CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL INSTITUTO POLITÉCNICO NACIONAL

DEPARTAMENTO DE INGENIERÍA ELÉCTRICA
SECCIÓN DE ELECTRÓNICA DEL ESTADO SÓLIDO

RED NEURONAL CELULAR EN TECNOLOGÍA DIGITAL

Tesis que presenta

ING. JOSÉ LUIS OCHOA PADILLA

para obtener el grado de

MAESTRO EN CIENCIAS

en la especialidad de

INGENIERÍA ELÉCTRICA

Asesores:

Dr. Felipe Gómez Castañeda Dr. José Antonio Moreno Cadenas

México, D.F. — 2006

AGRADECIMIENTOS

Esta sección será escrita en la versión final.

CONTENIDO

Introducción 1

Capítulo 1. La Red Neuronal Celular Fundamental.

1.1. Conceptos de Redes Neuronales Celulares 2
1.1.1. Redes Neuronales Celulares 2
1.1.2. Arquitectura de las CNNs 2
1.1.3. Estabilidad y Convergencia de las CNNs 7
1.1.4. CNNs Localmente Regulares 7
1.2. Optimización y Diseño Robusto de Mascarillas 8
1.2.1. Robustez Absoluta y Relativa 9
1.2.2. Optimización de Mascarillas 9
1.2.3. Diseño Óptimamente Robusto 12
1.2.4. Implementación de un Algoritmo de Optimización en MATLAB 13
1.3. Conclusiones 14

Capítulo 2. Red Neuronal Celular Extendida.

2.1. Definición de Problema Particular de Control de Matriz de Conmutación para FPS 17
2.2. Red Neuronal Celular Extendida 19
2.2.1. Arquitectura de CNN extendida 19
2.2.2. Estabilidad y Convergencia 21
2.2.3. Análisis en Simulink de CNN extendida en TC 21
2.3. Discretización en Tiempo del Modelo de CNN Extendida 23
2.3.1. Método de Euler para discretización 23
2.3.2. Análisis en Simulink de CNN Extendida en TD 25
2.4. Implementación de un Simulador de CNNs 26
2.4.1. Diseño del Simulador 27
2.4.2. Comparación de Resultados y Análisis de Desempeño 27
2.5 Conclusiones 28

Capítulo 3. Diseño Digital de CNN Extendida.

3.1. Planteamiento y Definición de la Arquitectura Digital 30
3.1.1. Definición del Modelo Digital 30
3.1.2. Elección del espacio de valores digitales 32
3.2. Módulos de Procesamiento 35
3.2.1. Sumador Multioperando 35
3.2.2. Multiplicador 38
3.2.3. Acumulador Saturado 42
3.2.4. Función No Lineal de Transferencia SATLIN 43
3.3. Conclusiones 43

Capítulo 4. Descripción de CNN Extendida en Lenguaje VHDL.
4.1. Traslación de los Módulos de Procesamiento a VHDL....45
4.1.1. Sumador Multioperando....45
4.1.2. Multiplicador....46
4.1.3. Función No Lineal de Transferencia SATLIN....47
4.1.4. Multiplexor 2:1 de 6 Canales....47
4.1.5. Multiplexor 5:1....47
4.1.6. Sumador de 2 Entradas con Saturación....47
4.2. Análisis de Respuesta de los Módulos....47
4.2.1. Sumador Multioperando....48
4.2.2. Multiplicador....48
4.2.3. Función No Lineal de Transferencia SATLIN....49
4.2.4. Multiplexor 2:1 de 6 Canales....50
4.2.5. Multiplexor 5:1....50
4.2.6. Sumador de 2 Entradas con Saturación....51
4.3. Neurona Digital....52
4.3.1. Diseño de la Neurona Digital....52
4.3.2. Respuesta de la Neurona Digital....53
4.4. Diseño de Módulo de Control....53
4.4.1. Memoria Secuencial....54
4.4.2. Contador Selector....54
4.4.3. Módulo de Control....54
4.4.4. Respuesta del Módulo de Control....55
4.5. Red Neuronal Digital Extendida....55
4.5.1. Diseño de la Red....55
4.5.2. Respuesta de la Red....56
4.6. Conclusiones....56

Capítulo 5. Conclusiones.
5.1. Análisis Global de Resultados....57
5.2. Optimización....57
5.3. Trabajo Futuro....58

Apéndices
A. Ejemplos de Optimización de Mascarillas....59
B. Ejemplos de Diseño Óptimo de Mascarillas....69
C. Programas de Optimización de Mascarillas en MATLAB....76
D. Diagramas de Simulación en Simulink....81
E. Listados de Código en Lenguaje VHDL....85

INTRODUCCIÓN

El objetivo de este trabajo de tesis es diseñar e implementar una red neuronal celular modificada en tecnología digital que permita controlar una matriz de conmutación utilizada frecuentemente en el campo de comunicaciones para la conmutación rápida de paquetes (FPS, fast packet switching).

En el capítulo 1 se dan los conceptos básicos de las redes neuronales celulares, sus propiedades y arquitectura y se revisa su estabilidad y convergencia. También se trata el tema de la optimización de las mascarillas de las CNNs y se presenta un programa en MATLAB que ayuda en el diseño robusto y optimización de las mascarillas. En el capítulo 2 se define el tema principal de la tesis, el control de una matriz de conmutación utilizado en comunicaciones para sistemas de FPS. Se define la arquitectura de una CNN extendida que permite obtener la solución al problema, se discretiza en tiempo el modelo de red y se implementa un simulador que permite resolver CNNs del tipo extendido como las que se requieren para nuestro caso. El capítulo 3 introduce el modelo digital de la red extendida, y plantea cada módulo digital de procesamiento requerido para la red. Asimismo, se diseña y analiza cada módulo. En el capítulo 4, cada módulo es trasladado al lenguaje VHDL y es analizado su desempeño. También se diseña e implementa el módulo de control de la red y se analizan los resultados globales de la red. Finalmente, en el capítulo 5 se discute la respuesta final de la red, se proponen algunas mejoras y optimizaciones necesarias para esta red y se menciona el trabajo a futuro.

CAPÍTULO 1

La Red Neuronal Celular Fundamental.

Equation Section 1

El concepto de *red neuronal celular* (CNN) fue introducido por Leon O. Chua y Lin Yang en 1988. [1] Las redes neuronales celulares poseen algunas de las características claves de las redes neuronales y tienen importantes aplicaciones potenciales en diversos campos, como son el reconocimiento de patrones, el procesamiento de imágenes y video, procesamiento no lineal en general y solución de ecuaciones diferenciales, entre otras. [2] En este capítulo veremos los conceptos, la arquitectura y las propiedades de las redes neuronales celulares básicas, así como un método para optimizar el desempeño de una CNN.

1.1. Conceptos de Redes Neuronales Celulares.

1.1.1. Redes Neuronales Celulares.

Una *red neuronal celular* es un sistema de procesamiento paralelo compuesto por una gran cantidad de unidades básicas de procesamiento llamadas *celdas*, las cuales están conectadas entre sí de manera directa sólo con las celdas más próximas. Esta característica de conectividad local permite su implementación en tecnología VLSI y facilita su diseño. Las características internas de las celdas (como sus ponderaciones de entrada y su función de transferencia), así como la conectividad entre ellas, determinan el comportamiento de la red y definen la tarea que realiza.

1.1.2. Arquitectura de las CNNs.

La unidad básica de una red neuronal celular es llamada una *celda*. La celda de una CNN analógica contiene elementos de circuito lineales y no lineales, típicamente capacitores lineales, resistores lineales, fuentes controladas lineales y no lineales, y fuentes independientes. La estructura de la red neuronal celular es similar a la hallada en los autómatas celulares; cualquier célula en una red neuronal celular está conectada sólo a sus vecinos más próximos. Las celdas adyacentes sólo pueden interactuar directamente con cada otra. Las celdas no directamente conectadas pueden afectar a otras celdas debido al efecto de propagación de la dinámica de tiempo continuo de la CNN. [1]

Considerando una red neuronal celular de dos dimensiones de $M \times N$, se tienen MN celdas en M filas y N columnas. Llamamos a la celda de la i-ésima fila y j-ésima columna $celda(i, j)$, y la denotamos por $C(i, j)$. Véase la Fig. 1.1 para un ejemplo de red neuronal celular de 4 x 5.

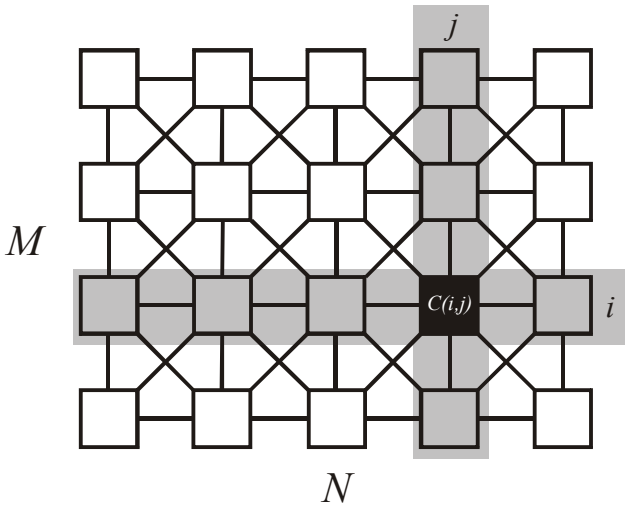


Fig. 1.1. *Ejemplo de una red neuronal celular de 4x5. Se muestra en negro la celda C(i,j).*

Vecindario en una CNN.

El *vecindario-r* de una celda $C(i,j)$ en una red neuronal celular está definido por

$$N_r(i,j)=\left\{C(k,l)\,|\,\max\left\{|k-i|,|l-j|\right\}\le r,1\le k\le M;1\le l\le N\right\}, \qquad (1.1)$$

donde *r* es un número entero positivo y es llamado *radio de vecindario* o *esfera de influencia*. El vecindario de una celda se extiende sobre una rejilla de tamaño $(2r+1)\times(2r+1)$. Aunque esto está definido para un arreglo cuadrado de celdas, se puede extender a otras topologías, v.g., para una topología hexagonal. En la Fig. 1.2 se observa el vecindario de radio 1 (sombreado) de una celda *C(i,j)* (en negro).

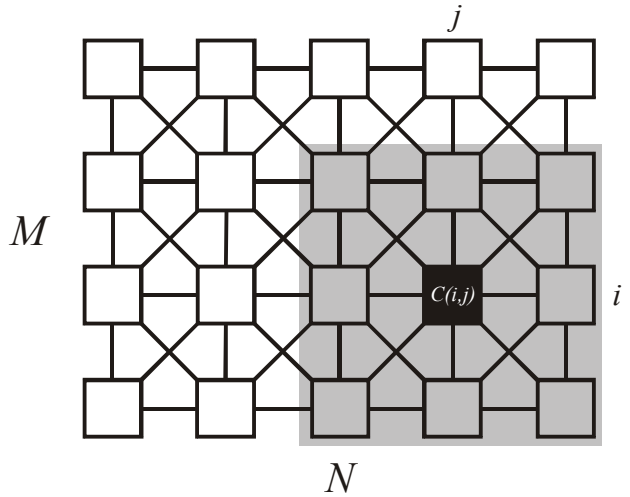


Figura 1.2. *CNN de 4x5, mostrando la celda C(i,j) (en negro) y su vecindario de radio 1 (sombreado).*

Se puede ver que el sistema de vecindario mostrado exhibe una propiedad de simetría en el sentido que si $C(i,j)\in N_r(k,l)$, entonces $C(k,l)\in N_r(i,j)$, para toda $C(i,j)$ y $C(k,l)$ en una red neuronal celular.

Estructura de la celda.

La estructura simbólica de una celda de una CNN se puede observar en la Fig. 1.3.. Se puede ver que en general consta de un número de entradas, cada una de las cuales es ponderada por un factor. El conjunto de estos factores es llamado *mascarilla*. Después, las señales son sumadas e integradas en un modelo de tiempo continuo. El resultado constituye el estado interno de la celda $x_{i,j}$. La función de transferencia *f* proporciona el valor de salida de la celda $y_{i,j}$.

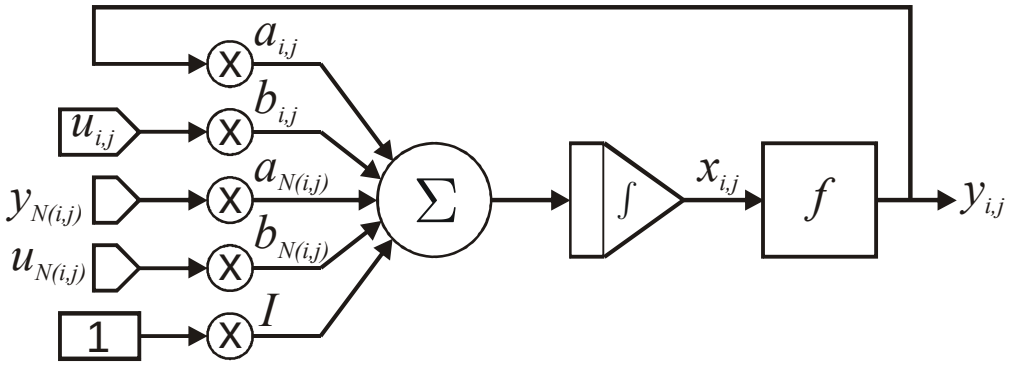


Figura 1.3. *Estructura a bloques de una celda.*

Ecuación de una CNN.

La dinámica describe la evolución temporal de cada celda, la cual es gobernada por una ecuación diferencial. De acuerdo a la definición original de Chua y Yang, la ecuación diferencial que gobierna la CNN es

$$C\frac{dv_{xij}(t)}{dt} = -\frac{1}{R_x}v_{xij}(t) + \sum_{C(k,l)\in N_r(i,j)} A(i,j;k,l)v_{ykl}(t) + \sum_{C(k,l)\in N_r(i,j)} B(i,j;k,l)v_{ukl} + I_{ij}, \quad (1.2)$$

donde

$$1 \le i \le M; 1 \le j \le N.$$

Haciendo $C = 1$ y $R_x = 1$ obtenemos la ecuación de estado normalizada

$$\frac{dv_{xij}(t)}{dt} = -v_{xij}(t) + \sum_{C(k,l)\in N_r(i,j)} A(i,j;k,l)v_{ykl}(t) + \sum_{C(k,l)\in N_r(i,j)} B(i,j;k,l)v_{ukl} + I_{ij}. \quad (1.3)$$

En estas ecuaciones, v_{xij} representa la variable de estado de la neurona $C(i,j)$, v_{ukl} son las entradas de las neuronas pertenecientes al vecindario $N_r(i,j)$, v_{ykl} son las salidas de las neuronas del vecindario y el término I_{ij} es el umbral. La matriz $A(i,j;k,l)$ representa la mascarilla de retroalimentación y la matriz $B(i,j;k,l)$ es la mascarilla de control.

De manera más general, podemos definir la ecuación diferencial normalizada

$$\frac{dx_\mu(t)}{dt} = -x_\mu(t) + \sum_{\nu \in \mathcal{N}_\mu} a_{\mu\nu}\, \mathrm{f}\left(x_\nu(t)\right) + \sum_{\nu \in \mathcal{N}_\mu} b_{\mu\nu} u_\nu + I_\mu, \tag{1.4}$$

donde $\mu = (\mu_1\, \mu_2 \cdots \mu_q)$ y $\nu = (\nu_1\, \nu_2 \cdots \nu_q)$ son multi-índices. El uso de estos multi-índices nos permite incluir la notación de CNN multicapa como un caso especial. En esta ecuación, definimos $\dim(\mu)$ como el orden de una CNN. Si $\dim(\mu) = 1$ hablamos de una CNN unidimensional, y si $\dim(\mu) = 2$ entonces se trata de una CNN plana. Las sumatorias son efectuadas sobre todas las celdas conectadas con la celda en la posición $\boldsymbol{\mu}$, denotada por C_μ, es decir, que se encuentran en el vecindario N_μ. El estado y la entrada de una celda C_μ son definidos por $x_\mu(t)$ y u_μ, respectivamente. Se asume que las entradas son independientes del tiempo. Por definición, $y_\mu(t) = \mathrm{f}\left(x_\mu(t)\right)$ es la salida de la celda C_μ en el tiempo *t*. En particular, denotamos por x_μ^* y y_μ^* el estado y la salida de C_μ en equilibrio, respectivamente.

