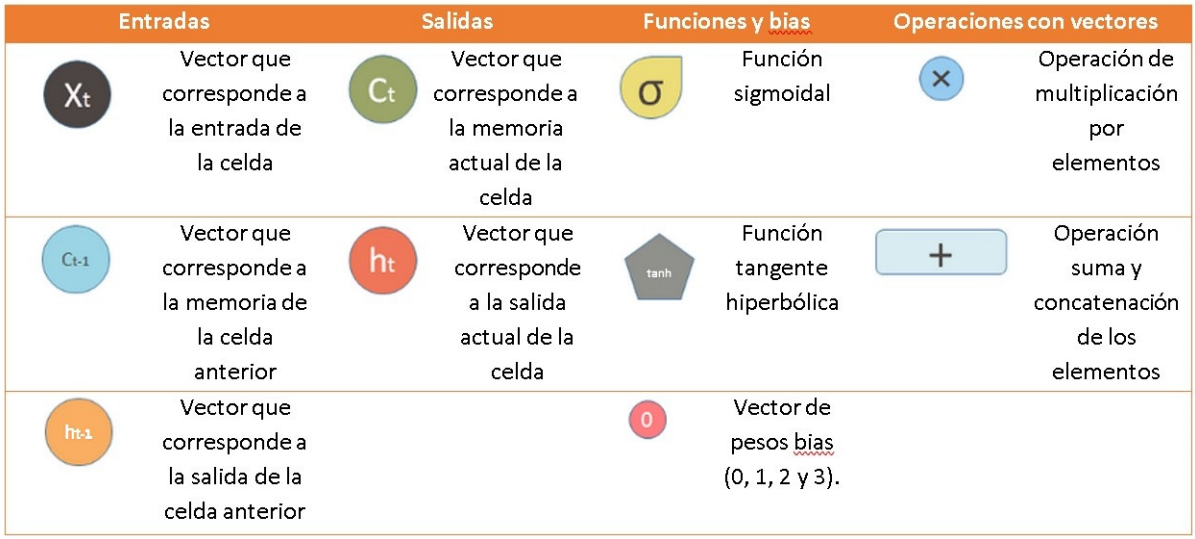

Figura 4.11: Leyenda de una celda LSTM en el tiempo t [23].

En el diagrama anterior, cada línea transporta un vector completo y una bifurcación de línea denota que su contenido se copia y las copias van a diferentes ubicaciones.

La celda LSTM recibe como entradas las salidas del estado oculto anterior $h_t - 1$ y el estado anterior de la celda $c_t - 1$, ambos corresponden respectivamente al valor de salida de la celda y al estado actual de la memoria, calculadas en el paso de tiempo anterior. El valor x_t corresponde a la entrada en el paso de tiempo actual [22]. Dentro de la celda, como se comenta anteriormente, existen tres puertas: la puerta de entrada, olvido y salida. Cada una está formada por una función de activación que se aplica al resultado de una suma pesada calculada por el producto de los valores de los pesos de entrada W_i, y la entrada x_t, más el producto de los valores de los pesos de estado U_i y el valor del estado oculto anterior $h_(t - 1)$, más el sesgo b_i [22].

La puerta de olvido se encargará de decidir que datos almacenados en la celda de estado se conservan olvidando el resto. Primero realiza el cálculo de la suma pesada y después aplica la función sigmoidal al resultado anterior [22]. En la figura 4.12, el subíndice f, corresponde a la puerta de olvido (forgate gate) en el paso de tiempo t.

$$f_t = sigmoid\,(U_f \cdot h_{t-1} + W_f \cdot x_t + b_f) \tag{4.12}$$

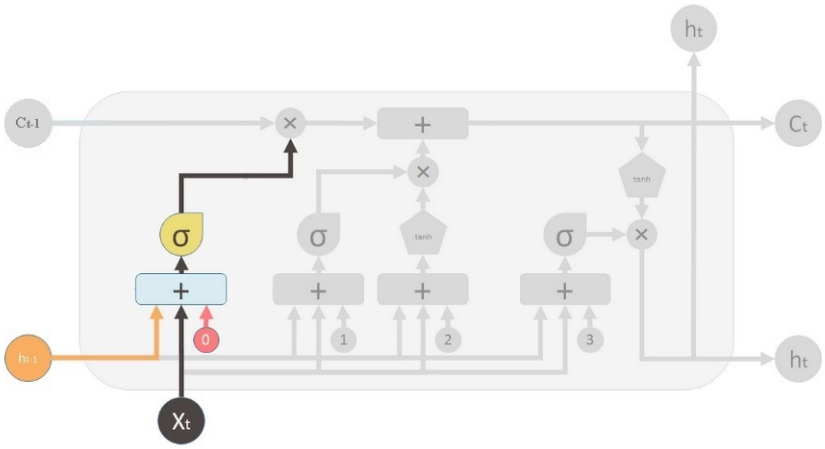

Figura 4.12: Puerta de olvido de una celda LSTM en el tiempo t [23].

Para decidir la nueva información que almacenará la celda de estado, la puerta de entrada evalúa qué datos deben actualizarse con la función de activación sigmoidal y la función de activación tangente hiperbólica generando un vector con los valores candidatos. Es decir, con la función sigmoidal de la puerta de entrada se decide qué se va a modificar y con la tangente hiperbólica que calcula el nuevo candidato, cómo se modifica dicho valor. Ambas funciones se aplican sobre el resultado de la suma pesada correspondiente a cada función. Por último, se realiza el producto con el resultado de ambas funciones para combinar los nuevos datos candidatos con los valores que se decidieron conservar [22]. En las fórmulas 4.13 y 4.14, los subíndices i y nc, corresponden a la puerta de entrada (input gate) y el cálculo del nuevo candidato (new candidate) respectivamente en el paso de tiempo t.

$$i_t = sigmoid\left(U_i \cdot h_{t-1} + W_i \cdot x_t + b_i\right) \tag{4.13}$$

$$nc_t = tanh\left(U_{nc} \cdot h_{t-1} + W_{nc} \cdot x_t + b_{nc}\right) \tag{4.14}$$

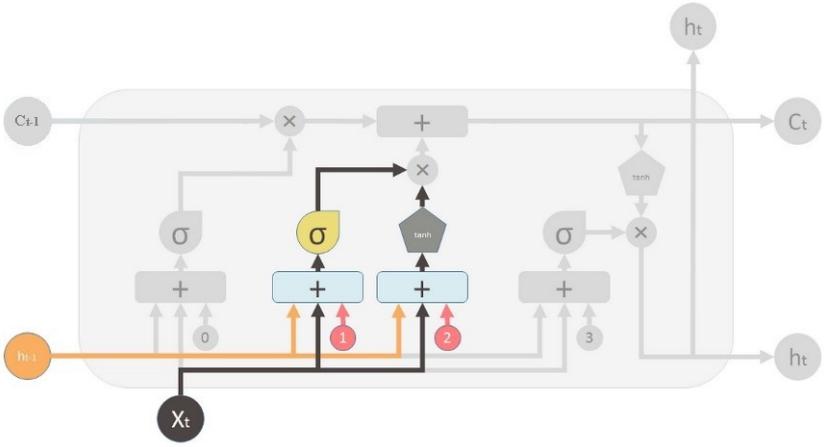

Figura 4.13: Puerta de entrada de una celda LSTM en el tiempo t [23].

El nuevo valor de la celda de estado se calcula con la concatenación de los vectores resultantes de: la salida de la puerta de olvido por el estado actual de la memoria y la salida de la puerta de entrada por el nuevo vector candidato, ambas operaciones entre vectores se realizan como un producto escalar. Si la multiplicación de la memoria actual $c_t - 1$ es con un vector, de la salida de la puerta de olvido cercano a 0, significará olvidar la mayor parte de dicha memoria. En caso contrario, cercano a 1, se preservará la mayor parte de la memoria actual. Por otro lado, la multiplicación de la puerta de entrada y el nuevo vector candidato, como se comenta en el apartado anterior, contienen la nueva información que se va a actualizar en la celda de estado [22]. En la fórmula 4.15, el subíndice t, corresponde al paso de tiempo t para la celda de estado (cell state).

$$c_t = f_t \cdot c_{t-1} + i_t \cdot nc_t \tag{4.15}$$

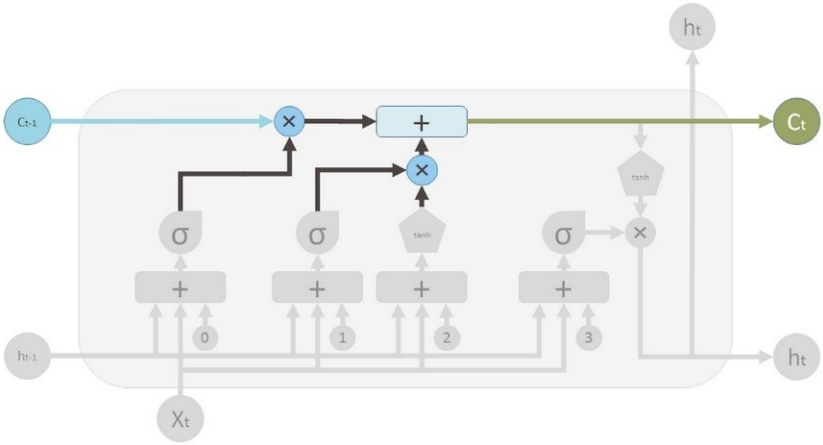

Figura 4.14: Celda de estado de una celda LSTM en el tiempo t [23].

La salida que genera la celda es el nuevo estado oculto referenciado como ht. Dicho estado es el resultado del producto entre los valores que genera la puerta de salida, la cual se calcula con la función de activación sigmoidal sobre la suma pesada correspondiente, y los datos actuales en la celda de estado acotados entre -1 y 1 [22]. En las fórmulas 4.16 y 4.17, los subíndices o y t, corresponden a la puerta output y el paso de tiempo t respectivamente.

$$h_t = o_t \cdot tanh\,(c_t) \tag{4.16}$$

$$o_t = sigmoid\,(U_o \cdot h_{t-1} + W_o \cdot x_t + b_o) \tag{4.17}$$

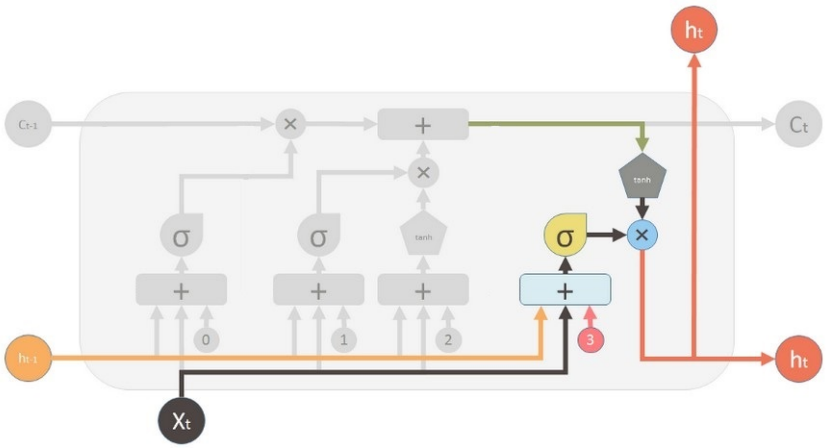

Figura 4.15: Puerta de salida de una celda LSTM en el tiempo t [23].

según [22] La celda LSTM utiliza dos de las funciones de activación que se explicaron en la sección 4.1:

- La función de activación sigmoidal se aplica a las tres compuertas para proteger y controlar el estado de la celda [18].
- La función de activación tangente hiperbólica se aplica para la elección del nuevo candidato [22].

En el entrenamiento de una red LSTM se elimina el problema del descenso de gradiente, producido por las repetidas multiplicaciones de jacobianos durante la propagación hacia atrás, al contar con la celda de estado que actúa de memoria a largo plazo. Gracias a la celda de estado se realizan actualizaciones aditivas, no multiplicativas, del estado de la celda, por lo tanto, evitan que los gradientes se desvanezcan. Por otro lado, aún pueden sufrir gradientes explosivos, explicados en el apartado 4.2 de las RNN [24].

4.4. Usos en la actualidad de las Redes Neuronales

Las redes neuronales se utilizan en un gran número de áreas gracias a su aprendizaje adaptativo capaz de realizar tareas basadas en un entrenamiento lo que les permite crear su propia organización de la información con gran tolerancia a fallos [16]. En el área de la implementación software de redes neuronales hemos encontrado [19]:

- Conversión de texto a voz cambiando los símbolos gráficos de un texto por lenguaje hablado.
- Procesado del lenguaje natural.
- Compresión de imágenes transformando los datos de una imagen a una representación distinta que necesite menos memoria.
- Reconocimiento de patrones en imágenes como la clasificación de objetos detectados por un sonar.
- Problemas de combinatoria dando soluciones mediante cálculos tradicionales que requieren un tiempo de proceso (CPU) exponencial con el número de entradas.

En el área de la implementación hardware hemos encontrado:

- Implementación de una red neuronal artificial tipo SOM (self-organizing maps) en una FPGA para la resolución de trayectorias tipo laberinto [25].
- Reconocimiento de caracteres [26].
- Algoritmos para la implementación hardware de redes neuronales compactas, que explota las propiedades especiales de las funciones booleanas, que, a su vez describen el funcionamiento de las neuronas artificiales con la función de activación por pasos [27].
- Implementación de hardware de una red neural artificial de perceptrón de múltiples capas totalmente digital que utiliza las matrices de puertas programables de campo de Xilinx [28].
- Implementación del hardware del controlador de una red neural inteligente con un chip de propósito general basado en una matriz de puertas programables (FPGA) y una placa de procesamiento de señales digitales (DSP) para resolver problemas de control de sistemas no lineales [29].

Parte III

IMPLEMENTACIÓN

En esta segunda parte de la memoria se presenta el diseño hardware de la red LSTM que proponemos. En el capítulo 5 se explica el estudio que se ha realizado en Python para decidir la arquitectura de la red. Esta arquitectura incluye el número y tipo de capas, el número de neuronas que las componen y el valor de sus pesos. En el capítulo 6 primero se explican los aspectos generales del diseño, como los protocolos de comunicación o la representación de los datos utilizados, y posteriormente se detalla el diseño siguiendo una aproximación jerárquica top-down, empezando por la interfaz y finalizando con los módulos más internos.

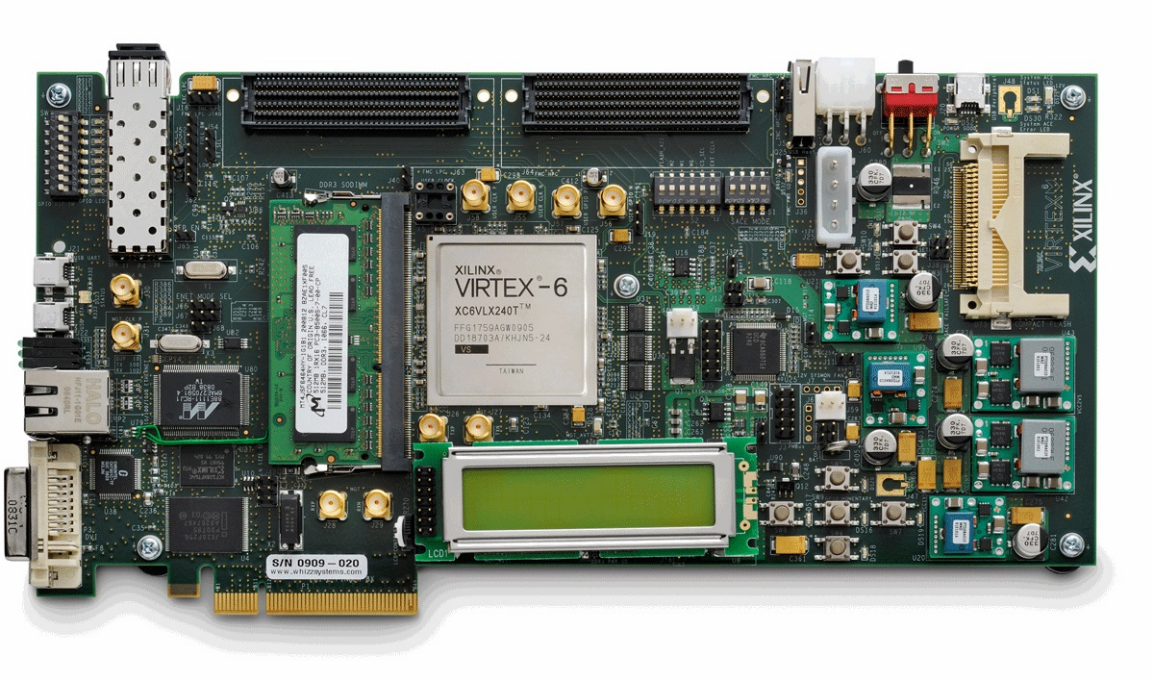

Figura 4.16: "Kit de evaluación Virtex®-6 FPGA ML605 incluye todos los componentes básicos de hardware, herramientas de diseño, IP y un diseño de referencia previamente verificado para diseños de sistemas que exigen alto rendimiento, conectividad en serie e interfaz de memoria avanzada-[30].

Capítulo 5

Modelado en Python

La implementación software de la red se realizó en Python. La razón es que este lenguaje cuenta con la librería Keras, que es una librería dedicada a redes neuronales muy completa. Esto nos permitió realizar un estudio minucioso pero rápido de diversas arquitecturas de red. El estudio realizado lo podemos dividir en tres partes:

- Creación del dataset a partir de la base de datos facilitada. Para que una red pueda llevar a cabo la predicción de glucemia, Keras requiere que los datos se encuentren en una estructura prefijada, con lo que debemos realizar diversas transformaciones sobre los datos. También cabe destacar el análisis sobre la base de datos, donde determinamos que información es relevante y por tanto introducimos en el dataset.
- Implementación mediante keras de diferentes redes LSTM con el objetivo de probar su validez en la predicción del nivel de glucosa en sangre y de obtener los pesos ya entrenados, para la posterior carga en la red hardware.
- Diseño e implementación de nuestra propuesta de red LSTM, a la que llamamos HwNN, sin utilizar las primitivas proporcionadas por Keras. El objetivo primario de implementar HwNN es confirmar la viabilidad de poder recrear una red implementada con Keras, integrando los pesos entrenados de Keras en HwNN y obtener los mismo resultados en las predicciones.

Creación del dataset

El primer paso para realizar un modelo en Python mediante la librería Keras es la creación de un dataset acorde con el modelo en cuestión. Para llevarlo a cabo, contamos con una base de datos de pacientes con diabetes donde se tiene registrada su glucemia, la ingesta de hidratos de carbono y el nivel de insulina en sangre. Estas mediciones están tomadas cada 5 minutos durante un periodo de tiempo.

Keras espera que el dataset de entrada, que representa la serie temporal, tenga la estructura de una matriz de tres dimensiones 5.1, en cambio, el dataset de salida es un vector

de una dimensión 5.2 donde se recoge el resultado esperado de la predicción por cada muestra de entrada.

$$[\text{ número de muestras, pasos de tiempo, número de características }] \tag{5.1}$$

$$[\text{ número de predicciones }] \tag{5.2}$$

Las entradas de todos los modelos implementados son: glucemia, hidratos de carbono e insulina en sangre. Llamamos paso de tiempo a la medición de los valores de las características en un instante concreto. El objetivo de nuestro modelo predictivo es hallar la glucemia a treinta minutos vista, por lo tanto, definimos el intervalo de estas mediciones en treinta minutos. En nuestro dataset, por cada registro de las características, se tiene un histórico desde dos horas atrás hasta las actuales. Entonces, si el paso de tiempo es de 30 minutos y cada muestra empieza con el registro de hace dos horas hasta las actuales tenemos 120 min / 30 min = 4 pasos de tiempo, y añadiendo el registro del instante actual, establecemos un total de 5 pasos de tiempo, lo que significa que esta muestra está conformada por cinco pasos de tiempo, conteniendo cada uno de ellos el registro de las tres características. A continuación, ilustramos el razonamiento mediante una figura y una breve explicación:

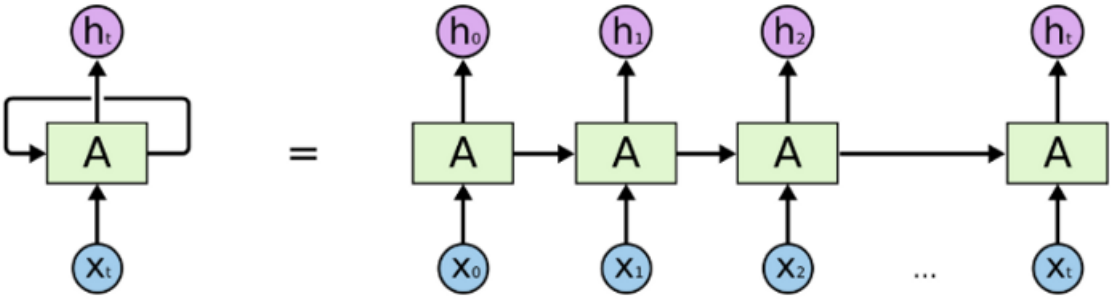

Figura 5.1: Desenrrollamiento temporal de una celda LSTM [18]

Dada una muestra de entrada con la estructura establecida, para que la celda LSTM realice una predicción de la glucemia a treinta minutos vista, esta se conforma de:

- X_0: entrada en el primer paso de tiempo, se trata de las tres características registradas dos horas atrás.
- X_1: entrada en el segundo paso de tiempo, se trata de las tres características registradas hace hora y media.
- X_2: entrada en el tercer paso de tiempo, se trata de las tres características registradas una hora antes.

- X_3: entrada en el cuarto paso de tiempo, se trata de las tres características registradas media hora antes.
- X_4: entrada en el quinto paso de tiempo, se trata de las tres características registradas en este instante.

Después de haber facilitado a la celda todo este conjunto de entradas propaga el resultado de la predicción de la glucemia dentro de 30 minutos.

Finalmente llevando a cabo este proceso, obtenemos el dataset de entrada y el dataset de salida con las siguientes dimensiones:

Data set Entrada			Data set Salida
Número de muestras	Pasos de tiempo	Número de características de la entrada	Resultados
56668	5	3	56668

Tabla 5.1: Dataset de entrada y salida

Después de haber estructurado nuestro dataset, se debe normalizar los datos en el rango [0,1] para que el modelo pueda trabajar de forma apropiada con ellos. Se realiza una normalización independiente a cada una de las características, es decir, se transforman los valores de la glucemia, hidratos de carbono y nivel de insulina con escalas distintas. Por último, determinamos qué porcentaje del dataset de entrada destinamos para el entrenamiento y cuánto para la validación del modelo. Formamos el conjunto de entrenamiento con el 67 % de los datos y con el 33 % restante el conjunto de test para su posterior validación, quedando así, un total de 37967 muestras en el conjunto de entrenamiento y 18701 en el de test.

Modelo LSTM de Keras

Keras es una librería capaz de implementar muchos tipos de redes neuronales. En nuestro estudio nos centramos en utilizar redes que solo contienen capas LSTM. En esta sección nos disponemos a explicar en que se basa nuestro procedimiento para llevar a cabo los diferentes modelos implementados con esta librería.

Todos los modelos realizados tienen la siguiente estructura:

- Capa de entrada: está constituida por una muestra de entrada a la cual se le aplica la función identidad.
- Capa oculta: es una capa LSTM, a la cual se la parametriza de distintas formas en las pruebas.
- Capa de salida: es una capa LSTM de un solo estado oculto.

Teniendo en cuenta todo lo descrito anteriormente, se define el modelo de la red con los siguientes parámetros:

- hidden neurons: es el número de estados ocultos que se le asignará a la capa oculta del modelo. Estos estados definen el tamaño de los vectores de salida (celda de estado y estado oculto), y de las matrices de pesos.
- batch size: es el tamaño del conjunto de datos que se le pasa como entrada.
- stateful: es una variable booleana para determinar si el modelo en cada muestra debe reiniciar sus estados (poniéndolos a 0) o no. Indicando así a la red si las muestras de cada batch del entrenamiento son consecutivas en el tiempo.
- return sequence: es un valor booleano que determina si la capa oculta propaga su estado en cada paso de tiempo, o solo en el último, indicando si se requiere una predicción por paso de tiempo, o no.

A continuación, se muestra la arquitectura de un modelo con 15 hidden_neuron 5 pasos de tiempo, 3 características de entrada, un tamaño de batch (igual a una muestra) y las variables de statefull y return_sequence activas:

```
Model: "Model_15_St"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input (InputLayer)           [(1, 5, 3)]               0
_________________________________________________________________
LSTM0 (LSTM)                 (1, 5, 15)                1140
_________________________________________________________________
LSTMout (LSTM)               (1, 1)                    68
=================================================================
Total params: 1,208
Trainable params: 1,208
Non-trainable params: 0
_________________________________________________________________
```

- **Salida:**
  - input(tamaño del batch, pasos de tiempo, características)
  - LSTM0(tamaño del batch, pasos de tiempo, estados ocultos LSTM0)
  - LSTMout(tamaño del batch, estados ocultos LSTMout)

Figura 5.2: Arquitectura del modelo en Python obtenida en jupyter

Una vez finalizada la definición del modelo toca entrenar la red haciendo uso del conjunto de entrenamiento. El número de iteraciones a realizar sobre el conjunto de entrenamiento (epoch como se aparece en la Figura 5.3) se establece en dos, puesto que, realizando

diferentes pruebas observamos que aumentando el número de épocas en más de dos no disminuye el error cuadrático medio (la función de coste con la que validamos el entrenamiento de la red), con lo que se puede concluir que se podría estar sobrentrenando la red.

```
Train on 37967 samples
Epoch 1/2
37967/37967 [==============================] - 310s 8ms/sample - loss: 0.0045
Epoch 2/2
37967/37967 [==============================] - 315s 8ms/sample - loss: 0.0035
<tensorflow.python.keras.callbacks.History at 0x1b1aa36d940>
```

Figura 5.3: Salida del entrenamiento en Python obtenido en jupyter

Para la validación del modelo realizamos la predicción tanto del conjunto de entrenamiento como el de test. Posteriormente calculamos el error cuadrático medio de la predicción de los dos conjuntos obteniendo los siguientes resultados:

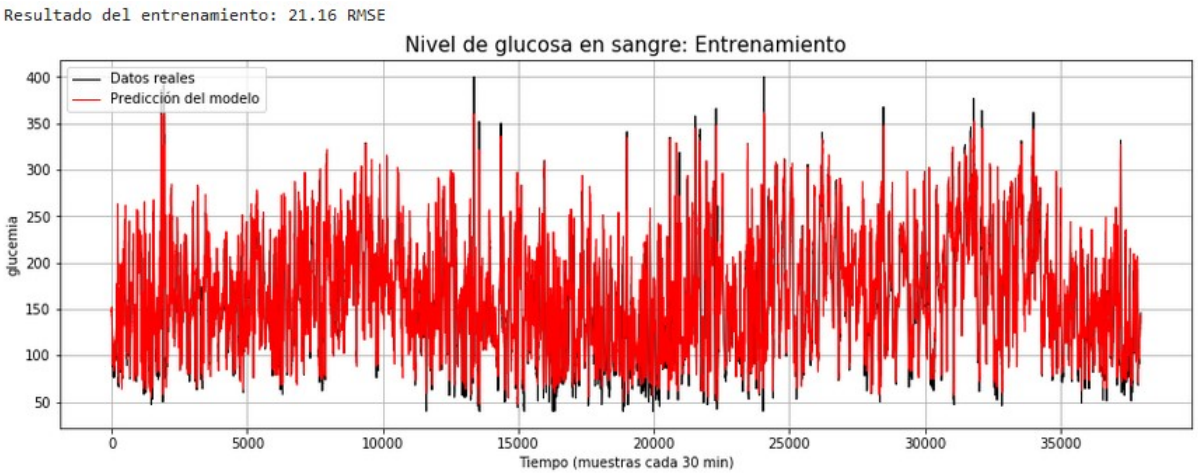

Figura 5.4: Gráfica que muestra el resultado del nivel de glucosa en sangre durante el test de entrenamiento obtenida en jupyter.

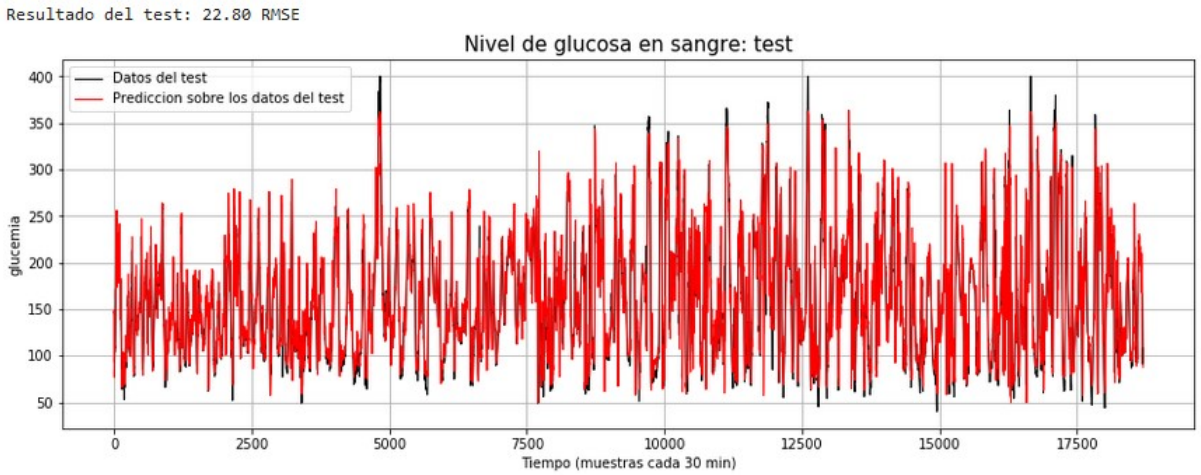

Figura 5.5: Gráfica que muestra el resultado del nivel de glucosa en sangre durante el test de evalución obtenida en jupyter.

Por último, extraemos los pesos del modelo de Keras propuesto, almacenándolos en ficheros para su posterior carga en el modelo de la red HwNN. Para ello hacemos uso de la librería Pandas con el fin de guardar los pesos en ficheros «.csv». Cabe destacar que la librería Keras clasifica los pesos de una capa LSTM en tres tipos: los pesos bias (b), los pesos de entrada (W) y los pesos de estado (U). Nosotros mantenemos esta misma clasificación almacenando cada tipo en un fichero distinto, para posteriormente llevar a cabo la carga de estos en nuestro modelo LSTM HwNN de una forma más sencilla.

Modelo LSTM HwNN

La implementación del modelo HwNN tiene como finalidad dos objetivos. El primero es realizar un estudio sobre como funciona una red LSTM, implementando los métodos necesarios para que sea capaz de predecir el nivel de glucosa en sangre. Cabe destacar que este modelo no es entrenable, la idea es incorporar los pesos ya entrenados de un módelo Keras, y llegar a ser capaces de replicar esa misma red obteniendo los mismos resultados en las predicciones. Con este objetivo confirmamos que es posible replicar un modelo implementado con Keras en un diseño hardware. El segundo es la generación de los ficheros «.coe», que contienen cada uno de los pesos del modelo ya organizados y adaptados para la inserción en las memorias ROM del diseño hardware. El tratamiento de los pesos se basa en convertirlos a punto fijo, con el formato y la longitud correspondientes al que utilizaremos en nuestro diseño hardware.

El modelo HwNN esta implementado con una estructura de datos a la cual denominamos Hw_NN que representa una red neuronal, a su vez, se conforma de otra estructura llamada LSTM que representa las capas LSTM. Dentro de Hw_NN se implementan las funciones necesarias para llevar a cabo el cálculo de una predicción, incluyendo la carga de los pesos entrenados por Keras.

A continuación, explicamos como se lleva a cabo la creación del modelo HwNN a partir de un modelo de Keras:

El primer paso es crear un nuevo objeto HwNN, instanciándolo con dos argumentos de entrada, el primero corresponde al nombre que se le quiera dar al objeto y el segundo es el número de capas LSTM de las que se conforma la red, cabe destacar que el número de capas LSTM pasado como argumento debe coincidir con el modelo creado con Keras que se desea copiar. El segundo paso tiene como objetivo clonar el modelo de Keras en el objeto HwNN instanciado previamente, llamando a la función load_Keras_Layers() con dos argumentos, el primero corresponde el modelo Keras a copiar y el segundo el HwNN.

1. Declaración del modelo HwNN Hw_model:

```
Hw_model = hw.Hw_NN(name=Hw_NN_Modelname+"_", nLayers = 2)
```

2. Copia de la red Keras "model" en "Hw_model":

```
load_Keras_Layers(model, Hw_model)
```

Figura 5.6: Declaración del modelo HwNN

La función load_Keras_Layers() tiene como cometido leer los ficheros donde están almacenados los pesos de cada una de las capas LSTM que conforman la red de Keras, replicar cada capa de Keras al modelo HwNN y finalmente llamar a la función load_weights(), que se encarga de introducir los pesos a cada capa de la red HwNN de la forma apropiada para que la red sea capaz de realizar las predicciones correctamente.

```
def load_Keras_Layers(model, Hw_model):
    i = 0
    for layer in model.layers:
        if "LSTM" in str(layer.name):
            # Cargamos los pesos extraidos previamente de Keras, en tres variables
            w = read_csv(Keras_path+Modelname+'_'+layer.name+'_W.csv',  engine='python')
            u = read_csv(Keras_path+Modelname+'_'+layer.name+'_U.csv',  engine='python')
            b = read_csv(Keras_path+Modelname+'_'+layer.name+'_b.csv',  engine='python')
            # Casteamos los dataframes de panda a numpy
            w, u, b = w.to_numpy(), u.to_numpy(), b.to_numpy()

            # Calculamos el número de estados ocultos (hunits) de la capa
            hunits= int(len(b)/4)
            # Calculamos el número de Características (nfeatures) de la entrada de la capa
            nfeatures=int((w.size/4)/hunits)

            # Añadimos la capa correspondiente de Keras al modelo Hw:
            Hw_model.addLSTMLayer(name=layer.name, hunits=hunits, nfeatures=nfeatures)

            # Redimensionamos los pesos de la forma apropiada para el modelo Hw (filas, columnas).
            w.shape = (nfeatures, hunits*4)
            u.shape = (hunits, hunits*4)
            b.shape = (hunits*4, )
            # Cargamos los pesos (w, u, b) en la capa del modelo Hw:
            Hw_model.LSTM[i].load_weights(w, u, b)
            i = i + 1
```

Figura 5.7: Función Load Keras Layer

La función load_weights del modelo HwNN se encarga de clasificar cada matriz de pesos en la puerta LSTM a la que corresponde, es decir, estos pesos son agrupados según a la puerta a la que pertenecen dentro de una matriz de 3 dimensiones, donde la primera dimensión de estas representa cada una de las puertas LSTM: puerta de entrada, de olvido, nuevo candidato y la de salida. Por lo tanto, nos quedan unas matrices con las siguientes dimensiones:

$$W = [\text{nº de puertas LSTM, nº de características, nº de estados ocultos}] \quad (5.3)$$

$$U = [\text{nº de puertas LSTM, nº de estados ocultos, nº de estados ocultos}] \quad (5.4)$$

$$b = [\text{nº de puertas LSTM, 1, nº de estados ocultos}] \quad (5.5)$$

```
def load_weights(self, W, U, b):
    # Establecemos las dimensiones de las matrices de pesos de la capa LSTM
    self.W = np.zeros((4, self.nfeatures, self.hunits))
    self.U = np.zeros((4, self.hunits, self.hunits))
    self.b = np.zeros((4, 1, self.hunits))
    self.W, self.U, self.b = self.W.astype('float16'), self.U.astype('float16'), self.b.astype('float16')
    # Asignamos a cada puerta LSTM sus pesos correspondientes
    for i1,i2 in enumerate(range(0,len(b),self.hunits)):
        self.W[i1] = w[:,i2:i2+self.hunits]
        self.U[i1] = u[:,i2:i2+self.hunits]
        self.b[i1] = b[i2:i2+self.hunits].reshape(1, self.hunits)
```

Figura 5.8: Función load weight

Una vez terminado todo este proceso, y para poder confirmar que ya tenemos una réplica exacta al modelo de Keras realizamos varias pruebas, estas consisten en comprobar que tanto el modelo de Keras como el HwNN predicen los mismos resultados dados unos datos de entrada. Para realizar estas pruebas se implementó una función que realizaba el cálculo de una predicción de una celda LSTM, la cual podemos ver en la siguiente figura:

```
def propagate(self, xt):
        # Calculamos las funciones de las puertas LSTM para la entrada xt
        i = sigmoid((xt.dot(self.W[0]) + self.h_t.dot(self.U[0]) + self.b[0]))
        f = sigmoid((xt.dot(self.W[1]) + self.h_t.dot(self.U[1]) + self.b[1]))
        nc = np.tanh((xt.dot(self.W[2]) + self.h_t.dot(self.U[2]) + self.b[2]))
        o = sigmoid((xt.dot(self.W[3]) + self.h_t.dot(self.U[3]) + self.b[3]))
        # Calculamos los estados Cell State y Hidden State
        self.c_t = i*nc + f*self.c_t
        self.h_t = o*np.tanh(self.c_t)
        return(self.h_t,self.c_t)
```

Figura 5.9: Función propagate

Finalmente cuando las pruebas que la red HwNN realiza las predicciones de forma correcta, nos disponemos a generar los ficheros «.coe», que contendrán los pesos en un formato adecuado, para la introducción de estos en las memorias Xilinx de nuestro diseño. Implementamos una función a la que llamamos COE_File_parser, en la cual se distinguen las dos tareas a llevar a cabo, la primera es representar los pesos de la manera adecuada,

es decir, representarlos en el formato con el que trabajará nuestro diseño hardware y la segunda es generar los ficheros y escribir en ellos los pesos ya con el formato pertinente. Sobre la representación de los datos, cabe destacar que se hace uso de la librería PyPhix para dar el formato de punto fijo con el que trabaja nuestro diseño y se realiza una conversión a binario de los pesos. La función genera por cada celda LSTM un total de 12 ficheros «.coe», que corresponden a la distinción de los tres tipos de pesos que contiene cada una de las puertas LSTM de la celda, que son cuatro. A continuación, se muestra en las figuras la función mencionada:

```
# Generador de archivos COE
def COE_File_parser(Weight, name, w_name, path):
    fileName = name                                  # Nombre del fichero
    ACCEPT_RADIX = '2'                               # base en la que se representan los datos
    c_data_width = 16                                # longitud en bits de los datos
    c_data_decimal = 12                              # longitud de la parte decimal de los datos
    c_data_int =3                                    # longitud de la parte entera de los datos
    fmt = fix.FixFmt(True, c_data_int, c_data_decimal) # Creación del formato de punto fijo
    fmt2= fix.FixFmt(True, c_data_width, 0)
    base = fix.FixNum((2**c_data_decimal), fmt2)
    gates = ["_i","_f","_c","_o"]
    wfx = fix.FixNum(Weight, fmt)              # Se establece el formato de punto fijo a los datos
    for i in range (4):
        w_file = path+fileName+gates[i]+w_name+".coe"
        try:
            fp = open(w_file, 'w')             # Creación del fichero COE
        except IOError:
            print("Error: Cannot open "+w_file+" for write\n")
            exit()
        fp.write("""; COE FILE TFG2019-20Hw_NN:
; NAME FILE:  {}\n""".format(fileName+gates[i]+w_name))
        fp.write('memory_initialization_radix={};\n'.format(ACCEPT_RADIX))
        fp.write('memory_initialization_vector=\n')
        for j in range ((Weight.shape[1])):
            for k in range ( Weight.shape[2]):
                # Conversión del dato a binario en puto fijo
                tmp_w = binConverter(int((wfx[i][j][k].value)*(base.value)), c_data_width)
                # Escritura del dato en el fichero con el formato apropiado
                fp.write("{}\n".format(tmp_w))
        fp.write(";")
        fp.close()
```

Figura 5.10: Funcion_coe_file_parser

```
#Convertirdor a binario
def binConverter(n, bits):
       s = bin(n & int("1"*bits, 2))[2:]
       return ("{0:0>%s}" % (bits)).format(s)
```

Figura 5.11: Función binConverter

Capítulo 6

Implementación hardware

6.1. Aspectos generales del diseño

La red neuronal implementada en hardware se basa en el modelo realizado en Python explicado en la sección anterior. Es decir, la red contiene tres capas: una capa de entrada, una capa oculta y una capa de salida. La capa oculta se parametriza con tres estados ocultos, mientras que la de salida solo contiene un estado.

La entrada de datos al sistema se conforma por muestras de 15 datos que a su vez se dividen en 5 vectores de 3 elementos cada uno. En cada paso de tiempo se recibe por el puerto de entrada X un vector compuesto por un dato de glucosa, hidratos de carbono e insulina los cuales se transmiten elemento a elemento. Cuando la red recibe un vector se le debe indicar que comience la ejecución, se le podrá transmitir el siguiente vector cuando esta notifique la finalización de la lectura del anterior. Este procedimiento se debe realizar hasta la transmisión de toda la muestra.

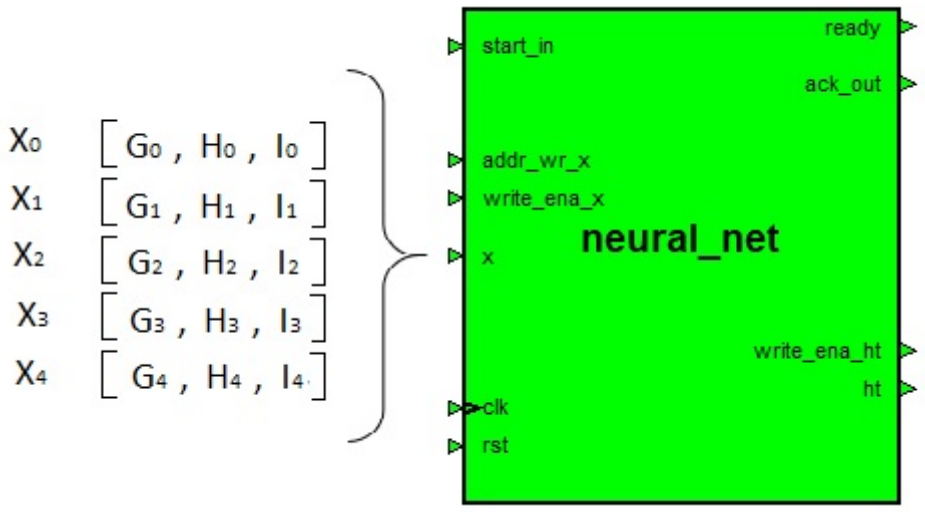

Figura 6.1: Ilustración de la entrada de datos al sistema

Inicialización de la red neuronal

Para inicializar el sistema se declara un conjunto de constantes que determinan la arquitectura de la red. Además se ha implementado en el diseño una función de carga que se utiliza para almacenar en las memorias ROM los pesos extraídos del modelo implementado en Python.

Control del sistema

El sistema tiene un control distribuido y jerárquico, por lo tanto cada módulo tiene su propia unidad de control que se comunica con sus submódulos mediante las señales de sincronización. El protocolo de comunicación implementado es tal que un módulo consumidor de datos espera a que el módulo generador asociado le indique que los datos están disponibles para empezar sus operaciones. cuando el módulo consumidor ha leído todos los datos, se lo comunica al módulo generador para que este pueda empezar de nuevo sus operaciones. Este protocolo se utiliza tanto para módulos del mismo nivel, como para módulos de diferentes niveles. Cabe destacar que todos los módulos están conectados a dos memorias, una conectada a su entrada y otra a su salida, haciendo estas de 'glue' entre los diferentes módulos. La memoria de entrada almacena los datos fuente y la memoria de salida almacena los resultados de la operación.

Si suponemos una cadena de módulos conectados por memorias glue, podemos definir un módulo generador como aquel que escribe los resultados en una memoria intermedia, mientras que podemos definir su módulo consumidor como aquel que lee los datos de esta memoria. Lógicamente, el mismo módulo es de manera alterna consumidor y generador. Visto esto, el protocolo de comunicación es el siguiente. Un módulo consumidor espera a la señal start_in enviada por su módulo generador. Esta señal indica que ya están disponibles los nuevos datos en su memoria de entrada. Cuando el modulo consumidor acaba de leer los datos de la memoria se lo comunica a su generador mediante la señal ack_out. Cuando el generador la recibe ya puede empezar a producir datos otra vez. Ahora el módulo consumidor pasa a ser generador puesto que al ejecutar su operación escribe los resultados en la memoria de salida. Cuando acaba la operación y todos los datos están escritos envía la señal start_out a su módulo consumidor (el siguiente módulo) para indicarle que tiene disponibles nuevos datos. Finalmente el módulo espera la llegada de la señal ack_in, enviada por su módulo sucesor para indicarle que ya ha leído los datos y puede reanudar la operación. Ahora el módulo vuelve a esperar a la señal start_in de su módulo generador, cerrando así el ciclo. Darse cuenta que la señal start_in que recibe un consumidor es la señal start_out que envía un generador. De la misma manera, la señal ack_out que envía un consumidor es la señal ack_in que recibe un generador.

En resumen:

- start_in: indica al módulo que ya tiene acceso a la nueva información para llevar a cabo su cometido.
- ack_out: el módulo informa a su generador que ya ha leído todos los datos, por lo que estos pueden ser sobre escritos.

- start_out: el módulo indica a su consumidor que tiene acceso a nueva información.
- ack_in: el consumidor informa al módulo que ya puede volver a escribir nueva información en su memoria de salida.

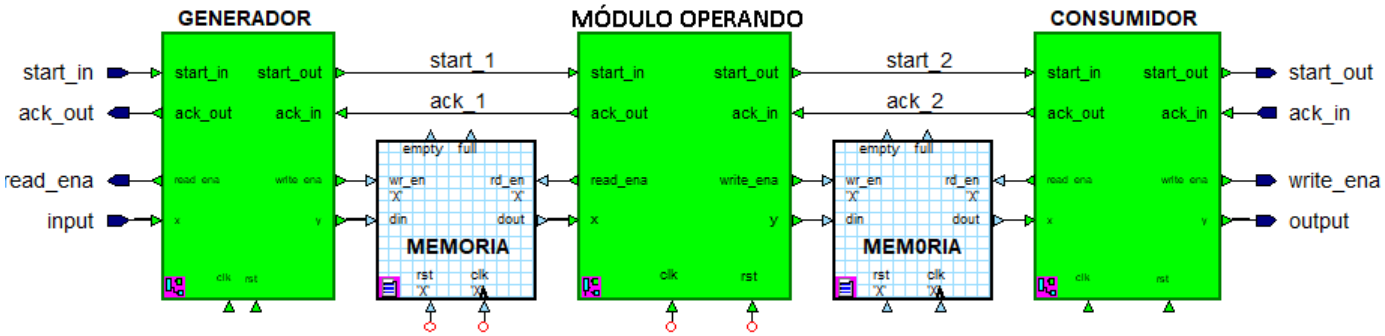

Figura 6.2: Ilustración de una posible estructura del sistema

Representación y almacenamiento de los datos

El sistema trabaja con datos en punto fijo de una longitud de 32 bits. Gracias al paquete fixed_pkg de la librería ieee_proposed se puede representar este tipo de datos y realizar sus operaciones aritméticas con gran facilidad. En concreto se utilizamos el tipo de dato sfixed caracterizado por reservar el bit más significativo para la representación del valor de signo.

La notación Q es un formato de representación de números binarios en punto fijo. En la nomenclatura de este formato el símbolo mQn representa un número en punto fijo donde m es el número de bits de la parte entera en la que se reserva el bit más significativo para la representación del valor de signo y n de la parte fraccionaria. Dicho todo esto, de los 32 bits que constituyen un dato del sistema, se determinan 25 para la parte decimal y 6 más 1 para la parte entera y la representación del signo, lo que corresponde en notación Q a 6Q25. Para representar esta notación Q en el tipo sfixed sería:

signal a: sfixed(6 downto -25);

donde el índice negativo -25 representa al número de bits de la parte fraccionaria, que va desde el bit -1 hasta el -25 y el índice de la izquierda representa a la parte entera, que va desde el 6 hasta el 0 haciendo un total de 7 bits incluyendo el bit reservado para el signo.

Respecto a cómo se realizan las operaciones aritméticas con el paquete fixed_pkg, éste se ha diseñado para que no exista la posibilidad de desbordamiento en sus operaciones. En la siguiente figura 6.3 se muestran los tamaños que se deben establecer para que se lleven a cabo:

Operation	Result Range
A + B	Max(A'left, B'left)+1 downto Min(A'right, B'right)
A - B	Max(A'left, B'left)+1 downto Min(A'right, B'right)
A * B	A'left + B'left+1 downto A'right + B'right
A rem B	Min(A'left, B'left) downto Min(A'right, B'right)
Signed /	A'left - B'right+1 downto A'right - B'left
Signed A mod B	Min(A'left, B'left) downto Min(A'right, B'right)
Signed Reciprocal(A)	-A'right downto -A'left-1
Abs(A)	A'left +1 downto A'right
- A	A'left +1 downto A'right
Unsigned /	A'left - B'right downto A'right - B'left -1
Unsigned A mod B	B'left downto Min(A'right, B'right)
Unsigned Reciprocal(A)	-A'right +1 downto -A'left

Figura 6.3: Tabla de operaciones en punto fijo [31]

Después de calcular el resultado de la operación correspondiente se hace uso de unas funciones del paquete denominadas resize que tienen como objetivo reconvertir el dato calculado en el formato pasado como argumento, es decir, modifica los tamaños de la parte entera y de la fraccionaria, con lo que podemos volver a dar a los datos calculados con este paquete el formato que trabaja el sistema.

En cuanto al almacenamiento de los datos en el sistema se debe introducir las matrices de pesos que se implementan como memorias lineales en las memorias ROM. Para acceder a un elemento de la matriz situado en las coordenadas (x,y) se debe aplicar la siguiente expresión:

$$\text{Dirección de memoria} = (x-1)\cdot C + (y-1)\,. \tag{6.1}$$

donde x e y son las coordenadas de la fila y la columna respectivamente y C el número de columnas que tiene la matriz.

6.2. Módulos Comunes

El proyecto está dividido en librerías que agrupan los módulos que implementan funcionalidades relacionadas con el diseño. En esta sección vamos a explicar alguno de los elementos de la librería Common, que incluye módulos de propósito general utilizados por todo el diseño, como por ejemplo sumadores, contadores o registros. También explicamos los IP de Xilinx que se incluyen en la librería IP Core.

En la librería Common se encuentra el paquete constants donde se definen las constates del sistema y el paquete functional donde se definen los tipos de datos y algunas funciones

del sistema. A continuación, en las tablas 6.1, 6.2 y 6.3 se visualiza el contenido de estos paquetes:

Constantes generales

Nombre	Valor	Descripción
c_data_width	32 bits	Ancho de los datos del sistema.
c_data_decimal_width	25 bits	Longitud de la parte decimal.
c_data_integer_width	7 bits	Longitud de la parte entera.
n_hidden_units	3	Número de estados ocultos de hidden layer.
n_input_features	3	Número de características de la entrada.

Tabla 6.1: Constantes generales

Constantes para los datos de tipo sfixed

Nombre	Valor	Descripción
c_sfixed_width	c_data_width	Ancho de los datos de tipo sfixed.
c_sfixed_high	6	Longitud de la parte entera de los datos tipo sfixed.
c_sfixed_low	-25	Longitud de la parte decimal de los datos tipo sfixed.

Tabla 6.2: Constantes de datos sfixed

Otras constantes

Nombre	Descripción
c_model_path	Ruta del modelo realizado en Python.
c_COE_path	Ruta de los ficheros COE del modelo.

Tabla 6.3: Otras constantes

Funciones

En el paquete functional se implementan varias funciones donde la mayoría de ellas tienen el propósito de reformatear los resultados de las operaciones aritmética implementadas con la librería fixed_pkg a la representación punto fijo del sistema. Log2 es otra función que calcula el logaritmo en base 2 del numero pasado como argumento de entrada, esta es utilizada mayoritariamente para definir los tamaños de los buses de direcciones del sistema.

También cabe destacar que en este paquete se encuentra implementada la función de carga de los pesos de la red, llamada read_weights, a la cual se le pasan dos argumentos de entrada: el número de elementos de la matriz y el nombre (o ruta) de la matriz de pesos a leer. Esta función devuelve un array de tipo sfixed que es introducido en la memoria ROM correspondiente del sistema.

Módulos de la librería Common

Como se comenta al principio de la sección los módulos de esta librería son de propósito general, encargados de tareas especificas y en su conjunto conformando módulos más complejos. No se verán todos ellos en detalle puesto que son muchos y la mayoría son más que conocidos por cualquier lector interesado en el campo tratado. Pero sí se expondrán a continuación algunos un poco más especiales:

- comparator_sfxd: es un comparador de datos del tipo sfixed. Proporciona una salida de out_cmp = '1' si el valor absoluto del dato A de entrada es mayor o igual al dato de entrada B u out_cmp = '0' en caso contrario. Se compone de una interfaz muy sencilla según se describe en la tabla 6.4.

Puerto	Tamaño	Tipo	Descripción
A	c_data_width	Entrada	Dato de entrada.
B	c_data_width	Entrada	Dato de entrada.
out_cmp	1 bit	Salida	Valor del bit de salida.

Tabla 6.4: Comparator_sfxd

- converter_nqm: es un conversor que codifica los datos de entrada con formato en punto fijo Q29 al formato nQm con la que trabaja el sistema. La n viene dada por el valor de la constante c_data_integer_width que indica la longitud en bits de la parte entera y la m por el valor de la constante c_data_decimal_widtht que indica el tamaño de la parte fraccionaria del dato. Proporciona una interfaz muy sencilla según se describe en la tabla 6.5.

Puerto	Tamaño	Tipo	Descripción
input	c_data_width	Entrada	Dato de entrada en formato 1Q29.
output	c_data_width	Salida	Dato codificado en la forma 6Q25.

Tabla 6.5: Converter_nqm

- converter_2qn: es un conversor que codifica los datos en punto fijo con formato nQm con los que trabaja el sistema a la forma 2Qn. En la tabla tabla 6.6 se puede ver las señales de la interfaz, su tipo y sus formatos.

Puerto	Tamaño	Tipo	Descripción
input	c_data_width	Entrada	Dato de entrada en formato 6Q25.
output	c_data_width	Salida	Dato codificado en la forma 2Q29.

Tabla 6.6: Converter_2qn

- sign_converter: su función es cambiar el bit de signo al dato de tipo sfixed pasado por el puerto de entrada. En la tabla 6.7 se puede ver las señales de la interfaz, su tipo y sus formatos.

Puerto	**Tamaño**	**Tipo**	**Descripción**
input	c_data_width	Entrada	Dato de entrada.
output	c_data_width	Salida	Dato con el bit de signo cambiado.

Tabla 6.7: Sign_converter

- counter_mod: es un contador con carga en paralelo, salida modulada y reset síncrono. A este contador se le puede configurar un valor inicial de cuenta y el valor a comparar por la cuenta actual. También tiene como característica la modulación de su salida (la cuenta actual). El valor del módulo a calcular se establece mediante el genérico G_MOD. En la tabla 6.8 se puede ver las señales de la interfaz, su tipo y sus formatos.

Puerto	**Tamaño**	**Tipo**	**Descripción**
G_DEPTH	-	Genérico	Tamaño del bus de salida.
G_MOD	-	Genérico	Valor del módulo que le aplica a la salida.
G_MAX	-	Genérico	Tamaño máximo de cuenta.
clk	1 bit	Entrada	Reloj del sistema
rst	1 bit	Entrada	Reset del sistema.
cnt_ena	1 bit	Entrada	Capacitación de cuenta.
load	1 bit	Entrada	señal de carga de cuenta inicial.
cmp_count	log2(G_MAX)	Entrada	Valor de fin de cuenta.
init_value	log2(G_MAX)	Entrada	Valor de cuenta inicial.
count	log2(G_DEPTH)	Salida	Valor de cuenta actual.
end_count	1 bit	Salida	Señal de fin de cuenta.

Tabla 6.8: Counter_mod

- common_cu: se trata de una unidad de control implementada como una maquina de estados finitos de tipo Moore con reset síncrono. Esta unidad de control es utilizada en varios módulos del diseño y se caracteriza por controlar la comunicación interna entre 2 o más submódulos permitiendo la comunicación con el exterior basada en el protocolo descrito en la sección anterior. En la tabla 6.9 se puede ver las señales de la interfaz, su tipo y sus formatos. La siguiente figura ilustra un ejemplo de un módulo con el componente Common CU:

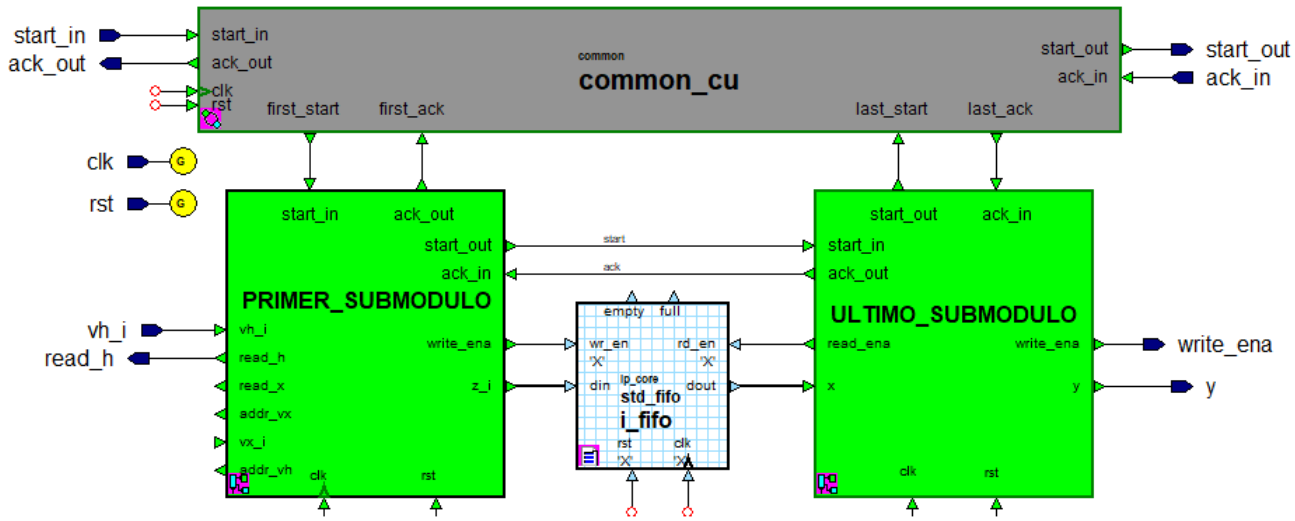

Figura 6.4: Estructura de Common CU

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Common CU que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Common CU informa al módulo generador ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Common CU informa al módulo consumidor que ya ha escrito su resultado en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Common CU que ya puede escribir en la memoria de salida.
first_start	1 bit	Salida	Common CU informa al primer submódulo que comience su ejecución.
first_ack	1 bit	Entrada	El primer submódulo indica a Common CU que ya ha leído todos los datos de la memoria de entrada.
last_start	1 bit	Entrada	El ultimo submódulo indica a Common CU que ya ha escrito su resultado en la memoria de salida.
last_ack	1 bit	Salida	Common CU informa al ultimo submódulo que ya puede escribir en memoria la memoria de salida.

Tabla 6.9: Interfaz de Common CU

La siguiente figura 6.5 ilustra el diagrama de transición estados.

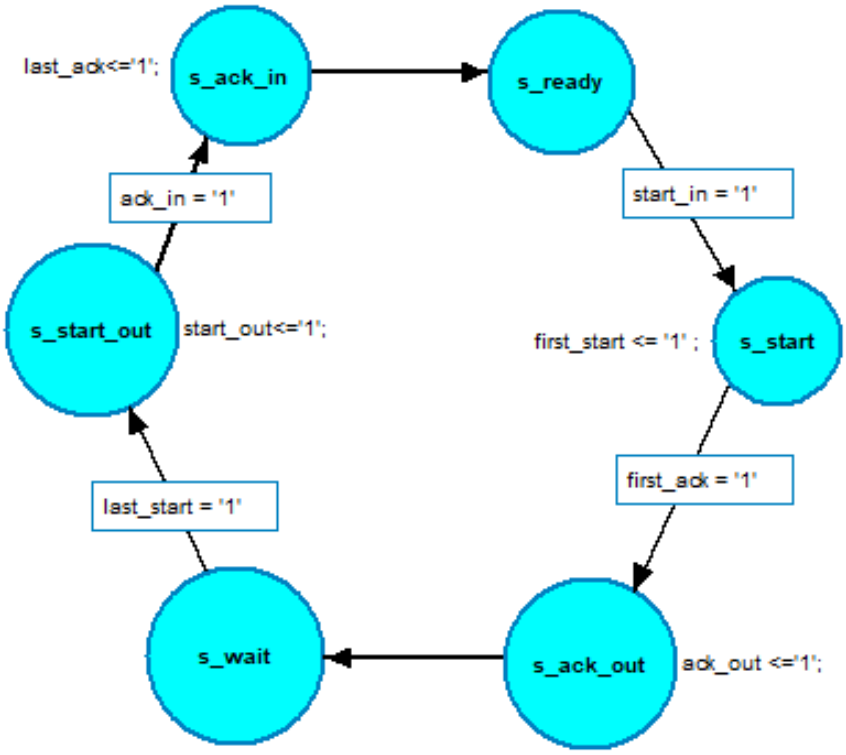

Figura 6.5: Diagrama de estados de Common CU

- Estado ready: marca la disponibilidad del módulo para poder realizar una nueva ejecución. Se espera la llegada de la señal start_in, que envia el módulo generador, para hacer una transición al estado start.
- Estado start: marca el comienzo de ejecución. Common CU activa la señal first_start informando al primer submódulo de que comience su ejecución. Espera en este estado hasta que llega la señal first_ack, que indica que el primer submódulo ya ha leído los datos de su memoria de entrada.
- Estado ack_out: Common CU genera la señal ack_out que indica al módulo generador que ya se han leído los datos de la memoria de entrada. Seguidamente se pasa al estado wait.
- Estado wait: espera a que llegue la señal last_start, que indica que el último submódulo que lo compone ya ha escrito en la memoria de salida los datos generados. El siguiente estado es el start_out.
- Estado start_out: Common CU genera la señal start_out que informa al módulo consumidor de la disponibilidad de nuevos datos en la memoria de salida. Se espera en este estado hasta que el módulo consumidor envíe la señal ack_in indicando que los datos de su memoria de salida ya han sido leídos y por tanto ya pueden ser sobre escritos. Después de la llegada de esta señal se pasa al estado ack_in.

- Estado ack_in: Common CU activa la señal last_ack informando al último submódulo de que ya puede volver a escribir nueva información en su memoria de salida. Después se hace la transición al estado ready terminando así el ciclo de ejecución.

En la siguiente tabla 6.10 se citan los demás módulos de esta librería:

Nombre	**Descripción**
adder	realiza la suma de dos sumandos de tipo sfixed.
counter	contador ascendente con carga en paralelo.
divisor	realiza la división de dos datos de tipo sfixed.
multiplexor2a1	multiplexor de tipo std_logic_vector de 2 entradas 1 salida. Se determina el tamaño de sus puertos mediante un genérico.
multiplexor4a1	multiplexor de tipo std_logic_vector de 4 entradas 1 salida. Se determina el tamaño de sus puertos mediante un genérico.
multiplier	realiza el producto de dos datos de tipo sfixed.
registr	registro que determina su tamaño mediante un genérico.
ROM	memoria ROM.

Tabla 6.10: Módulos comunes

IPs de Xilinx

A continuación, listamos los módulos de Xilinx utilizados en el diseño acompañados de una breve descripción.

Nombre	**Descripción**
CORDIC	Componente que calcula funciones hiperbólicas.
std_fifo	Se trata de una cola de memoria que implementa el algoritmo first in first out.
RAM_sdp	Memoria RAM distribuida simple dual port.
RAM_16x32	Memoria RAM distribuida single port.