Mascarillas.

Se asume que los parámetros de la CNN en la celda C_μ son independientes del tiempo y están contenidos en un conjunto denotado por $\mathcal{T}_\mu$, el *conjunto de mascarillas* en la celda C_μ, que consiste de

$$\mathcal{T}_\mu = \left\{a_{\mu\nu}, b_{\mu\nu}, I_\mu \mid \nu \in \mathcal{N}_\mu\right\}. \tag{1.5}$$

Específicamente, $a_{\mu\nu}$ son llamados *parámetros de retroalimentación*, $b_{\mu\nu}$ son los *parámetros de control* y I_μ es el término de *bias*, *polarización* o *umbral*. En el caso de CNNs rectangulares de radio de vecindario 1 y considerando parámetros espacial y temporalmente invariables (el cual es el caso más común), se pueden definir las matrices de retroalimentación, control y término de polarización, respectivamente, como:

$$A = \begin{bmatrix} a_{i-1,j-1} & a_{i-1,j} & a_{i-1,j+1} \\ a_{i,j-1} & a_{i,j} & a_{i,j+1} \\ a_{i+1,j-1} & a_{i+1,j} & a_{i+1,j+1} \end{bmatrix} \quad B = \begin{bmatrix} b_{i-1,j-1} & b_{i-1,j} & b_{i-1,j+1} \\ b_{i,j-1} & b_{i,j} & b_{i,j+1} \\ b_{i+1,j-1} & b_{i+1,j} & b_{i+1,j+1} \end{bmatrix} \quad I_{ij}\,. \tag{1.6}$$

Cada elemento $a_{k,l}$ de la matriz *A* se multiplica por la salida $y_{k,l}$ de la neurona $C_{k,l}$, y cada elemento $b_{k,l}$ de la matriz *B* se multiplica por la entrada $u_{k,l}$ de la neurona $C_{k,l}$, como se vió en la Fig. 1.3 y se describe en las Ecs. (1.3) y (1.4).

Funciones de Transferencia.

El único elemento no lineal en cada celda es la función de transferencia *f*, la cual toma el estado interno de la celda $x_{i,j}$ y produce la salida $y_{i,j}$. Por lo general se utiliza la *función de saturación de ganancia unitaria* como función de transferencia. Está definida como

$$\mathrm{f}(x) = \frac{1}{2}\left(|x+1| - |x-1|\right), \tag{1.7}$$

por lo que la ecuación de salida de la celda se define por

$$y_{i,j} = f\left(x_{i,j}(t)\right) = \frac{1}{2}\left(|x_{i,j}(t)+1| - |x_{i,j}(t)-1|\right). \tag{1.8}$$

Otras funciones comúnmente utilizadas son listadas en la Tabla 1.

Funciones de Transferencia Comunes		
Nombre	**Símbolo**	**Definición**
Lineal (purelin)		$\mathrm{purelin}(x) = x$
Saturación Lineal Positiva (satlin)		$\mathrm{satlin}(x) = \begin{cases} 0 & if\ x \le 0 \\ x & if\ 0 < x < 1 \\ 1 & if\ x \ge 1 \end{cases}$
Saturación Lineal Simétrica (satlins)		$\mathrm{satlins}(x) = \frac{1}{2}\left(|x+1| - |x-1|\right)$
Signo (hardlims)		$\mathrm{hardlims}(x) = \begin{cases} -1 & if\ x < 0 \\ 1 & if\ x > 0 \end{cases}$
Signo positiva (hardlim)		$\mathrm{hardlim}(x) = \begin{cases} 0 & if\ x < 0 \\ 1 & if\ x > 0 \end{cases}$
Sigmoidal Tangente Hiperbólica (tansig)		$\mathrm{tansig}(x) = \frac{2}{1+e^{-2x}} - 1$
Sigmoidal Logarítmica (logsig)		$\mathrm{logsig}(x) = \frac{1}{1+e^{-x}}$

Tabla 1.1. *Funciones de Transferencia Comunes en Redes Neuronales.*

Condiciones de Frontera.

Todas las *celdas internas* de una red neuronal celular tienen las misma estructura circuital y valores de elementos. Una *celda interna* es aquella que tiene $(2r+1)^2$ celdas vecinas, donde *r* está definida en la Eq. (1.1). Todas las demás celdas son llamadas *celdas de frontera*. Existen tres tipos de condiciones de frontera que son las más comúnmente utilizadas:

1. **Condición de frontera fija (Dirichlet).** Los valores de la variable de estado y la salida de cada celda de frontera están asignado a un valor constante.

2. **Condición de frontera de Newmann.** Los valores de estado de las celdas perpendiculares a las fronteras están restringidas a ser iguales las de unas con las otras.
3. **Condición de frontera periódica.** Las celdas de frontera están conectadas a las del extremo opuesto, sean verticales u horizontales.

Por lo general, el tipo de condición de frontera más utilizada es el de frontera fija. Es necesario tomar en cuenta el tipo de frontera al definir la operación de la red.

Condiciones Iniciales.

Como en todo sistema continuo, se deben especificar las condiciones iniciales para obtener el funcionamiento deseado. En el caso de las redes neuronales celulares, es necesario definir el estado inicial de la variable de estado $x_{i,j}$ para cada celda, e incluso se puede definir como una segunda entrada de la celda.

1.1.3. Estabilidad y Convergencia de las CNNs.

El trabajo original de Chua y Yang hace uso de una técnica para analizar las propiedades de convergencia de circuitos dinámicos no lineales: el método de Lyapunov. En éste, se define una función de energía $E(t)$, que se interpreta como la "energía generalizada" de la red neuronal celular, y a continuación se demuestra que la función está acotada y es monótonamente decreciente. Posteriormente, se demuestra que el límite de la función en un tiempo infinito es constante y se concluye con el teorema que permite asegurar la estabilidad y convergencia de la red. Este teorema afirma que si los parámetros de circuito satisfacen la condición

$$A(i,j;i,j) > \frac{1}{R_x}, \tag{1.9}$$

entonces cada celda de la red se establecerá en un punto de equilibrio estable después de que el transitorio haya decaído a cero.

El teorema anterior es significativo por que implica que la red no oscilará o se volverá caótica, y que los valores de salida serán binarios.

1.1.4. CNNs Localmente Regulares.

El nombre localmente regular se deriva de las reglas locales, las cuales son usadas para definir una tarea de CNN.

Un conjunto de mascarillas es *localmente regular* si su operación se puede caracterizar por un conjunto de reglas locales invariantes en el tiempo. La operación es entonces llamada localmente regular. Una *regla local* describe cuando la derivada $\dot{x}_{i,j}(t)$ del estado de la celda debe de ser negativa o positiva para una configuración bipolar particular de los valores de salida y entrada de las celdas adyacentes.

Definimos que una celda $C_{i,j}$ está *directamente conectada* a una celda $C_{m,n}$ si $|i-m| \leq 1$ y $|j-n| \leq 1$, esto es, cuando $m,n \in \mathcal{N}_{i,j}$ y $a_{i-m,j-n} \neq 0$. También definimos que una celda lineal es una celda para la cual $|x(t)| \leq 1$, o equivalentemente, $x(t) = y(t)$. Finalmente, podemos decir que un conjunto de mascarillas son localmente regulares si, para todas las celdas lineales, no hay ninguna otra celda lineal directamente conectada. Esto se debe a que si alguna celda C_L y alguna otra celda directamente conectada son lineales al mismo tiempo, la dinámica $x_L(t)$ es descrita por una ecuación diferencial variante en el tiempo, ya que es influenciada por la salida no bipolar y variante en el tiempo de su vecino lineal. Por tanto, el signo de $\dot{x}_L(t)$ no puede ser determinado por reglas locales invariantes en el tiempo.

1.2. Optimización y Diseño Robusto de Mascarillas.

El problema del diseño de las mascarillas es un tema clave en la investigación de las CNNs. Los métodos que se han investigado desde la invención de las redes neuronales celulares pueden ser clasificados en *métodos analíticos*, *algoritmos de aprendizaje local* y *algoritmos de aprendizaje global*.

Los métodos analíticos están basados en un conjunto de reglas locales que caracterizan la dinámica de la celda dependiendo en las condiciones de las celdas vecinas. Estas reglas son transformadas en un conjunto afín de ecuaciones que debe de ser resuelto a fin de obtener mascarillas funcionales.

Los *métodos de aprendizaje locales* están derivados de los métodos desarrollados para otros tipos de redes neuronales, tales como los perceptrones multicapa, mientras que sus contrapartes *globales* usan mayoritariamente técnicas de optimización estocástica, como algoritmos genéticos o recocido simulado.

La implementación de la ecuación de la CNN tiene un cierto número de limitaciones que deben tomarse en cuenta a fin de garantizar una operación correcta y eficiente. Las mascarillas por lo general pueden ser implementadas con sólo un 5% a 10% de precisión de su valor nominal, y usualmente, sólo está disponibles un conjunto discreto de posibles valores. Algunas otras fuentes de error son perturbaciones en entradas y valores de estado inicial, inexactitudes en la función de salida, diferencias en la constante de tiempo de las diferentes celdas, y el máximo valor de la variable de estado.

El requerimiento de que un conjunto de mascarillas lleven a cabo una tarea bajo estas circunstancias pone obstáculos adicionales al diseño de las mascarillas. Una operación en la CNN que se lleva a cabo confiablemente a pesar de todas las imperfecciones es llamada una *operación robusta*.

En esta sección nos restringiremos al análisis y diseño de mascarillas para las CNNs de clase bipolar, donde $\mathbf{u}, \mathbf{y}(t \to \infty) \in \mathbb{B}^n$ y $\mathbf{x}(0) \in \mathbb{Z}_0^n$ para $\mathbb{B} = \{-1,1\}$ y $\mathbb{B}_0 = \{-1,0,1\}$.

1.2.1. Robustez Absoluta y Relativa.

La *robustez* de un conjunto de mascarillas de CNN es una medida que cuantifica el grado por el cual los valores de las mascarillas pueden ser alterados mientras aún se produce la salida deseada.

Definamos un vector **p** que contiene todos los *m* elementos diferentes de cero de un conjunto de mascarillas $\mathcal{T}$, con el elemento central de la matriz *A* como el primer elemento $\left(p_1 := a_c\right)$, y el resto de los $m-1$ elementos en orden arbitrario. Llamaremos a las salidas finales de la CNN programada con **p** como $\mathbf{y}_\infty\left(\mathbf{p}\right)$. La *robustez absoluta* ε de un conjunto de mascarillas es

$$\varepsilon\left(\mathbf{p}\right) = \max_{\alpha}\left\{\alpha \mid \mathbf{y}_\infty\left(\mathbf{p}\right) = \mathbf{y}_\infty\left(\mathbf{p} + \alpha\mathbf{1}^{\pm}\right)\ \forall \mathbf{1}^{\pm} \in \mathbb{B}^m\right\}, \tag{1.10}$$

donde $\mathbb{B}$ es el conjunto $\{-1,1\}$ y $\mathbf{1}^{\pm}$ es un vector con elementos de $\mathbb{B}$. Los efectos de tolerancia de hardware debidos a imperfecciones físicas y de fabricación dan lugar a errores de parámetro ligeramente proporcionales al valor absoluto del respectivo parámetro. Por tanto, consideramos que la *robustez relativa D* de un conjunto de mascarillas es

$$D\left(\mathbf{p}\right) = \max_{\alpha}\left\{\alpha \mid \mathbf{y}^*\left(\mathbf{p} \circ \left(\mathbf{1} + \alpha\mathbf{1}^{\pm}\right)\right) = \mathbf{y}^*\left(\mathbf{p}\right)\ \forall \mathbf{1}^{\pm} \in \mathbb{B}^m\right\}, \tag{1.11}$$

donde $\circ$ denota una multiplicación de componentes vectoriales.

1.2.2. Optimización de Mascarillas.

La definición de una tarea para CNN bipolares está completamente caracterizada si se definen los signos de $\dot{x}$ para todas las posibles configuraciones de valores de salidas y entradas de las celdas vecinas, incluyendo la celda central. De la misma forma, el comportamiento de la CNN programada con un conjunto de mascarillas **p** puede ser analizado calculando $\dot{x}$ para todas las configuraciones.

Introduzcamos aquí un vector **p** ligeramente modificado, al que llamaremos $\tilde{\mathbf{p}}$, que comprenda todos los valores diferentes de cero de $\mathcal{T}$, pero incluyendo $a_c - 1$ en lugar de a_c (el elemento central de *A*) como primer elemento, i.e.,

$$\tilde{p}_1 := p_1 - 1 = a_c - 1 \quad \tilde{p}_i := p_i \ 2 \le i \le m, \tag{1.12}$$

que también puede ser descrito por $\tilde{\mathbf{p}} := \mathbf{p} - \mathbf{e}_1$, donde $\mathbf{e}_1$ es un vector unitario en dirección del incremento de a_c.

Codifiquemos todos los posibles valores de entradas y salidas de las celdas vecinas en un vector bipolar $\mathbf{v} \in \{-1,1\}^m$. La derivada del estado de una celda, dada una constelación de sus vecinos, puede ser escrita ahora como

$$\dot{x}\big|_{\mathbf{v}} = \mathbf{v}^T \cdot \tilde{\mathbf{p}} \ \forall \mathbf{v} \in \{-1,1\}^m . \tag{1.13}$$

En el caso de tener un parámetro de polarización *I* diferente de cero, el número de constelaciones es $n = 2^{m-1}$, en caso contrario $n = 2^m$, ya que el umbral no se multiplica por ningún valor dependiente de la celda. Introduciendo un índice *i*, dado por $1 \le i \le n$ para todo posible **v**, definimos una matriz $\mathbf{K} \in \{-1,1\}^{n \times m}$, la cual será

$$\mathbf{K} = \begin{bmatrix} \operatorname{sgn}\left(v_1^T \tilde{p}\right) \cdot v_1^T \\ \operatorname{sgn}\left(v_2^T \tilde{p}\right) \cdot v_2^T \\ \vdots \\ \operatorname{sgn}\left(v_n^T \tilde{p}\right) \cdot v_n^T \end{bmatrix}, \tag{1.14}$$

donde se define $\operatorname{sgn}(0) = 1$. Por medio de **K**, la caracterización de la CNN se simplifica al siguiente sistema lineal y homogéneo de desigualdades:

$$\left(\mathbf{K}\tilde{\mathbf{p}}\right)_i > 0 \quad \forall 1 \le i \le n , \tag{1.15}$$

el cual define un subconjunto $\mathcal{R} \in \mathbb{R}^m$. Dentro de este subconjunto, las mascarillas operan correctamente. Este subconjunto está delimitado por los hiperplanos definidos por los vectores fila $\mathbf{k}_i^T = \operatorname{sgn}\left(\mathbf{v}_i^T \tilde{\mathbf{p}}\right) \mathbf{v}_i^T$ de **K**. Los vectores fila son, de hecho, los vectores normales a estos hiperplanos.