Tabla 6.11: IPs de Xilinx

6.3. Jerarquía del diseño

En esta sección se explica de manera jerárquica el diseño de la red neuronal.

6.3.1. Neural net

El módulo Neuronal net es el nivel superior de la jerarquía e implementa una red neuronal cuyo objetivo es realizar predicciones de la concentración de la glucosa en sangre. Esta red se compone de tres capas, input layer, hidden layer, y output layer.la primera capa es una memoria RAM en la que se almacenan los datos suministrados por el bus de entrada X. El layer hidden y el output se comunican a través de una memoria RAM de manera que primera escribe sus resultados en la memoria mientras que la segunda los lee. La predicción son los valores que salen por el bus ht del output. A continuación, se expone su estructura 6.6 e interfaz 6.12:

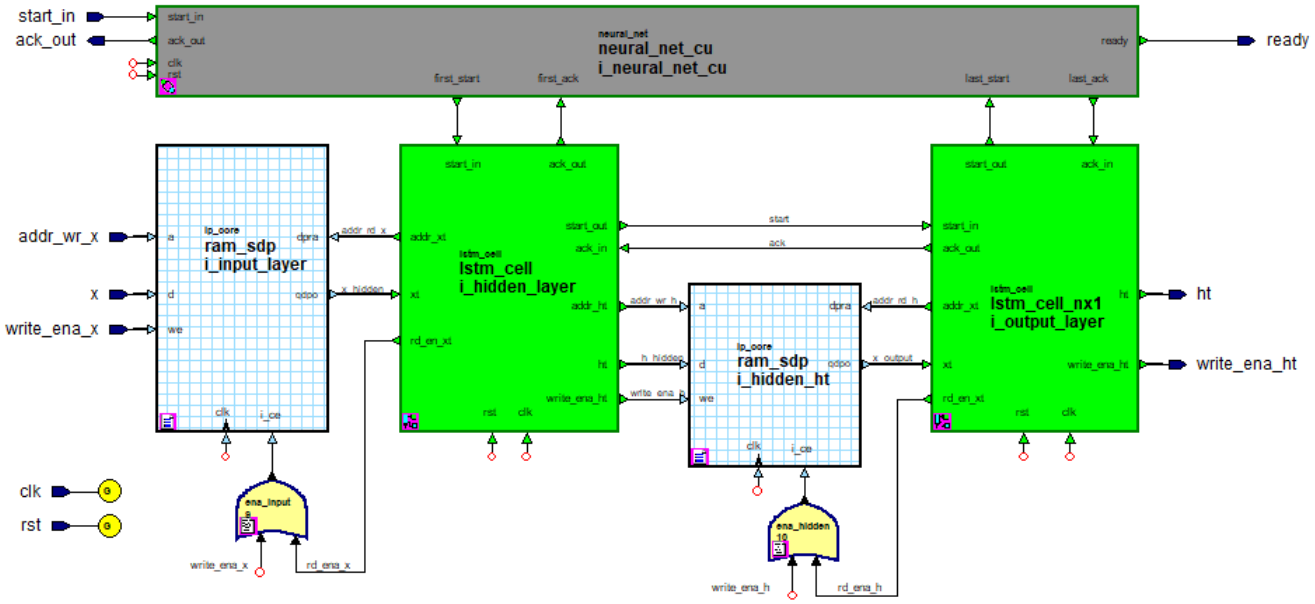

Figura 6.6: Diagrama de bloques de Neural net

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Neural net que ya tiene los datos disponibles en Input layer y que por lo tanto puede realizar la predicción.
ack_out	1 bit	Salida	Neural net informa al módulo generador que ya ha leído todos los datos de Input layer.
ready	1 bit	Salida	Neural net informa al módulo generador que esta listo para una nueva ejecución.
write_ena_x	1 bit	Entrada	Capacitación de memoria Input layer.
addr_wr_x	4 bits	Entrada	Dirección de memoria del dato x.
x	c_data_width	Entrada	Dato x.
write_ena_ht	1 bit	Salida	Capacitación de memoria de ht.
ht	c_data_width	Salida	Dato ht.

Tabla 6.12: Interfaz neural net

Neural net está formado por los siguientes componentes descritos brevemente a continuación:

- Input_layer: es la capa de entrada representada por una memoria RAM_sdp donde se almacena el número de datos que conforman una muestra de entrada. Se hace uso de una RAM simple dual port ya que esta cuenta con un puerto de direccionamiento lectura y otro de direccionamiento de escritura independientes, con lo que el direccionamiento de escritura es manejado por el módulo generador, y el de lectura por Hidden layer.
- Hidden_layer: es la capa oculta y se trata del módulo LSTM cell que realiza los primeros cálculos sobre las entradas facilitadas por la capa de entrada. Estos cálculos son transmitidos a hidden_ht.
- Hidden_ht: es la memoria RAM_sdp donde se almacenan los resultados de la capa oculta. Se hace uso de una RAM simple dual port ya que esta cuenta con un puerto de direccionamiento lectura y otro de direccionamiento de escritura independientes, con lo que el direccionamiento de escritura es manejado por Hidden layer, y el de lectura por Output layer.
- Output_layer: es la capa de salida y se trata del módulo LSTM cell nx1 que realiza sus cálculos y predicción definitiva sobre los datos facilitados por hidden_ht. Se encarga de propagar los datos por el puerto de salida ht.
- Neural_net_cu: es la unidad de control de Neural net. En la siguiente subsección hablamos en detalle sobre ella.

6.3.1.1. Neural net CU

Es la unidad de control que gestiona la ejecución de Neural net y se encarga de llevar a cabo la comunicación con el exterior mediante las señales start y ack. Su interfaz se detalla la siguiente tabla 6.13:

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Neural net que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Neural net informa al módulo generador que ya ha leído todos los datos de Input layer.
ready	1 bit	Salida	Neural net informa al módulo generador que esta listo para una nueva ejecución.
first_start	1 bit	Salida	Neural net CU informa a Hidden layer que comience su ejecución.
first_ack	1 bit	Entrada	Hidden Layer indica a Neural net CU ya ha leído todos los datos.
last_start	1 bit	Entrada	Output layer indica a Neural net CU que ya ha escrito su resultado en la memoria de salida.
last_ack	1 bit	Salida	Neural net CU informa a Output layer que su resultado ya ha sido leído de la memoria de salida.

Tabla 6.13: Interfaz de Neural net CU

Se ha implementado como una máquina de estados finitos de tipo Moore con reset síncrono y 5 estados. La siguiente figura ilustra el diagrama de transición de estados y posteriormente se describe cada uno de sus estados 6.11.

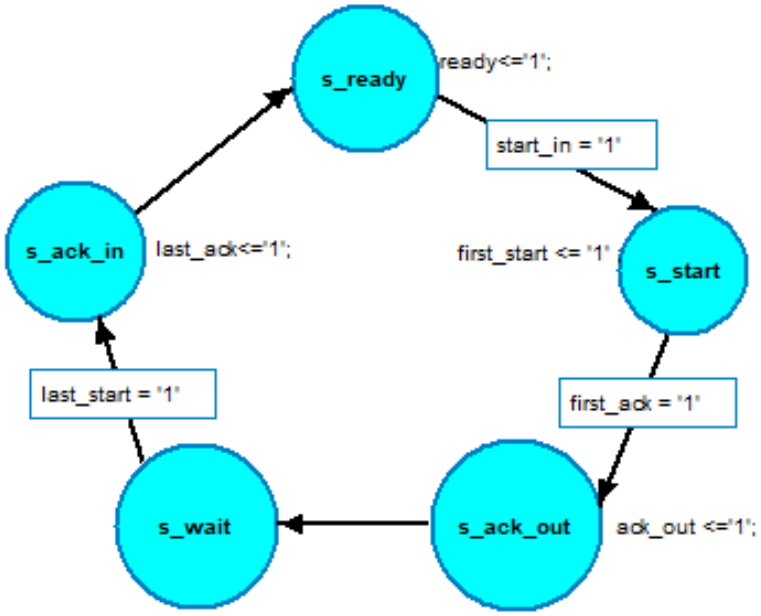

Figura 6.7: Diagrama de transición de estados de Neural net CU

- Estado ready: Neural net está listo para empezar a trabajar y esto lo comunica mediante la señal ready. Se cambia al estado start cuando el módulo generador envié la señal start_in.
- Estado start: marca el comienzo de ejecución. Neural net CU activa la señal first_start informando a Hidden layer que comience su ejecución y se espera a la activación de la señal first_ack que indica que Hidden layer ha leído todos los datos de entrada de Input layer.
- Estado ack_out: Neural net CU informa al módulo generador que ya ha leído todos los datos de Input layer mediante la activación de la señal ack_out. Seguidamente se pasa al estado wait.
- Estado wait: se espera a la llegada de la señal last_start la cual indica que Output layer ya ha escrito en la memoria de salida su resultado. En el momento que llegue dicha señal se hace la transición al estado ack_in.
- Estado ack_in: Neural net CU activa la señal last_ack informando a Output layer la acción de poder volver a escribir nueva información en su memoria de salida. Después se hace la transición al estado ready terminando así el ciclo de ejecución.

6.3.2. LSTM cell

Este módulo implementa una celda LSTM cuya estructura y modo de funcionamiento se explicó en la primera parte de la memoria. Con el objetivo de hacer un diseño más flexible y propiciar su adaptabilidad a diferentes tipos problemas se puede parametrizar mediante una serie de genéricos, el número de estados ocultos y el número de características de la entrada. Las operaciones que lleva a cabo son el cálculo del estado de celda (cell state) y el cálculo del estado oculto (hidden state). La salida es el estado oculto que se propaga al exterior por el puerto ht. A continuación se pueden ver su estructura 6.8 e interfaz 6.14:

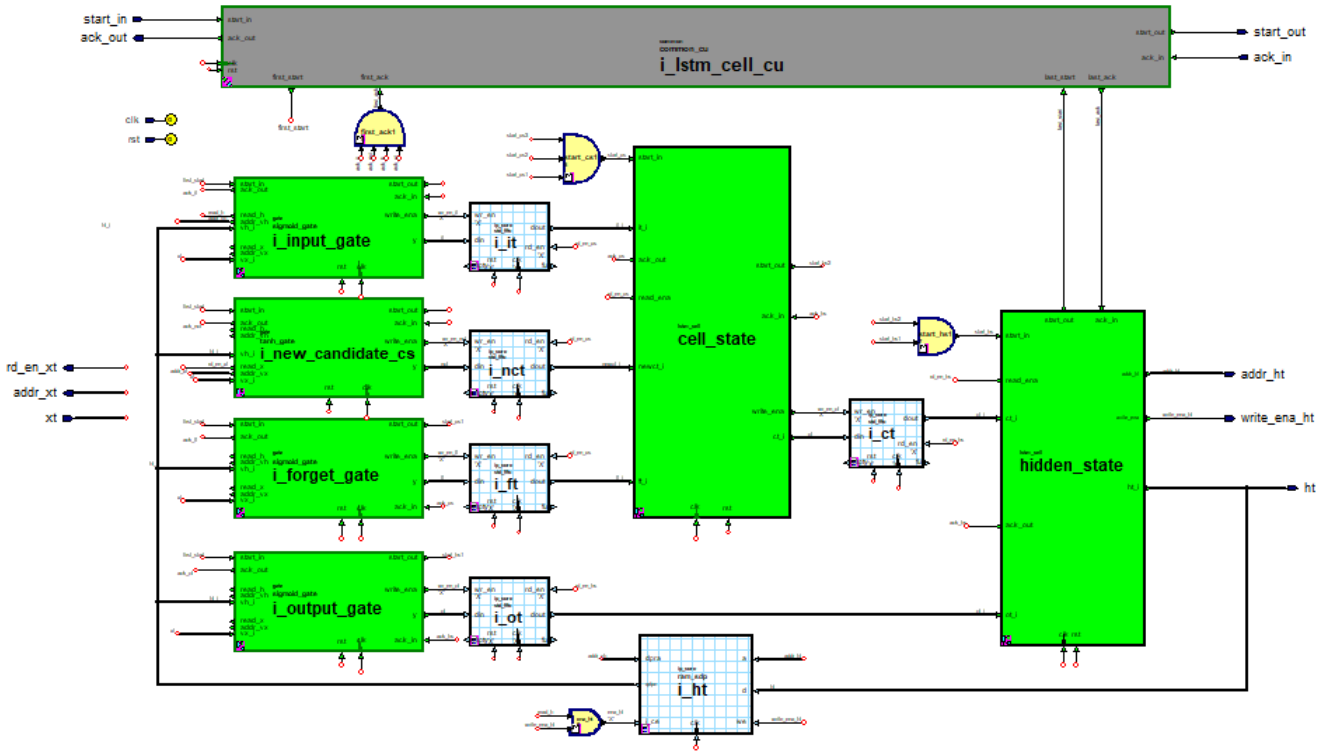

Figura 6.8: Diagrama de bloques de LSTM cell

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a LSTM cell que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	LSTM cell informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	LSTM cell informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a LSTM cell que ya puede escribir en la memoria de salida.
rd_ena_xt	1 bit	Salida	Capacitación de lectura de un dato de la entrada xt.
addr_xt	4 bits	Salida	Dirección de memoria del dato xt.
xt	c_data_width	Entrada	Dato de entrada xt.
write_ena_ht	1 bit	Salida	Capacitación de escritura de un dato ht.
addr_ht	4 bits	Salida	Dirección de memoria del dato ht.
ht	c_data_width	Salida	Dato de salida ht.

Tabla 6.14: Interfaz de LSTM cell

LSTM cell está formado por los siguientes componentes descritos brevemente a continuación:

- Forget gate: se trata del módulo Sigmoid gate y representa la puerta de olvido LSTM. Esta puerta permite el paso de los nuevos datos que superan el umbral de la función de activación sigmoidal para posteriormente guardarlos en el estado de la celda, en definitiva, cuanto mayor sea su valor más impacto tendrá la muestra de entrada sobre los estados de la celda.
- ft: se trata de una std_fifo donde se almacena temporalmente el vector resultado de la puerta de olvido.
- Input gate: se trata del módulo Sigmoid gate y representa la puerta de entrada LSTM que controla qué información nueva entra en los estados de la celda.
- it: se trata de una std_fifo donde se almacena temporalmente el vector resultado de la puerta de entrada.
- Output gate: se trata del módulo Sigmoid gate y en este caso representa a la puerta de salida LSTM que determina cuanto se utiliza en el resultado la información contenida en ella.
- ot: se trata de una std_fifo donde se almacena temporalmente el vector resultado de la puerta de salida.

- New candidate cs: se trata del módulo Tanh gate que representa el nuevo candidato a cell state y se encarga de controlar cuanta información nueva va entrar en la celda.
- nct: se trata de una std_fifo donde se almacena temporalmente el vector resultado del nuevo candidato a cell state.
- Cell state: se trata del módulo Cell state que representa al estado de la celda encargado de calcular el mismo.
- ct: se trata de una std_fifo donde se almacena temporalmente el vector resultado del estado de la celda.
- Hidden State: se trata del módulo Hidden state que representa al estado oculto de la celda encargado de calcularlo y propagarlo por el puerto de salida ht.
- ht: se trata de una memoria RAM_sdp, donde se guarda el vector resultado del estado oculto de la celda. Se hace uso de una RAM simple dual port ya que esta cuenta con un puerto de direccionamiento lectura y otro de direccionamiento de escritura independientes, con lo que el direccionamiento de escritura es manejado por Hidden state, y el de lectura por las puertas LSTM.
- LSTM cell CU: se trata de la unidad de control de LSTM cell que a su vez es el módulo Common CU.

6.3.3. LSTM cell nx1

LSTM cell nx1 también representa una celda LSTM pero con diferente estructura a la explicada el la subsección anterior. El sufijo nx1 significa que se puede establecer un número de características de la entrada variable, pero esta diseñado con solo un estado oculto, lo que da a lugar a que el resultado de sus cálculos sea un escalar. Todos los módulos con el sufijo nx1 expresan lo mismo y son componentes de éste.

La función de este módulo es calcular el estado de la celda (cell state nx1) y el estado oculto (hidden state nx1). Solo el estado oculto es propagado al exterior por el puerto ht. A continuación se pueden ver su estructura 6.9 e interfaz 6.15:

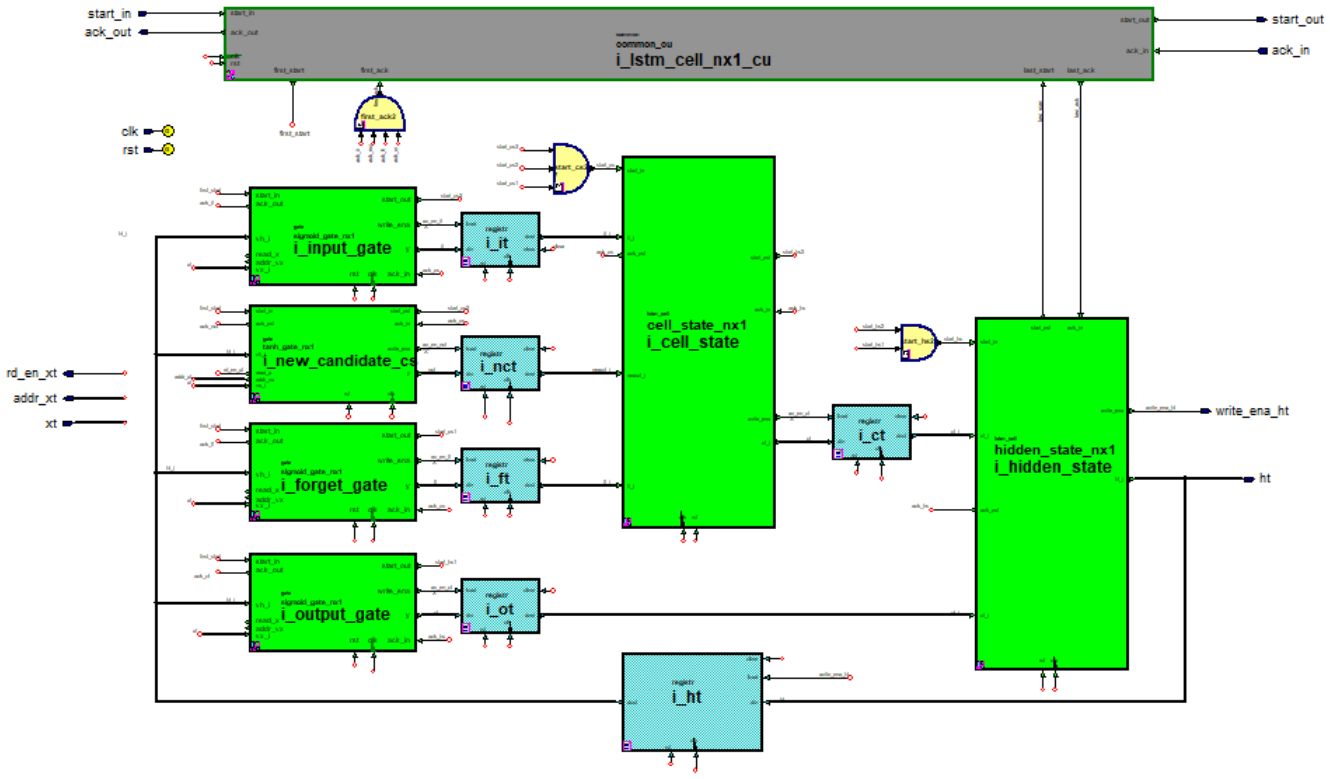

Figura 6.9: Diagrama de bloques de LSTM cell nx1

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a LSTM cell nx1 que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	LSTM cell nx1 informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	LSTM cell nx1 informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a LSTM cell nx1 que ya puede escribir en la memoria de salida.
rd_ena_xt	1 bit	Salida	Capacitación de lectura de un dato de la entrada xt.
addr_xt	4 bits	Salida	Dirección de memoria del dato xt.
xt	c_data_width	Entrada	Dato de entrada xt.
write_ena_ht	1 bit	Salida	Capacitación de escritura del dato ht.
ht	c_data_width	Salida	Dato de salida ht.

Tabla 6.15: Interfaz de LSTM cell nx1

LSTM cell nx1 está formado por los siguientes componentes descritos brevemente a continuación:

- Forget gate: se trata del módulo Sigmoid gate nx1 y representa la puerta de olvido LSTM. Esta puerta permite el paso de los nuevos datos que superan el umbral de función de activación sigmoidal para guardarlos en el estado de la celda. En definitiva, cuanto mayor sea su valor más impacto tendrá la muestra de entrada sobre los estados de la celda.
- ft: se trata de un registro donde se guarda el vector resultado compuesto por un solo elemento correspondiente a la puerta de olvido.
- Input gate: se trata del módulo Sigmoid gate nx1 y representa la puerta de entrada LSTM, ésta controla qué información nueva se almacena en los estados de la celda.
- it: se trata de un registro donde se guarda el vector resultado compuesto por un solo elemento correspondiente a la puerta de entrada.
- Output gate: se trata del módulo Sigmoid gate nx1 y en este caso representa la puerta de salida LSTM la cual determina cuanto se utiliza en el resultado la información contenida en ella.
- ot: se trata de un registro donde se guarda el vector resultado de un solo elemento correspondiente a la puerta de salida.

- New candidate cs: se trata del módulo Tanh gate nx1 que representa el nuevo candidato a cell state y se encarga de controlar cuanta información nueva va entrar en la celda.
- nct: se trata de un registro donde se guarda el vector resultado compuesto por un solo elemento correspondiente al nuevo candidato.
- Cell state: se trata del módulo Cell state nx1 que representa al estado de la celda y se encarga de calcular este.
- ct: se trata de un registro donde se guarda el vector resultado compuesto por un solo elemento correspondiente a estado de la celda.
- Hidden state: se trata de módulo Hidden state nx1 que representa al estado oculto de la celda y se encarga de calcularlo y propagarlo por el puerto de salida ht.
- ht: se trata de un registro donde se guarda el vector resultado compuesto por un solo elemento correspondiente al estado oculto de la celda.
- LSTM cell nx1 CU: se trata de la unidad de control de LSTM cell nx1 que a su vez es módulo Common CU.

6.3.4. Hidden state

Se encarga de calcular el estado oculto del módulo LSTM cell con el siguiente producto escalar:

$$h_t = o_t \cdot \tanh(c_t) \tag{6.2}$$

donde:

- o_t: es el vector resultado de la puerta de salida en el instante t.
- tanh: corresponde a la función tangente hiperbólica.
- c_t: es el vector que representa el estado de la celda (cell state) en el instante t.
- h_t: vector que representa el estado oculto en el instante t.

A continuación se pueden ver su estructura 6.10 e interfaz 6.17:

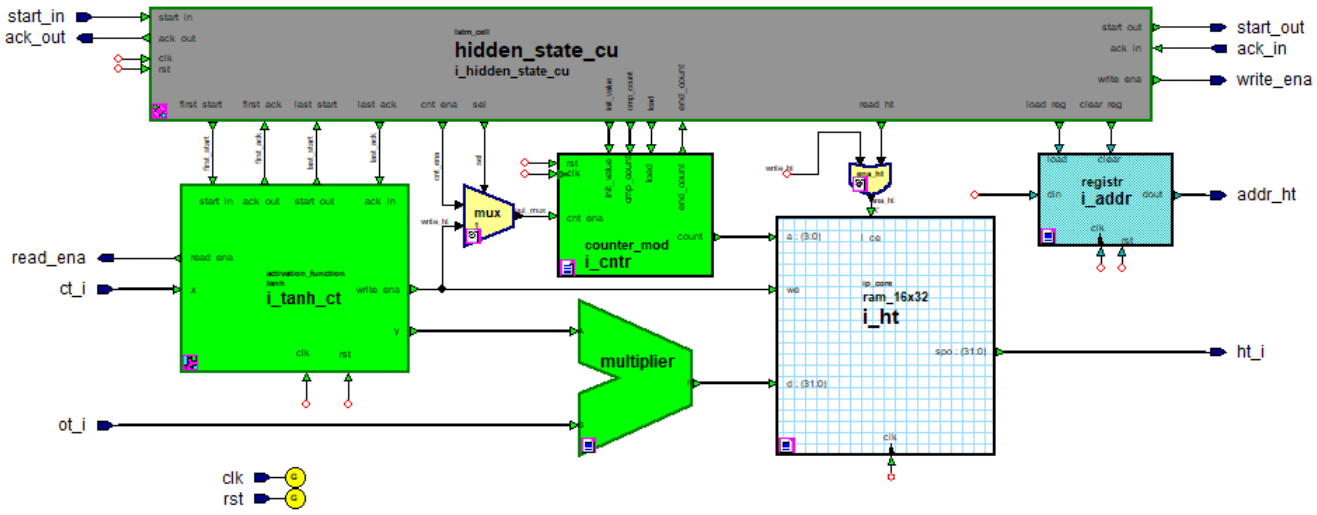

Figura 6.10: Diagrama de bloques de Hidden State

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Hidden state que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Hidden state informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Hidden state informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Hidden state que ya puede escribir en la memoria de salida.
rd_ena	1 bit	Salida	Capacitación de lectura de los datos de entrada.
ct_i	c_data_width	Entrada	Dato de entrada de c_t.
ot_i	c_data_width	Entrada	Dato de entrada de o_t.
write_ena_ht	1 bit	Salida	Capacitación de escritura del dato h_t.
addr_ht	4 bits	Salida	Dirección de memoria del dato h_t.
ht_i	c_data_width	Salida	Dato de salida h_t.

Tabla 6.16: Interfaz de Hidden state

Hidden state está formado por los siguientes componentes descritos brevemente a continuación:

- Tanh ct: módulo Tanh encargado de realizar la función tangente hiperbólica al dato c_t. También cabe destacar que este módulo se encarga de la escritura del resultado correspondiente al estado oculto en la memoria interna ht mediante la señal write_ht.
- cntr: contador con salida modulada que se encarga de direccionar los datos del vector resultado h_t tanto para la memoria ht propia del módulo como para la memoria de salida.
- multiplier: multiplicador donde se realiza el producto escalar de los dos vectores.
- addr: registro conectado a la salida del contador que almacena la dirección correspondiente al dato de h_t para realizar la escritura en la memoria de salida de forma correcta.
- ht: se trata de una memoria RAM_16x32 donde se almacena el estado oculto de la celda.
- Hidden state cu: unidad de control de Hidden state que a su vez se trata del módulo Hidden state cu. En la siguiente subsección hablamos en detalle sobre ella.

6.3.4.1. Hidden state CU

Es la unidad de control que gestiona la ejecución de Hidden state. Se encarga de la comunicación con el exterior llevando a cabo el protocolo de la forma esperada y de gestionar la escritura para las memorias de salida, exteriores al módulo del resultado h_t.

Se ha implementado como una máquina de estados finitos de tipo Moore con reset síncrono y 7 estados. La siguiente figura 6.11 ilustra el diagrama de transición de estados y posteriormente se describe cada uno de ellos.

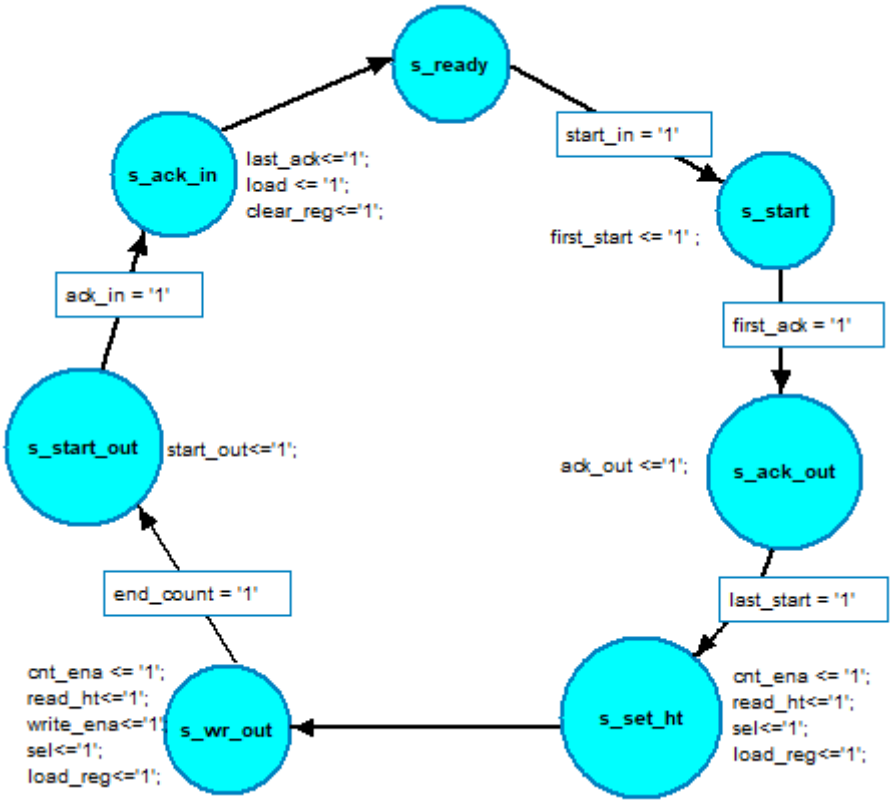

Figura 6.11: FSM de Hidden State CU

- Estado ready: Hidden state está disponible para realizar un nuevo cálculo del estado oculto de la celda LSTM. Espera que el módulo generador envíe la señal start_in por puerto de entrada para cambiar al estado start.
- Estado start: marca el comienzo del cálculo. Se activa la señal first_start informando a Tanh ct que comience su ejecución. Se espera la la activación de la señal first_ack que indica que Tanh ct ya ha leído todos los datos de entrada.
- Estado ack_out: Hidden state CU informa al módulo generador que Hidden state ya ha leído todos los datos de la memoria de entrada mediante la activación de la señal ack_out y se espera a la llegada de la señal last_start conectada a Tanh ct para pasar al estado set ht.
- Estado set_ht: el nuevo estado oculto ya se ha calculado, se prepara la memoria interna ht para propagar el vector resultante con el objetivo de escribirlo en la memoria de salida. En el siguiente ciclo se hace una transición al estado wr_out.
- Estado wr_out: durante este estado se escribe en la memoria de salida el resultado del nuevo estado oculto. Se permanece en este estado hasta la llegada de la señal end_count, fin de cuenta del contador, indica que ya se ha escrito todo el vector. El siguiente estado es start out.
- Estado start_out: Hidden state CU informa al módulo consumidor la escritura de los nuevos datos mediante la señal start_out. Se espera a que el módulo consumidor envíe la señal ack_in y en ese momento se pasa al estado ack_in.

- Estado ack_in: indica que ya se han leído los datos escritos en la memoria de salida, Hidden state CU informa al componente Tanh ct con la señal interna last_ack y se pasa al estado ready para llevar a cabo una nueva ejecución cuando se precise.

6.3.5. Hidden state nx1

Se encarga de calcular el estado oculto del módulo LSTM cell nx1 con la ecuación:

$$h_t = o_t \cdot \tanh(c_t) \tag{6.3}$$

donde:

- o_t: es el vector resultado de la puerta de salida en el instante t.
- tanh: corresponde a la función tangente hiperbólica.
- c_t: es el vector que representa el estado de la celda (cell state) en el instante t.
- h_t: vector que representa el estado oculto en el instante t.