Sin embargo, no todos estos hiperplanos son necesarios para definir a $\mathcal{R}$, ya que algunas desigualdades son trivialmente satisfechas mientras que otras permanecen. A fin de obtener una matriz **K** más pequeña, removemos estos renglones redundantes. Un vector es redundante si y sólo si puede ser expresado como una combinación lineal positiva de algún(os) otro(s) vector(es). Los vectores fila $\mathbf{k}_i^t$ son redundantes si

$$\exists \boldsymbol{\lambda} \in \left(\mathbb{R}_0^+\right)^n \text{ tal que } \mathbf{k}_i^t = \sum_{\substack{j=1 \\ j \ne i}}^{n} \lambda_j \mathbf{k}_j^t . \tag{1.16}$$

Tras eliminar estos renglones linealmente dependientes positivos terminamos con una matriz $\hat{\mathbf{K}} \in \mathbb{R}^{\hat{n} \times m}$, más pequeña y no redundante .Debe notarse que $\hat{n}$ no puede ser determinado

fácilmente, ya que en $\mathbb{R}^q$, para $q \geq 3$, existen una infinidad de vectores linealmente dependientes positivos. En general, $\hat{n} \square m$.

Margen de Seguridad.

Por medio del conjunto

$$\mathcal{R}' = \left\{ \tilde{\mathbf{p}} \in \mathbb{R}^m \mid \left(\mathbf{K} \cdot \tilde{\mathbf{p}} \right)_i \geq 0 \ \forall 1 \leq i \leq n \right\} \supset \mathcal{R}, \tag{1.17}$$

el cual incluye las fronteras de $\mathcal{R}$, introducimos el término de *margen de seguridad*; lo denotamos por $\gamma(\mathcal{T})$ y lo definimos formalmente por

$$\gamma(\mathcal{T}) = \gamma(\tilde{\mathbf{p}}) = \min_{1 \leq i \leq n} \left\{ \left(\mathbf{K} \cdot \tilde{\mathbf{p}} \right)_i \right\}, \tilde{\mathbf{p}} \in \mathcal{R}'. \tag{1.18}$$

Método de Optimización.

Supóngase que se cuenta con un conjunto de mascarillas que realiza correctamente una tarea definida. A fin de hacer robusto este conjunto de mascarillas, primero obtenemos el vector $\tilde{\mathbf{p}}$ como se definió en la Ec. (1.12), luego se definen todas las posibles constelaciones **v** y se obtiene la matriz **K**. Después se eliminan los renglones redundantes como se explicó anteriormente a fin de obtener la matriz $\hat{\mathbf{K}}$. $\left(\hat{\mathbf{K}} \cdot \tilde{\mathbf{p}} \right)_i$ es una medida de la robustez absoluta de un conjunto de mascarillas. Mientras mayor sea este valor, mayor será la distancia a la frontera del subconjunto en el cual opera correctamente el conjunto de mascarillas. Si tomamos el mínimo sobre todas las *i* y lo dividimos entre $\|\mathcal{T}\|_1$ tendremos la robustez relativa *D*:

$$D(\tilde{p}) = \frac{\min_i \left(\hat{\mathbf{K}} \cdot \tilde{\mathbf{p}} \right)_i}{\|\mathcal{T}\|_1}, \tag{1.19}$$

donde $\|\mathcal{T}\|_1 = \|\tilde{\mathbf{p}}\|_1 + 1$.

Si todas las $\left(\hat{\mathbf{K}} \cdot \tilde{\mathbf{p}} \right)_i$ son iguales, i.e., el margen de seguridad γ es el mismo, entonces las mascarillas son óptimamente robustas para una norma L_1 dada. Escalando el vector de mascarillas $\tilde{\mathbf{p}}$ por algún factor $q > 0$ no afecta ni su correcta operación ni su propiedad de ser óptimamente robusto y, al contrario, incrementa (para $q > 1$) la robustez relativa ligeramente. Por otro lado, escalando el vector obviamente conduce a tener una norma L_1 mayor. Así pues, tenemos que resolver

$$\hat{\mathbf{K}}\tilde{\mathbf{p}} = \gamma \mathbf{1}^m, \tag{1.20}$$

donde γ es una constante positiva y $\mathbf{1}^m$ es el vector $\begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix}^T \in \mathbb{R}^m$. De manera general, obtenemos

$$\tilde{\mathbf{p}}_{opt}(\gamma) = \left(\hat{\mathbf{K}}^T\hat{\mathbf{K}}\right)^{-1}\hat{\mathbf{K}}^T\gamma\mathbf{1}^m \tag{1.21}$$

como solución. Hay que notar que aunque esta es una solución por mínimos cuadrados, la solución será exacta si $\hat{\mathbf{K}}$ no contiene renglones redundantes, lo cual se asegura con el procedimiento expuesto antes.

El método de solución de (1.20) depende de m, $\hat{n}$ y el rango de $\hat{\mathbf{K}}$. Tenemos cinco casos:

1. $\operatorname{rank}\hat{\mathbf{K}} = m = \hat{n}$. $\hat{\mathbf{K}}$ es directamente invertible, es el caso básico y la solución está dada por $\tilde{\mathbf{p}}_{opt}(\gamma) = \gamma\hat{\mathbf{K}}^{-1}\mathbf{1}^m$.
2. $\operatorname{rank}\hat{\mathbf{K}} < m = \hat{n}$. El sistema tiene una solución de dimensión $\left(m - \operatorname{rank}\hat{\mathbf{K}}\right)$. Puede ser resuelto directamente usando descomposición *QR*, por ejemplo.
3. $\operatorname{rank}\hat{\mathbf{K}} = m < \hat{n}$. El sistema tiene más desigualdades que parámetros; está sobredeterminado pero tiene rango completo. Si existe una solución, la más robusta es $\tilde{\mathbf{p}}_{opt}(\gamma) = \left(\hat{\mathbf{K}}^T\hat{\mathbf{K}}\right)^{-1}\hat{\mathbf{K}}^T\gamma\mathbf{1}^m$. Si no es una solución, la tarea no está bien definida y no puede realizarse en esta clase de CNNs.
4. $\operatorname{rank}\hat{\mathbf{K}} < m < \hat{n}$. En este caso, el número de parámetros debe ser reducido por álgebra matricial para lograr que $m = \operatorname{rank}\hat{\mathbf{K}}$. Entonces será posible resolver usando el método descrito.
5. $\operatorname{rank}\hat{\mathbf{K}} \le n < m$. Si el número de parámetros excede el número de desigualdades, siempre hay una solución con dimensión $\left(\hat{n} - \operatorname{rank}\hat{\mathbf{K}}\right)$.

Ahora podemos escalar arbitrariamente $\tilde{\mathbf{p}}_{opt}$, dependiendo del máximo rango de valores disponibles para implementar las mascarillas, tal vez en un chip VLSI. La robustez relativa es

$$D\left(\tilde{\mathbf{p}}_{opt}(\gamma)\right) = \frac{\gamma}{\left\|\tilde{\mathbf{p}}_{opt}(\gamma)\right\|_1 + 1} = \frac{1}{\left\|\tilde{\mathbf{p}}_{opt}(1)\right\|_1 + \frac{1}{\gamma}}. \tag{1.22}$$

Esto demuestra que la robustez estrictamente monótonamente creciente cuando se incrementa γ, y que $1/\left\|\tilde{\mathbf{p}}_{opt}(1)\right\|_1$ es el límite superior teórico para la robustez si $\gamma \to \infty$.

Véase el Apéndice A para ejemplos de optimización de mascarillas.

1.2.3. Diseño Óptimamente Robusto.

Ya se ha visto en la sección anterior el método para obtener un conjunto de mascarillas óptimas, partiendo de otro conjunto que realice la tarea sin importar su robustez. Ahora veremos el procedimiento para diseñar desde cero un conjunto de mascarillas que realice la tarea de manera óptimamente robusta, partiendo de sus reglas locales de operación.

Método de Diseño.

Primero es necesario definir, basándose en el tipo de tarea a realizar, qué elementos de las mascarillas serán necesarios. Por ejemplo, si la tarea requiere propagación, será necesario utilizar elementos de la matriz *A* de retroalimentación. Si la tarea únicamente depende de sus entradas, solo será necesario utilizar elementos de la matriz *B*.

Ya seleccionados los elementos, se definen también las condiciones iniciales y se obtienen las desigualdades, basándose en las reglas locales que definen la tarea, y tomando en cuenta todas las combinaciones (constelaciones) posibles de entradas y salidas de las celdas vecinas. Estas desigualdades, de la forma $\left(\mathbf{K}\tilde{\mathbf{p}}\right)_i > 0$, pueden ser ajustadas de acuerdo al signo de la desigualdad y pueden tomar la forma $\left(\mathbf{K}\tilde{\mathbf{p}}\right)_i \geq 0$ para, a lo más, $m-1$ desigualdades. Estas desigualdades forman posteriormente la matriz **K**.

A partir de la matriz obtenida, es posible aplicar los métodos descritos en la sección anterior a fin de obtener un conjunto de mascarillas, la cual permitirá programar una CNN para que realice la tarea de manera óptima. Se reduce la matriz **K** a fin de eliminar las desigualdades redundantes y se obtiene la solución por cualquiera de los métodos descritos. Obtendremos al final una solución $\tilde{\mathbf{p}}_{opt}\left(\gamma\right)$, la cual puede ser escalada, también de acuerdo a la sección anterior, tomando en cuenta el rango de posibles valores de mascarillas disponibles para su implementación.

En el Apéndice B se muestran algunos ejemplos resueltos de diseño óptimamente robusto de mascarillas para tareas comunes.

1.2.4. Implementación de un Algoritmo de Optimización en MATLAB.

Durante la optimización o el diseño de mascarillas óptimamente robustas nos podemos enfrentar a la dificultad de decidir cuáles desigualdades de la matriz **K** son redundantes, y sobre todo cuando trabajamos con mascarillas con una gran cantidad de elementos diferentes de cero, ya que se dan muchas combinaciones que analizar. A fin de simplificar y acelerar el diseño y optimización de mascarillas para CNNs bipolares, se implementaron dos algoritmos en MATLAB basados en los procedimientos explicados en esta sección. El primero permite ingresar un conjunto de mascarillas ya existentes en la forma de un vector **p**, indicando si se incluye el elemento central y el umbral, y se procede a optimizar el conjunto. El segundo algoritmo permite ingresar un conjunto de desigualdades de diseño en la forma de matriz **K** y, de la misma forma, se analiza y se obtiene un conjunto de mascarillas óptimas.

Optimizador de Mascarillas.

Este algoritmo optimiza un conjunto de mascarillas ya existente. El programa principal es `robustify.m`, modificando la sección de entrada del programa permite introducir el conjunto de mascarillas a optimizar y especificar si se incluyen el elemento central de *A* y el umbral. Este programa utiliza los cuatro siguientes subprogramas dependientes:

1. `vmatrix.m`. Crea todas las posibles constelaciones para el conjunto de mascarillas.
2. `sgn.m`. Función signo modificada, devuelve1 para el valor de cero.
3. `redrem.m`. Remueve las desigualdades redundantes de la matriz **K**.
4. `sslv.m`. Resuelve el sistema para una matriz **K** no redundante, dependiendo de su rango.

El código del programa se muestra en el Apéndice C.

Diseño Robusto de Mascarillas.

Este algoritmo permite obtener un conjunto de mascarillas óptimamente robustas a partir de las desigualdades obtenidas de las reglas de diseño. La entrada se hace por medio de la matriz **K**. Utiliza los mismos subprogramas que el optimizador. Su programa principal es `robdes.m`.

El código del programa se muestra en el Apéndice C.

1.3. Conclusiones.

En este capítulo se han cubierto los conceptos básicos de las redes neuronales celulares, su arquitectura y se ha discutido su estabilidad y convergencia. Estos conceptos servirán de base para el resto del documento de tesis. También se han visto los conceptos de robustez de operación de una CNN, y se ha analizado un método para obtener conjuntos de mascarillas que realizan las tareas de forma óptima. Finalmente se ha mostrado el programa que se desarrolló para efectuar el proceso de optimización. Se espera que éste algoritmo pueda contribuir para acelerar y simplificar el diseño de tareas para las CNNs bipolares.

Referencias

1. L. O. Chua and L.Yang, “Cellular neural networks: Theory”, IEEE Trans. Circuits Syst., vol. 35, no. 10, pp. 1257–1272, 1988.
2. L. O. Chua and L. Yang, “Cellular neural networks: Applications”, IEEE Trans. Circuits Syst., vol. 35, no. 10, pp. 1273–1290, 1988.
3. M. Hänggi and G. Moschytz, “Cellular neural networks: Analysis, design and optimization”, Kluwer Academic Publishers, 2000.
4. _____, “An exact and direct analytical method for the design of optimally robust CNN templates”, IEEE Trans. Circuits Syst. – 1: Fundamental Theory and App., vol. 46, no. 2, pp. 304–311, 1999.
5. _____, “Optimization of CNN template robustness”, IEEE Trans. Fundamentals, vol. E82-A, no. 9, pp.1897–1899, 1999.

CAPÍTULO 2

Red Neuronal Celular Extendida.

Equation Section (Next)

En el área de redes de computadoras y telecomunicaciones, el paradigma dominante es la conmutación de paquetes, en el cual los paquetes (unidades portadoras de información) son dirigidos individualmente hacia nodos sobre un enlace de datos, los cuales pueden ser compartidos por varios nodos. Esto contrasta con el otro paradigma principal, la conmutación de circuitos, en el cual se establece una conexión entre dos nodos para su uso exclusivo durante todo el transcurso de su comunicación. La conmutación de paquetes se utiliza para optimizar el uso del ancho de banda disponible en una red, para disminuir la latencia e incrementar la robustez de las comunicaciones. La Conmutación Rápida de Paquetes (FPS, Fast Packet Switching) es una técnica de conmutación que incrementa la cantidad de datos transmitidos al eliminar los encabezados de los paquetes. Esto se logra al implementar el control de flujo y corrección de errores en los nodos de datos.

Entre las varias aplicaciones de las redes neuronales, el campo de las comunicaciones merece especial interés. Las redes de comunicaciones basadas en el Modo Asíncrono de Transferencia (ATM, Asynchronous Transfer Mode) deben manejar tráfico con una amplia gama de requerimientos en cuanto a Calidad de Servicio (QoS, Quality of Service). La Conmutación Rápida de Paquetes es una herramienta básica para alcanzar los requerimientos de QoS a través de un eficiente enrutamiento de paquetes. Para maximizar el flujo de datos en una red de FPS es necesario tomar un gran número de decisiones óptimas de conmutación en un corto periodo de tiempo, i.e., dentro de una ranura de tiempo cuya duración suele ser del orden de microsegundos. Tal tarea es difícil de resolver algorítmicamente en una computadora secuencial, pero gracias a la habilidad de tomar decisiones binarias óptimas en tiempo real, las redes neuronales se convierten en una buena elección para el control de las redes FPS.

Se han propuesto soluciones de redes neuronales del tipo Hopfield (HNNs) para el control de redes FPS, pero debido a que su diseño se basa esencialmente en argumentos de energía, no se garantiza que se obtengan respuestas óptimas. Sin embargo, con el uso de la metodología de diseño de redes neuronales celulares (CNNs) es posible evitar los problemas de las redes HNN al programar las CNNs con un conjunto preestablecido de puntos de equilibrio binarios estables, lo que nos lleva a tener un control preciso sobre las salidas deseadas para la red bajo un conjunto de condiciones de entrada.

En éste capítulo se define el problema de control de matriz de conmutación para FPS y se describe y analiza la solución propuesta en [1] y [2] utilizando una CNN extendida, haciendo uso de un modelo de simulación en SIMULINK para verificar la validez de la solución. Posteriormente se describe un método para producir un modelo de tiempo discreto de la red, paso que es necesario si se desea implementar la CNN en un procesador digital. Basándose en el modelo, se realizan simulaciones para verificar su estabilidad y convergencia, así como su velocidad de procesamiento, y se desarrolla un simulador de CNNs basado en software, que permite simular tanto el modelo extendido como la mayoría de los tipos comunes de CNNs. Finalmente se harán demostraciones de casos de control sobre redes de 4x4 y se determinarán sus tiempos de procesamiento.

2.1. Definición de Problema Particular de Control de Matriz de Conmutación para FPS.

En la figura 2.1 se muestra el diagrama esquemático de una Red de Conmutación Rápida de Paquetes(FPS) con N líneas de entrada y salida. La información digital que será transmitida está constituida por pequeñas unidades de información llamadas paquetes o celdas. Se considera una red síncrona, donde el tiempo se subdivide en ranuras de igual longitud, y los paquetes son dirigidos de un puerto de entrada a uno de salida en cada ranura de tiempo. Es necesario tener un almacenamiento de los paquetes para evitar su pérdida, por lo que se colocan buffers. En nuestra red de conmutación los buffers serán colocados en las entradas. Los paquetes que llegan a un puerto de entrada y que tienen diferentes destinos de salida son subdivididos en *N* partes separadas, uno para cada destino de salida posible. Debido al sincronismo entre las líneas de entrada y salida, sólo puede transmitirse, a lo más, un paquete a cada puerto de entrada y sólo puede recibirse, a lo más, un paquete por puerto de salida en cada ranura de tiempo. Esto representa la restricción básica para la realización física de la red FPS.

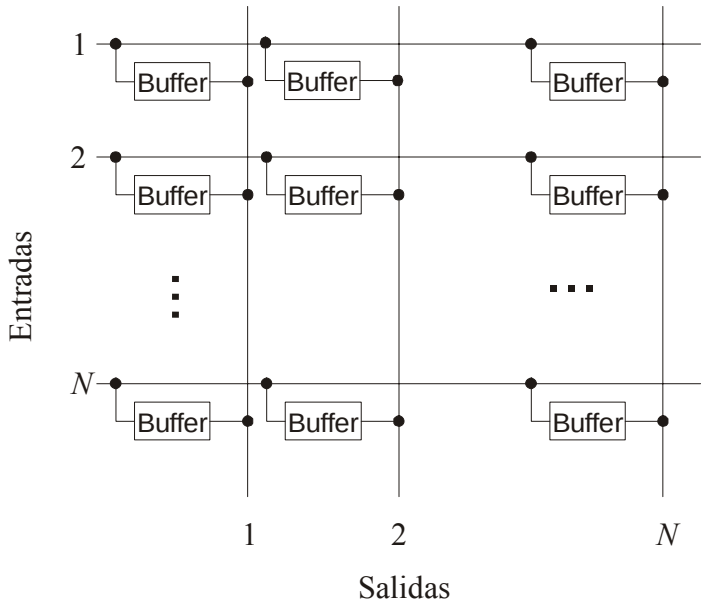


Fig. 2.1. *Diagrama esquemático de una Red de Conmutación Rápida de Paquetes con N líneas de entrada y salida.*

En cada ranura de tiempo, el estado de los $N \times N$ paquetes en los buffers de entrada puede ser descrito por una matriz binaria de $N \times N$, dada por $S = \left[s_{ij} \right] \in \mathbb{B}$ y llamada matriz de requisición de entrada. Una entrada $s_{ij} = 1$ indica que hay por lo menos un paquete de datos en el puerto de entrada *i* esperando a ser transferido al puerto de salida *j*. De otro modo, si no existen paquetes en el buffer, $s_{ij} = 0$. Un ejemplo de matriz de requisición de entrada para una red FPS de 4×4 es:

$$S = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}. \tag{2.1}$$

La red neuronal tendrá por tanto que decidir a cual puerto de salida deberá direccionar el siguiente paquete que se encuentre en el buffer de entrada. Dada la restricción principal mencionada anteriormente, la matriz de solución consistirá en una matriz de permutación, es decir, será una matriz de ceros con sólo un elemento 1 por fila y por columna. Así, las decisiones óptimas para conmutar los paquetes corresponden a cualquiera de las tres matrices de permutación P_1, P_2 y P_3 que podemos extraer de la matriz de entrada S:

$$P_1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}; P_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}; P_3 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}. \tag{2.2}$$

Estas selecciones nos permiten maximizar el flujo de datos en la ranura de tiempo sin violar las restricciones de realización física.

En aplicaciones prácticas, es importante maximizar el flujo de datos de acuerdo a ciertas restricciones y optimizar al mismo tiempo una función de costo asociada. En cada ranura de tiempo supongamos que asociamos un peso positivo w_{ij} a cada elemento $s_{ij} = 1$ de S. También definimos $w_{ij} = 0$ cuando $s_{ij} = 0$. El problema de optimización que deseamos resolver puede definirse como sigue. Considere la función de costo definida sobre el conjunto $\mathcal{P}$ de matrices de permutación

$$C(P, W) = \sum_{i,j=1,\ldots,N} p_{ij} w_{ij}, \qquad P \in \mathcal{P} \tag{2.3}$$

donde $W = \left[w_{ij} \right]$ es la matriz de pesos de $N \times N$. Estamos interesados en encontrar la matriz de permutación $\mathcal{D} \in \mathcal{P}$ extraída de S que maximice la función de costo (2.3). Las entradas d_{ij} de D representan las decisiones de conmutación óptimas, tomadas en la ranura de tiempo considerada.

En la aplicación de comunicaciones aquí considerada, la función de costo (2.3) nos permite manejar diferentes requerimientos de QoS (*Quality of Service*, calidad del servicio), que se dan en las aplicaciones de tráfico de multimedia, por ejemplo, en las redes de comunicaciones de voz y datos. Para este fin, damos valores más grandes a los pesos w_{ij} cuyos paquetes tienen mayor prioridad. De manera similar, si estamos interesados en minimizar el retardo de transmisión de los paquetes, damos valores de w_{ij} más grandes a los paquetes que sufran de

mayor retraso en los buffers de entrada. Más generalmente, w_{ij} puede ser una función que tome en cuenta tanto el retraso de transmisión como la prioridad de los paquetes.

Tomemos como ejemplo la siguiente matriz de pesos, que asociaremos a la matriz de requisición de entrada (2.1),

$$W = \begin{bmatrix} 0.9 & 0.3 & 0.4 & 0.5 \\ 0.4 & 0.6 & 0 & 0 \\ 0 & 0 & 0 & 0.7 \\ 0 & 0.1 & 0.4 & 0 \end{bmatrix}. \tag{2.4}$$

La solución óptima que maximiza la función *C* corresponde a $D = P_2$, donde P_2 está dada por (2.2).

Para redes de conmutación grandes, el problema de optimización es difícil de resolver algorítmicamente en tiempo real, por lo que recurrimos al uso de una red neuronal. Haremos uso de un arreglo cuadrado de $N \times N$ celdas, donde cada celda (i, j) estará dedicada a realizar la decisión óptima d_{ij} sobre la conmutación del paquete en tiempo real.

2.2. Red Neuronal Celular Extendida.

2.2.1. Arquitectura de CNN extendida.

Dado que se realizan $N \times N$ decisiones de conmutación, consideramos una red neuronal de $N \times N$ celdas en un arreglo cuadrado, donde cada celda $(i, j), i, j = 1, \ldots, N$ satisface la ecuación diferencial

$$\dot{x}_{ij} = -x_{ij} + \alpha y_{ij} + A\left[\sum_{h=1,\ldots,N;h\neq j} y_{ih} + \sum_{k=1,\ldots,N;k\neq i} y_{kj}\right] + \beta w_{ij} + B\left[\sum_{h=1,\ldots,N:h\neq j} w_{ih} + \sum_{k=1,\ldots,N;k\neq i} w_{kj}\right] + I \tag{2.5}$$

En esta ecuación, x_{ij} es el estado de la celda, mientras que $y_{ij} = g(x_{ij})$ es la salida de la celda. Asumimos que *g* es una función de activación lineal a segmentos con niveles de saturación 0 y +1, y ganancia unitaria en la región lineal (función **poslin**), i.e., definida como sigue:

$$g(\rho) = \begin{cases} 0 & \rho < 0 \\ \rho & 0 \le \rho \le 1 \\ 1 & \rho > 1 \end{cases}$$

Se consideran valores normalizados para la resistencia y la capacitancia internas de la celda ($C_x = 1F$, $R_x = 1\Omega$).

Cada celda está conectada sólo con celdas en la misma fila y columna, y el parámetro *A* representa el peso de la conexión. Suponemos que *A* es negativo, ya que la celda está sujeta a retroalimentación competitiva de las celdas en la misma fila y columna. El término α es la retroalimentación propia. Los términos w_{ij} son entradas a la celda que representan la prioridad de la conexión y se mantienen constantes durante la ranura de tiempo. Las cantidades β y *B* son positivas y constituyen parámetros que controlan la influencia ejercida por w_{ij} sobre la dinámica de la red. Mientras mayores sean estos valores con respecto a $|A|$, mayor será la influencia de los pesos w_{ij} de entrada sobre la toma de decisión de la red. Además de w_{ij}, las entradas s_{ij} de la matriz de requisición de entrada deben ser alimentadas como entradas a la red a través de las condiciones iniciales $x_{ij}(0)$, las cuales son elegidas de modo que $y_{ij}(0) = g(x_{ij}(0)) = s_{ij}$.

Diseño de la Red.

El diseño de la red consiste en determinar los coeficientes *A*, *B*, α, β e *I* que permiten que la CNN lleve a cabo la tarea descrita. De acuerdo con [2], estos coeficientes se determinan como sigue:

$$\begin{aligned} &I > 0 \\ &\alpha > 1 \\ &A < 1 - \alpha - I < 0 \\ &0 < B < \Gamma = \frac{1}{2}\frac{1-\alpha-I-A}{\rho + N - 1} \end{aligned} \quad , \tag{2.6}$$

donde *N* es el número de entradas y salidas de la red, y ρ está dado por

$$\rho = \frac{\beta}{2B} > 1 \, . \tag{2.7}$$

Una vez que se establecen α e *I* de acuerdo a (2.6), los únicos puntos estables de equilibrio de la red son matrices de permutación, cuando a cada celda se le da la suficiente retroalimentación competitiva *A* de las celdas en la misma fila y columna, y cuando el parámetro de control *B* para las entradas w_{ij} no es muy grande. Así, la red neuronal puede considerarse como una memoria asociativa especial cuyos patrones almacenados son matrices de permutación.

Hay que hacer notar que Γ es el límite superior para el parámetro *B*, y para nuestro caso es conveniente tomar el máximo valor posible, ya que incrementando *B* hacia el límite Γ causa un drástico aumento de la cercanía de las decisiones tomadas por la red hacia las decisiones óptimas.

2.2.2. Estabilidad y Convergencia.

Simplifiquemos la notación de la ec. (2.5), definiendo el operador

$$\Phi\left(M;i,j\right)=\sum_{h=1,\ldots,N;h\neq j} m_{ih}+\sum_{k=1,\ldots,N;k\neq i} m_{kj}\,, \tag{2.8}$$

que asocia una matriz *M* de $N\times N$ y un índice i,j dado a la sumatoria de las entradas de *M* en la *i*-ésima fila y la *j*-ésima columna, excluyendo a m_{ij}. El modelo neural de (2.5) puede reescribirse como

$$\dot{x}_{ij}=-x_{ij}+\alpha y_{ij}+A\Phi\left(Y;i,j\right)+\beta w_{ij}+B\Phi\left(W;i,j\right)+I\,, \tag{2.9}$$

donde $Y=\left[\,y_{ij}\,\right]$ es la matriz de variables de salidas de las celdas y $W=\left[\,w_{ij}\,\right]$ es la matriz de prioridades definida antes.

Debe notarse que la simetría de la matriz de interconexiones neuronales se satisface en el modelo definido por la ec. (2.9). Esto implica que la red neuronal es completamente estable, i. e., se garantiza que alcanzará un punto de equilibrio tras un transitorio. Debido a la homogeneidad de la interconexión neuronal (mismos pesos) y la estructura de interconexión esencialmente dispersa, el modelo (2.9) puede ser considerado una CNN extendida.

2.2.3. Análisis en Simulink de CNN extendida en tiempo continuo.

A fin de verificar el procedimiento de diseño que se presentó para la CNN extendida, se construyó un modelo de simulación en tiempo continuo de la red en Simulink. Véase 1 y 2 del Apéndice D para los diagramas esquemáticos del modelo de Simulink. Se eligió un tamaño de red de 4×4 a fin de hacer el modelo manejable y verificar los resultados reportados. Para el diseño del modelo de simulación se trasladó la ecuación diferencial (2.5) a sus correspondientes bloques de simulación. Integrando ambos miembros de (2.9) se obtiene

$$x_{ij}=\int\left[-x_{ij}+\alpha y_{ij}+A\Phi\left(Y;i,j\right)+\beta w_{ij}+B\Phi\left(W;i,j\right)+I\right]dt\,, \tag{2.10}$$

donde se observa directamente que se puede utilizar un integrador de tiempo continuo para implementar el modelo. Compárese esta ecuación con el modelo interno de la celda en 2 del Apéndice D. Ya que los parámetros *A* y *B* multiplican a varios términos, se hace la sumatoria de los términos de entrada antes de multiplicarlos por dichos parámetros. Posteriormente son sumados todos los productos parciales.

El modelo de la celda es instanciado para formar la red neuronal completa, haciendo la interconexión entre celdas y con las entradas y salidas. Los valores de entrada (matriz de requisición de entrada *S*, matriz de prioridades *W* y parámetros de red) son alimentados por

medio de la lectura de archivos binarios de MATLAB que contienen los valores correspondientes para la simulación de la CNN durante una ranura de tiempo. Igualmente, las salidas de la CNN, correspondientes a los estados y las salidas de las celdas, son escritos a otros archivos binarios y posteriormente analizados.

Para el modelo de CNN de 4×4, los parámetros de la red obtenidos por el procedimiento de la sección 2.2.1 son:

$$\alpha = 2 \quad \beta = 20 \quad A = -23 \quad B = 1 \quad I = 1. \tag{2.11}$$

Las simulaciones mostraron que el modelo es válido y que la CNN realiza las decisiones de conmutación correctamente. Utilizando como ejemplo la matriz de requisición de entrada (2.1) y la matriz de prioridades (2.4), se observó la evolución de la red mostrada en las figuras 2.2 y 2.3.