A continuación se expone su estructura 6.12 e interfaz 6.17:

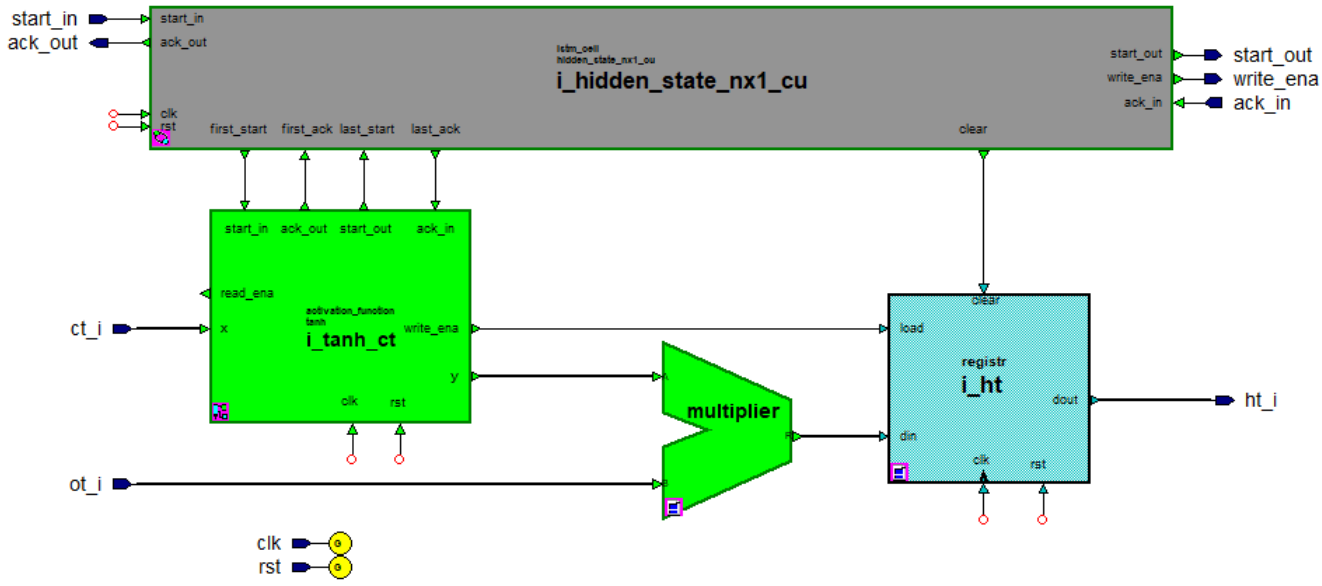

Figura 6.12: Diagrama de bloques de Hidden State nx1

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Hidden state nx1 que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Hidden state nx1 informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Hidden state nx1 informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Hidden state nx1 que ya puede escribir en la memoria de salida.
ct_i	c_data_width	Entrada	Dato de entrada de c_t.
ot_i	c_data_width	Entrada	Dato de entrada de o_t.
write_ena	1 bit	Salida	Capacitación de escritura del dato h_t.
ht_i	c_data_width	Salida	Dato de salida h_t.

Tabla 6.17: Interfaz de Hidden state nx1

Hidden state nx1 está formado por los siguientes componentes descritos brevemente a continuación:

- Tanh ct: módulo Tanh encargado de realizar la función tangente hiperbólica al dato c_t.
- multiplier: multiplicador donde se realiza el producto de los datos.
- ht: se trata de un registro donde se almacena el estado oculto de la celda.
- Hidden state nx1 cu: unidad de control de Hidden state nx1 descrita más en detalle en la siguiente subsección.

6.3.5.1. Hidden state nx1 CU

Es la unidad de control que gestiona la ejecución de Hidden state nx1. Se encarga de la comunicación con el exterior llevando a cabo el protocolo de la forma esperada y de realizar la escritura en la memorias exteriores al módulo del resultado h_t.

Se ha implementado como una máquina de estados finitos de tipo Moore con reset síncrono y 6 estados. La siguiente figura 6.13 ilustra el diagrama de transición de estados y posteriormente se describen cada uno de sus estados.

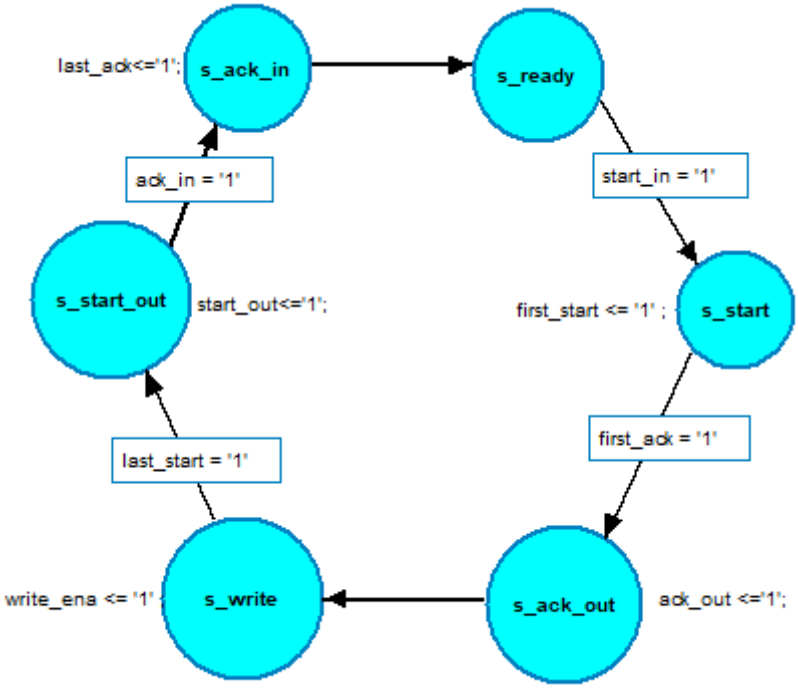

Figura 6.13: FSM de Hidden State nx1 CU

- Estado ready: Hidden state nx1 está esperando la señal start_in que envía el módulo generador de datos para comenzar calcular el nuevo estado oculto de la celda LSTM. Cuando esta señal llega se cambia al estado start.
- Estado start: En este estado Hidden State nx1 CU envía la señal first_start a Tanh ct para que comience su ejecución. Se espera a la llegada de la señal first_ack que indica que Tanh ct ya ha leído todos los datos de entrada.
- Estado ack_out: Hidden State nx1 CU genera la señal ack_out para indicar al módulo generador que ya se han leído todos los datos de entrada y se pasa al estado write.
- Estado write: durante este estado se escribe en la memoria de salida el nuevo estado oculto. Para realizar esta acción se genera la señal write en. Se cambia al estado start out cuando recibe la señal last_start generada por Tanh ct.
- Estado start_out: mediante la señal start_out Hidden State nx1 CU informa al módulo consumidor de que se tienen datos nuevos en la memoria de salida . Se cambia al estado ack_in cuando se recibe la señal ack_in por parte del módulo consumidor.
- Estado ack_in: indica que ya se han leído los datos escritos en la memoria de salida, Hidden State nx1 CU informa al componente Tanh ct con la señal interna last_ack y se pasa al estado ready para llevar a cabo una nueva ejecución cuando se precise.

6.3.6. Cell state

Se encarga de calcular el estado de la celda del módulo LSTM cell con la siguiente operación, la cual se basa en la suma de dos productos escalares:

$$c_t = c_{t-1} \cdot f_t + i_t \cdot newc_t \tag{6.4}$$

donde:

- c_{t-1}: es el vector que representa el estado de la celda en el instante t-1.
- f_t: es el vector resultado de la puerta de olvido en el instante t.
- i_t: es el vector resultado de la puerta de entrada en el instante t.
- $newc_t$: es el vector resultado de la puerta del nuevo candidato en el instante t.
- c_t: es el vector que representa el estado de la celda en el instante t.

El módulo se ha implementado como una máquina de estados de tipo Moore puesto que el objetivo de este módulo es calcular su siguiente estado el cual solo depende de su estado actual y sus entradas. A continuación se expone su estructura 6.14 e interfaz 6.18:

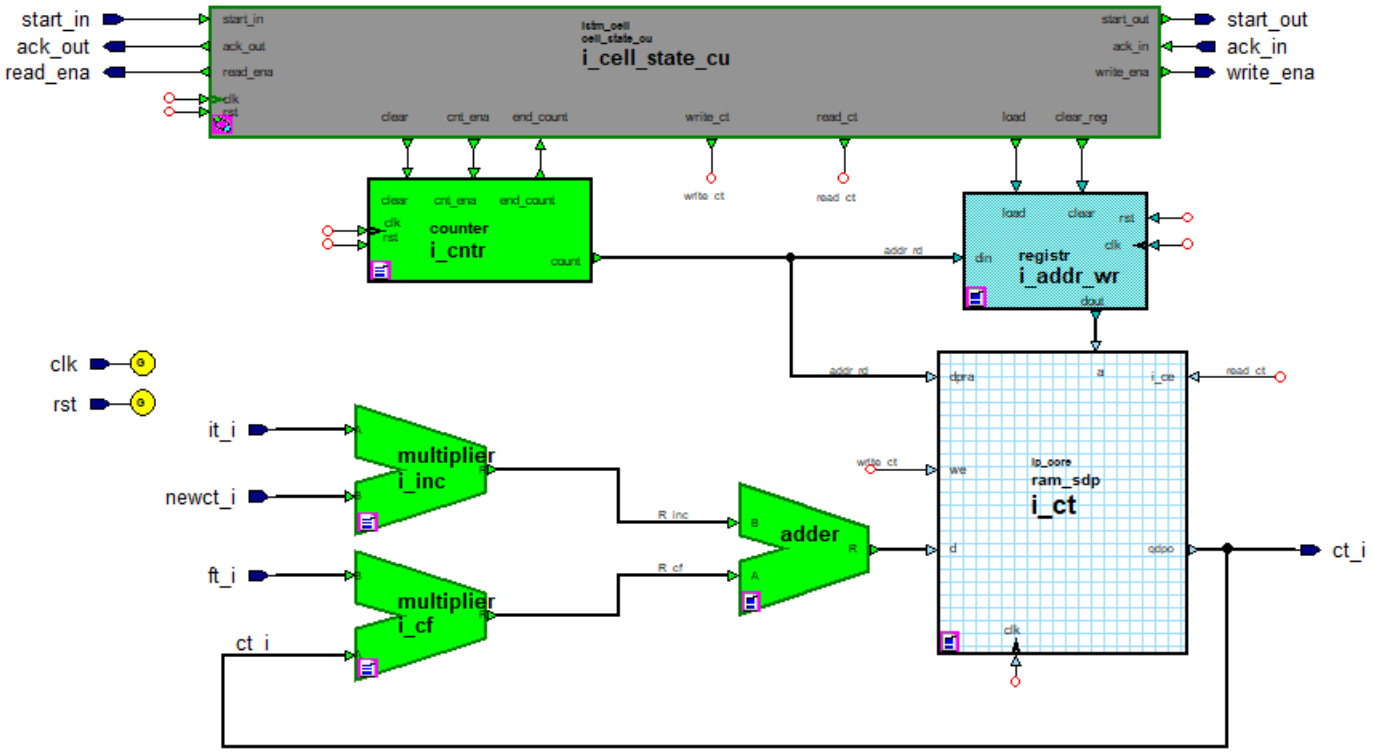

Figura 6.14: Diagrama de bloques de Cell State

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Cell state que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Cell state informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Cell state informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Cell state que ya puede escribir en la memoria de salida.
read_ena	1 bit	Salida	Capacitación de lectura de los datos de entrada.
newct_i	c_data_width	Entrada	Dato de entrada de $newc_t$.
it_i	c_data_width	Entrada	Dato de entrada de i_t.
ft_i	c_data_width	Entrada	Dato de entrada def_t.
write_ena	1 bit	Salida	Capacitación de escritura del dato c_t.
ct_i	c_data_width	Salida	Dato de salida c_t.

Tabla 6.18: Interfaz de Cell state

Cell state está formado por los siguientes componentes descritos brevemente a continuación:

- cf: multiplicador donde se realiza el producto de los datos de la puerta de olvido y el estado de la celda. Se trata del módulo común multiplier, que va recibiendo las componentes de los vectores y realizando el producto de estas.
- inc: multiplicador donde se realiza el producto de los datos de la puerta de entrada y el nuevo candidato. Se trata del módulo común multiplier, que va recibiendo las componentes de los vectores y realizando el producto de estas.
- adder: realiza la suma vectorial de los vectores resultantes de los productos de cf y inc, que consiste en sumar las componentes de los dos vectores, las cuales se le van pasando dato a dato.
- cntr: contador encargado de indexar los datos de la memoria ct.
- addr wr: registro conectado a la salida del contador que almacena la dirección correspondiente al dato de ct_i para realizar la escritura en la memoria ct de forma correcta.
- ct: se trata de una memoria RAM_sdp donde se almacena el estado de la celda, c_t. Es necesario el uso de una RAM simple dual port ya que esta cuenta con un puerto de direccionamiento lectura y otro de direccionamiento de escritura independientes,

que permiten realizar operaciones de lectura y escritura en la memoria simultáneamente. Para calcular el nuevo estado de la celda, se necesita leer el estado anterior de la memoria y escribir el nuevo estado de la celda en la memoria, sobrescribiendo cada dato previamente leído por el nuevo calculado.

- Cell state cu: unidad de control de Cell state que a su vez es el módulo Cell state CU, descrita en detalle en la siguiente subsección.

6.3.6.1. Cell state CU

Es la unidad de control que gestiona la ejecución de Cell state. Se ha implementado como una máquina de estados finitos de tipo Moore con reset síncrono y 7 estados. La siguiente figura 6.15 ilustra el diagrama de transición de estados y posteriormente se describen cada uno de sus estados.

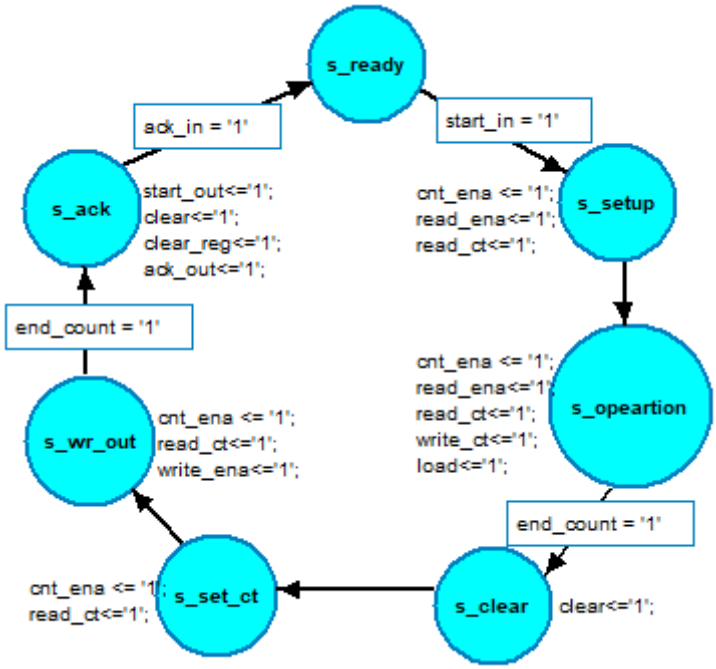

Figura 6.15: FSM de Cell State CU

- Estado ready: Cell state está disponible para realizar un nuevo cálculo del estado de la celda LSTM. Espera la llegada de la señal start_in, enviada por el módulo generador, para cambiar al estado setup.
- Estado setup: se prepara los datos para realizar el cálculo del nuevo estado, se habilita la cuenta del contador y las respectivas señales de capacitación de lectura de las memorias, seguidamente se pasa al estado operation.
- Estado operation: se realiza el cálculo del nuevo estado. El final del cálculo lo marca la señal end_count que indica que el fin de cuenta del contador. Cuando llega esta señal se pasa al estado clear.

- Estado clear: se reinicia el contador para llevar a cabo la escritura del nuevo estado en la memoria de salida. El siguiente estado es set ct.
- Estado set ct: se prepara la memoria interna ct, capacitando a esta y habilitando la cuenta del contador, para propagar el vector resultante con el objetivo de escribirlo en la memoria de salida y se pasa al estado wr_out.
- Estado wr_out: se escribe en la memoria de salida el resultado del estado de la celda activando la señal write_ena. Se permanece en este estado hasta la llegada se la señal end_count, que indica que ya se ha escrito todo el vector. El siguiente estado es ack.
- Estado ack: Cell State CU activa las señales start_out y ack_out, la primera informa al módulo consumidor de que ya tiene disponible el nuevo estado en la memoria de salida; la segunda informa al módulo generador de que ya se han leído todos los datos de la memoria de entrada. Se espera la señal ack_in, enviada por el módulo consumidor, para terminar la ejecución volviendo al estado ready.

6.3.7. Cell state nx1

Se encarga de calcular el estado de la celda correspondiente al módulo LSTM cell nx1 con la ecuación:

$$c_t = c_{t-1} \cdot f_t + i_t \cdot newc_t \tag{6.5}$$

donde:

- c_{t-1}: es el vector que representa el estado de la celda en el instante t-1.
- f_t: es el vector resultado de la puerta de olvido en el instante t.
- i_t: es el vector resultado de la puerta de entrada en el instante t.
- $newc_t$: es el vector resultado de la puerta del nuevo candidato en el instante t.
- c_t: es el vector que representa el estado de la celda en el instante t.

Su implementación también esta basada en una maquina de estados finitos de tipo Moore como el módulo Cell state. A continuación se expone su estructura 6.16 e interfaz 6.19:

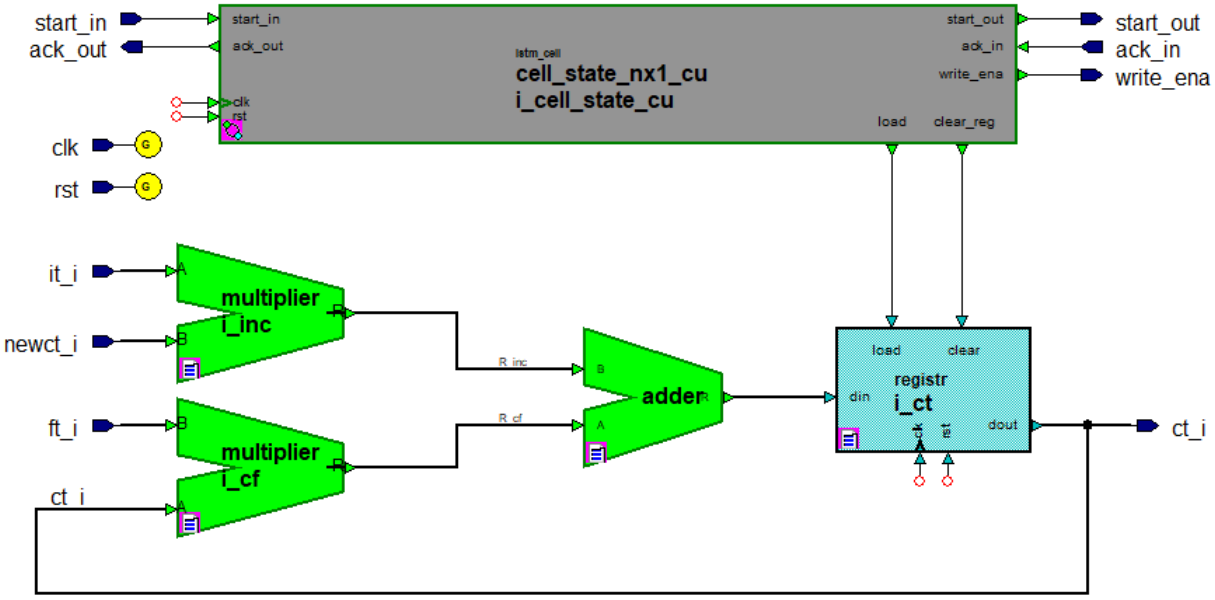

Figura 6.16: Diagrama de bloques de Cell state nx1

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Cell state nx1 que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Cell state nx1 informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Cell state nx1 informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Cell state nx1 que ya puede escribir en la memoria de salida.
newct_i	c_data_width	Entrada	Dato de entrada de $newc_t$.
it_i	c_data_width	Entrada	Dato de entrada de i_t.
ft_i	c_data_width	Entrada	Dato de entrada def_t.
write_ena	1 bit	Salida	Capacitación de escritura del dato c_t.
ct_i	c_data_width	Salida	Dato de salida c_t.

Tabla 6.19: Interfaz de Cell state nx1

Cell state nx1 está formado por los siguientes componentes descritos brevemente a continuación:

- cf: se trata del módulo común multiplier, encargado de realizar el producto entre las únicas componentes de los vectores de la puerta de olvido y el estado de la celda.
- inc: se trata del módulo común multiplier, encargado de realizar el producto entre las únicas componentes de los vectores de la puerta de entrada y el nuevo candidato.
- adder: suma los dos resultados de los productos.
- ct: se trata de un registro donde se almacena el estado de la celda, c_t.
- Cell state nx1 CU: unidad de control de Cell state nx1. En la siguiente subsección hablamos en detalle sobre ella.

6.3.7.1. Cell state nx1 CU

Es la unidad de control que gestiona la ejecución de Cell state nx1. Se ha implementado como una máquina de estados finitos de tipo Moore con reset síncrono y 4 estados. La siguiente figura 6.17 ilustra el diagrama de transición de estados y posteriormente se describen cada uno de sus estados.

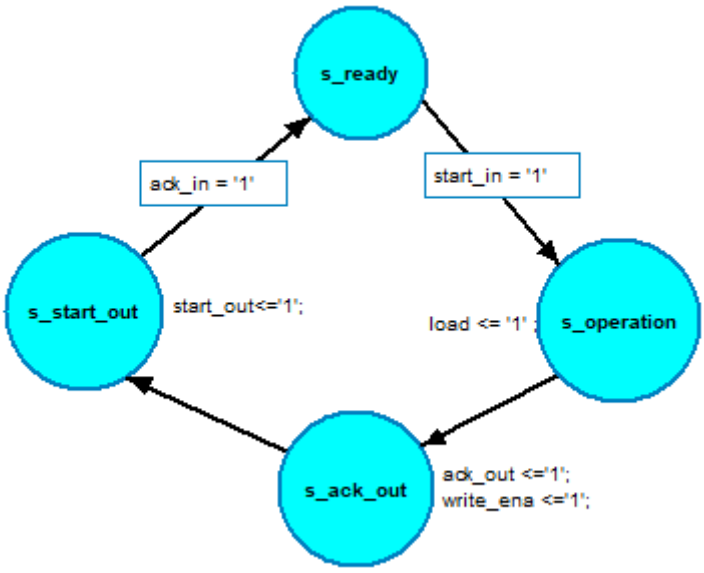

Figura 6.17: FSM de Cell State nx1 CU

- Estado ready: Cell state nx1 está disponible para realizar un nuevo cálculo del estado de la celda LSTM. Espera la llegada de la señal start_in, enviada por el módulo generador, para cambiar al estado operation.
- Estado operation: se realiza el cálculo del nuevo estado que a su vez carga el resultado en el registro ct y se hace la transición al estado ack_out.

- Estado ack_out: Cell State nx1 CU activa el puerto de salida write_ena para llevar acabo la escritura del nuevo estado en la memoria de salida y también el puerto de salida ack_out para informar al módulo generador de la lectura de los datos correspondientes en la memoria de entrada. El siguiente estado es start_out.
- Estado start_out: estado en el que Cell state nx1 CU informa al módulo consumidor de la escritura del nuevo estado en la memoria de salida mediante la activación de la señal start_out. Se termina la ejecución volviendo al estado ready en el momento que el módulo consumidor active la señal ack_in.

6.3.8. Sigmoid gate

Se trata de una puerta LSTM que tiene como función de activación la función sigmoidea. Este componente se encuentra en el módulo LSTM cell y se encarga de calcular la siguiente ecuación:

$$y_t = sigmoid\left(U_y \cdot h_{t-1} + W_y \cdot x_t + b_y\right) \tag{6.6}$$

donde:

- x_t: vector con los valores de entrada en el instante t.
- W_y: matriz de pesos de entrada para la puerta y.
- h_{t-1}: vector del estado oculto en el instante a t-1.
- U_y: matriz que contiene los pesos de estado para la puerta y.
- b_y: vector que contiene los pesos bias de la puerta y.
- $sigmoid()$: función sigmoidea, que se aplica a todos los componentes del vector resultante.
- y_t: vector resultante.

A continuación se pueden ver su estructura 6.32 e interfaz 6.29:

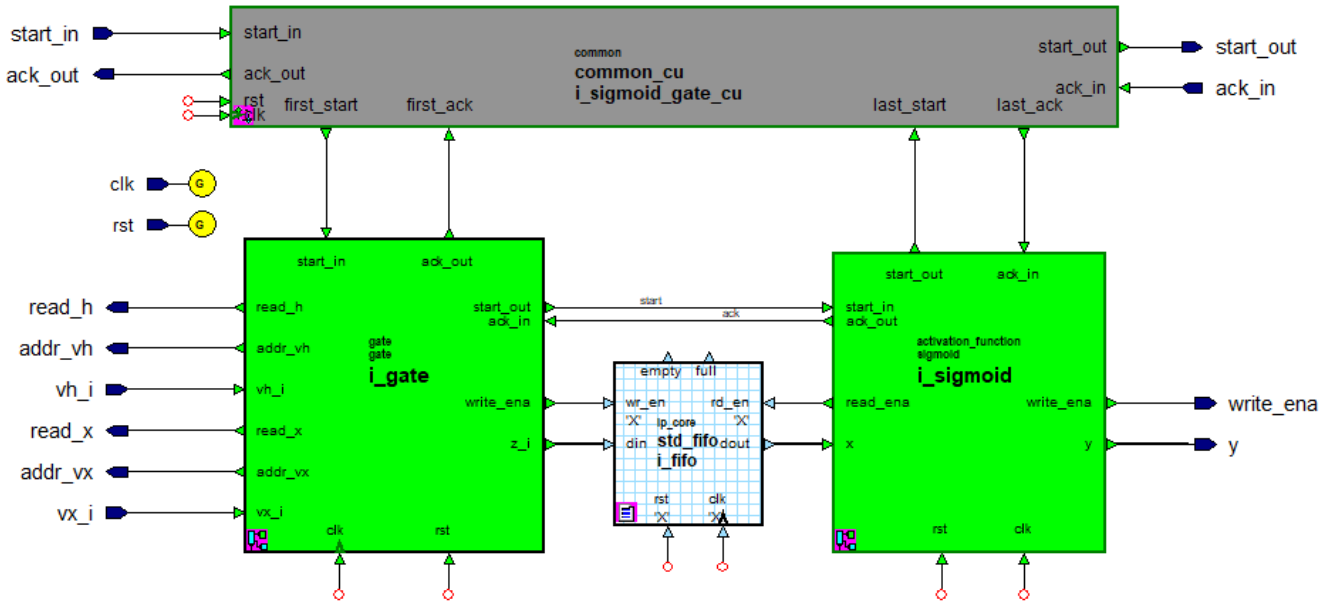

Figura 6.18: Diagrama de bloques de Sigmoid gate

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Sigmoid gate que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Sigmoid gate informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Sigmoid gate informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Sigmoid gate que ya puede escribir en la memoria de salida.
read_x	1 bit	Salida	Capacitación de lectura de un dato de la entrada vx.
addr_vx	4 bits	Salida	Dirección de memoria del dato vx_i.
vx_i	c_data_width	Entrada	Dato de entrada vx_i.
read_h	1 bit	Salida	Capacitación de lectura de un dato de la entrada vh.
addr_vh	4 bits	Salida	Dirección de memoria del dato vh_i.
vh_i	c_data_width	Entrada	Dato de entrada vh_i.
write_ena	1 bit	Salida	Capacitación de escritura de un dato del vector y.
y_i	c_data_width	Salida	Dato de salida y_i.

Tabla 6.20: Interfaz de Sigmoid gate

Sigmoid gate se forma por los siguientes componentes descritos a continuación:

- Gate: se trata del módulo Gate que representa una puerta LSTM sin función de activación que realiza la suma ponderada de una puerta LSTM.
- Fifo: se trata de una fifo donde se almacena temporalmente el vector resultado de Gate.
- Sigmoid: el módulo Sigmoid se encarga de realizar la función sigmoidal a todos los elementos del vector resultante de Gate. Lee los datos de la fifo y después propaga su resultado por el puerto de salida y.
- Sigmoid gate CU: unidad de control de Sigmoid gate que a su vez es el módulo Common CU.

6.3.9. Sigmoid gate nx1

Se trata de una puerta LSTM que tiene como función de activación la función sigmoidea. Este componente se encuentra en el módulo LSTM cell nx1 y se encarga de calcular la siguiente ecuación:

$$y_t = sigmoid\left(U_y \cdot h_{t-1} + W_y \cdot x_t + b_y\right) \tag{6.7}$$

donde:

- x_t: vector con los valores de entrada en el instante t.
- W_y: matriz de pesos de entrada para la puerta y.
- h_{t-1}: vector del estado oculto en el instante a t-1.
- U_y: matriz que contiene los pesos de estado para la puerta y.
- b_y: vector que contiene los pesos bias de la puerta y.
- $sigmoid()$: función sigmoidea, que se aplica a todos los componentes del vector resultante.
- y_t: vector resultante.

A continuación exponemos su estructura 6.19 e interfaz 6.21:

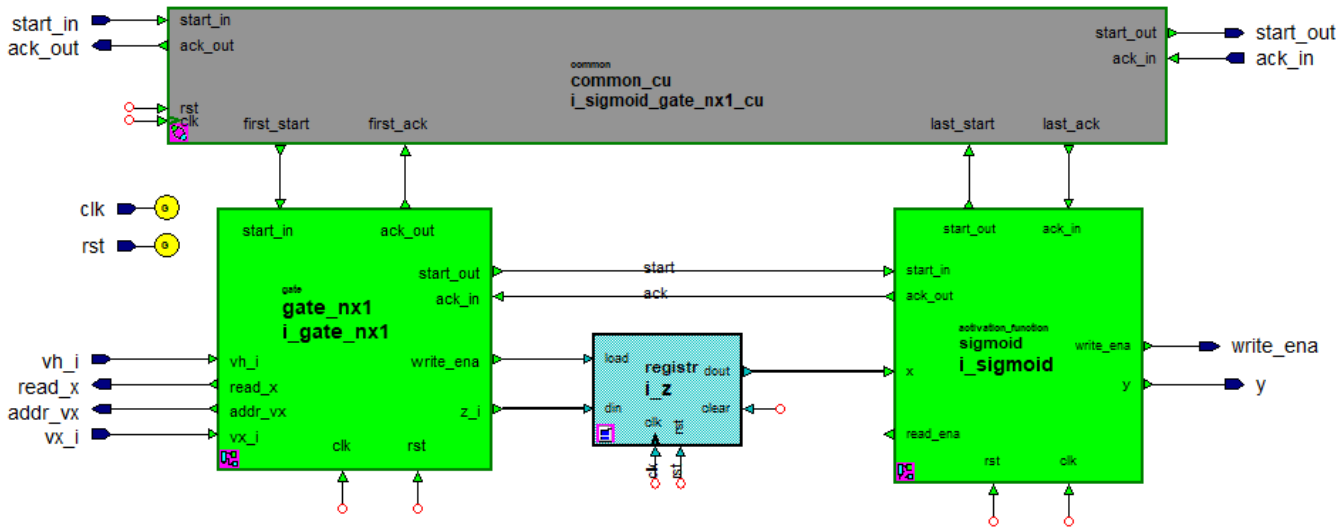

Figura 6.19: Diagrama de bloques de Sigmoid gate nx1

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El modulo generador indica a Sigmoid gate nx1 que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución
ack_out	1 bit	Salida	El módulo Sigmoid gate nx1 informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	El módulo Sigmoid gate nx1 informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	el módulo consumidor indica al módulo Sigmoid gate nx1 que ya puede escribir en la memoria de salida.
read_x	1 bit	Salida	Capacitación de lectura de un dato de la entrada vx.
addr_vx	4 bits	Salida	Dirección de memoria del dato vx_i.
vx_i	c_data_width	Entrada	Dato de entrada de vx.
vh_i	c_data_width	Entrada	Dato de entrada de vh.
write_ena	1 bit	Salida	Capacitación de escritura de un dato del vector y.
y	c_data_width	Salida	Dato de salida y.

Tabla 6.21: Interfaz de Sigmoid gate nx1

Sigmoid gate nx1 está formado por los siguientes componentes descritos brevemente a continuación:

- Gate nx1: se trata del módulo Gate nx1, puerta LSTM sin función de activación que realiza la suma ponderada de una puerta LSTM.
- Z: se trata de un registro donde se guarda el resultado de Gate nx1.
- Sigmoid: el módulo Sigmoid se encarga de realizar la función sigmoidal al dato resultante de Gate nx1. Lee el dato del registro z y después propaga su resultado por el puerto de salida y.
- Sigmoid gate nx1 CU: unidad de control de Sigmoid gate nx1 que a su vez es el módulo Common CU.

6.3.10. Tanh gate

Se trata de una puerta LSTM que tiene como función de activación la función tangente hiperbólica. Este componente se encuentra en el módulo LSTM cell y se encarga de calcular el nuevo candidato de la celda de estado con la siguiente ecuación:

$$y_t = \tanh\left(U_y \cdot h_{t-1} + W_y \cdot x_t + b_y\right) \tag{6.8}$$

donde:

- x_t: vector con los valores de entrada en el instante t.
- W_y: matriz de pesos de entrada para la puerta y.
- h_{t-1}: vector del estado oculto en el instante a t-1.
- U_y: matriz que contiene los pesos de estado para la puerta y.
- b_y: vector que contiene los pesos bias de la puerta y.
- tanh(): función tangente hiperbólica, que se realiza a todos los componentes del vector resultante.
- y_t: vector resultante.

A continuación exponemos su estructura 6.20 e interfaz 6.22:

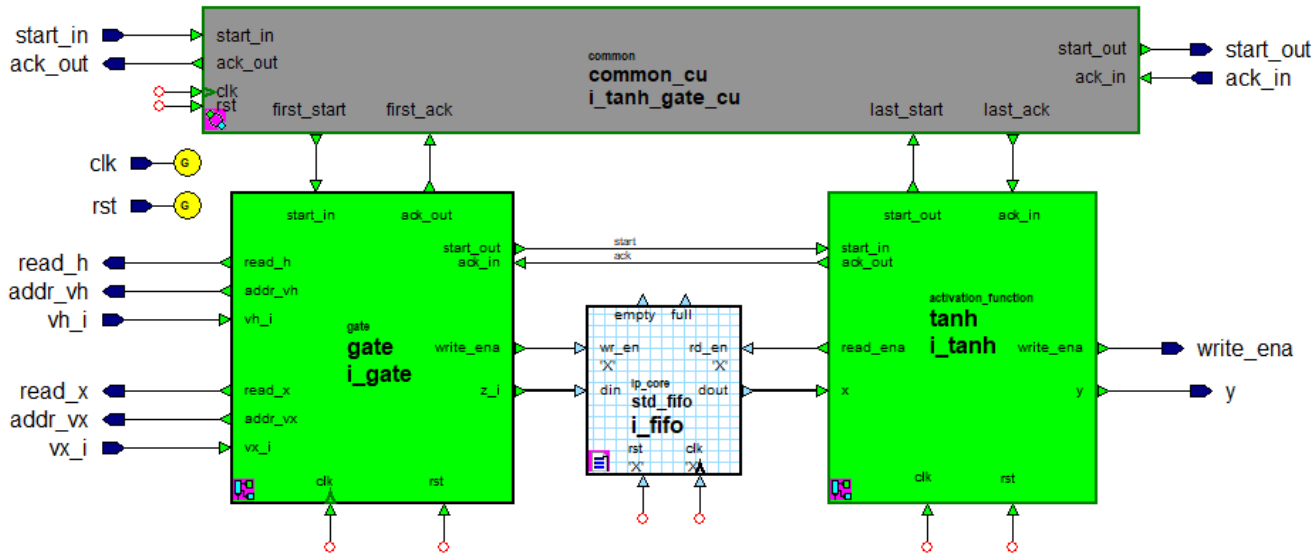

Figura 6.20: Diagrama de bloques de Tanh gate

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Tanh gate que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Tanh gate informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Tanh gate informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Tanh gate que ya puede escribir en la memoria de salida.
read_x	1 bit	Salida	Capacitación de lectura de un dato de la entrada vx.
addr_vx	4 bits	Salida	Dirección de memoria del dato vx_i.
vx_i	c_data_width	Entrada	Dato de entrada vx_i.
read_h	1 bit	Salida	Capacitación de lectura de un dato de la entrada vh.
addr_vh	4 bits	Salida	Dirección de memoria del dato vh_i.
vh_i	c_data_width	Entrada	Dato de entrada vh_i.
write_ena	1 bit	Salida	Capacitación de escritura de un dato del vector y.
y_i	c_data_width	Salida	Dato de salida y_i.

Tabla 6.22: Interfaz de Tanh gate

Tanh gate está formado por los siguientes componentes que se describen brevemente a continuación:

- Gate: se trata del módulo Gate, puerta LSTM sin función de activación que realiza la suma ponderada de una puerta LSTM.
- FIFO: se trata de una fifo donde se almacena temporalmente el vector resultado de Gate.
- Tanh: el módulo Tanh se encarga de realizar la función tangente hiperbólica a todos los elementos del vector resultante de Gate.
- Tanh gate CU: unidad de control de Tanh gate que a su vez es el módulo Common CU.

6.3.11. Tanh gate nx1

Se trata de una puerta LSTM que tiene como función de activación la función tangente hiperbólica. Este componente se encuentra en el módulo LSTM cell nx1 y se encarga de calcular el nuevo candidato de la celda de estado con la siguiente ecuación:

$$y_t = \tanh\left(U_y \cdot h_{t-1} + W_y \cdot x_t + b_y\right) \tag{6.9}$$

donde:

- x_t: vector con los valores de entrada en el instante t.
- W_y: matriz de pesos de entrada para la puerta y.
- h_{t-1}: vector del estado oculto en el instante a t-1.
- U_y: matriz que contiene los pesos de estado para la puerta y.
- b_y: vector que contiene los pesos bias de la puerta y.
- tanh(): función tangente hiperbólica, que se realiza a todos los componentes del vector resultante.
- y_t: vector resultante.

A continuación se expone su estructura 6.21 e interfaz 6.23:

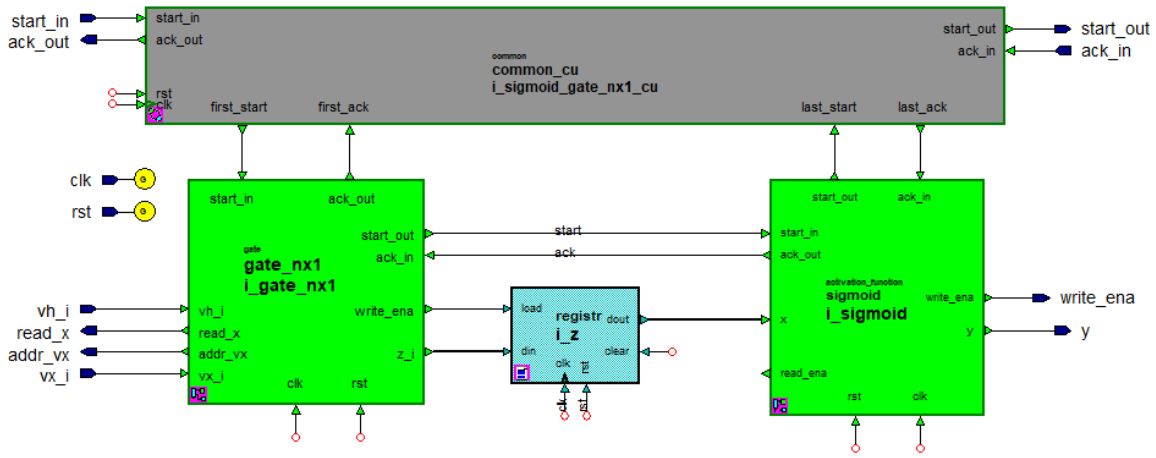

Figura 6.21: Diagrama de bloques de Tanh gate nx1

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Tanh gate nx1 que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Tanh gate nx1 informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Tanh gate nx1 informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Tanh gate nx1 que ya puede escribir en la memoria de salida.
read_x	1 bit	Salida	Capacitación de lectura de un dato de la entrada vx.
addr_vx	4 bits	Salida	Dirección de memoria del dato vx_i.
vx_i	c_data_width	Entrada	Dato de entrada de vx.
vh_i	c_data_width	Entrada	Dato de entrada de vh.
write_ena	1 bit	Salida	Capacitación de escritura de un dato del vector y.
y	c_data_width	Salida	Dato de salida y.

Tabla 6.23: Interfaz de Tanh gate nx1

Tanh gate nx1 está formado por los siguientes componentes que se describen brevemente a continuación:

- Gate nx1: se trata del módulo Gate nx1, puerta LSTM sin función de activación que realiza la suma ponderada de una puerta LSTM.
- Z: se trata de un registro donde se guarda el resultado de Gate nx1.
- Tanh: el módulo Tanh se encarga de realizar la función tangente hiperbólica al dato resultante de Gate nx1. Lee el dato del registro z y después propaga su resultado por el puerto de salida y.
- Tanh gate nx1 CU: se trata del módulo Common CU que a su vez controla el módulo Tanh gate nx1.

6.3.12. Gate

Este módulo se encuentra en Sigmoid gate y Tanh gate. Se encarga de calcular la suma ponderada de una puerta LSTM con la siguiente ecuación:

$$z_t = (U_z \cdot h_{t-1} + W_z \cdot x_t + b_z) \tag{6.10}$$

donde:

- x_t: vector con los valores de entrada en el momento t.
- W_z: matriz de pesos de entrada para la puerta z.
- h_{t-1}: vector con los valores del estado oculto en el instante t-1.
- U_z: matriz que contiene los pesos de estado para la puerta z.
- b_z: vector que contiene los pesos bias de la puerta z.
- z_t: vector resultante.

A continuacióne se expone su estructura 6.22 e interfaz 6.24:

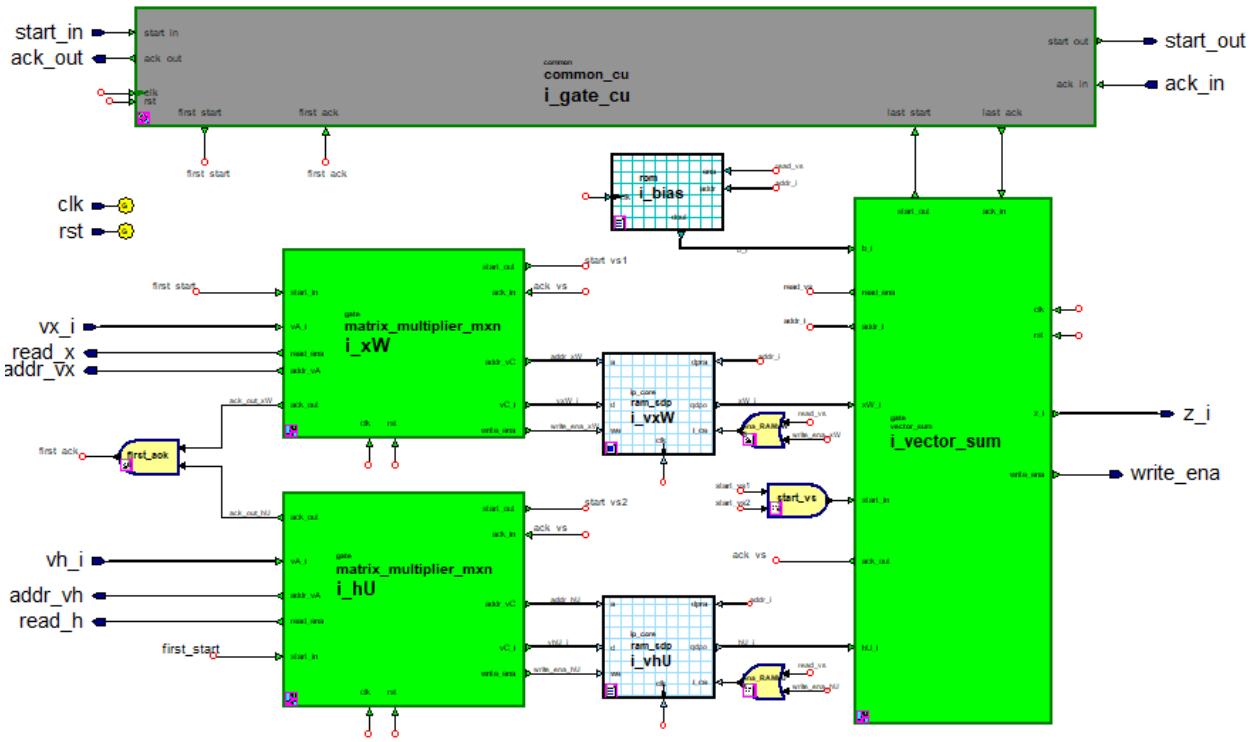

Figura 6.22: Diagrama de bloques de Gate

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Gate que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Gate informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Gate informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Gate que ya puede escribir en la memoria de salida.
read_x	1 bit	Salida	Capacitación de lectura de un dato de la entrada vx.
addr_vx	4 bits	Salida	Dirección de memoria del dato vx_i.
vx_i	c_data_width	Entrada	Dato de entrada vx_i.
read_h	1 bit	Salida	Capacitación de lectura de un dato de la entrada vh.
addr_vh	4 bits	Salida	Dirección de memoria del dato vh_i.
vh_i	c_data_width	Entrada	Dato de entrada vh_i.
write_ena	1 bit	Salida	Capacitación de escritura de un dato del vector z.
z_i	c_data_width	Salida	Dato de salida z_i.

Tabla 6.24: Interfaz de Gate

Gate está formado por los siguientes componentes que se describen brevemente a continuación:

- xW: módulo Matrix multiplier mxn encargado de realizar el producto de matrices de la entrada xt por la matriz de pesos W.
- hU: módulo Matrix multiplier mxn encargado de realizar el producto de matrices de la entrada ht por la matriz de pesos U.
- Bias: se trata de una memoria ROM donde se guarda el vector bias.
- vxW: se trata de una memoria RAM simple dual port donde se guarda el vector resultado de xW. El direccionamiento de escritura es manejado por xW, y el de lectura por Vector sum.
- vhU: se trata de una memoria RAM simple dual port donde se guarda el vector resultado de hU. El direccionamiento de escritura es manejado por hU, y el de lectura por Vector sum.
- Vector sum: módulo Vector sum que realiza la suma vectorial de los vectores xW+hU+bias. Se encarga de leer los datos de las memorias y después propagar el resultado por el puerto de salida z_i.

- Gate CU: se trata del módulo Common CU, unidad que controla a Gate.

6.3.13. Vector sum

Módulo encargado de realizar la suma vectorial del módulo Gate. A continuación se expone su estructura 6.23 e interfaz 6.25:

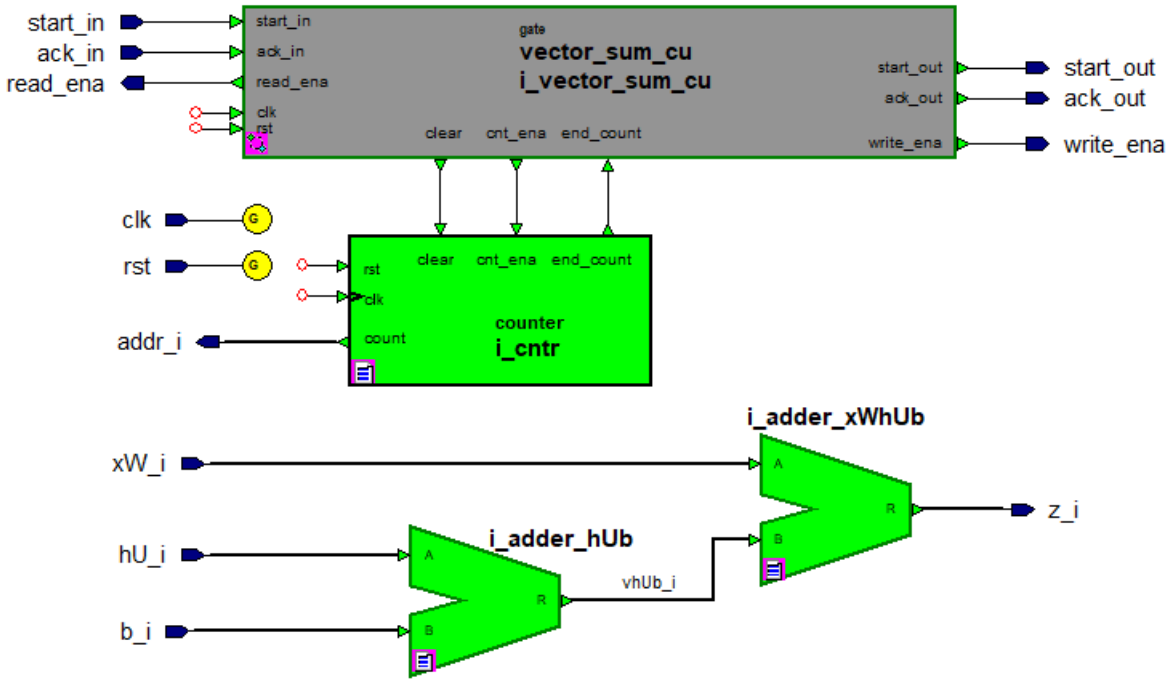

Figura 6.23: Diagrama de bloques de Vector sum

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Vector sum que ya tiene los datos disponibles y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Vector sum informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Vector sum informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Vector sum que ya puede escribir en la memoria de salida.
read_ena	1 bit	Salida	Capacitación de lectura de un dato de entrada.
addr_i	4 bits	Salida	Dirección de memoria de los datos de entrada.
xW_i	c_data_width	Entrada	Dato de entrada xW_i.
hU_i	c_data_width	Entrada	Dato de entrada hU_i.
b_i	c_data_width	Entrada	Dato de entrada $bias_i$.
write_ena	1 bit	Salida	Capacitación de escritura de un dato del vector z.
z_i	c_data_width	Salida	Dato de salida z_i.

Tabla 6.25: Interfaz de Vector sum

Vector sum está formado por los siguientes componentes que se describen brevemente a continuación:

- Adder hUb: sumador que realiza la operación hU+bias, que consiste en sumar las componentes de los dos vectores, las cuales se le van pasando data a dato, realizando una suma por ciclo.
- Adder xWhUb: sumador que realiza la suma del resultado (hU+bias)+xW, que consiste en sumar las componentes de los dos vectores, las cuales se le van pasando data a dato, realizando una suma por ciclo.
- Cntr: contador encargado de indexar los datos de las memorias de entrada.
- Vector sum CU: unidad de control de Vector Sum. En la siguiente subsección hablamos en detalle sobre ella.

6.3.13.1. Vector sum CU

Es la unidad de control de Vector sum.Genera las señales necesarias para la implementación del protocolo de sincronización con los módulos generador y consumidor, así como las señales que permiten la correcta ejecución de las operaciones internas del módulo. Se ha implementado como una máquina de estados finitos de tipo Moore con reset síncrono y 4 estados. La siguiente figura 6.24 ilustra el diagrama de transición de estados y posteriormente se describen cada uno de ellos.

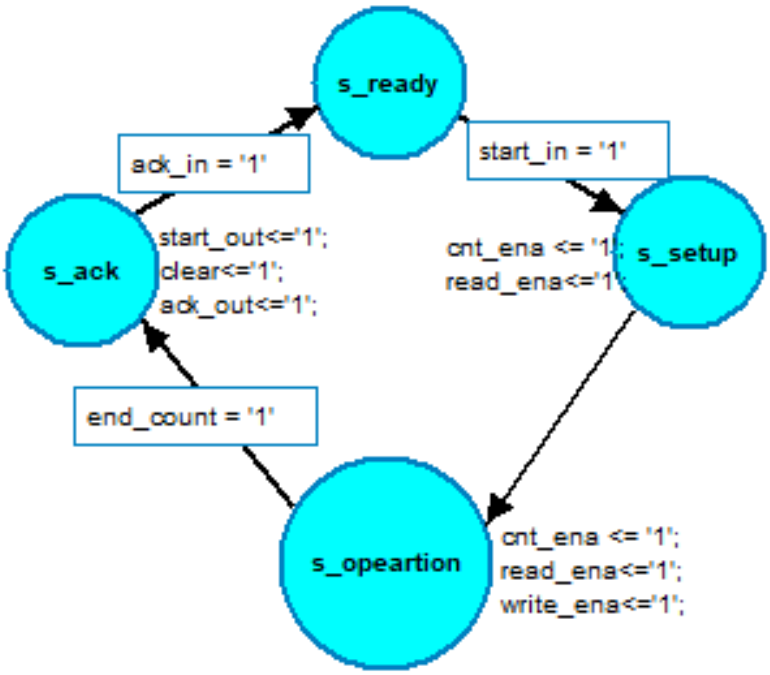

Figura 6.24: FSM de Vector sum

- Estado ready: Vector sum está disponible para realizar la suma vectorial de las entradas ponderadas mas el bias de Gate. Espera la llegada de la señal start_in, enviada por el módulo generador, para cambiar al estado set up.
- Estado set up: se preparan los componentes para realizar un nuevo cálculo activando la cuenta del contador y habilitando el puerto de salida read_ena. Después se hace la transición al estado operation.
- Estado operation: se realiza el cálculo, se van leyendo los datos y escribiendo simultáneamente; en el momento que llegue la señal end_count, fin de cuenta del contador, se cambia al estado ack.
- Estado ack: estado en el que Vector sum CU informa al módulo consumidor de la escritura del nuevo vector resultado en la memoria de salida mediante la activación de la señal start_out. También se informa al módulo generador de que todos los datos de las memorias de entrada han sido leídos, mediante la señal ack_out. Se

espera a que el módulo consumidor envíe la señal ack_in por puerto de entrada para terminar la ejecución volviendo al estado ready.

6.3.14. Gate nx1

Este módulo se encuentra en Sigmoid gate nx1 y Tanh gate nx1. Se encarga de calcular la suma ponderada de una puerta LSTM de un solo estado oculto con la ecuación:

$$z_t = (U_z \cdot h_{t-1} + W_z \cdot x_t + b_z) \tag{6.11}$$

donde:

- x_t: vector con los valores de entrada en el momento t.
- W_z: matriz de pesos de entrada para la puerta z.
- h_{t-1}: vector con los valores del estado oculto en el instante t-1.
- U_z: matriz que contiene los pesos de estado para la puerta z.
- b_z: vector que contiene los pesos bias de la puerta z.
- z_t: vector resultante.

A continuación se expone su estructura 6.25 e interfaz 6.26:

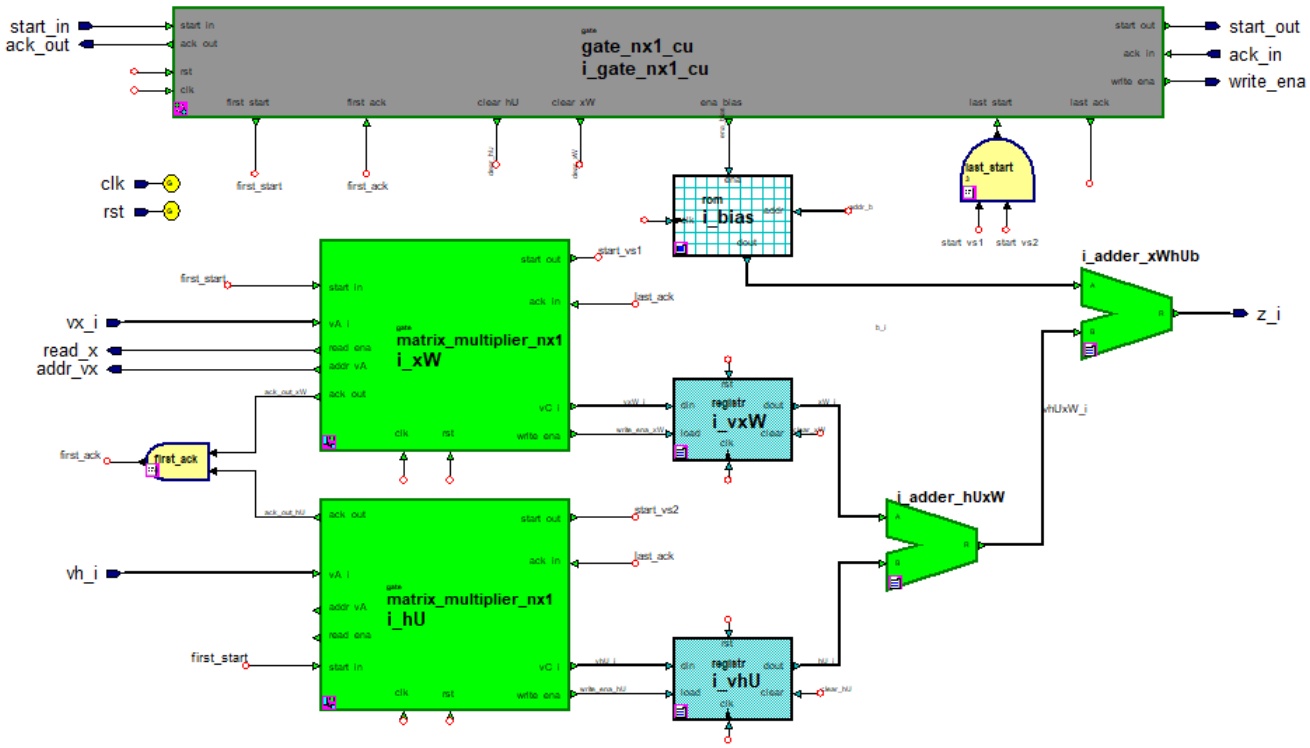

Figura 6.25: Diagrama de bloques de Gate nx1

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Gate nx1 que ya tiene los datos disponibles en la memoria de entada y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Gate nx1 informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Gate nx1 informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Gate nx1 que ya puede escribir en la memoria de salida.
read_x	1 bit	Salida	Capacitación de lectura de un dato de la entrada vx.
addr_vx	4 bits	Salida	Dirección de memoria del dato vx_i.
vx_i	c_data_width	Entrada	Dato de entrada vx_i.
vh_i	c_data_width	Entrada	Dato de entrada vh_i.
write_ena	1 bit	Salida	Capacitación de escritura de un dato del vector z.
z_i	c_data_width	Salida	Dato de salida z.

Tabla 6.26: Interfaz de Gate nx1

Gate nx1 está formado por los siguientes componentes que se describen brevemente a continuación:

- xW: módulo Matrix multiplier nx1 encargado de realizar el producto de matrices de la entrada xt por la matriz de pesos W.
- hU: módulo Matrix multiplier nx1 encargado de realizar el producto de matrices de la entrada ht por la matriz de pesos U.
- Bias: se trata de una memoria ROM donde se guarda el vector bias.
- vxW: se trata de un registro donde se guarda el vector resultado, de un solo elemento de xW.
- vhU: se trata de un registro donde se guarda el vector resultado de un solo elemento de hU.
- Adder hUb: sumador que realiza la operación hU+bias.
- Adder xWhUb: sumador que realiza la suma del resultado (hU+bias)+xW.
- Gate nx1 CU: unidad de control de Gate nx1, en la siguiente subsección hablamos en detalle sobre ella.

6.3.14.1. Gate nx1 CU

Es la unidad de control de Gate nx1.Genera las señales necesarias para la implementación del protocolo de sincronización con los módulos generador y consumidor, así como las señales que permiten la correcta ejecución de las operaciones internas del módulo. Se ha implementado como una máquina de estados finitos de tipo Moore con reset síncrono y 7 estados. La siguiente figura 6.26 ilustra el diagrama de transición de estados y posteriormente se describen cada uno de sus estados.

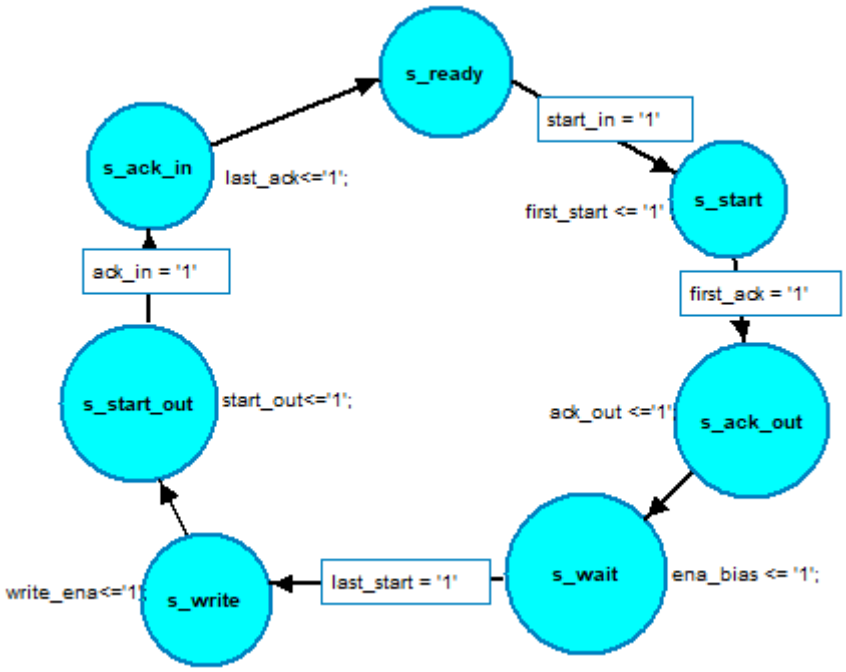

Figura 6.26: FSM de Gate nx1

- Estado ready: Gate nx1 está disponible para realizar una nueva ejecución. Espera la llegada de la señal start_in, enviada por el módulo generador, para cambiar al estado start.
- Estado start: marca el comienzo de ejecución. Se activa la señal first_start informando a los módulos xW y hU que comiencen su ejecución. Se espera la la activación de la señal first_ack que indica que xW y hU ya han leído todos los datos de entrada.
- Estado ack_out: Gate nx1 CU informa al módulo generador que Gate nx1 ya ha leído todos los datos de la memoria de entrada, mediante la activación de la señal ack_out. Seguidamente se pasa al estado wait.
- Estado wait: espera de la activación de la señal last_start conectada a los módulos xW y hU para pasar al siguiente estado write. Cabe destacar que en estado se activa la ROM bias y así tener el dato preparado para realizar la suma vectorial de los datos.

- Estado write: escritura en la memoria de salida del resultado de Gate nx1. En este mismo estado se realiza la suma vectorial de los resultados de xW+hU+bias. Se pasa al siguiente estado start out.
- Estado start_out: Gate nx1 CU informa al módulo consumidor de la escritura de los nuevos datos mediante la señal start_out. Se espera que el módulo consumidor envíe la señal ack_in. En ese momento se pasa al estado ack_in.
- Estado ack_in: indica que ya se han leído los datos escritos en la memoria de salida, se informa a los componentes xW y hU con la señal interna last_ack y se pasa al estado ready para llevar a cabo una nueva ejecución cuando se precise.

6.3.15. Matrix multiplier mxn

Este módulo es un componente de Gate y se encarga de que se lleve a cabo un producto matricial tal y como se puede observar en la siguiente ecuación:

$$vC = vA \cdot MB \tag{6.12}$$

donde:

- vA: vector A, que corresponde con la entrada x_t o con el estado oculto h_{t-1} de Gate, los cuales son un vector de 3 componentes.
- MB: matriz B, se trata de la matriz de pesos W o bien la matriz U.
  La matriz W tiene la forma de:

  $$W = [\text{ nº de características de } x_t, \text{ nº de estados ocultos }] \tag{6.13}$$

  En cambio la matriz U tiene la forma:

  $$U = [\text{ nº de estados ocultos, nº de estados ocultos }] \tag{6.14}$$

  Ambas corresponden a la dimensión de $3 * 3$.
- vC: vector resultante C, que tendrá 3 componentes.