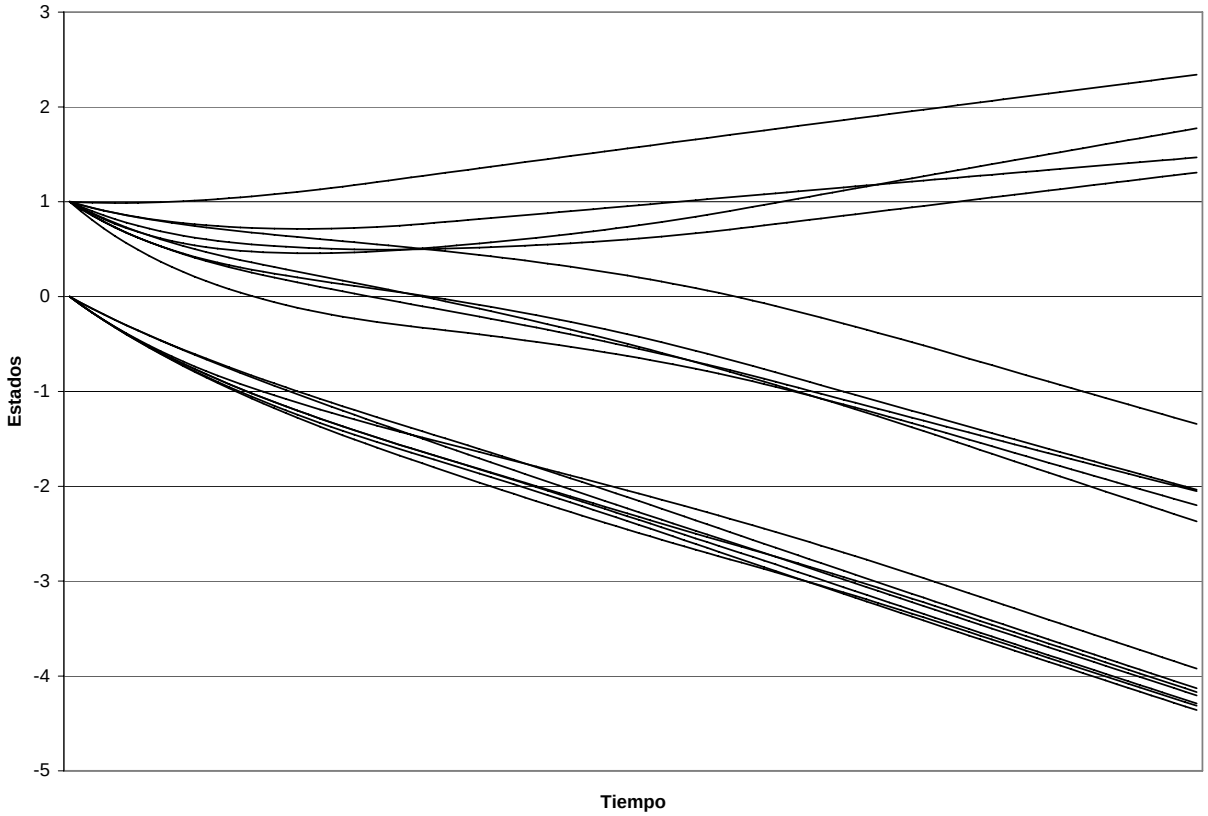


Figura 2.2. *Evolución de los estados internos de las celdas en la CNN.*

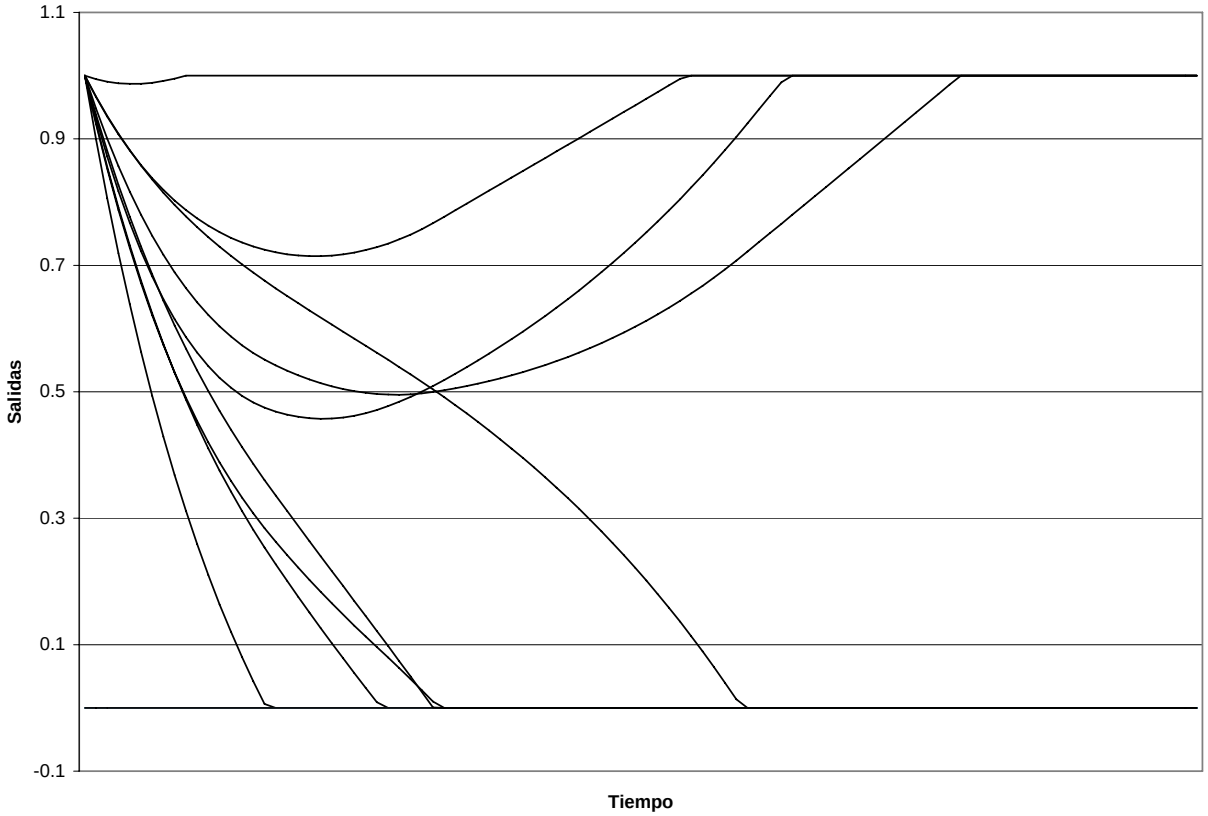


Figura 2.3. *Evolución de las salidas de la CNN.*

2.3. Discretización en Tiempo del Modelo de CNN Extendida.

2.3.1. Método de Euler para Discretización.

Ya que la CNN se pretende implementar en un dispositivo lógico programable, es necesario discretizar tanto el tiempo como el espacio de valores de entradas, estados y salidas de las celdas. Esto es debido a que los procesadores digitales utilizan tanto pasos de tiempo como valores de señales discretos. Comenzaremos por discretizar, en este capítulo, el tiempo. Posteriormente se hará la discretización del espacio de valores. Dado que la ecuación (2.5) es una ecuación diferencial y se tienen valores iniciales para los estados, necesitaremos un método que nos permita aproximar numéricamente la solución de la ecuación.

Una de las técnicas más sencillas para aproximar soluciones del problema de valor inicial

$$y' = f(x, y), \quad y(x_0) = y_0 \tag{2.12}$$

se llama *método de Euler* o *método de las tangentes*. Aplica el hecho de que la derivada de una función $y(x)$, evaluada en un punto x_0, es la pendiente de la tangente a la gráfica de $y(x)$ en

ese punto. Como el problema de valor inicial establece el valor de la derivada de la solución en (x_0, y_0), la pendiente de la tangente a la curva de solución en este punto es $f(x_0, y_0)$. Si recorremos una distancia corta por la línea tangente obtenemos una aproximación a un punto cercano de la curva de solución. A continuación se repite el proceso en el punto nuevo. Para formalizar este procedimiento se emplea la linealización

$$L(x) = y'(x_0)(x - x_0) + y_0 \tag{2.13}$$

de $y(x)$ en $x = x_0$. La gráfica de esta linealización es una recta tangente a la gráfica de $y = y(x)$ en el punto (x_0, y_0). Ahora se define *h* como un incremento positivo sobre el eje *x*. Reemplazamos *x* con $x_1 = x_0 + h$ en (2.13) y llegamos a

$$L(x_1) = y'(x_0)(x_0 + h - x_0) + y_0 = y_0 + hy_0' \tag{2.14}$$

esto es

$$y_1 = y_0 + hf(x_0, y_0) \tag{2.15}$$

en donde $y_0' = y'(x_0) = f(x_0, y_0)$ y $y_1 = L_1(x)$. El punto (x_1, y_1) sobre la tangente es una aproximación al punto $(x_1, y(x_1))$ en la curva de solución; esto es, $L(x_1) \approx y(x_1)$ o $y_1 \approx y(x_1)$ es una aproximación lineal local de $y(x)$ en x_1. La exactitud de la aproximación depende del tamaño *h* del incremento. Por lo general se escoge una magnitud de paso "razonablemente pequeña". Si a continuación repetimos el proceso, identificando al nuevo punto de partida (x_1, y_1) como (x_0, y_0) de la descripción anterior, obtenemos la aproximación

$$y(x_2) = y(x_0 + 2h) = y(x_1 + h) \approx y_2 = y_1 + hf(x_1, y_1) \tag{2.16}$$

La consecuencia general es que

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{2.17}$$

en donde $x_n = x_0 + nh$. Esta es la fórmula de Euler.

Aplicaremos la fórmula (2.17) a la ecuación (2.9), la cual depende implícitamente del tiempo *t*. Identificamos al estado x_{ij} como la variable y_n, al tiempo *t* como la variable x_n y la función $f(x_n, y_n)$ es el lado derecho de la ecuación (2.9). Por tanto, la ecuación diferencial se transforma en

$$x_{ij}[n+1] = x_{ij}[n] + hf(t[n], x_{ij}[n]), \tag{2.18}$$

donde se utilizan los brackets para indicar el paso de tiempo discreto o iteración. Como la variable tiempo no interviene directamente en el cálculo del estado siguiente, se tiene

$$x_{ij}[n+1] = x_{ij}[n] + hf\left(x_{ij}[n]\right). \tag{2.19}$$

Esta es la forma de tiempo discreto de la ecuación diferencial de la celda (i, j).

2.3.2. Análisis en Simulink de CNN Extendida en Tiempo Discreto.

A fin de verificar la validez del nuevo modelo de tiempo discreto, se construyó un modelo en tiempo discreto de la CNN definida por la ec. (2.19). El modelo se puede observar en 3 y 4 del Apéndice D. Se ha reemplazado el integrador en tiempo continuo por un retardo unitario y un sumador, formando un acumulador. También se agrega un multiplicador para incluir el paso de tiempo *h* del proceso numérico. El resto del modelo es esencialmente el mismo. La evolución de la red para el ejemplo de la matriz de requisición de entrada (2.1) y la matriz de prioridades (2.4) se muestra en las figuras 2.4 y 2.5. El tamaño del paso es $h = 0.01$.

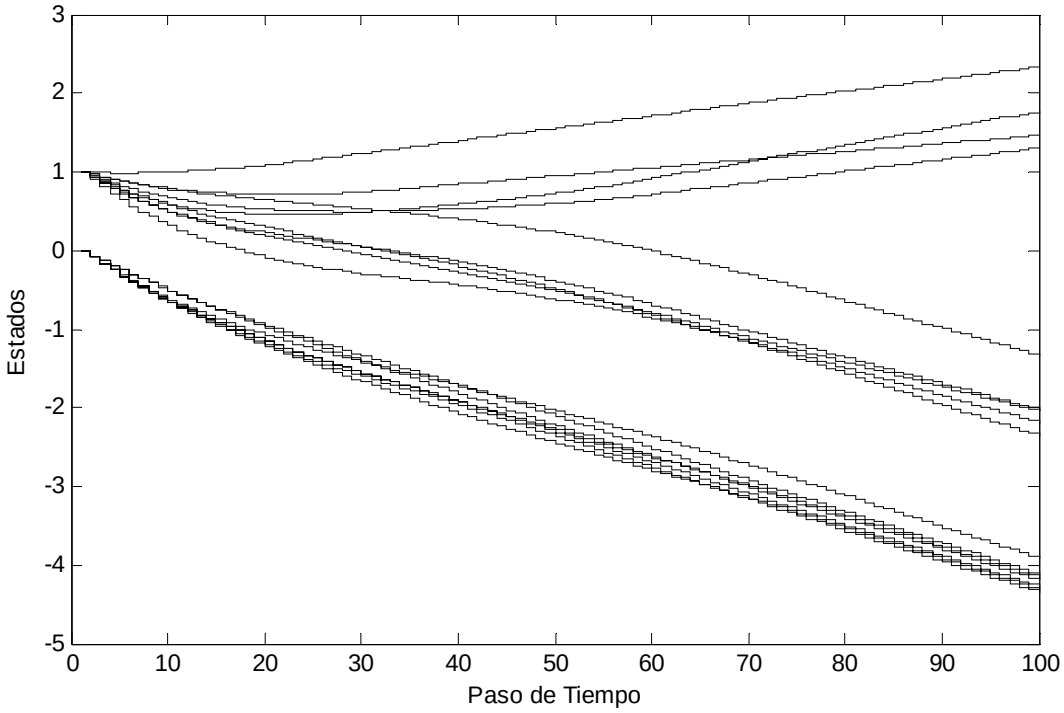


Figura 2.4. *Evolución de los estados internos de las celdas en la CNN de tiempo discreto.*

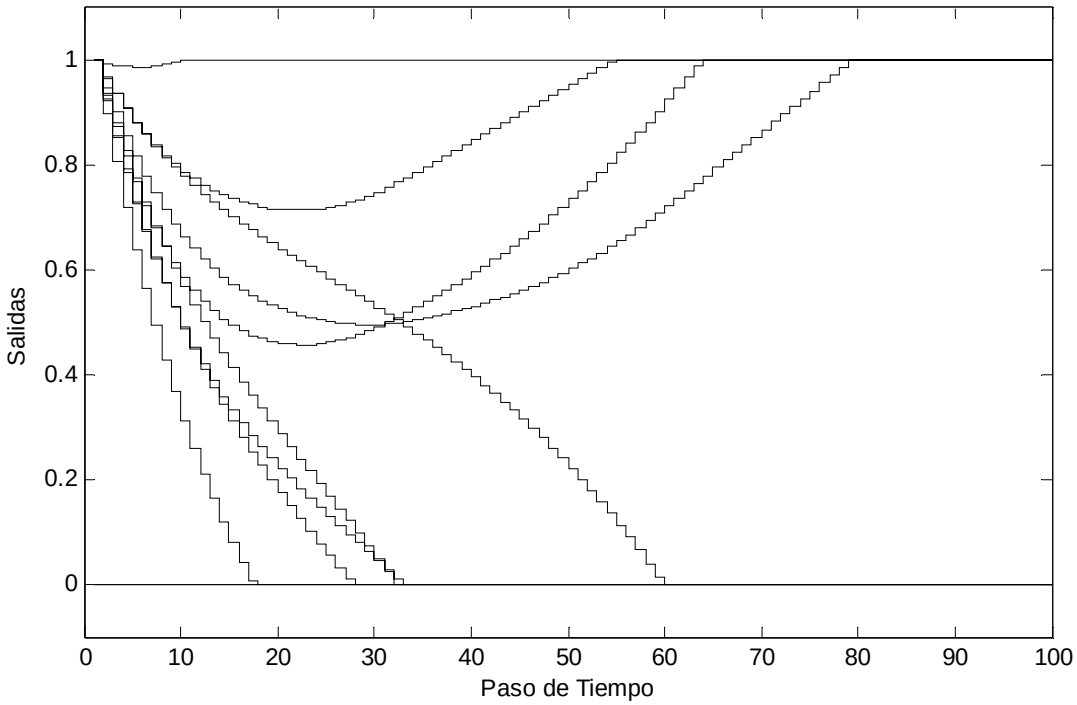


Figura 2.5. *Evolución de las salidas de la CNN en tiempo discreto.*

En [2] se menciona que el tiempo de convergencia de la red para una constante de tiempo $\tau = R_x C_x$ es de aproximadamente 0.3τ. De acuerdo a las simulaciones del modelo discretizado, donde se ha definido un paso de tiempo $h = 0.01$ con el cual se alcanza el valor de τ en 100 iteraciones, el tiempo de convergencia es de aproximadamente 0.8τ. Esto es debido al método de discretización usado. El método de Euler genera errores de discretización más grandes que otros métodos de mayor orden. Sin embargo, este método prueba tener la convergencia necesaria para generar la salida correcta de la red, a costa de un tiempo de procesamiento un poco mayor, pero que aún así se mantiene dentro de un margen aceptable. Además, el método de Euler genera un algoritmo numérico reducido que no necesita una gran cantidad de componentes al ser implementado en hardware.