Solo se le debe facilitar por su puerto de entrada los valores del vector A puesto que la matriz B está contenida en una memoria ROM del propio módulo. Para implementar este producto matricial se hace uso de una tabla de verdad, que genera las direcciones de memoria de los operandos a los que se tiene que acceder. Esta tabla tiene tantas entradas como lecturas de memoria se deben ejecutar. La entrada a la tabla se corresponde con la cuenta de un contador que indica el orden de operación, las salidas indican las direcciones de memoria a las que se deben acceder. La siguiente figura ilustra el contenido de la tabla de verdad para el cálculo de la primera componente del vector C:

	A	B	C	D
1	count	addr_MB	addr_vA	addr_vC
2	"0000"	"0000"	"0000"	"0000"
3	"0001"	"0011"	"0001"	"0000"
4	"0010"	"0110"	"0010"	"0000"

Figura 6.27: tabla de verdad Matrix Multiplier mxn

A continuación se expone la estructura e interfaz de Matrix multiplier mxn:

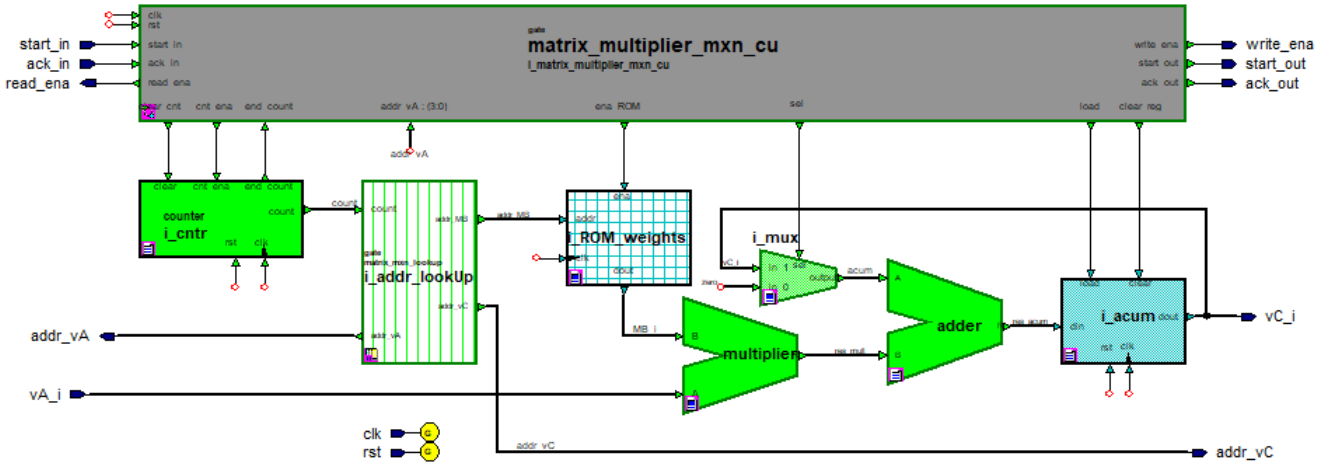

Figura 6.28: Diagra de bloques de Matrix Multiplier mxn

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Matrix multiplier mxn que ya tiene los datos disponibles en la memoria de entada y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Matrix multiplier mxn informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Matrix multiplier mxn informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Matrix multiplier mxn que ya puede escribir en la memoria de salida.
read_ena	1 bit	Salida	Capacitación de lectura de un dato de la entrada vA.
addr_vA	4 bits	Salida	Dirección de memoria del dato vA_i.
vA_i	c_data_width	Entrada	Dato de entrada vA_i.
write_ena	1 bit	Salida	Capacitación de escritura de un dato de vC.
addr_vC	4 bits	Salida	Dirección de memoria del dato vC_i.
vC_i	c_data_width	Salida	Dato de salida vC_i.

Tabla 6.27: Interfaz de Matrix multiplier mxn

Matrix multiplier mxn se forma por los siguientes componentes de los que se realiza una breve descripción:

- Cntr: contador conectado a Addr lookup que lleva la cuenta del número de operaciones que se deben efectuar para llevar a cabo el producto matricial.
- Addr lookup: tabla de verdad donde están almacenados las direcciones de memoria de los operandos que se deben operar. Gracias a esta tabla podemos indexar los componentes de cada una de los operandos aunque no correspondan a la misma dirección, es decir, podemos recorrer una de la matrices por sus filas y a la otra por sus columnas realizando así la operación correctamente.
- ROM weights: memoria ROM donde está almacenada la matriz B.
- Multiplier: realiza el producto de las componentes de las matrices.
- Mux: multiplexor que determina que debe entrar al sumador si el valor de la cuenta acumulada o '0'.
- Adder: sumador de los productos y la cuenta acumulada.
- Acum: registro donde se almacenan los resultados de cada componente del vector resultante C.

- Matrix multiplier mxn CU: unidad de control de Matrix multiplier descrita posteriormente en la siguiente subsección.

6.3.15.1. Matrix multiplier mxn CU

Es la unidad de control de Matrix multiplier mxn.Genera las señales necesarias para la implementación del protocolo de sincronización con los módulos generador y consumidor, así como las señales que permiten la correcta ejecución de las operaciones internas del módulo. Se ha implementado como una máquina de estados finitos de tipo Moore con reset síncrono y 5 estados. La siguiente figura 6.29 ilustra el diagrama de transición de estados y posteriormente se describen cada uno de sus estados.

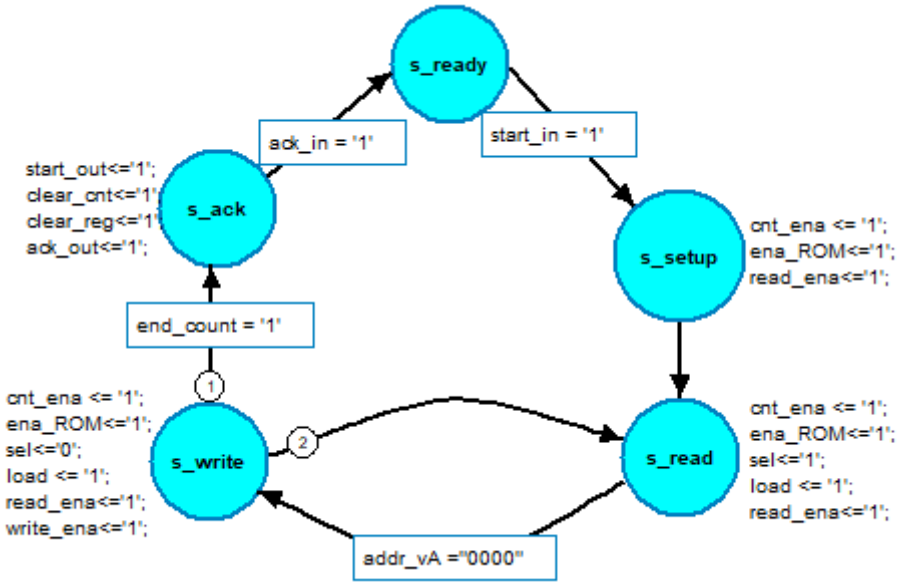

Figura 6.29: FSM de Matrix multiplier mxn

- Estado ready: Matrix multiplier mxn está disponible para realizar un nuevo producto matricial. Espera la llegada de la señal start_in, enviada por el módulo generador, para cambiar al estado start.
- Estado setup: comienzo de ejecución. Se prepara el contador, la memoria ROM y se capacita la lectura de la memoria de entrada para comenzar la operación. El siguiente estado es read.
- Estado read: se van leyendo los datos y operando simultáneamente. En el momento que la Addr lookup marque la dirección ”0000”significa que ha terminado de operar una componente del vector resultante C y se pasa al estado write.
- Estado write: escritura en la memoria de salida del resultado de Matrix multiplier. Por otro lado se sigue operando si todavía no se ha terminado el producto matricial, la señal end_count, fin de cuenta del contador, marca el fin de la operación; si se activa se hace una transición al estado ack, en caso contrario se vuelve al estado read.

- Estado ack: indica el fin de la operación. Matrix multiplier mxn CU activa por puerto de salida las señales ack_out y start_out, informando al módulo consumidor la disponibilidad de nuevos datos en la memoria de salida y al módulo generador de la lectura de los datos de la memoria de entrada. Se espera a que el módulo consumidor envíe la señal ack_in para volver al estado ready para llevar a cabo una nueva ejecución cuando se precise.

6.3.16. Matrix multiplier nx1

Este módulo es un componente de Gate nx1 y se encarga de que se lleve a cabo un producto matricial con la siguiente operación:

$$vC = vA \cdot MB \tag{6.15}$$

donde:

- vA: vector A, que corresponde con la entrada x_t o con el estado oculto h_{t-1} de Gate nx1, donde x_t es un vector de 3 componentes, y h_{t-1} solo una componente .
- MB: matriz B, se trata de la matriz de pesos W o bien la matriz U.
  La matriz W tiene la dimensión de 3x1:

  $$W = [\ \text{nº de características de } x_t, \text{ nº de estados ocultos }\ ] \tag{6.16}$$

  En cambio la matriz U tiene la dimensión de 1x1:

  $$U = [\ \text{nº de estados ocultos, nº de estados ocultos }\ ] \tag{6.17}$$

- vC: vector resultante C, que tendrá solo una componente.

Dado que la Matriz B está formada por solo una columna hace que la operación sea más sencilla y que el vector resultante C solo conste de una componente, sin necesidad de hacer uso de una tabla de verdad. A continuación se expone su estructura 6.30 e interfaz 6.28:

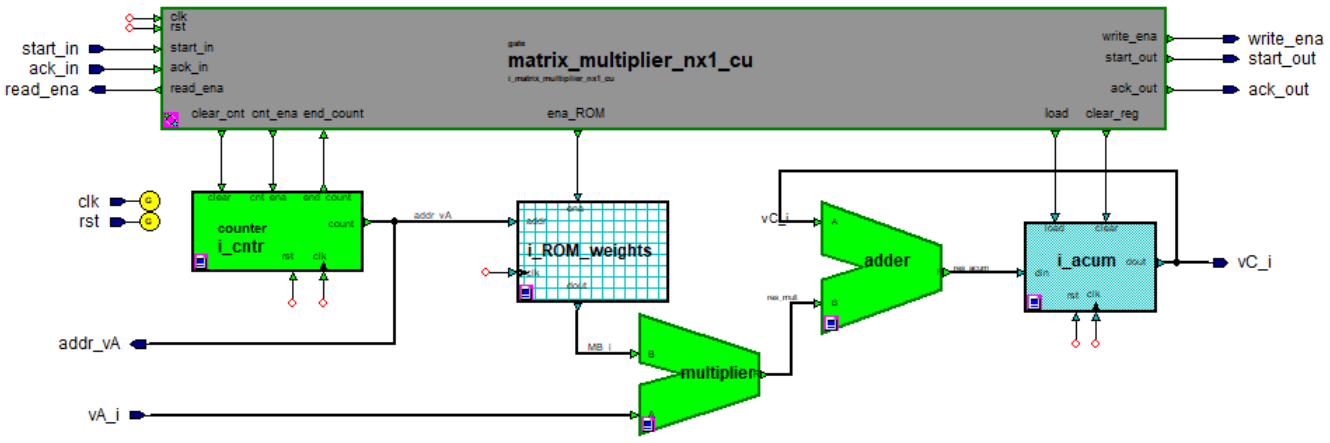

Figura 6.30: Diagra de bloques de Matrix Multiplier nx1

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Matrix multiplier nx1 que ya tiene los datos disponibles en la memoria de entada y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Matrix multiplier nx1 informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Matrix multiplier nx1 informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Matrix multiplier nx1 que ya puede escribir en la memoria de salida.
read_ena	1 bit	Salida	Capacitación de lectura de un dato de la entrada vA.
addr_vA	4 bits	Salida	Dirección de memoria del dato vA_i.
vA_i	c_data_width	Entrada	Dato de entrada vA_i.
write_ena	1 bit	Salida	Capacitación de escritura de un dato de vC.
vC_i	c_data_width	Salida	Dato de salida vC_i.

Tabla 6.28: Interfaz de Matrix multiplier nx1

Matrix multiplier nx1 está formado por los siguientes componentes de los que hacemos una breve descripción:

- Cntr: contador que indexa las componentes de los operandos.
- ROM weights: memoria ROM donde está almacenada la matriz B.
- Multiplier: realiza el producto de las componentes de las matrices.
- Adder: sumador de los productos y la cuenta acumulada.
- Acum: registro donde se almacena el resultado del vector resultante C.
- Matrix multiplier nx1 CU: unidad de control de Matrix multiplier nx1. En la siguiente subsección hablamos en detalle sobre ella.

6.3.16.1. Matrix multiplier nx1 CU

Es la unidad de control de Matrix multiplier nx1. Se encarga de la comunicación con el exterior llevando a cabo el protocolo de la forma esperada y gestionar que se realice el cálculo de la forma correcta así como su posterior escritura del resultado. Se ha implementado como una máquina de estados finitos de tipo Moore con reset síncrono y 6 estados.

La siguiente figura 6.31 ilustra el diagrama de transición de estados y posteriormente se describen cada uno de sus estados.

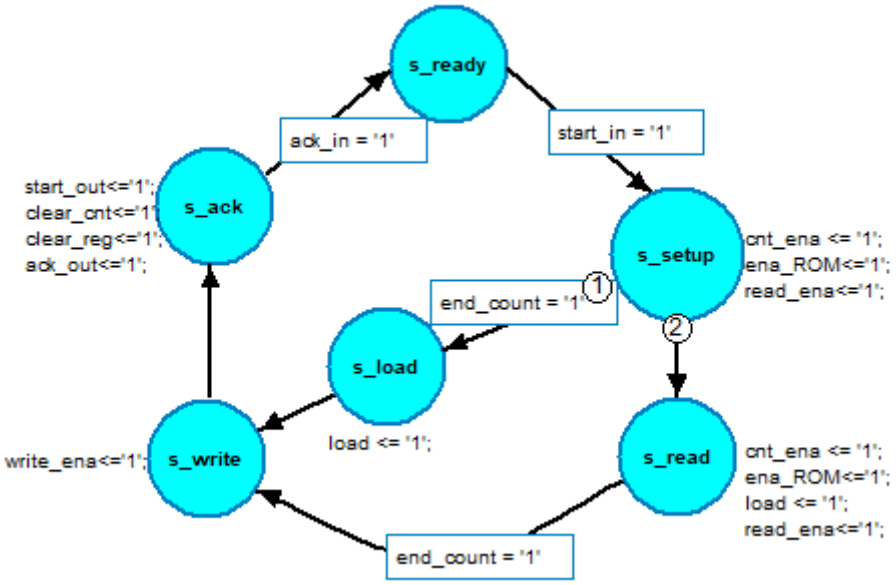

Figura 6.31: FSM de Matrix multiplier nx1

- Estado ready: Matrix multiplier nx1 está disponible para realizar un nuevo producto matricial. Espera la llegada de la señal start_in, enviada por el módulo generador, para cambiar al estado start.
- Estado setup: comienzo de ejecución. Se capacita el contador, la memoria ROM y también la lectura de la memoria de entrada para comenzar la operación. En el caso de que tanto vA y MB solo estuvieran formados por un elemento, se activaría la señal end_count, fin de cuenta del contador y el siguiente estado sería load; si no, el siguiente estado es read.
- Estado load: se activa la señal load del registro Acum, para almacenar el resultado y se pasa al estado write.
- Estado read: lectura, se van leyendo los datos y operando simultáneamente. En el momento que la señal end_count se active, fin de cuenta del contador, ha terminado de operar el vector resultante C, y se pasa al estado write.
- Estado write: escritura en la memoria de salida del resultado de Matrix multiplier. Se hace una transición al estado ack.
- Estado ack: indica el fin de la operación. Matrix multiplier nx1 CU activa por puerto de salida las señales start_out y ack_out, la primera informa al módulo consumidor la disponibilidad de nuevos datos en la memoria de salida y la segunda al módulo generador de la lectura de los datos de la memoria de entrada. Se espera a que el módulo consumidor envíe la señal ack_in para volver al estado ready para llevar a cabo una nueva ejecución cuando se precise.

6.3.17. Sigmoid

Realiza la función sigmoidea:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{6.18}$$

Debido a la dificultad de implementar en hardware series exponenciales como la función dada, se suele proponer como posible aproximación a la solución la implementación mediante tablas lookup pero estas requieren una gran cantidad de recursos hardware por lo que buscamos otra alternativa para llevar a cabo esta función. Este módulo consta de dos componentes distintos para calcular su resultado Cordic sigmoid y Taylor sigmoid. Dependiendo del valor de la entrada hace uso de uno u otro. A continuación se expone su estructura 6.32 e interfaz 6.29:

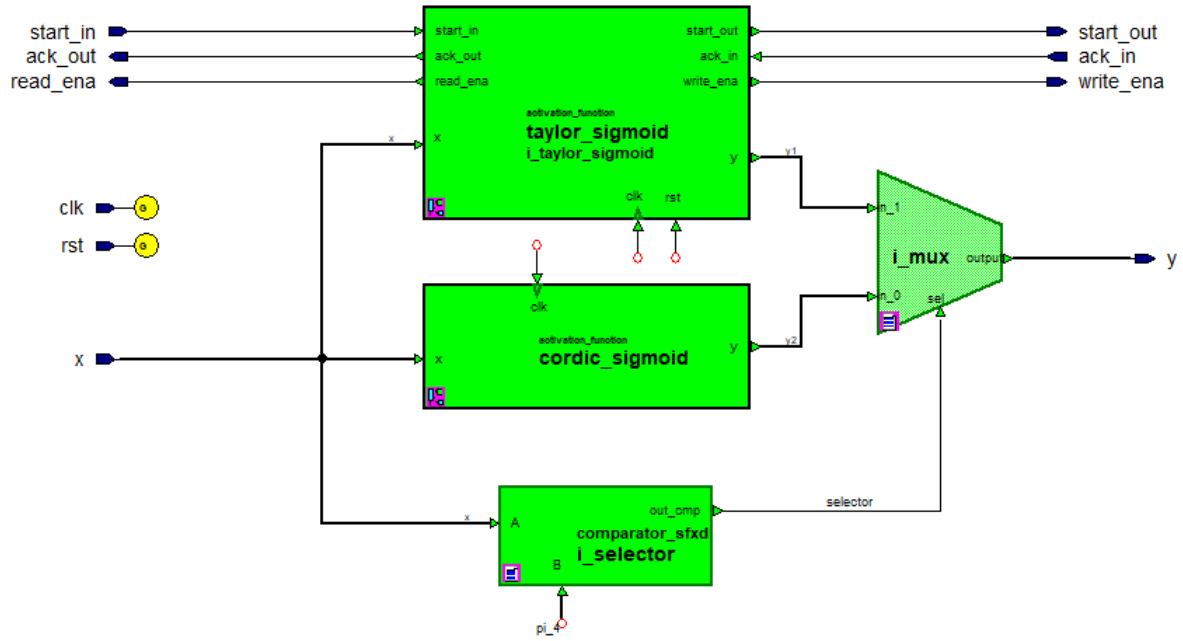

Figura 6.32: Diagrama de bloques de Sigmoid

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Sigmoid que ya tiene los datos disponibles en la memoria de entada y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Sigmoid informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Sigmoid informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Sigmoid que ya puede escribir en la memoria de salida.
read_ena	1 bit	Salida	Capacitación de lectura de un dato de la entrada x.
x	c_data_width	Entrada	Dato de entrada x.
write_ena	1 bit	Salida	Capacitación de escritura de un dato y.
y	c_data_width	Salida	Dato de salida y.

Tabla 6.29: Interfaz de Sigmoid

Sigmoid está formado por los siguientes componentes de los que hacemos una breve descripción:

- Taylor sigmoid: el módulo Taylor sigmoid se encarga de calcular la función sigmoidea si el valor de entrada x está fuera del rango [-pi/4, pi/4].
- CORDIC sigmoid: el módulo CORDIC sigmoid Se encarga de calcular la función sigmoidea si el valor de entrada x está entre el rango [-pi/4, pi/4].
- Selector: se trata del módulo comparator_sfxd que realiza la comparación del valor del dato de entrada x con el valor pi/4. Su salida esta conectada al puerto de selección de Mux, con lo que su salida determina el valor del puerto y.
- Mux: es un multiplexor 2 a 1 con sus entradas conectadas a las salidas de los módulos Taylor y CORDIC sigmoid y su salida conectada directamente al puerto de salida y.

6.3.18. Taylor sigmoid

Calcula la función sigmoidea mediante la siguiente ecuación:

$$\begin{aligned}\sigma(x) =& 0{,}924142 + 0{,}07010(x-2{,}5) \\ & - 0{,}02973(x-2{,}5)^2 + 0{,}00677(x-2{,}5)^3 \\ & - 0{,}000393(x-2{,}5)^4 - 0{,}0003(x-2{,}5)^5\end{aligned} \tag{6.19}$$

Obtenemos esta función mediante un desarrollo de Taylor de orden 5 centrado en el punto x=2.5 consiguiendo abarcar el rango entre [0, 5]. En el caso de que el valor de x sea negativo obtenemos el valor de la función calculando el valor de x positivo y realizamos la resta de uno menos el resultado, con lo que obtenemos el valor correspondiente de la función para el valor de x con signo negativo:

$$\sigma(x) = y \tag{6.20}$$

$$1 - y = \sigma(-x) \tag{6.21}$$

A continuación se expone su estructura 6.33 e interfaz 6.30:

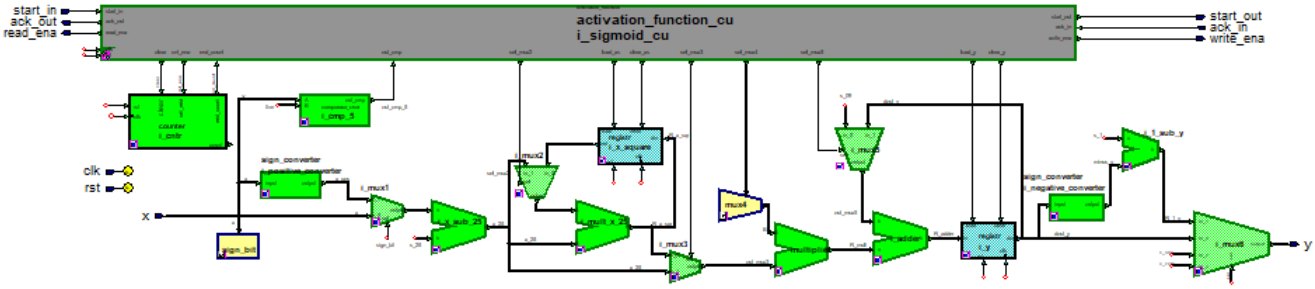

Figura 6.33: Diagrama de bloques de Taylor sigmoid

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Taylor sigmoid que ya tiene los datos disponibles en la memoria de entada y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Taylor sigmoid informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Taylor sigmoid informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Taylor sigmoid que ya puede escribir en la memoria de salida.
read_ena	1 bit	Salida	Capacitación de lectura de un dato de la entrada x.
x	c_data_width	Entrada	Dato de entrada x.
write_ena	1 bit	Salida	Capacitación de escritura de un dato y.
y	c_data_width	Salida	Dato de salida y.

Tabla 6.30: Interfaz de Taylor sigmoid

Taylor sigmoid se forma por los siguientes componentes descritos brevemente a continuación:

- Cnt: contador que cuenta el numero de datos de la entrada x.
- Cmp 5: comparator sfx que compara el valor de entrada x con el valor 5.
- Sign bit: recoge el bit de signo del valor de entrada x.
- Positive converter: componente Sign converter. En el caso de que la entrada x tenga un valor negativo se convierte a positivo.
- Mux[0-6]: contiene un total de 6 multiplexores que facilitan llevar a cabo el cálculo con el mínimo número de componentes.
- Adder[0-3]: contiene tres sumadores. Uno solo se encarga de realizar la resta (x – 2.5). Otro se de realizar la resta (1 - y). El ultimo encarga de las demás sumas de la ecuación.
- Multiplier x2: contiene dos multiplicadores. Uno se encarga exclusivamente de realizar las potencias de (x – 2.5). El otro efectúa los demás productos de la operación.
- X square: registro donde se almacenan las potencias de (x – 2.5).
- Y: registro donde se almacena el resultado de las demás operaciones.

- Negative converter: componente sign converter. Convierte el resultado y en negativo. Este valor solo se utiliza en el caso de que el bit de signo recogido por sign bit sea negativo
- Taylor sigmoid CU: módulo Activation function CU es la unidad de control del módulo. En la siguiente subsección hablamos de ella en detalle.

6.3.18.1. Activation function CU

Esta unidad controla los módulos Sigmoid y Tanh. se encarga de implementar el protocolo de sincronización con los módulo generador y consumidor puesto que se encarga de la comunicación con el exterior en estos llevando a cabo el protocolo de la forma esperada. También gestiona que se realice el cálculo de la forma correcta así como su posterior escritura del resultado en Taylor sigmoid y Taylor tanh. Se ha implementado como una máquina de estados finitos de tipo Moore con reset síncrono y 10 estados. La siguiente figura 6.34 ilustra el diagrama de transición de estados y posteriormente se describen cada uno de sus estados.

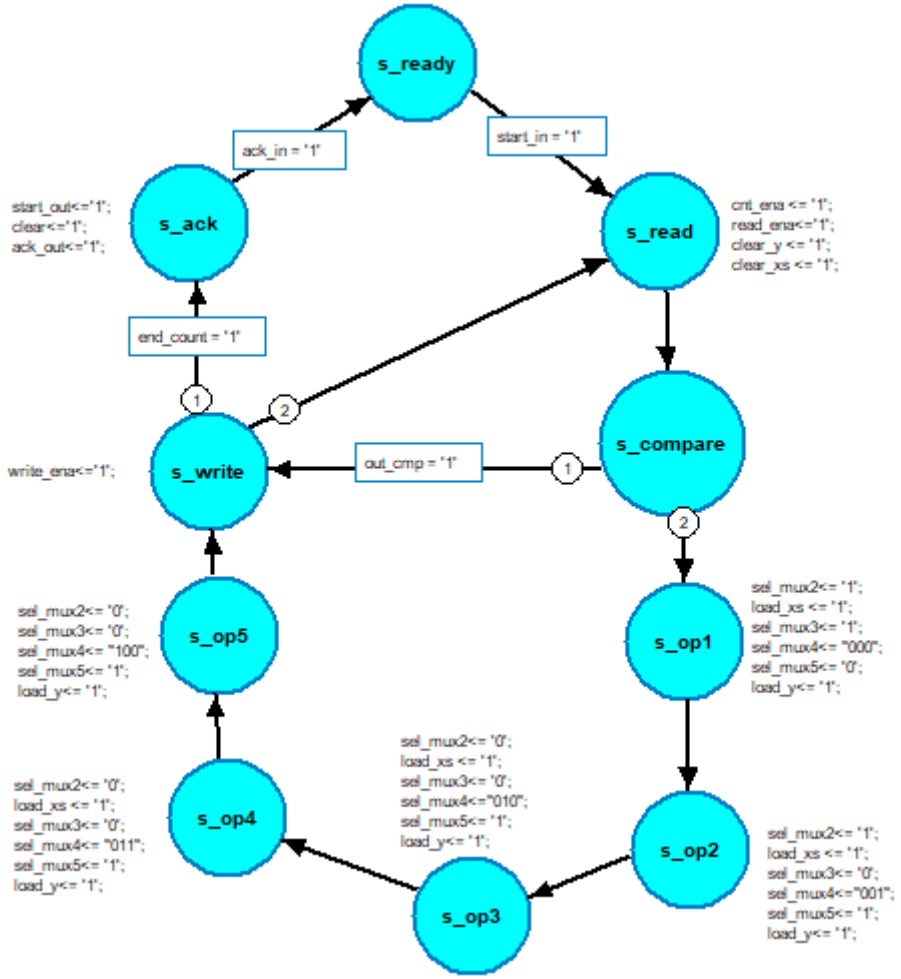

Figura 6.34: FSM de Activation function CU

- Estado ready: el módulo está disponible para realizar un nuevo cálculo de la función de activación. Espera la llegada de la señal start_in, enviada por el módulo generador, para cambiar al estado read.
- Estado read: se realiza la lectura de un dato de la memoria de entrada y se pasa al estado copmpare.
- Estado compare: estado donde se realiza la comparación del dato de entrada, en caso de que el resultado de la comparación sea '1' el siguiente estado es write en caso contrario es op1.
- Estado op1: primer estado del cálculo de la función de Taylor. Se pasa al estado op2.
- Estado op2: segundo estado del cálculo de la función de Taylor. Se pasa al estado op3.
- Estado op3: tercer estado del cálculo de la función de Taylor. Se pasa al estado op4.
- Estado op4: cuarto estado del cálculo de la función de Taylor. Se pasa al estado op5.
- Estado op5: quinto estado del cálculo de la función de Taylor. Se pasa al estado write.
- Estado write: escritura en la memoria de salida del módulo, Activation function CU activa la señal write_ena. Si llega la señal end_count, fin de cuenta del contador, se hace la transición al estado ack; en caso contrario se vuelve al estado read para calcular un nuevo dato.
- Estado ack: estado en el Activation function CU informa al módulo generador de la lectura de todos los datos de la memoria de entrada, mediante la señal ack_out; y también se informa al módulo consumidor de la escritura de nueva información en la memoria de salida del módulo, con la señal start_out. Se espera a que el módulo consumidor envíe la señal ack_in para pasar al estado ready.

6.3.19. CORDIC sigmoid

Módulo donde se calcula la función sigmoidea mediante el componente CORDIC con la siguiente ecuación:

$$\sigma(x) = \frac{1}{1 + cosh(x) - sinh(x)} \tag{6.22}$$

A continuación se explica la obtención de la misma.

La función sigmoidea viene dada por la expresión 6.18. Por otro lado, medainte CORDIC se pude obtener:

$$sinh(x) = \frac{e^{x} - e^{-x}}{2} \tag{6.23}$$

$$cosh(x) = \frac{e^{x} + e^{-x}}{2} \tag{6.24}$$

si restamos el coseno menos el seno obtenemos:

$$cosh(x) - sinh(x) = \frac{-e^{x} + e^{-x} + e^{x} + e^{-x}}{2} = e^{-x} \tag{6.25}$$

finalmente obtenemos 6.22 sustituyendo en 6.18

Gracias al CORDIC podemos calcular dicha función pero tiene la limitación de que el valor de entrada x tiene que estar en el rango [-pi/4, pi/4]. Fuera de ese rango el componente no es capaz de calcular el seno y coseno hiperbólicos. A continuación se expone su estructura 6.35 e interfaz 6.31:

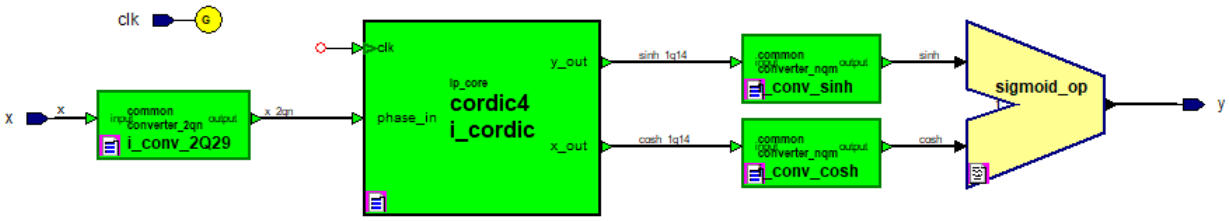

Figura 6.35: Diagrama de bloques de CORDIC sigmoid

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
x	c_data_width	Entrada	Dato de entrada x.
y	c_data_width	Salida	Dato de salida y.

Tabla 6.31: Interfaz de CORDIC sigmoid

CORDIC sigmoid se forma con los siguientes componentes descritos brevemente a continuación:

- Conv 2Q29: componente converter 2qn que codifica los valores en el formato 2q29.
- CORDIC: Componente CORDIC. Se trata de la IP de Xilinx configurada para el cálculo de seno y coseno hiperbólicos dado un ángulo 'phase_in'. El dato de entrada 'phase_in' se trata del ángulo hiperbólico que representa la superficie que hay bajo el vector(x,y). Tiene un tamaño de 32 bits, y debe estar codificado en el formato punto fijo 2Qn. Devuelve el resultado por los puertos 'x_out' e 'y_out', que corresponden con el coseno y seno hiperbólicos en la codificación punto fijo 1Qn.
- Conv sinh: componente converter nqm que convierte el resultado del seno hiperbólico dado por el CORDIC en el formato 1qn a la forma con la que trabaja el sistema nqm.
- Conv cosh: componente converter nqm que convierte el resultado del coseno hiperbólico dado por el CORDIC en el formato 1qn a la forma con la que trabaja el sistema nqm.
- Sigmoid op: componente encargado de realizar la siguiente ecuación para hallar la función sigmoidea:

$$sigmoid(x) = 1/\left(1 + cosh(x) - sinh(x)\right). \tag{6.26}$$

  recibiendo por puerto de entrada el seno y coseno hiperbólicos.

6.3.20. Tanh

Módulo donde se realiza la función de la tangente hiperbólica:

$$tanh(x) = \frac{\sinh(\mathrm{x})}{\cosh(\mathrm{x})} \tag{6.27}$$

Para llevar a cabo la función seguimos la misma idea que en el módulo Sigmoid. Este módulo consta de dos componentes distintos para calcular su resultado CORDIC tanh y Taylor tanh que dependiendo del valor de la entrada se hace uso de uno u otro. A continuación se detalla su estructura 6.36 e interfaz 6.32.

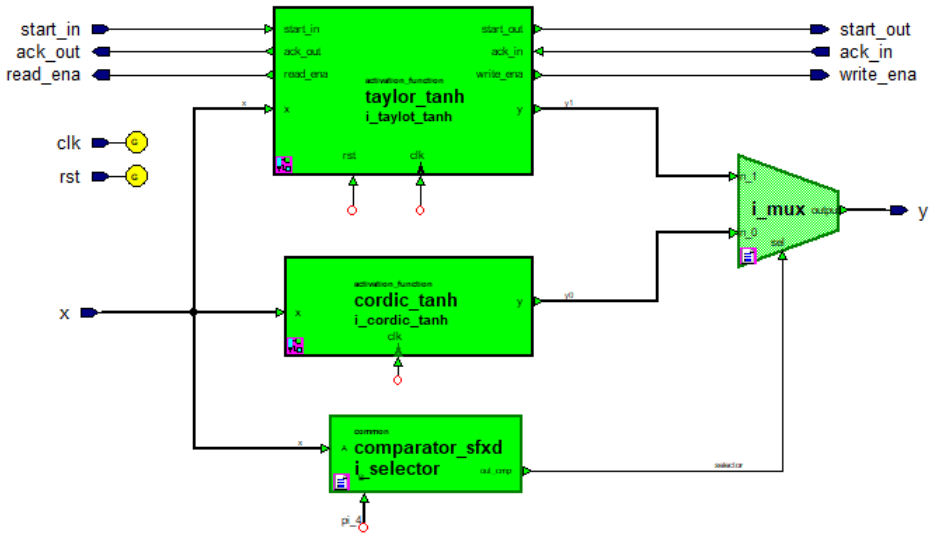

Figura 6.36: Diagrama de bloques de Tanh

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Tanh que ya tiene los datos disponibles en la memoria de entada y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Tanh informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Tanh informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Tanh que ya puede escribir en la memoria de salida.
read_ena	1 bit	Salida	Capacitación de lectura de un dato de la entrada x.
x	c_data_width	Entrada	Dato de entrada x.
write_ena	1 bit	Salida	Capacitación de escritura de un dato y.
y	c_data_width	Salida	Dato de salida y.

Tabla 6.32: Interfaz de Tanh

Tanh se forma por los siguientes componentes que se describen brevemente a continuación:

- Taylor tanh: módulo Taylor tanh que se encarga de calcular la función tangente hiperbólica si el valor de entrada x está fuera del rango [-pi/4, pi/4].
- CORDIC tanh: módulo CORDIC tanh que se encarga de calcular la función tangente hiperbólica si el valor de entrada x está entre el rango [-pi/4, pi/4].

- Selector: se trata del módulo comparator_sfxd que realiza la comparación del valor del dato de entrada x con el valor pi/4. Su salida esta conectada al puerto de selección de Mux con lo que su salida determina el valor del puerto y.
- Mux: multiplexor 2 a 1 donde sus entradas están conectadas a las salidas de los módulos Taylor y CORDIC tanh y su salida conectada directamente al puerto de salida y.

6.3.21. Taylor tanh

Calcula la función de la tangente hiperbólica mediante la siguiente ecuación:

$$\begin{aligned}\tanh(x) = &- 0{,}046307446320614(x)^5 + 0{,}259249358467676(x)^4 \\ &- 0{,}470318523869344(x)^3 - 0{,}001316120216492(x)^2 \\ &+ 1{,}028101951387410(x) - 0{,}007815063492871\end{aligned} \tag{6.28}$$

Obtenemos esta función mediante un desarrollo de Taylor de orden 5 centrado en el punto x=1 consiguiendo abarcar el rango entre [0, 2].En el caso de que el valor de x sea negativo se obtiene el valor de la función calculando el valor de x positivo e invertimos el signo al resultado, obteniendo el valor correspondiente de la función para el valor de -x. Esto se debe a que la función de la tangente hiperbólica es impar. Una función impar, desde un punto de vista geométrico, se dice que tiene una simetría rotacional con respecto al origen de coordenadas, lo que quiere decir que su gráfica no se altera si se rota 180º. Por lo tanto, satisface la siguiente relación:

$$\tanh(-x) = -\tanh(x) \tag{6.29}$$

A continuación se expone su estructura 6.37 e interfaz 6.33:

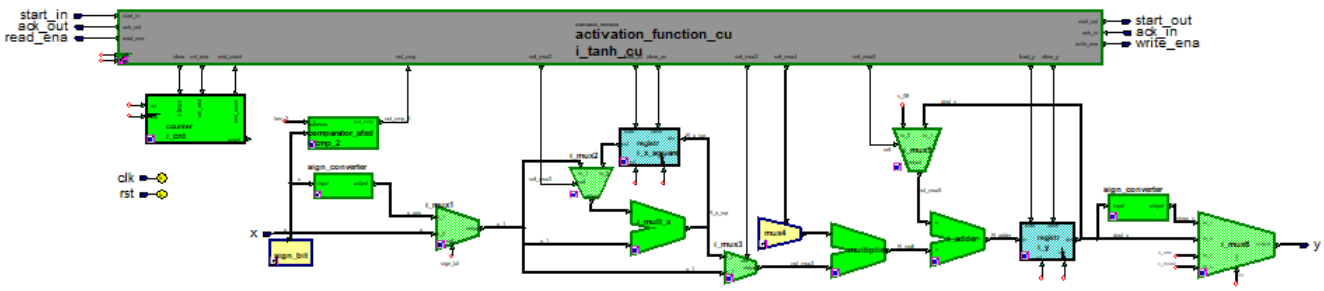

Figura 6.37: Diagrama de bloques de Taylor tanh

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
rst	1 bit	Entrada	Reset del sistema.
start_in	1 bit	Entrada	El módulo generador indica a Taylor tanh que ya tiene los datos disponibles en la memoria de entada y que por lo tanto puede empezar la ejecución.
ack_out	1 bit	Salida	Taylor tanh informa al módulo generador que ya ha leído todos los datos de la memoria de entrada.
start_out	1 bit	Salida	Taylor tanh informa al módulo consumidor que ya ha escrito todos los datos en la memoria de salida.
ack_in	1 bit	Entrada	El módulo consumidor indica a Taylor tanh que ya puede escribir en la memoria de salida.
read_ena	1 bit	Salida	Capacitación de lectura de un dato de la entrada x.
x	c_data_width	Entrada	Dato de entrada x.
write_ena	1 bit	Salida	Capacitación de escritura de un dato y.
y	c_data_width	Salida	Dato de salida y.

Tabla 6.33: Interfaz de Taylor tanh

Taylor tanh se forma con los siguientes componentes que se describen brevemente a continuación:

- Cnt: contador que cuenta el numero de datos de la entrada x.
- Cmp 2: comparator sfx que compara el valor de entrada x con el valor 2.
- Sign bit: recoge el bit de signo del valor de entrada x.
- Positive converter: componente Sign converter. En el caso de que la entrada x tenga un valor negativo, se convierte a positivo.
- Mux[0-6]: contiene un total de 6 multiplexores, que facilitan llevar a cabo el cálculo, con el mínimo número de componentes.
- Adder: contiene un sumador que realiza las sumas de la ecuación.
- Multiplier x2: contiene dos multiplicadores. Uno se encarga exclusivamente de realizar las potencias de x. El otro efectúa los demás productos de la operación.
- X square: registro donde se almacenan las potencias de x.
- Y: registro donde se almacena el resultado de las demás operaciones.

- Negative converter: componente sign converter. Convierte el resultado y en negativo. Este valor solo se utiliza en el caso de que el bit de signo, recogido por sign bit, sea negativo
- Taylor sigmoid CU: módulo Activation function CU, unidad de control del módulo.

6.3.22. CORDIC tanh

Calcula la función de la tangente hiperbólica mediante el componente CORDIC el cual calcula el seno y coseno hiperbólicos y después mediante un divisor obtenemos el resultado.

Gracias al CORDIC podemos calcular la función, pero tiene la limitación de que el valor de entrada x tiene que estar en el rango [-pi/4, pi/4]. Fuera de ese rango el componente no es capaz de calcular el seno y coseno hiperbólicos. A continuación se expone su estructura 6.38 e interfaz 6.34:

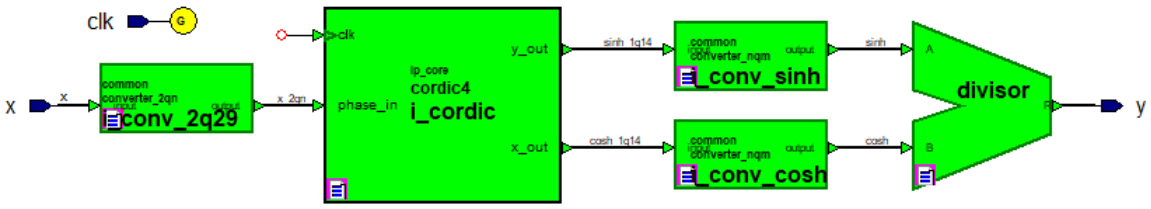

Figura 6.38: Diagrama de bloques de CORDIC tanh

Puerto	Tamaño	Tipo	Descripción
clk	1 bit	Entrada	Reloj del sistema.
x	c_data_width	Entrada	Dato de entrada x.
y	c_data_width	Salida	Dato de salida y.

Tabla 6.34: Interfaz de CORDIC tanh

CORDIC tanh está formado por los siguientes componentes de los que hacemos una breve descripción:

- Conv 2Q29: componente converter 2qn, que codifica los valores en el formato 2q29.
- CORDIC: Componente CORDIC. Se trata de la IP de Xilinx configurada para el cálculo de seno y coseno hiperbólicos dado un ángulo 'phase_in'. El dato de entrada 'phase_in', se trata del ángulo hiperbólico que representa la superficie que hay bajo el vector(x,y). Tiene un tamaño de 32 bits, y debe estar codificado en el formato punto fijo 2Qn. Devuelve el resultado por los puertos 'x_out' e 'y_out', que corresponden con el coseno y seno hiperbólicos, en la codificación punto fijo 1Qn.

- Conv sinh: componente converter nqm, convierte el resultado del seno hiperbólico dado por el CORDIC en el formato 1qn a la forma con la que trabaja el sistema nqm.
- Conv cosh: componente converter nqm, convierte el resultado del coseno hiperbólico dado por el CORDIC en el formato 1qn a la forma con la que trabaja el sistema nqm.
- Divisor: Componente encargado de realizar la división del seno entre el coseno hiperbólicos brindados por CORDIC.

Parte IV

RESULTADOS EXPERIMENTALES, SÍNTESIS DEL HARDWARE Y CONCLUSIONES

Capítulo 7

Resultados experimentales

En este apartado hablamos de los resultados obtenidos por nuestra red neuronal en las pruebas realizadas. Primero explicaremos los resultados obtenidos en las predicciones con varios modelos realizados en Python y terminaremos hablando de los resultados obtenidos en las simulaciones del diseño hardware. Las pruebas realizadas al diseño hardware consisten en comparar los resultados obtenidos por este con los de Python, con lo que comprobaremos si el diseño hardware funciona como se espera.

Resultados en las predicciones

Las predicciones de glucemia a 30 minutos vista calculadas por los modelos implementados con Keras no son totalmente acertadas, sin embargo obtenemos un coeficiente de error de 17.17 RMSE, que se ajusta a los resultados del ranking 7.1 del concurso BGLP Challenge organizado por la Universidad de Ohio en 2020.

Official ranking (April 26, 2020).

	30 min		60 min				
Paper ID	RMSE	MAE	RMSE	MAE	**Overall**	Online	Personalized
13	18.22	12.83	31.66	23.60	86.31	No	Yes
6	19.21	13.08	31.77	23.09	87.15	No	Yes
16	18.34	13.37	32.21	24.20	88.12	No	Yes
15	19.05	13.50	32.03	23.83	88.41	No	Yes
1	18.23	14.37	31.10	25.75	89.45	No	No
14	19.37	13.76	32.59	24.64	90.36	Yes	Yes
7	19.60	14.25	34.12	25.99	93.96	No	Yes
9	20.03	14.52	34.89	26.41	95.85	Yes	Yes

Figura 7.1: Ranking del desafío BGLP por la Universidad de Ohio [32]

A continuación exponemos nuestros 3 modelos: en el primero, la capa oculta consta de una celda LSTM con tres estados ocultos; en el segundo, la celda LSTM de la capa oculta tiene cinco estados ocultos; y en el tercero la capa oculta tiene quince estados ocultos.

Para comprobar la validez de los mismos hemos calculado la raíz del error cuadrático medio (RMSE) en los resultados obtenidos y esperados. En la validación del modelo donde la red predice los datos del conjunto de test obtenemos los siguientes resultados:

- Modelo con tres estados ocultos: obtenemos un error de 17.17 RMSE.
- Modelo con cinco estados ocultos: obtenemos un error de 17.44 RMSE.
- Modelo con quince estados ocultos: obtenemos un error de 17.35 RMSE.

A continuación, mostramos en la gráfica 7.2 los resultados del test realizado al modelo con tres estados ocultos:

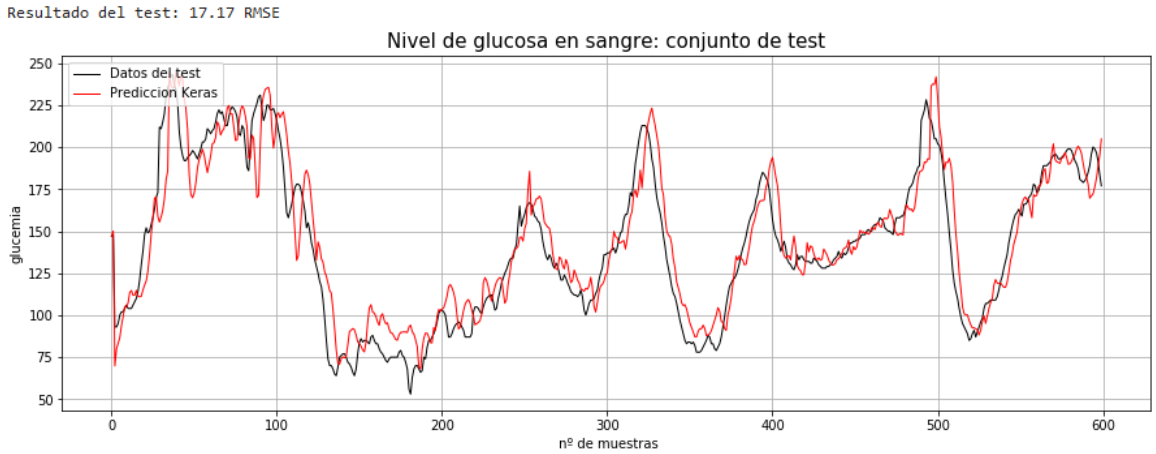

Figura 7.2: Test del modelo Keras con tres estados ocultos

La obtención de estos resultados se debe a la dificultad de realizar un modelo que prediga el nivel de glucosa en sangre para cualquier paciente independientemente del tipo de diabetes que padezca, ya que en nuestra base de datos no se registra el tipo de diabetes que padecen los pacientes, por lo que la red no es consciente de que existen dos tipos de diabetes. También podemos concluir que no solo con la parametrización de la glucemia, la ingesta de hidratos de carbono y el nivel de insulina en sangre es suficiente. El nivel de glucosa en sangre se ve alterado por muchos más factores, como por ejemplo el tipo de insulina con la que el paciente regula su glucemia, su actividad física, el metabolismo del paciente, el estrés, etc [33].

Para reafirmar esta conclusión cabe destacar que realizamos diferentes modelos con Keras, variando la arquitectura de la red como se explicó en el capitulo anterior e incluso con redes más grandes a la escogida obtenemos unos resultados muy parecidos.

Resultados del diseño hardware

Para realizar las pruebas del diseño hardware implementamos un banco de pruebas (test bench) que es capaz de leer los datos de entrada de un fichero y posteriormente escribir los resultados en otro fichero, y así poder realizar comprobaciones de un número importante de datos. En las pruebas utilizamos los datos del conjunto de test, con el que validamos la red de Keras, que tiene un total de 18701 muestras.

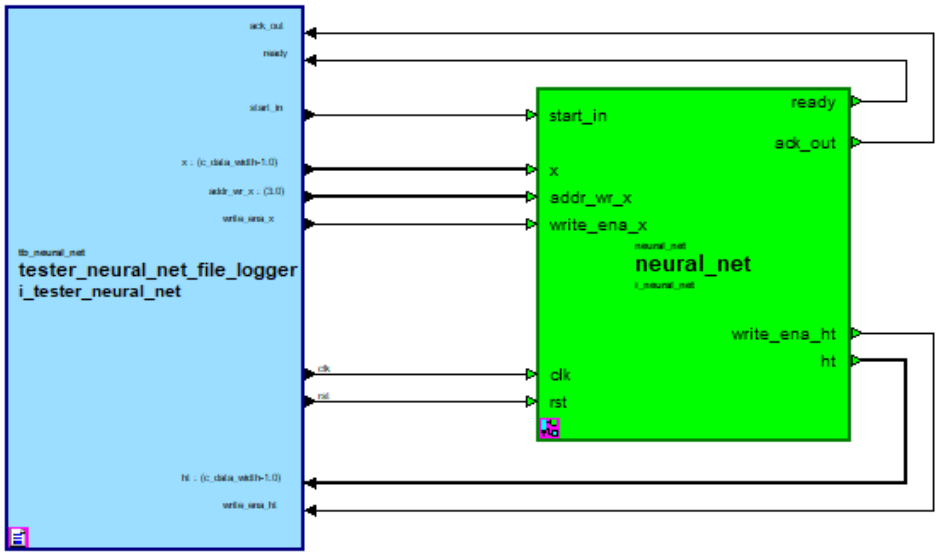

Figura 7.3: test bench neural net

Las pruebas que a continuación vamos a exponer las realizamos con la red explicada en el capitulo anterior, la cual tiene una arquitectura de tres capas. Una capa de entrada donde se almacenan las muestras de entrada, una capa oculta que consta de una celda LSTM con tres estados ocultos y una capa de salida que tiene una celda LSTM con solo un estado oculto.

El objetivo de estas pruebas es comparar los resultados obtenidos por la red hardware y la Keras utilizando como entradas las mismas, recogidas en nuestro conjunto de test.

En la primera prueba el sistema tenía las siguientes características, por un lado el bus de datos tenía un ancho de 16 bits y el formato de los datos de punto fijo 3Q12; por el otro las funciones de activación Tanh y Sigmoid no utilizaban el IP core CORDIC de Xilinx para realizar sus cálculos, sólo utilizaban las aproximación de Taylor, de orden 5 para la sigmoidea y de orden 3 para la Tanh. En estas pruebas observamos que las primeras predicciones de la red hardware eran muy cercanas a las obtenidas con Keras, pero a medida de que se iban realizando más predicciones los resultados no coincidían. Después de realizar un análisis exhaustivo sobre los resultados de la simulación, nos dimos cuenta de que Tanh era uno de los problemas, ya que la función implementada era poco precisa cuando el valor de la entrada estaba próximo a 0, dando resultados como:

tanh(0.0285) = - 0.02, da un valor negativo, mientras que debería ser positivo.
tanh(0.051) = 0.007, debería ser 0.05.

Parecen errores muy pequeños, pero no lo son, estos errores se van acumulando, y al final producen resultados muy dispares.

Como solución decidimos implementar una aproximación de Taylor de orden 5 para la función tangente hiperbólica, con el objetivo de que esta fuera más precisa y paliar el error.

Nos dispusimos a volver a realizar una nueva prueba, y sí, los resultados de Tanh eran más precisos, pero no eran suficientemente buenos, a continuación expongo el fallo más llamativo del calculo de la tangente hiperbólica con la nueva ecuación de la interpolación de Taylor:

$$\begin{aligned}\tanh(x) = &- 0{,}046307446320614(x)^5 + 0{,}259249358467676(x)^4 \\ &- 0{,}470318523869344(x)^3 - 0{,}001316120216492(x)^2 \\ &+ 1{,}028101951387410(x) - 0{,}007815063492871\end{aligned} \tag{7.1}$$

Como se puede observar en el último monomio se realiza la resta:

$$1{,}028101951387410(x) - 0{,}007815063492871 \tag{7.2}$$

por lo que seguíamos manteniendo el mismo problema con valores próximos a 0, ya que si x es igual a 0, el resultado de la ecuación es:

$$\tanh(0) = -0{,}007815063492871 \tag{7.3}$$

Entonces en ese momento nuestros tutores nos comentaron la existencia de la IP de Xilinx CORDIC, capaz de calcular funciones hiperbólicas, y nos aconsejaron su implementación. Sin más miramientos nos pusimos a implementar estos componentes tanto para la función sigmoidea como para la tangente hiperbólica. Los nuevos diseños de los módulos Tanh y Sigmoid ya realizaban sus cálculos tanto con CORDIC como con la función de Taylor.

Una vez implementados la nueva versión de estos módulos volvimos a realizar una prueba. Los resultados seguían sin ser buenos, y analizando los resultados de la simulación, nos dimos cuenta de que en la nueva interpolación de taylor de la tangente hiperbólica se producía un desbordamiento en la parte entera cuando la entrada a la función era mayor o igual a 1.52, a continuación explicamos el problema:
En la función se realiza la potencia de x elevado a cinco, por tanto:

$$1{,}52^5 = 8{,}1136812032 \tag{7.4}$$

Recordamos que el sistema trabajaba con un ancho de bus de 16 bits, en el formato 3Q12. Lo que quiere decir que empleábamos solo 3 bits para la representación de la parte entera, en este formato como máximo se pude representar el valor 7.99999.

Decidimos cambiar la representación de los datos a 4Q11, sabiendo que el sistema iba a perder precisión en los cálculos (que era muy necesaria por los resultados que estamos obteniendo). Al cambiar el formato y realizar una nueva simulación observamos otro problema, se producía un underflow en la parte fraccionaria en la interpolación de Taylor de la función sigmoidea con valores de entrada cercanos a 2.5, ya que la ecuación es de la siguiente forma:

$$\begin{aligned} f(x) = &+ 0{,}924142 + 0{,}07010(x - 2{,}5) \\ &- 0{,}02973(x - 2{,}5)^2 + 0{,}00677(x - 2{,}5)^3 \\ &- 0{,}000393(x - 2{,}5)^4 - 0{,}0003(x - 2{,}5)^5 \end{aligned} \tag{7.5}$$

Como se puede observar, en estas operaciones se requiere un mínimo de precisión en la parte fraccionaria, ya que las operaciones pueden devolver valores muy pequeños, y se producía un underflow:

$$0{,}00677(x - 2{,}5)^3 - 0{,}000393(x - 2{,}5)^4 - 0{,}0003(x - 2{,}5)^5 \tag{7.6}$$

Dados estos problemas, la solución que decidimos llevar a cabo fue el aumento del ancho del bus del sistema a una longitud de 32 bits. Representamos los datos en el formato 6Q25, para solventar los problemas de desbordamiento que estábamos teniendo. Volvimos a realizar una nueva prueba, y finalmente obtuvimos los resultados esperados. La prueba consistió en que realizara la predicción de las 664 primeras muestras de entrada del conjunto de test, obteniendo los siguientes resultados:

- Raíz del error cuadrático medio: 0.84
- Media del error absoluto: 0.51
- Media del porcentaje de error: 0.42 %

Valores que consideramos muy aceptables.

A continuación, mostramos en una gráfica la comparativa de las predicciones hechas por el modelo de Keras y la red hardware:

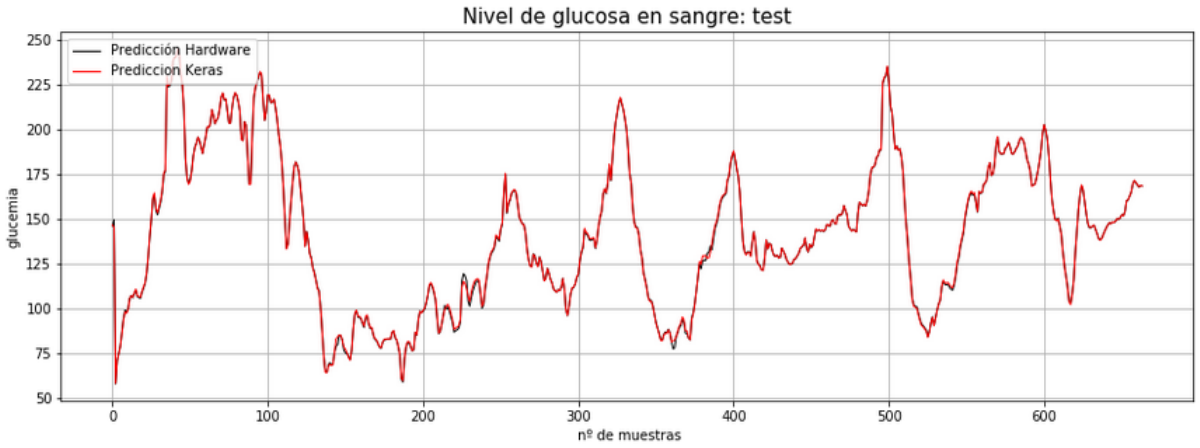

Figura 7.4: Comparativa de los resultados del diseño hardware con los resultados de Keras

Por último queremos concluir, que la red de Keras realiza sus cálculos con el tipo de datos float propio del lenguaje Python, el cual se trata de un tipo de dato con coma flotante y un tamaño de 64 bits. En cambio nuestra red hardware trabaja con datos en punto fijo con un tamaño de 32 bits. Con esto queremos justificar el pequeño error que obtenemos en las pruebas, ya que que la red de Keras tiene un nivel de precisión mucho mayor en sus cálculos comparado con el que nosotros hemos implementado en hardware.

Capítulo 8

Síntesis del hardware

En este capitulo vamos a exponer los resultados de sintetizar nuestro diseño hardware. Las dos primeras secciones muestran dos síntesis, la primera se caracteriza por tener un bus de datos de 16 bits de longitud, en cambio la segunda tiene un bus de 32 bits. En la siguiente sección se hace un análisis de las dos síntesis expuestas, y la última sección habla sobre el camino crítico del diseño hardware.

Sintetizamos nuestro diseño en una FPGA Xilinx Virtex-6 ML-605 Evaluation Board con velocidad -1. La red se compone de dos capas LSTM, la capa oculta consta de tres estados ocultos y la capa de salida solo de uno. A continuación, exponemos los resultados:

8.1. Red neuronal con el bus de 16 bits

Informe sobre el reloj

- Periodo mínimo: 80.742 ns (Frecuencia máxima 12.285 MHz)
- Tiempo mínimo de la entrada de datos antes de un evento de reloj: 1.68 ns
- Tiempo necesario de la salida de datos después de un evento del reloj: 1.493 ns
- Retardo del camino combinacional más largo: No existe.

Recursos utilizados

Los recursos presentados en la siguiente tabla corresponden con el informe antes de la optimización del circuito.

Componente	Cantidad
RAMs	26
Multiplicadores	42
Sumadores	593
Contadores	9
Registros	1306
Comparadores	412
Multiplexores	12268
FSMs	53
Xors	382

Tabla 8.1: Componentes utilizados con un bus de 16 bits

En cuanto al porcentaje de los recursos utilizados del dispositivo después de la optimización:

```
Slice Logic Utilization:
  Number of Slice Registers:              2,251 out of 301,440    1%
  Number of Slice LUTs:                  34,867 out of 150,720   23%
    Number used as logic:                34,576 out of 150,720   22%
    Number used as Memory:                  161 out of  58,400    1%

Slice Logic Distribution:
  Number of occupied Slices:             11,581 out of  37,680   30%
  Number of LUT Flip Flop pairs used:    35,285
    Number with an unused Flip Flop:     33,497 out of  35,285   94%
    Number with an unused LUT:              418 out of  35,285    1%
    Number of fully used LUT-FF pairs:    1,370 out of  35,285    3%

IO Utilization:
  Number of bonded IOBs:                     43 out of     600    7%

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:               0 out of     416    0%
  Number of RAMB18E1/FIFO18E1s:               9 out of     832    1%
  Number of BUFG/BUFGCTRLs:                   1 out of      32    3%
  Number of DSP48E1s:                        42 out of     768    5%
```

Figura 8.1: Recursos utilizados con un bus de 16 bits

Después de realizar el proceso de Place and Route sobre el dispositivo, el informe nos facilita un layout de la FPGA con los recursos usados en el diseño en color azul:

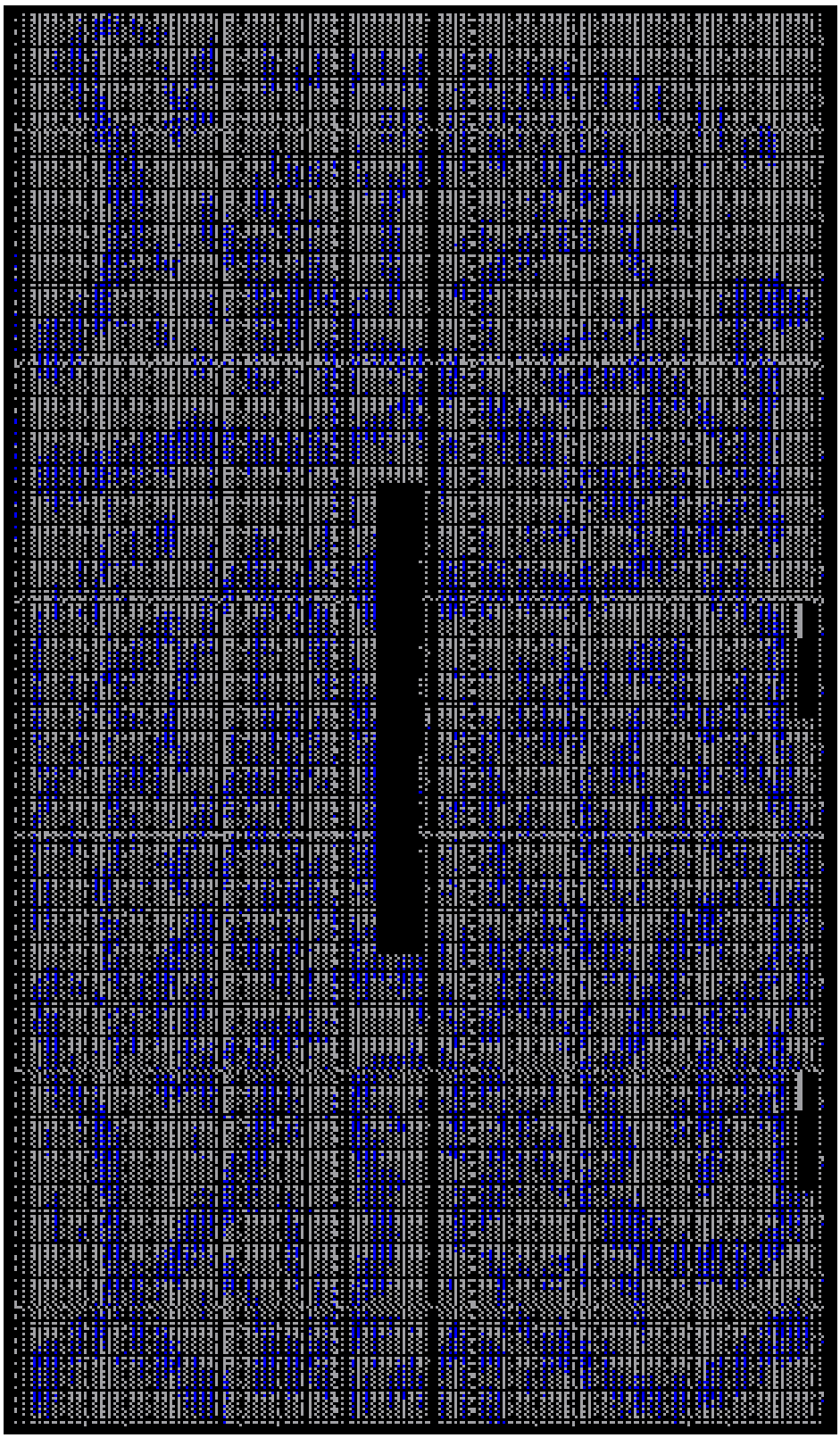

Figura 8.2: Layout del diseño con los recursos utilizados en color azul

8.2. Red neuronal con el bus de 32 bits

Informe sobre el reloj

- Periodo mínimo: 163.747 ns (Frecuencia máxima 6.107 MHz)
- Tiempo mínimo de la entrada de datos antes de un evento de reloj: 1.642 ns
- Tiempo necesario de la salida de datos después de un evento del reloj: 1.572 ns
- Retardo del camino combinacional más largo: No existe.