Como se puede observar a partir de estos resultados, la implementación de esta CNN en una dispositivo programable requerirá de, aproximadamente, 80 iteraciones del algoritmo para obtener una respuesta. Por supuesto, el tiempo requerido por iteración dependerá del diseño lógico final que se implemente, lo cual será tratado posteriormente al realizar la discretización del espacio de valores y seleccionar los componentes internos de la red digital.

2.4. Implementación de un Simulador de CNNs.

Si bien la simulación de una red neuronal completa en un entorno de simulación de propósito general como Simulink resulta eficiente y proporciona resultados confiables, resulta difícil hacer cambios rápidos a los parámetros de red, así como proporcionar los datos de entrada

para la red y visualizar sus salidas. Por esto se ha implementado un simulador completo de CNNs que permite simular tanto la red extendida de interés para este trabajo como redes neuronales celulares generales. Presenta la ventaja de tener una interfaz gráfica que permite la entrada directa de datos de entrada a la red y visualización de resultados y estados internos de la red. También permite hacer fácilmente cambios a los parámetros de red. Además de todo esto, la programación del simulador permite entender de manera muy clara el algoritmo necesario para la implementación posterior del modelo de tiempo discreto de CNN extendida en un procesador.

2.4.1. Diseño del Simulador.

La implementación del simulador fue realizada en Microsoft Visual Basic, un lenguaje de programación con eficientes interfaces gráficas para Windows. En la figura 2.6 se muestra la pantalla principal de simulación. Permite simular redes de hasta 200 x 200 celdas, con radio de vecindario de tamaño arbitrario. Cuenta con 8 funciones de transferencia para el modelo de la neurona, puede fijar la configuración de frontera de la red en 3 modos de continuidad distintos y con valores definidos por el usuario. Permite elegir el método de simulación, ofreciendo métodos de discretización de hasta cuarto grado.

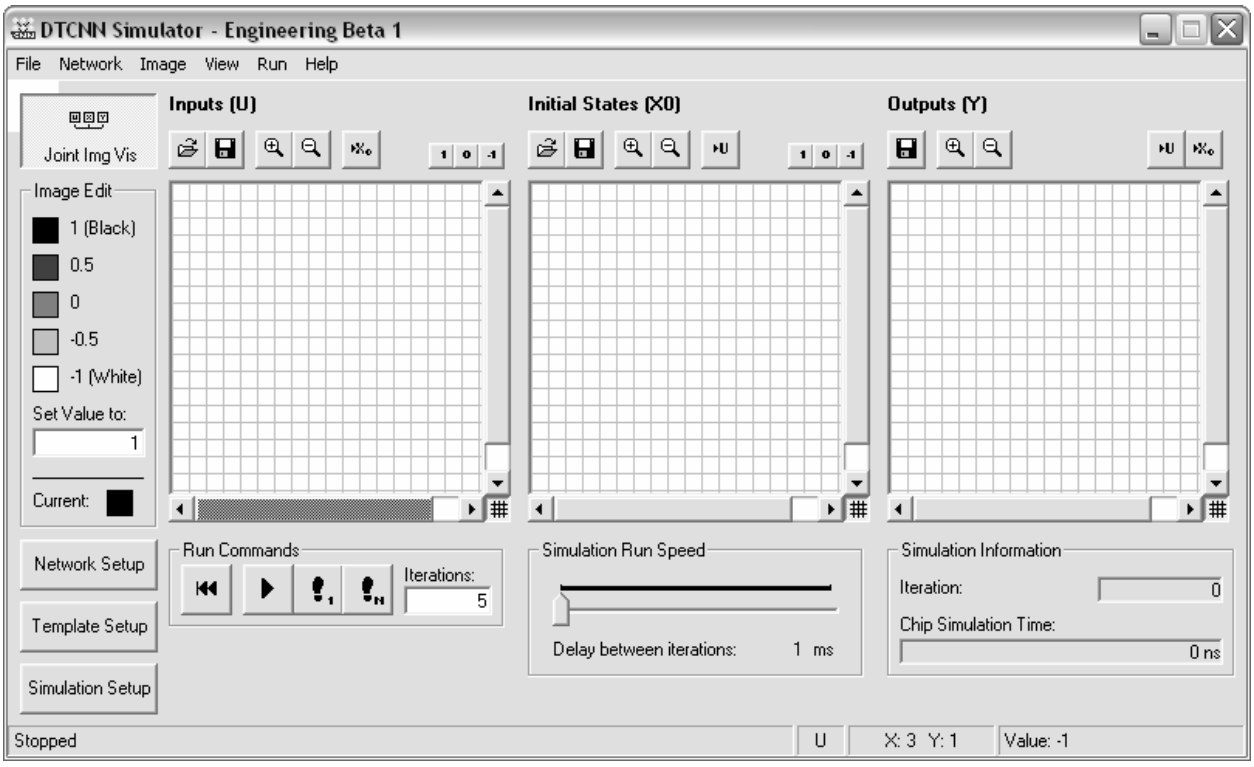


Figura 2.6. *Interfaz del Simulador de CNNs.*

2.4.2. Comparación de Resultados y Análisis de Desempeño.

A pesar de que el simulador fue implementado en un lenguaje que no es óptimo para cálculos numéricos extensivos, muestra una buena respuesta, comparable a otros simuladores

similares. Los resultados obtenidos al simular patrones de prueba fueron coherentes con los resultados obtenidos por la simulación de los modelos de Simulink.

2.5. Conclusiones

En este capítulo se ha descrito el problema central de la tesis: la implementación de una red neuronal celular extendida para el control de una red de conmutación rápida de paquetes. Se describió su estructura y propiedades y se demostró su estabilidad. Posteriormente se llevó a cabo el primer paso para su implementación en un dispositivo programable, que consiste en derivar el modelo de tiempo discreto de la red. Se realizaron simulaciones utilizando Simulink y un simulador creado específicamente para este problema, y se obtuvieron resultados satisfactorios.

Referencias

1. R. Fantacci, M. Forti and M. Marini, “A Cellular Neural Network for Packet Selection in a Fast Packet Switching Fabric with Input Buffers”, IEEE Trans. Communications, vol. 44, no. 12, pp. 1649–1652, 1996.
2. R. Fantacci, M. Forti, M. Marini and Luca Pancani, “Cellular Neural Network Approach to a Class of Communication Problem”, IEEE Trans. Circuits Syst. – 1: Fundamental Theory and App., vol. 46, no. 12, pp. 1457–1467, 1999.
3. Zill, Dennis, “Ecuaciones Diferenciales con Aplicaciones de Modelado”, Internacional Thomson Editores, 1997.

CAPÍTULO 3

Diseño Digital de Red CNN Extendida.

Equation Section (Next)

En el capítulo anterior se llevó a cabo la discretización en tiempo del modelo de CNN para la red de FPS. Para la creación de un sistema digital que realice las funciones requeridas necesitamos discretizar también el espacio de valores de toda la red. Para este fin, en este capítulo describiremos primero la estructura general que tendrá la neurona digital, basándonos en el modelo desarrollado en el capítulo anterior. Posteriormente describiremos y seleccionaremos los componentes necesarios para conformar la red, y finalmente definiremos el espacio de valores digitales que caracterizará a la red.

3.1. Planteamiento y Definición de Arquitectura Digital.

A fin de definir la estructura general de la CNN digital, comenzaremos con el modelo de tiempo discreto mostrado en 4 del Apéndice D. Este modelo se puede traducir directamente a un sistema digital si se seleccionan adecuadamente los tamaños de palabra y los componentes necesarios, pero tiene el inconveniente de requerir varios multiplicadores y sumadores, lo que significa que requerirá de grandes recursos —en términos de compuertas en un dispositivo programable o área en un diseño *custom* VLSI— si se traslada directamente. Debido a esto es necesario llegar a un compromiso entre área de dispositivo y tiempo de procesamiento, a fin de que nuestra red pueda ser implementada en un dispositivo de tamaño adecuado para que el costo no sea un factor determinante.

Para poder disminuir los recursos requeridos, se opta por una arquitectura que permita compartir recursos que generalmente utilizan mucho espacio, como los multiplicadores y los sumadores multioperando. Para este fin se utilizan multiplexores que permitan el adecuado acceso a los recursos compartidos. También se hacen otras consideraciones que simplifican el diseño de la neurona digital.

3.1.1 Definición del Modelo Digital.

El modelo digital se definirá en base a las siguientes consideraciones:

A fin de minimizar el tamaño de la neurona, se utilizará un solo multiplicador, el cual será compartido por todos los términos de entrada y de realimentación del estado, y que operará de manera secuencial sobre los mismos. Dicho multiplicador deberá operar sobre términos con signo, ya que la mascarilla tiene elementos que pueden tomar valores tanto positivos como negativo, e igualmente el estado interno de la neurona. Para acumular la suma de los productos se utilizará un acumulador saturado con signo, a fin de realizar la suma secuencial. Esto se hará con un sumador saturado y un registro. La saturación es una característica que se debe incluir, ya que es posible una inversión en el estado de la neurona al transcurrir un tiempo de ejecución muy prolongado, debido a las características del sistema de complemento a dos.

Para las entradas y las salidas de las neuronas vecinas es necesario hacer la suma de seis términos. Se utilizará un solo sumador de seis operandos, el cual será compartido por ambos conjuntos de entradas. Ya que la neurona utiliza una función de transferencia de salida de tipo saturación positiva, los términos son fracciones positivas, $u_n, y_n \in [0,1]$, y por tanto el sumador asume que todos los términos son positivos, lo que hace innecesario un bit de signo.

A partir de la ecuación (2.19) podemos ver que el estado se actualiza en cada iteración, sumándose la expresión $hf\left(x_{ij}[n]\right)$ al estado actual. La función $f\left(x_{ij}[n]\right)$ se puede expandir en

$$f\left(x_{ij}[n]\right) = -x_{ij}[n] + g\left(x_{ij}[n]\right), \tag{3.1}$$

y al sustituir y agrupar términos se obtiene que el estado siguiente de la neurona es

$$x_{ij}[n+1] = (1-h)x_{ij}[n] + g\left(x_{ij}[n]\right), \tag{3.2}$$

por lo que resulta posible ahorrar un paso de suma al multiplicar el estado actual por $(1-h)$ y sumarlo al resto de los términos, denotados por $g\left(x_{ij}[n]\right)$.

Para evitar la multiplicación de la suma total por *h*, es posible programar la red con los valores previamente multiplicados $(A', B', \alpha', \beta', I')$ a fin de eliminar un paso de multiplicación. Tanto el término $(1-h)$ como estos valores de la mascarilla serán almacenados en la red y serán multiplicados secuencialmente por sus términos de entrada correspondientes.

La salida del multiplicador deberá ser truncada a fin de obtener sólo el número de bits requeridos para alcanzar la precisión deseada, por lo que se utilizará un bloque de redondeo o truncamiento para retener solo los bits necesarios.

Finalmente, se agrega un registro que contendrá el estado en cada iteración de la red, y se desarrollará la lógica para definir la función de saturación a la salida de la neurona digital. También debe ser definido el estado inicial de la neurona, lo cual se hará suministrando una entrada al registro del estado para definir su valor al inicio del ciclo de solución.

En base a las consideraciones anteriores, obtenemos una estructura general como la que se muestra en la Fig. 3.1. Esta estructura corresponde a cada neurona de la red, y está ejemplificada para una red de 4x4. Ahora consideraremos las estructuras aritméticas, como los sumadores, multiplicadores y funciones. Las consideraciones sobre el módulo de control, fuentes de datos para la estructura neuronal y programación de las mascarillas se tratarán en el siguiente capítulo, así como las estructuras de multiplexado.

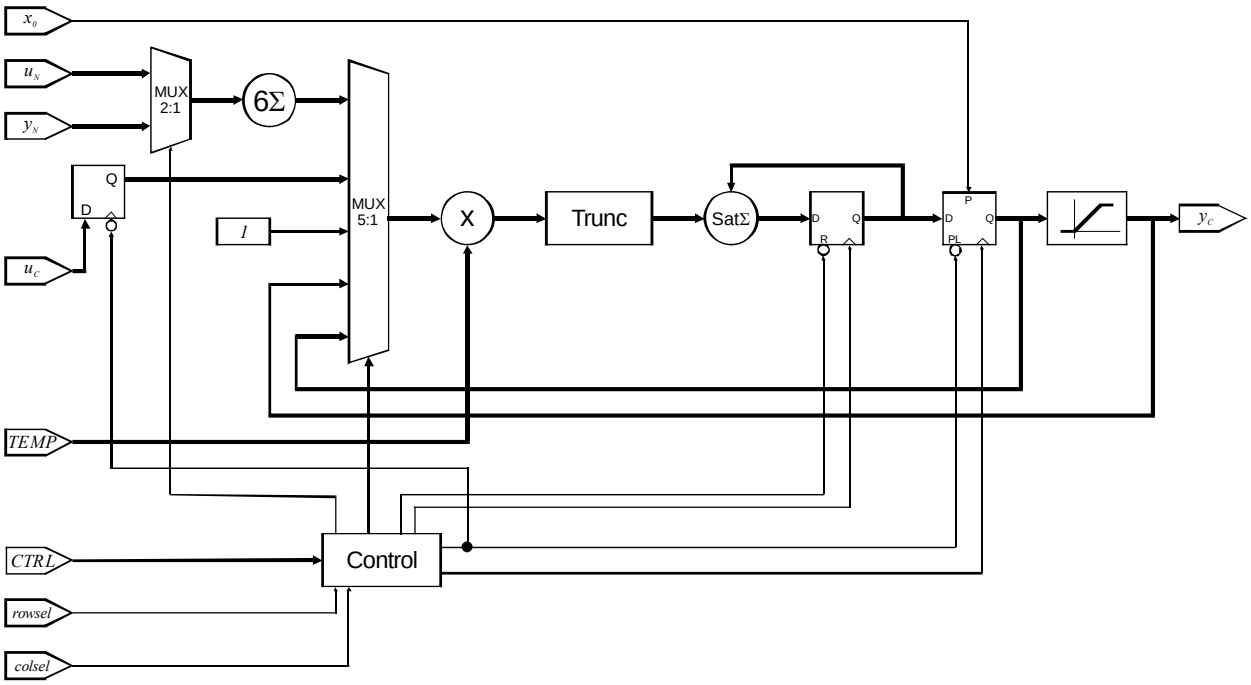


Figura 3.1. *Estructura general de la neurona digital.*

3.1.2. Elección del espacio de valores digitales.

A fin de discretizar el espacio de valores de la neurona digital, es necesario asignar los tamaños de palabra para cada uno de los componentes y señales que constituyen el modelo digital. Estos tamaños de palabra deben proporcionar la suficiente resolución de forma que la operación de la neurona se la correcta y la red neuronal completa llegue a una convergencia en su operación.

Es necesario definir los tamaños de palabra de los siguientes elementos:

- Estado interno de la neurona (x_{ij}).
- Elementos de las mascarillas.
- Paso de tiempo (h).
- Señales interneuronales (y_{ij}).
- Señales de entrada (u_{ij}).

La definición de sus tamaños de palabra se hará de acuerdo a los valores para los elementos de las mascarillas obtenidos por medio de las fórmulas (2.6), el paso de integración h necesario y la resolución deseada para la prioridad de los paquetes dados en la matriz W.

Señales de Entrada.