Recursos utilizados

Los recursos presentados en la siguiente tabla corresponden con el informe antes de la optimización del circuito.

Componente	Cantidad
RAMs	26
Multiplicadores	42
Sumadores	913
Contadores	9
Registros	2586
Comparadores	732
Multiplexores	43948
FSMs	53
Xors	702

Tabla 8.2: Componentes utilizados con un bus de 32 bits

En cuanto al porcentaje de los recursos utilizados del dispositivo después de la optimización:

Slice Logic Utilization:				
Number of Slice Registers:	3,956	out of	301,440	1%
Number of Slice LUTs:	109,423	out of	150,720	72%
Number used as logic:	108,954	out of	150,720	72%
Number used as Memory:	313	out of	58,400	1%
Slice Logic Distribution:				
Number of occupied Slices:	30,668	out of	37,680	81%
Number of LUT Flip Flop pairs used:	109,757			
Number with an unused Flip Flop:	106,561	out of	109,757	97%
Number with an unused LUT:	334	out of	109,757	1%
Number of fully used LUT-FF pairs:	2,862	out of	109,757	2%
Number of unique control sets:	180			
Number of slice register sites lost to control set restrictions:	641	out of	301,440	1%
IO Utilization:				
Number of bonded IOBs:	75	out of	600	12%
Specific Feature Utilization:				
Number of RAMB36E1/FIFO36E1s:	0	out of	416	0%
Number of RAMB18E1/FIFO18E1s:	9	out of	832	1%
Number of BUFG/BUFGCTRLs:	1	out of	32	3%
Number of DSP48E1s:	168	out of	768	21%

Figura 8.3: Recursos utilizados con un bus de 32 bits

Después de realizar el proceso de Place and Route sobre el dispositivo, el informe nos facilita un layout de la FPGA con los recursos usados en el diseño en color rojo:

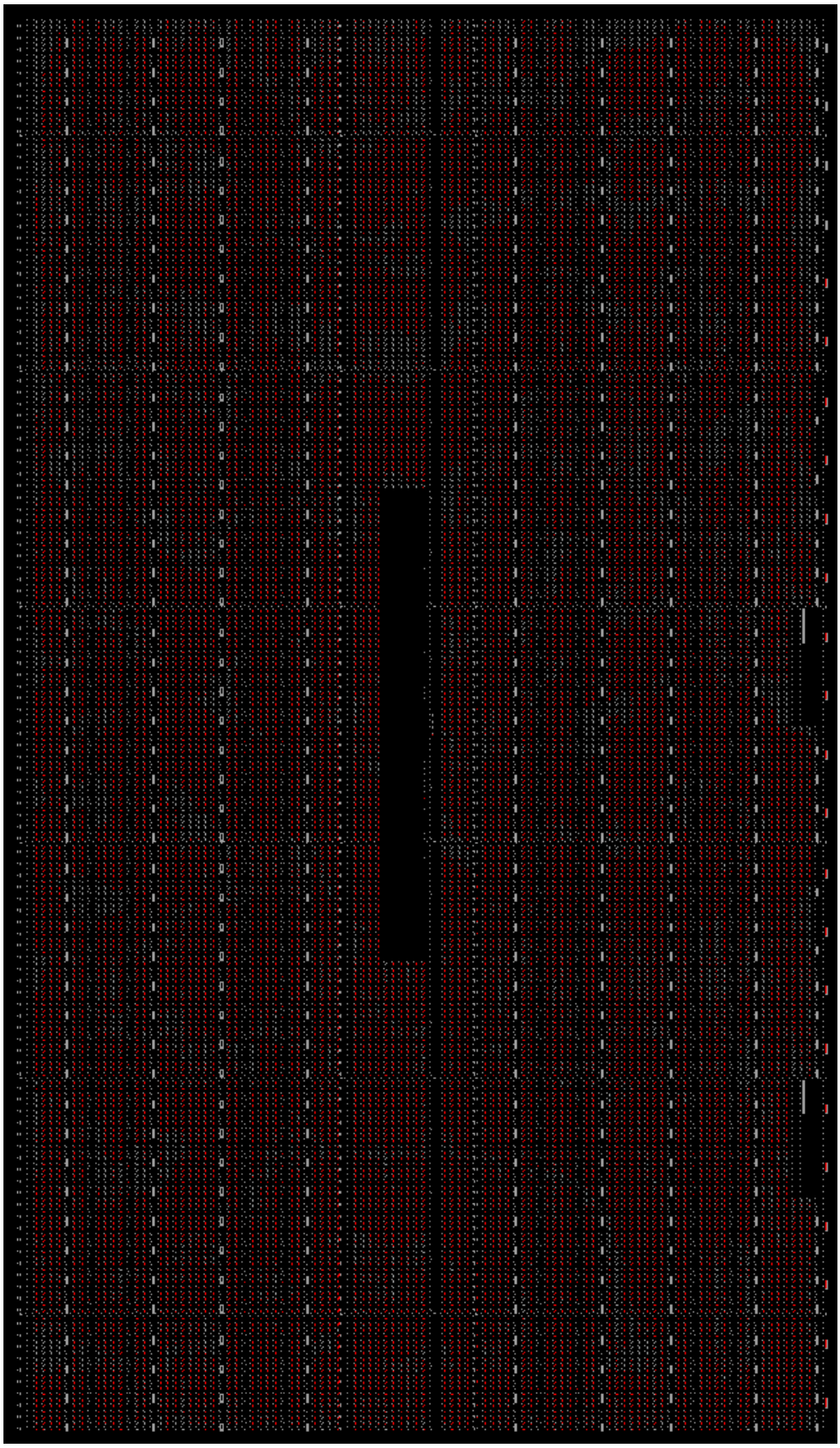

Figura 8.4: Layout del diseño con los recursos utilizados en color rojo

8.3. Análisis de la síntesis

La gran diferencia que se puede observar entre las dos síntesis es el gran incremento de recursos utilizados del dispositivo. Como es lógico esto se debe al aumento del ancho del bus de datos. En la figura 8.1 observamos que el porcentaje de los recursos utilizados en la red con el ancho del bus de 16 bits está dentro del rango esperado, con un total del 30 % de los recursos del dispositivo. En cambio, la red del bus de 32 bits, como se expone en la figura 8.3, sí se sale de lo esperado ocupando un 81 % del espacio.

Como hablamos en el capitulo anterior, la red neuronal con un bus de datos de 16 bits de longitud no es capaz de calcular las predicciones de glucemia correctamente, por ese motivo nos centramos en las posibles soluciones para llegar a un punto intermedio entre las dos síntesis y encontrar un equilibrio entre obtener unos resultados óptimos en las predicciones de la glucemia y disminuir el consumo de recursos de la FPGA.

El aspecto más importante para llegar a este equilibrio entre las dos consiste en establecer una longitud del bus datos que satisfaga los dos aspectos citados. Debido a los problemas de plazos en los que nos encontrábamos, desgraciadamente no podemos exponer ningún tipo de prueba evidente, pero con la experiencia que hemos ido obteniendo durante la implementación del diseño, nos atrevemos a decir que este equilibrio se encuentra en establecer un ancho del bus alrededor de los 20 bits.

El segundo aspecto a mejorar es el cambio de arquitectura de la IP de Xilinx CORDIC, con el fin de que estos componentes consuman el menor número de recursos del dispositivo. Xilinx brinda la posibilidad de poder generar estas IPs con distintas arquitecturas, dependiendo de como se generen, estas pueden realizar sus cálculos con diferentes latencias. En nuestro caso decidimos generar el componente CORDIC con una arquitectura en paralelo y no segmentada debido a que era la opción idónea, ya que no requería ningún reajuste importante para llevar a cabo la integración de estos en los módulos de cálculo de las funciones de activación. Para mejorar el consumo de los recursos el objetivo sería regenerar el componente CORDIC con una arquitectura en serie y segmentada, este cambio de arquitectura supondría que la realización de sus cálculos pasarían de tener una latencia de dos ciclos a tener una latencia de unos veinte ciclos aproximadamente por operación, lo que conllevaría la creación de una nueva unidad de control y otros ajustes en el diseño para el correcto funcionamiento del mismo.

A continuación queremos exponer en la siguiente figura 8.5 la diferencia de los recursos utilizados por CORDIC con sus diferentes arquitecturas y así justificar que este cambio en el diseño tendría un gran impacto en cuanto al consumo de recursos. Cabe destacar que en nuestro diseño hacemos uso de diez componentes CORDIC. La figura ilustra el consumo de recursos en una FPGA Xilinx Virtex-5, la cual no es nuestro dispositivo, por lo que los datos son simplemente orientativos:

Function	Architecture	Input/Output Width	Round Mode	LUT-FF pairs	Maximum Clock Frequency (Mhz)
Cos and Sin	Word Serial	16	Truncate	481	235
		32	Nearest Even	983	184
		48	Truncate	1376	149
	Parallel	16	Truncate	1093	366
		32	Nearest Even	3812	296
		48	Truncate	8126	220

Figura 8.5: Recursos utilizados por CORDIC [34]

Concluimos con el cálculo de la diferencia de las parejas de LUTs - flip flops utilizadas en las dos arquitecturas con datos de longitud de 32 bits, tal y como se puede observar en la figura:

- Arquitectura en paralelo con longitud de 32 bits: 3812 LUT-FF
- Arquitectura en serie con longitud de 32 bits: 983 LUT-FF

Lo que hace una diferencia de 2874 parejas de LUT-FF, y teniendo en cuenta que hacemos uso de 10 CORDICs, la diferencia quedaría en 28740 parejas de LUT-FF liberadas en el dispositivo.

Otro aspecto a mejorar sería hacer uso de las memorias primitivas de la FPGA (Block RAMs) y reducir el uso de memorias RAM distribuidas con las que almacenamos los datos por todo el diseño, puesto que como se observa en los informes de síntesis no hacemos uso de Bolck RAMs quedando estas sin usar. Este cambio repercutiría también en la liberación de LUTs del dispositivo ya que la memorias distribuidas se implementan mediante estas LUTs.

8.4. Camino crítico

El camino crítico es la ruta con el tiempo combinacional más largo entre dos registros. Es un aspecto muy importante del diseño que marca la frecuencia máxima a la que puede trabajar el sistema. Este camino viene indicado en el informe de síntesis de Xilinx. En nuestro diseño, el camino crítico corresponde a la ruta que comienza en el componente CORDIC tanh contenido en Tanh ct que se encuentra en el módulo Hidden state perteneciendo a su vez a LSTM cell del módulo Hidden layer y termina en la memoria RAM ht del mismo módulo Hidden state. A continuación mostramos el camino crítico marcado en rojo en las siguientes figuras:

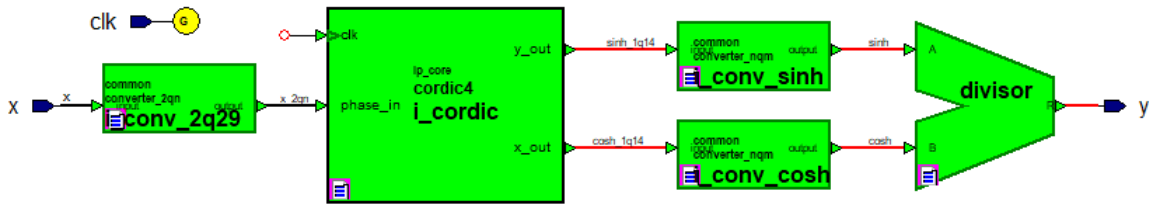

Figura 8.6: Sección del camino crítico dentro de CORDIC tanh

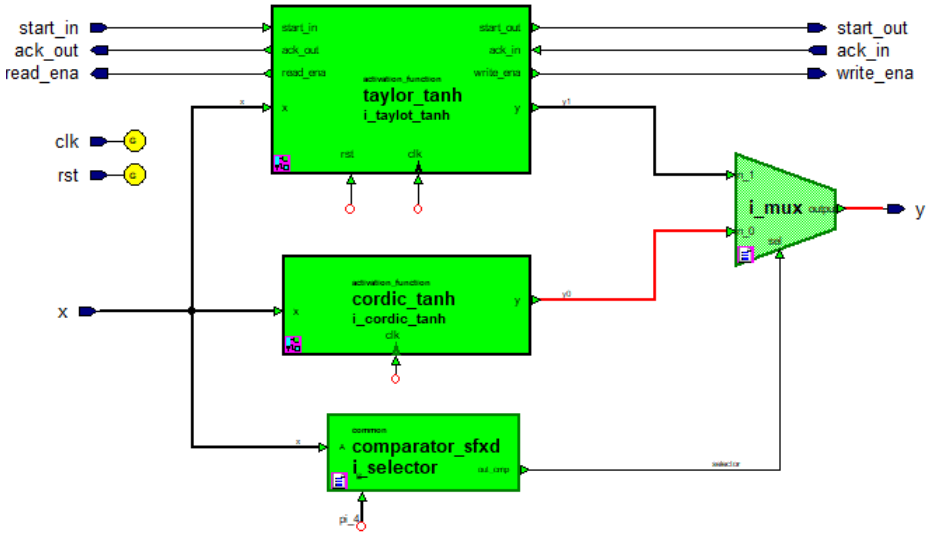

Figura 8.7: Sección del camino crítico dentro de Tanh ct

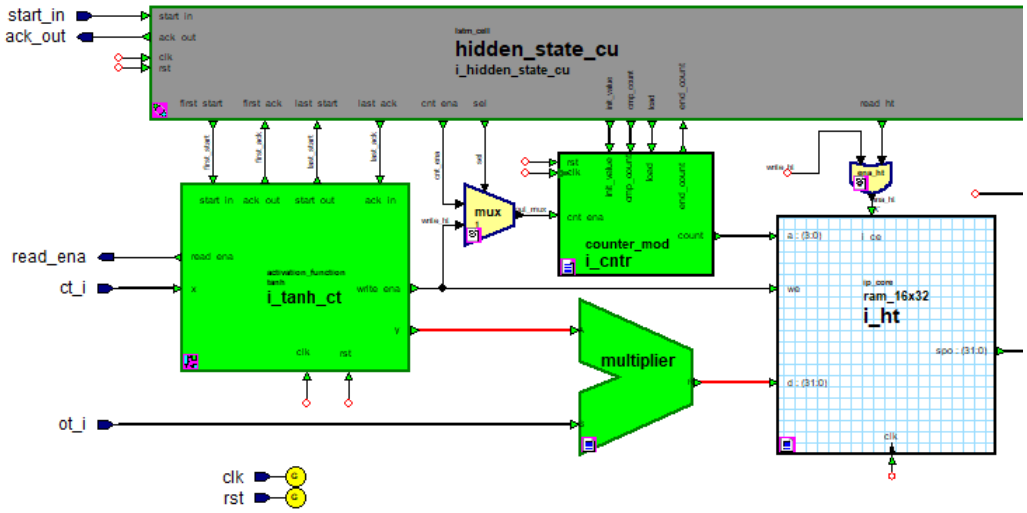

Figura 8.8: Sección del camino crítico dentro de Hidden state

Dado el informe sobre el reloj que nos muestra que el periodo mínimo con el que puede trabajar el sistema es muy grande. Para intentar que el sistema sea capaz de trabajar a una mayor frecuencia se debe segmentar el camino critico. La inclusión de registros de segmentación conllevaría la reestructuración de los módulos y requeriría reevaluar cómo impactaría a la temporización sobre el resto del diseño. Debido a los problemas de plazos en el que nos encontrábamos, desgraciadamente no ha sido posible llevar a cabo esta tarea, no obstante, dado que la idea es que al diseño se le transmitan las muestras registradas cada cinco minutos, no tenemos ninguna preocupación respecto a los tiempos de ejecución ya que debe realizar una predicción cada cinco minutos, por tanto no habría ningún tipo de problema en que pudiera realizar su cometido.

Capítulo 9

Conclusiones

En la realización de este trabajo de fin de grado hemos conseguido varios objetivos y resultados como:

- Hemos comprendido mucho mejor en que consiste la enfermedad de la diabetes.
- Hemos realizado un estudio en profundidad sobre el funcionamiento de una red neuronal LSTM tanto en la teoría como en la práctica con la implementación software y hardware desarrollada. Hemos aprendido como aproximar las funciones tangente hiperbólica y sigmoidal con la interpolación de Taylor.
- Hemos conseguido realizar y familiarizarnos con diferentes tareas como la creación de shell scripts, realización de notebooks de Python con Jupiter, scripts para la creación de las librerías del proyecto, compilación de las librerías de Xilinx y del proyecto.
- Hemos aprendido a desarrollar VHDL, a manejar la herramienta HDL Designer, a controlar las versiones con GitHub, a realizar simulaciones y a generar scripts de Questa Sim para formar ventanas de visualización de ondas y a generar documentos con LATEX.
- Finalmente hemos conseguido una red neuronal LSTM hardware sintetizable para poder ser implementada en una FPGA capaz de recrear un modelo software, desarrollado en Python, con un porcentaje de error del 0.42 % en los resultados de las predicciones correspondiente con la glucemia a treinta minutos vista.

Conclusions

During the realization of this Degree Final Project we have achieved several objectives and results as:

- We have a much better understanding of the disease of diabetes.
- We have managed to perform different tasks such as the creation of shell scripts, developing Python notebooks using Jupiter, scripts for the creation of the project's libraries, compilation of the Xilinx libraries and the project.
- We have learned how to develop VHDL, how to use the HDL Designer tool, how to manage file versions with GitHub, how to perform simulations and how to generate Questa Sim scripts to display waveforms windows and how to generate LATEX documents.
- Finally we have achieved a hardware LSTM neural network synthesizable to be implemented in an FPGA capable of recreating a software model, developed in Python, with an error rate of 0.42 % in the results of the glycemia predictions at thirty minutes in advance.

Capítulo 10

Contribuciones

La distribución y la contribución del proyecto entre ambos integrantes se describe con los puntos principales del proyecto donde hubo una participación por ambas partes:

- Implementación Python
- Implementación Hardware
- Redacción de la Memoria

A continuación, cada participante realizó:

10.1. Jorge López Melchor

10.1.1. Implementación Python

- Primero se realizó un estudio previo sobre las redes neuronales. Parte de mi función consistió en entender, comprender y decidir en equipo que tipo de red neuronal se ajustaba mejor en nuestro proyecto. Una vez que se tomó la decisión de utilizar las redes neuronales LSTM, tuvimos que hacer un estudio de la implementación de este tipo de redes con la librería Keras.
- Llevé a cabo el análisis sobre la base de datos seleccionando la información relevante y después el respectivo tratamiento de los datos requerido por la librería Keras para su posterior uso.
- Una vez quedo implementado el modelo HwNN y se realizaron las pruebas de este, mi función consistió en implementar los métodos para la generación de los ficheros COE con los pesos del modelo y el tratamiento de datos realizados en el fichero LSTM_Hw_NN.py.

10.1.2. Implementación Hardware

- Antes de llevar a cabo la implementación de los diferentes módulos me encargaba de realizar un análisis sobre las posibles implementaciones. Mi papel principal fue la esquematización de la celda LSTM con su funcionamiento interno.
- Participación conjunta en la organización de librerías, implementación de todos los módulos (diagrama de bloques, interfaz, unidad de control, etc.) y desarrollo de los ficheros de prueba para cada uno de los módulos.

  En concreto mi trabajo ha sido más acentuado en el desarrollo de los componentes: Tanh, Tylor tanh, Cordic Tanh , Activation function CU, Adder, Common CU, Comparator sfxd, Converter 2qn, Converter nqm, Counter, Counter mod, Divisor, Multiplexor2a1, Multiplexor4a1, Multiplier, Registr, Rom, Sign converter, Gate nx1, Gate nx1 CU, Matrix multiplier nx1, Matrix multiplier nx1 CU, Sigmoid gate nx1, Tanh gate nx1, , Vector sum, Vector sum CU, Cell state nx1, Cell state nx1 CU, Hidden state nx1, Hidden state nx1 CU, Lstm cell nx1.

10.1.3. Memoria

Realicé la implementación y maquetación de la memoria en Látex donde redacté gran parte del estado del arte, y conjuntamente con mi compañero las partes de implementación en Python e implementación e Hardware junto con todos los apartados comunes.

10.2. Octavio Sales Calvo

10.2.1. Implementación Python

- Primero se realizó un estudio previo sobre las redes neuronales. Parte de mi función consistió en entender, comprender y decidir en equipo que tipo de red neuronal se ajustaba mejor en nuestro proyecto. Una vez que se tomó la decisión de utilizar las redes neuronales LSTM mi papel principal fue la investigación de proyectos y pruebas con las redes LSTM y el uso de la librería Keras.
- Realización de diferentes modelos con Keras, su entrenamiento y validación de estos; así como el estudio e implementación del proceso para extraer los pesos entrenados de los modelos.
- Implementación del modelo HwNN. Realicé todos los métodos necesarios para que dado un modelo de Keras poder crear una nueva red a partir de este.

10.2.2. Implementación Hardware

- Participación conjunta en la organización de librerías, implementación de todos los módulos (diagrama de bloques, interfaz, unidad de control, etc.) y desarrollo de los ficheros de prueba para cada uno de los módulos.

  En concreto mi trabajo ha sido más acentuado en el desarrollo de los componentes: Sigmoid, Tylor sigmoid, CORDIC sigmoid, cordic4, RAM16x32, ram_sdp, std_fifo, Gate, Matrix multiplier mxn, Matrix multiplier mxn CU, Matrix mxn lookup, Sigmoid gate, Tanh gate, Cell state, Cell state CU, Hidden state, Hidden state CU, lstm cell, Neural net CU, Neural net.
  También el desarrollo de los paquetes constants y functional.

10.2.3. Memoria

Participé en la redacción de la parte del estado del arte e implementación en Python y sobre todo una gran aportación en la redacción de la parte de implementación Hardware. Conjuntamente con mi compañero realizamos todos los apartados comunes.

Bibliografía

[1] P. G. A. Uribe, "Deficiencia de acción insulina - relacsis | ops/oms," Servicios de Salud de Morelos, Calle Montes Urales, Tech. Rep. 440, May 2018. [Online]. Available: https://www.paho.org/relacsis/index.php/es/foros-relacsis/foro-becker-fci-oms/61-foros/consultas-becker/902-deficiencia-de-accion-insulina

[2] M. A. R. Herrera and M. D. B. Pomar, *La diabetes mellitus en la práctica clínica.* Buenos Aires; Madrid: Panamericana, 2009, p. 380. [Online]. Available: https://books.google.es/books?id=m8dcQYBF3UQC&pg=PA381&dq=diabetes+estudios+que+es&hl=es&sa=X&ved=0ahUKEwjOopbK-OrpAhXu0eAKHTM7DBEQ6AEIZjAI#v=onepage&q=diabetes%20estudios%20que%20es&f=fals

[3] N. Zozaya, R. Villoro, Álvaro Hidalgo, J. Oliva, M. Rubio, and S. García, "Estudio de coste de la diabetes tipo 2: una revisión de la literatura," Agencia de Evaluación de Tecnologías Sanitarias. Instituto de Salud Carlos III. Ministerio de Economía y Competitividad, Monforte de Lemos,5- Pabellón 8, Tech. Rep. 7251500341, abr 2015. [Online]. Available: http://gesdoc.isciii.es/gesdoccontroller?action=download&id=26/05/2015-28ff538b32

[4] J. L. López, "Implementación de algoritmos de identificación y predicción para glucmodel," Master's thesis, Facultad de Informática, Universidad Complutense de Madrid, Madrid, jun 2016/17. [Online]. Available: https://eprints.ucm.es/49967/1/Memoria%20TFG%20Javier%20Lesaga%20L%C3%B3pez%2003937913-Z%20-%20Implementaci%C3%B3n%20de%20algoritmos%20de%20identificaci%C3%B3n%20y%20predicci%C3%B3n%20para%20glUCModel.pdf

[5] F. E. de Diabetes (FEDE), "Diabetes e insulina," https://fedesp.es/diabetes/insulina/, 2020, accessed: 2020-21-06.

[6] D. Trujillo, A. J. Rivera, F. Charte, and M. J. del Jesus, "Una primera aproximación a la predicción de variables turísticas con deep learning," in *IX Simposio de Teoría y Aplicaciones de la Minería de Datos*, ser. Dialnet. Granada,España: CAEPIA, 2018, pp. 939–943. [Online]. Available: https://simidat.ujaen.es/sites/default/files/biblio/2018-CAEPIA-TurismoLSTMs.pdf

[7] R. P. Diez, A. G. Gómez, and N. de Abajo Martínez, *Introducción a la inteligencia artificial: Sistemas Expertos, Redes Neuronales Artificiales y Computación Evolutiva.* Oviedo: Universidad de Oviedo, 2001, p. 29. [Online]. Available: https://books.google.es/books?hl=es&lr=&id=RKqLMCw3IUkC&oi=fnd&pg=

PA10&dq=definici%C3%B3n+de+redes+neuronales+&ots=iGMzh5v08U&sig=hXnl48Gk3I7RePRBkNLXTOirjz4#v=onepage&q=red%20neuronal&f=false

[8] F. S. Caparrini, "Redes neuronales:una visión superficial," http://www.cs.us.es/ fsancho/?e=72, 2019, accessed: 2020-16-05.

[9] adrileon18, "Historia de la inteligencia artificial timeline." https://www.timetoast.com/timelines/historia-de-la-inteligencia-artificial-41109cc4-94c2-4f6d-9c28-440c6a016062, 2018, a2007-2020 Timetoast timelines.

[10] A. L. Cerpa, "Análisis de sistemas complejos usando machine learning," Master's thesis, Grado en Matemáticas, Universidad de Sevilla, Sevilla, jun 2018. [Online]. Available: https://idus.us.es/bitstream/handle/11441/81656/Luque%20Cerpa%20Alejandro%20TFG.pdf?sequence=1&isAllowed=y

[11] J. Bernstein, "A.i." *The New Yorker*, 1981. [Online]. Available: https://www.newyorker.com/magazine/1981/12/14/a-i

[12] M. Megías, P. Molist, and M. Pombal, "Tipos celulares. neurona," in *Atlas de Histología Vegetal y Animal*. Vigo, España: Departamento de Biología Funcional y Ciencias de la Salud, Universidad de Vigo, 2018. [Online]. Available: https://mmegias.webs.uvigo.es/descargas/tipos-cel-neurona.pdf

[13] A. del Profesor, "Sistemas inteligentes," 2020.

[14] L. F. Bertona, "Entrenamiento de redes neuronales basado en algoritmos evolutivos," Ph.D. dissertation, Universidad de Buenos Aires, Argentina, nov 2005. [Online]. Available: http://laboratorios.fi.uba.ar/lsi/bertona-tesisingenieriainformatica.pdf

[15] M. Villasana, "Introducción a las redes neuronales (neurales)," http://gecousb.com.ve/guias/GECO/Redes%20Neuronales%20(EC-5721)/Material%20Te%C3%B3rico%20(EC-5721)/EC-5721%20Introducci%C3%B3n.pdf, 2007, accessed: 2020-05-09.

[16] D. J. Matich, "Redes neuronales: Conceptos básicos y aplicaciones," Ph.D. dissertation, Universidad Tecnológica Nacional-Facultad Regional Rosario, Rosario, Mar. 2001. [Online]. Available: https://www.frro.utn.edu.ar/repositorio/catedras/quimica/5_anio/orientadora1/monograias/matich-redesneuronales.pdf

[17] J. Torres, "Redes neuronales recurrentes," https://torres.ai/redes-neuronales-recurrentes/, 2019, accessed: 2020-05-06.

[18] A. M. Mañas, "notes sobre pronóstico del flujo de tráfico en la ciudad de madrid," Master's thesis, Escuela Técnica Superior de Ingeniería Informática.UNED, Madrid, Jun. 2019. [Online]. Available: https://bookdown.org/amanas/traficomadrid/m%C3%A9todos-basados-en-deep-learning.html

[19] X. B. Olabe, "Redes neuronales artificiales y sus aplicaciones," 2008, accessed: 2020-07-08. [Online]. Available: https://ocw.ehu.eus/pluginfile.php/9047/mod_resource/content/1/redes_neuro/contenidos/pdf/libro-del-curso.pdf

[20] R. G. García, "Reconocimiento automático de idioma mediante redes neuronales," Master's thesis, Universidad Autónoma de Madrid, Madrid, Jun. 2018. [Online]. Available: https://repositorio.uam.es/bitstream/handle/10486/688478/go%20mez_garci%20a_ricardo_tfg.pdf?sequence=1

[21] J. Brownlee, "Why training a neural network is hard," https://machinelearningmastery.com/why-training-a-neural-network-is-hard/, 2019, accessed: 2020-06-06.

[22] C. Olah, "Understanding lstm networks," http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015, accessed: 2020-08-01.

[23] S. Yan, "Understanding lstm and its diagrams," https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714, 2016, accessed: 2020-03-18.

[24] Bccrwp, "Lstm vs rnn," https://es.bccrwp.org/solution/lstm-vs-rnn/, 2020, accessed: 2020-08-14.

[25] C. Iván, P. Juan, R. Juan, H. Ferney, and D. Fabio, "Implementación de una red neuronal artificial tipo som en una fpga para la resolución de trayectorias tipo laberinto," in *2013 II International Congress of Engineering Mechatronics and Automation (CIIMA)*, 2013, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6682790

[26] C. R. Ordoñez, J. J. Santiago, and J. F. Jaimes, "Reconocimiento de caracteres por medio de una red neuronal artificial," *DIALNET*, vol. 1, pp. 30–39, 14. [Online]. Available: https://dialnet.unirioja.es/servlet/articulo?codigo=5461232

[27] A. Dinu, M. N. Cirstea, and S. E. Cirstea, "Direct neural-network hardware-implementation algorithm," *IEEE Transactions on Industrial Electronics*, vol. 57, no. 5, pp. 1845–1848, 2010. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5280233

[28] N. M. Botros and M. Abdul-Aziz, "Hardware implementation of an artificial neural network using field programmable gate arrays (fpga's)," *IEEE Transactions on Industrial Electronics*, vol. 41, no. 6, pp. 665–667, 1994. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/334585

[29] S. Jung and S. s. Kim, "Hardware implementation of a real-time neural network controller with a dsp and an fpga for nonlinear systems," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 1, pp. 265–271, 2007. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4084734

[30] Xilinx, "Virtex-6 fpga ml605 evaluation kit," https://www.xilinx.com/products/boards-and-kits/ek-v6-ml605-g.html, 2020, accessed: 2020-08-23.

[31] Vipin, "Fixed point operations in vhdl," https://vhdlguru.blogspot.com/2010/03/fixed-point-operations-in-vhdl-tutorial_29.html, 2010, 10-02-2019.

[32] O. University, "The bglp challenge: Results, papers, and source code," http://smarthealth.cs.ohio.edu/bglp/bglp-results.html, 2020, 15-09-2020.

[33] UCFS, "El azúcar en sangre y otras hormonas," https://dtc.ucsf.edu/es/tipos-de-diabetes/diabetes-tipo-2/comprension-de-la-diabetes-tipo-2/como-procesa-el-azucar-el-cuerpo/el-azucar-en-sangre-y-otras-hormonas, 2007-2020, 02-08-2020.

[34] *LogiCORE IP CORDIC v4.0 product specification*, Xilinx, San Jose, California, U.S., 2011. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/cordic_ds249.pdfs

PASCAL
ENERO 2018
Ult. actualización 10 de noviembre de 2020
LaTeX lic. LPPL & powered by TEFLON CC-ZERO