Las señales de entrada corresponden a los valores de la matriz W, es decir, las prioridades de conexión de los paquetes. Estos valores entrarán a la neurona por medio de las entradas de control *u*. Podemos definir el número mínimo de bits necesarios para esta señal de acuerdo a

$$n_u = \lceil \log_2 l \rceil \tag{3.3}$$

donde *l* es el número de niveles de prioridad que se desea tener para el sistema. Por ejemplo, si deseamos tener 10 valores de prioridad desde 0.1 hasta 1.0 (espaciamiento de 0.1), el número mínimo de bits requeridos será de $\lceil \log_2 10 \rceil = \lceil 3.32 \rceil = 4$. Como en esta red empleamos sólo valores positivos, no incluimos el bit de signo.

Elementos de las Mascarillas.

Para determinar el tamaño de palabra mínimo para los elementos de mascarillas tenemos que considerar todos los elementos que conforman la mascarilla, y elegir un número de bits que permita representar estos valores con precisión. En general, el número mínimo de bits necesario será

$$n_t = \left\lceil \left| \log_2 \left(\max f_i \left(|\mathbf{p}| \right) \right) \right| \right\rceil + \left\lceil \left| \log_2 \left(\min f_f \left(|\mathbf{p}| \right) \right) \right| \right\rceil + 1 \tag{3.4}$$

donde **p** está definido como en la sección 1.2.1 y representa el conjunto de valores de las mascarillas diferentes de cero. Las funciones f_i y f_f obtienen las partes enteras y fraccionarias, respectivamente. Esta ecuación tiene en cuenta la inclusión de un bit de signo y los sumandos deben evaluarse sólo si existe la correspondiente parte entera o parte fraccionaria. Por ejemplo, si nuestro conjunto de mascarillas fuera $\mathbf{p} = [1 \quad 4 \quad -2.2 \quad 1.1 \quad -2 \quad 1]$, la evaluación de la función resultaría en

$$\begin{aligned} n_t &= \left\lceil \left| \log_2 \left(\max f_i \left(\left| [1 \quad 4 \quad -2.2 \quad 1.1 \quad -2 \quad 1] \right| \right) \right) \right| \right\rceil + \\ &\quad \left\lceil \left| \log_2 \left(\min f_f \left(\left| [1 \quad 4 \quad -2.2 \quad 1.1 \quad -2 \quad 1] \right| \right) \right) \right| \right\rceil + 1 \\ &= \left\lceil \left| \log_2 \left(\max (1 \quad 4 \quad 2 \quad 1 \quad 2 \quad 1) \right) \right| \right\rceil + \left\lceil \left| \log_2 \left(\min (0 \quad 0 \quad 0.2 \quad 0.1 \quad 0 \quad 0) \right) \right| \right\rceil + 1 \\ &= \left\lceil \left| \log_2 4 \right| \right\rceil + \left\lceil \left| \log_2 0.1 \right| \right\rceil + 1 = \lceil 2 \rceil + \lceil 3.32 \rceil + 1 = 7 \text{ bits} \end{aligned}$$

En nuestro ejemplo de la sección 2.2.3, los elementos son puramente fraccionarios, requiriendo 7 bits más 1 bit de signo.

Paso de tiempo.

El paso de tiempo estará incluido dentro de nuestras mascarillas, por lo que necesitaremos, en la sección anterior, definir los tamaños de palabra para los elementos de mascarillas multiplicados por el paso de tiempo *h*. Igualmente, tenemos que definir el tamaño

necesario para el parámetro $(1-h)$, el cual estará considerado como otro elemento más de mascarillas. Podemos definir el número de bits necesarios de la misma manera que en el paso anterior, incluyendo el valor de *h* entre los parámetros de mascarillas, pero debemos tomar en cuenta que debemos programar el valor de $(1-h)$. Al obtener el tamaño necesario para *h* aseguramos que la precisión necesaria para el parámetro $(1-h)$ será obtenida.

Estado Interno.

El número de bits necesarios para el estado interno dependerá, por un lado, de los máximos valores, tanto positivos como negativos, que alcance el estado de la neurona durante su operación y hasta que se logre la convergencia de la red, y por otro lado, de la resolución necesaria para lograr, de nuevo, la convergencia de la red. Ya que esta red trabaja con estados tanto positivos como negativos, necesitaremos también incluir un bit de signo. La forma más conveniente de determinar los máximos valores obtenidos es por simulación, ya que el estimado obtenido por medio de la determinación del rango dinámico de la neurona resulta muy por encima de los valores requeridos, esto debido a que nuestra red tiene tiempos de convergencia muy cortos [4].

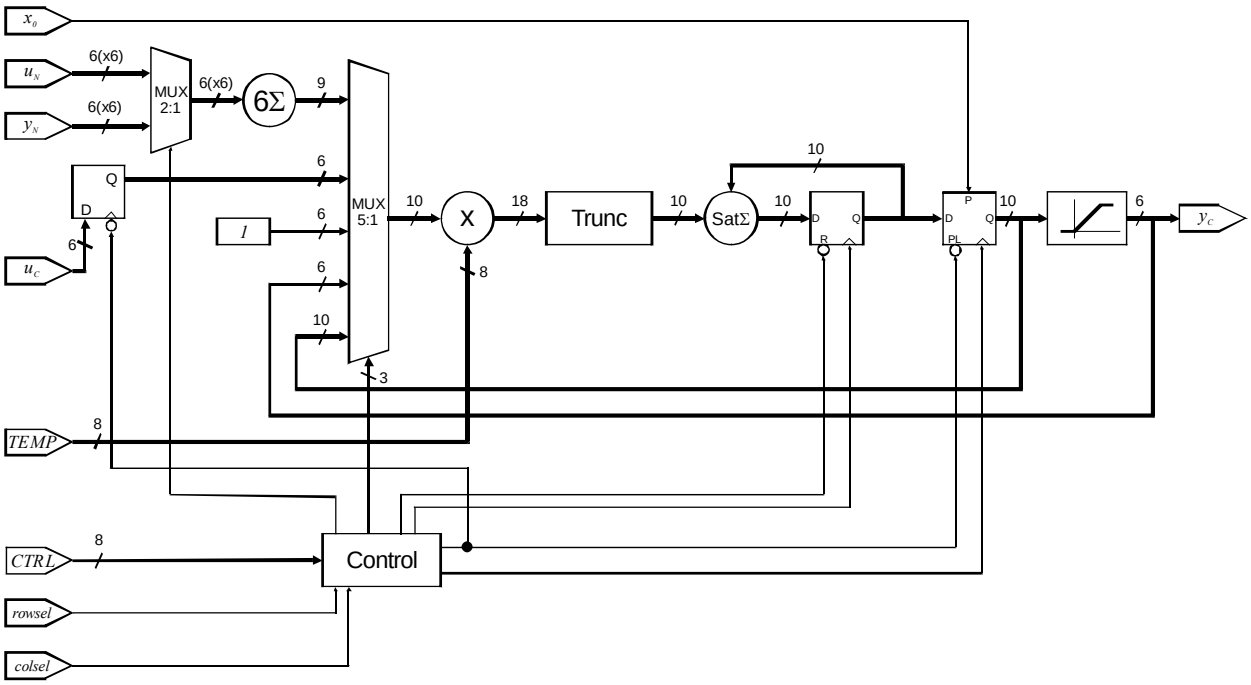
Utilizando los valores ejemplo (2.11) de la sección 2.2.3, los valores extremos aproximados para el estado resultan ser de 3 y -8. El máximo absoluto (aunque negativo) de 8 indica la necesidad de utilizar 3 bits más un bit de signo para la parte entera, mientras que para la parte fraccionaria podemos utlizar el mismo criterio que para la obtención de los bits necesarios para la parte fraccionaria de las mascarillas. Para nuestro ejemplo, 10 bits resultan ser una elección adecuada.

Señales Interneuronales.

Las señales interneuronales se refieren a las salidas *y* de las neuronas, que se conectan a las entradas de otras neuronas. Ya que trabajaremos con la función de transferencia tipo saturación lineal positiva, sólo tendremos valores positivos. Es por tanto conveniente que el número de bits para estas señales sea igual al número de bits requeridos para la parte fraccionaria del estado, mas un bit para indicar el valor de 1.

Neurona Digital Ejemplo.

Continuando con el ejemplo de la sección 2.2.3, la neurona para una red de 4x4 con los tamaños de palabra resultantes se muestra a continuación.



3.2. Módulos de Procesamiento.

Ya se han definido los módulos de procesamiento que constituirán la neurona digital. Ahora es necesario definir las características de cada módulo y, en base a estas, seleccionar y desarrollar las arquitecturas más eficientes.

3.2.1. Sumador multioperando.

A fin de acelerar el proceso de adición a la entrada de la neurona, se hará uso de un sumador multioperando de tipo Almacenamiento de Acarreo (CSA). Este sumador se seleccionó debido a que ofrece una obvia ventaja en cuanto a velocidad sobre un proceso de adición secuencial. Aunque su complejidad es mayor, su estructura regular no consume demasiados recursos de área.

Hay que notar que, debido a que la neurona cuenta con una función de activación no lineal de tipo saturación positiva, y como todas sus entradas serán positivas, todos los valores de entrada a la red y que serán sumados por este sumador serán positivos. Por lo mismo, se tomarán en cuenta todos los bits de salida. Si se quisiera implementar una red con entradas y salidas de signos tanto positivos como negativos se podría utilizar este mismo sumador, con la única diferencia de que se tendrá que ignorar el bit más significativo (MSB) del resultado.

Sumador de Almacenamiento de Acarreo (CSA). [1]

Cuando tres o más operandos son sumados simultáneamente usando sumadores de dos operandos, la propagación del acarreo debe repetirse varias veces, lo cual consume mucho

tiempo. Si el número de operandos es *k*, los acarreos deberán propagarse $(k-1)$ veces. Se han propuesto varias técnicas para adición de múltiples operandos que intentan reducir el tiempo de propagación del acarreo. La técnica más comúnmente usada es la Adición con Almacenamiento de Acarreo (*Carry-Save Addition*). En este tipo de adición los acarreos se propagan sólo en el último paso, mientras que en todos los otros pasos generamos sumas parciales y una secuencia de acarreos separadamente. Así pues, un sumador CSA acepta tres operandos de *n* bits y genera dos resultados de *n* bits: una suma parcial de *n* bits y un acarreo de *n* bits. Un segundo CSA acepta estas dos secuencias de bits y otro operando, y genera una nueva suma parcial y un acarreo. Un CSA es por tanto capaz de reducir el número de operando a ser sumados de 3 a 2 sin necesidad de propagar un acarreo.

Un sumador CSA puede ser implementado de diferentes maneras. En su implementación más simple, el elemento básico del CSA es un sumador completo (FA, *Full Adder*) con tres entradas, *x*, *y* y *z*, y cuya operación aritmética se puede describir por

$$x + y + z = 2c + s\,, \tag{3.5}$$

donde *s* y *c* son la suma y el acarreo, respectivamente. Sus valores son

$$s = (x + y + z) \bmod 2 \quad y \quad c = \frac{(x + y + z) - s}{2}. \tag{3.6}$$

Las salidas son la representación binaria ponderada del número de 1's en las entradas. Llamamos entonces al FA un contador $(3,2)$, como se muestra en la figura 3.2. Un CSA de *n* bits consiste entonces de *n* contadores $(3,2)$ operando en paralelo sin conexiones de acarreo ente ellos.

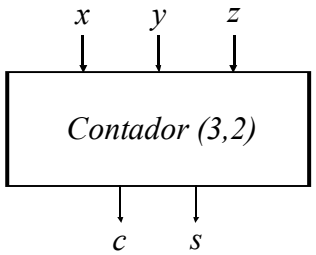


Figura 3.2. *Contador (3,2).*

Un ejemplo de CSA para cuatro operandos de cuatro bits se muestra en la figura 3.3. Los dos niveles superiores son CSAs de 4 bits, mientras que el tercer nivel es un Sumador con Propagación de Acarreo (CPA, *Carry-Propagate Adder*). Hay que notar que los bits de suma parciales y los bits de acarreo deben interconectarse de modo que sólo los bits con el mismo peso sean sumados por el mismo contador $(3,2)$.

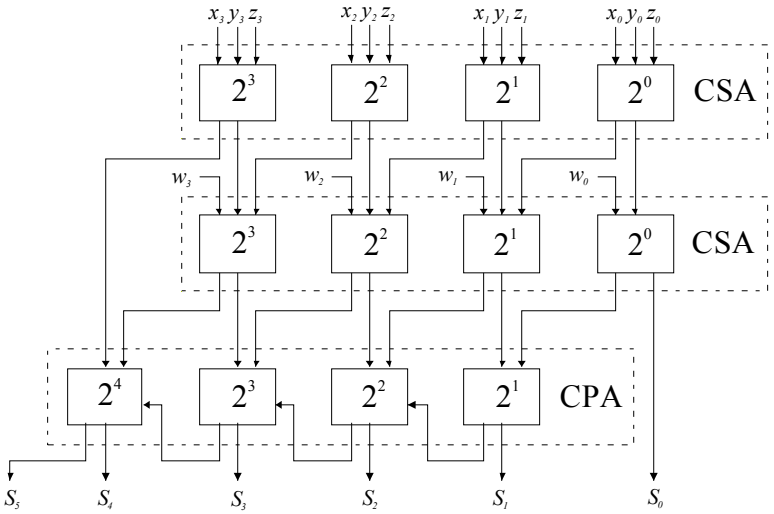


Figura 3.3. *CSA para cuatro operandos de cuatro bits.*

A fin de sumar *k* operandos $X_1, X_2, \ldots, X_k$, necesitamos $(k-2)$ unidades CSA y un CPA. Si éstos se arreglan en cascada como en la figura 3.3, el tiempo de suma de los *k* operandos es

$$(k-2)T_{CSA} + T_{CPA} \tag{3.7}$$

donde T_{CPA} es el tiempo de operación del CPA elegido y T_{CSA} es el tiempo de operación del CSA, el cual es equivalente a un sumador completo. El resultado final alcanzará una longitud máxima de

$$n + \log_2 k \tag{3.8}$$

bits, ya que la suma de *k* operandos de *n* bits puede ser tan grande como $(2^n - 1)k$.

Árbol de Wallace. [1]

Una mejor manera de organizar los CSAs a fin de reducir el tiempo de operación es formar una estructura de árbol, conocida como árbol de Wallace. Un ejemplo de seis operandos, como el requerido para nuestro ejemplo, se muestra en la figura 3.4. Las flechas izquierdas en las salidas de los CSAs indican que estas deben ser desplazadas a la izquierda antes de ser alimentadas a los siguientes niveles, de forma similar a como se muestra en la figura 3.3. Obviamente, las salidas deben ser alimentadas a un CPA posteriormente, así como los acarreos no procesados por las unidades CSA.

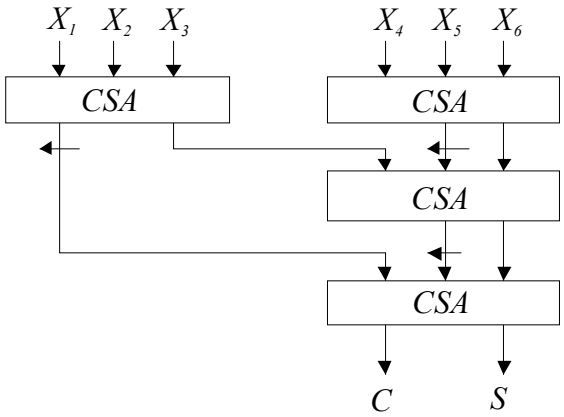


Figura 3.4. *Arbol de Wallace para seis operandos.*

En el árbol de Wallace, el número de operandos es reducido por un factor de 2/3 en cada nivel. Consecuentemente, el número de niveles requerido será aproximado por

$$\frac{\log(k/2)}{\log(3/2)} \tag{3.9}$$

Esta ecuación provee sólo un estimado del número de niveles, ya que en cada nivel el número de operandos debe ser un entero. Así, si N_i es el número de operandos en el nivel *i*, el número de operandos en el nivel superior $(i+1)$ puede ser, a lo más, $\lfloor N_i \cdot 3/2 \rfloor$ (donde la función $\lfloor x \rfloor$ de un número *x* es el entero más grande que es más pequeño que o igual a *x*). El número de operandos a nivel base (nivel 0) es 2, así que el máximo número de operandos en el nivel 1 es 3, en el nivel 2 es 4, y así sucesivamente. La secuencia resultante es 2, 3, 4, 6, 9, 13, 19, 28, etc. Esto se deberá tomar en cuenta para el diseño del sumador multioperando de entrada para redes neuronales de diferentes tamaños.

3.2.2. Multiplicador.

El multiplicador requiere un diseño especial no simétrico, ya que se requiere multiplicar dos números en forma de complemento a dos y de tamaños diferentes. También requerimos que la multiplicación se realice de manera paralela, sin utilizar el método de sumas y desplazamientos sucesivos, ya que la multiplicación es el paso que toma más tiempo dentro del algoritmo neuronal y además se realizarán multiplicaciones y acumulaciones sucesivas.

Dentro de los multiplicadores que pueden trabajar directamente con números en forma de complemento a dos, se seleccionó un multiplicador de arreglo Baugh-Wooley. Su principal ventaja es que el signo de todos los sumandos es positivo, permitiendo que el arreglo se construya enteramente con sumadores tipo 0, es decir, sin entradas complementadas. Esto permite también que, al momento de implementarlo en un dispositivo programable, se obtenga un arreglo bastante regular y uniforme. Consideremos ahora la forma en que este arreglo se obtiene.

Multiplicador Baugh-Wooley de Complemento a Dos. [2]

Considérense dos enteros en complemento a 2, un multiplicando de *m* bits $\mathbf{A} = \left(a_{m-1}a_{m-2}\ldots a_1a_0\right)_2$ y un multiplicador de *n* bits $\mathbf{B} = \left(b_{n-1}b_{n-2}\ldots b_1b_0\right)_2$. Los valores de **A** y **B** denotados por $\mathbf{A}_v$ y $\mathbf{B}_v$ pueden escribirse como

$$\begin{aligned}\mathbf{A}_v &= -a_{m-1}2^{m-1} + \sum_{i=0}^{m-2} a_i 2^i \\ \mathbf{B}_v &= -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i\end{aligned} \tag{3.10}$$

El valor $\mathbf{P}_v$ del producto $\mathbf{P} = \mathbf{A}\cdot\mathbf{B} = \left(p_{m+n-1}p_{m+n-2}\ldots p_1p_0\right)_2$ en notación de complemento a dos puede ser escrito como sigue en términos de los productos de los coeficientes de a_i's y b_j's con los factores de ponderación apropiadamente agregados

$$\begin{aligned}\mathbf{P}_v &= -p_{m+n-1}2^{m+n-1} + \sum_{i=0}^{m+n-2} p_i 2^i = \mathbf{A}_v\mathbf{B}_v \\ &= \left(-a_{m-1}2^{m-1} + \sum_{i=0}^{m-2} a_i 2^i\right)\left(-b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i\right) \\ &= a_{m-1}b_{n-1}2^{m+n-2} + \sum_{i=0}^{m-2}\sum_{j=0}^{n-2} a_ib_j 2^{i+j} - \sum_{i=0}^{m-2} a_ib_{n-1}2^{n-1+i} - \sum_{i=0}^{n-2} a_{m-1}b_i 2^{m-1+i}\end{aligned} \tag{3.11}$$

En esta ecuación, los signos de los sumandos a_ib_{n-1} para $i = 0,\ldots,m-2$, y de $a_{m-1}b_j$ para $j = 0,\ldots,n-2$ son todos negativos. Colocando todos los sumandos negativos en los dos últimos renglones como se muestra en la figura 3.5, el producto se puede formar sumando los primeros $n-2$ renglones de sumandos y sustrayendo los dos últimos renglones. En vez de sustraer los sumandos negativos, se agrega la negación de los sumandos.

					a_{m-1}	a_{m-2}		...	a_3	a_2	a_1	a_0	**=A**
				x			b_{n-1}		...	b_2	b_1	b_0	**=B**
						$a_{m-2}b_0$		...	a_3b_0	a_2b_0	a_1b_0	a_0b_0	
					$a_{m-2}b_1$		...	a_3b_1	a_2b_1	a_1b_1	a_0b_1		
P=AxB				$a_{m-2}b_2$		...	a_3b_2	a_2b_2	a_1b_2	a_0b_2			Sumandos
				⋰					⋰				Positivos
	$a_{m-1}b_{n-1}$	0	$a_{m-2}b_{n-2}$		...	a_2b_{n-2}	a_1b_{n-2}	a_0b_{n-2}					
0	0	$a_{m-2}b_{n-1}$	$a_{m-3}b_{n-1}$	...		a_1b_{n-1}	a_0b_{n-1}						Sumandos
0	0	$a_{m-1}b_{n-2}$	$a_{m-1}b_{n-3}$	...	$a_{m-1}b_0$								Negativos
P_{m+n-1}	P_{m+n-2}	P_{m+n-3}	P_{m+n-4}	...	P_{m-1}	...	P_{n-1}		...	P_2	P_1	P_0	**=P**

Figura 3.5. *Segregación de sumandos positivos y negativos en un multiplicador de complemento a dos de m por n.*

Podemos reemplazar la sustracción del tercer término de la ecuación (3.11)

$$\sum_{i=0}^{m-2} a_i b_{n-1} 2^{n-1+i} = 2^{n-1}\left(-0\cdot 2^m + 0\cdot 2^{m-1} + \sum_{i=0}^{m-2} a_i b_{n-1} 2^i\right) \tag{3.12}$$

por la adición de

$$2^{n-1}\left(-1\cdot 2^m + 1\cdot 2^{m-1} + 1 + \sum_{i=0}^{m-2} \overline{a}_i b_{n-1} 2^i\right) \tag{3.13}$$

Note que esta ecuación tiene el valor

$$\begin{cases} 0 & \text{para } b_{n-1} = 0 \\ 2^{n-1}\left(-2^m + 2^{m-1} + 1 + \sum_{i=0}^{m-2} \overline{a}_i 2^i\right) & \text{para } b_{n-1} = 1 \end{cases} \tag{3.14}$$

De esta ecuación, la ecuación (3.13) puede ser rescrita como

$$2^{n-1}\left(-2^m + 2^{m-1} + \overline{b}_{n-1} 2^{m-1} + b_{n-1} + \sum_{i=0}^{m-2} \overline{a}_i b_{n-1} 2^i\right) \tag{3.15}$$

Esto implica que el penúltimo renglón de la figura 3.5

$$0 \quad 0 \quad a_{m-2}b_{n-1} \quad a_{m-3}b_{n-1} \quad \cdots \quad a_0 b_{n-1} \tag{3.16}$$

puede reemplazarse por la suma de los dos siguientes vectores fila

$$\begin{array}{ccccccc} 0 & \overline{b}_{n-1} & \overline{a}_{m-2}b_{n-1} & \overline{a}_{m-3}b_{n-1} & \ldots & \overline{a}_0 b_{n-1} \\ 1 & 1 & 0 & 0 & \ldots & b_{n-1} \end{array} \tag{3.17}$$

Similarmente, podemos reemplazar la sustracción del cuarto término $\sum_{i=0}^{m-2} a_{m-1} b_i \times 2^{m-1+i}$ en la ecuación (3.13) por la adición de

$$2^{m-1}\left(-2^n + 2^{n-1} + \overline{a}_{m-1} 2^{n-1} + a_{m-1} + \sum_{i=0}^{n-2} a_{m-1} \overline{b}_i 2^i\right) \tag{3.18}$$

Esto implica que el último vector fila de la figura 3.5,

$$0 \quad 0 \quad a_{m-1}b_{n-2} \quad a_{m-1}b_{n-3} \quad \ldots \quad a_{m-1}b_0 \tag{3.19}$$

puede reemplazarse por la suma de los siguientes vectores fila

$$\begin{array}{ccccccc} 0 & \overline{a}_{m-1} & a_{m-1}\overline{b}_{n-2} & a_{m-1}\overline{b}_{n-3} & \cdots & a_{m-1}\overline{b}_0 \\ 1 & 1 & 0 & 0 & \cdots & a_{m-1} \end{array} \tag{3.20}$$

Podemos combinar el segundo renglón de la ecuación (3.17) y el segundo renglón de la ecuación (3.20) en un solo vector fila como se muestra abajo, ignorando el acarreo saliente de la columna P_{m+n-1} debido a la naturaleza de la adición en complemento a dos

$$\begin{array}{cccccccccccccc} & 1 & 0 & \ldots & 0 & a_{m-1} & 0 & \ldots & 0 & b_{n-1} & 0 & \ldots & 0 & \\ & \uparrow & & & & \uparrow & & & & \uparrow & & & \uparrow & \\ \textit{Columna} & p_{m+n-1} & & & & p_{m-1} & & & & p_{n-1} & & & p_0 & \end{array} \tag{3.21}$$

Tras la sustitución de los dos renglones en la figura 3.5 usando las fórmulas (3.17), (3.20) y (3.21), obtenemos la nueva matriz de sumandos como se muestra en la figura siguiente:

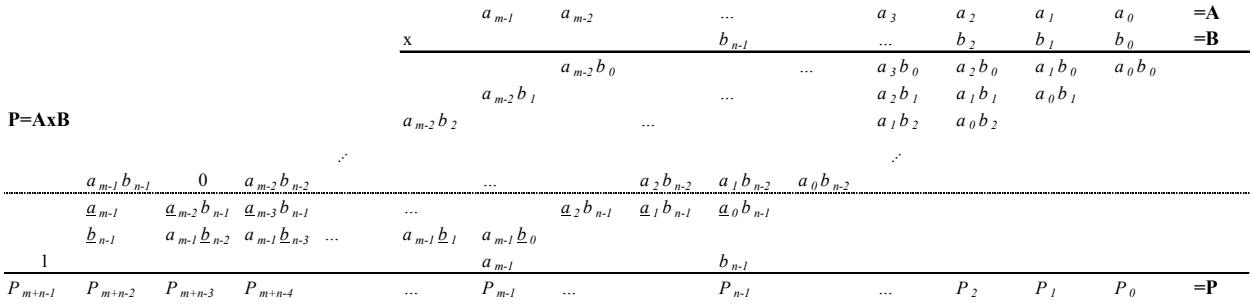


Figura 3.6. *El algoritmo de Baugh-Wooley para multiplicación en complemento a dos con todos los sumandos positivos, obtenido usando el complemento a dos de los sumandos negativos en la figura 3.5 (Se asume que $m > n$).*

la característica principal de esta nueva matriz es su uniformidad, ya que consiste solamente en sumandos positivos. Por tanto, el producto se puede obtener realizando la adición sólo con sumadores completos tipo 0 (sin entradas complementadas).Como ejemplo, un diagrama de circuito lógico para un multiplicador de arreglo Baugh-Wooley de complemento a dos se muestra en la figura 3.7. Note que si las entradas complementarias no están directamente disponibles en el bus de datos, se requerirán inversores. En general, un multiplicador Baugh-Wooley de *m* por *n* requerirá de $m(n-1)+3$ sumadores completos para su implementación. Su retardo de operación es de $2(m+n)\Delta$.

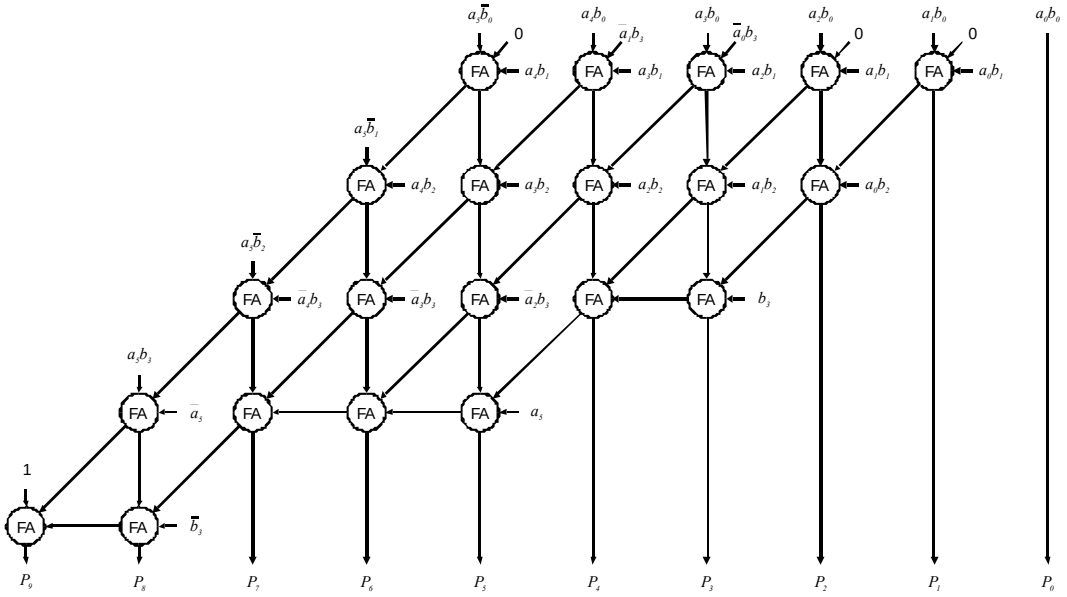


Figura 3.7. *Circuito esquemático lógico de un multiplicador de arreglo Baugh-Wooley para complemento a dos.*

3.2.3. Acumulador Saturado.

Debido a las características del sistema de complemento a dos, la neurona puede sufrir una inversión de estado al realizar acumulaciones sucesivas durante un tiempo prolongado. Para evitar esta situación, el acumulador se implementa de manera que, al recibir un sumando que podría causar una inversión, se genere una condición de saturación que coloque el acumulador en su valor máximo. De manera general, el sumador saturado se podría definir como sigue:

$$\text{satsum}(a,b) = \begin{cases} a+b & \text{si } minval \le (a+b) \le maxval \\ maxval & \text{si } (a+b) > maxval \\ minval & \text{si } (a+b) < minval \end{cases} \tag{3.22}$$

donde los valores de *maxval* y *minval* son el máximo positivo y el mínimo negativo, respectivamente, de los números representables en complemento a dos con un número dado de bits y que corresponde al tamaño de palabra del acumulador.

La implementación del acumulador saturado se puede realizar fácilmente con un sumador estándar y un registro, retroalimentando la salida del registro y utilizándola como sumando. Se utilizan las siguientes reglas:

- Si los signos de los sumandos son diferentes, se acumula la suma.
- Si los signos de los sumandos son iguales y el signo de la suma es igual al signo de cualquiera de ellos, se acumula la suma.

- Si los signos de los sumandos son iguales y el signo de la suma es diferente al signo de cualquiera de ellos, se transfiere el signo de cualquier sumando al signo del acumulador, y se rellenan el resto de los bits con el inverso de dicho signo.

  Siguiendo estas reglas, la implementación es directa.

3.2.4. Función No Lineal de Transferencia SATLIN.

La función no lineal de transferencia de tipo saturación lineal positiva está definida por la ecuación

$$\text{satlin}(x) = \begin{cases} 1 & \text{si } x > 1 \\ x & \text{si } 0 \le x \le 1 \\ 0 & \text{si } x < 0 \end{cases} \tag{3.23}$$

Esta función es fácil de implementar a nivel programación en VHDL, utilizando comparadores de magnitud incorporados en el lenguaje, por lo que no nos extenderemos en este tema. Solo queda aclarar que la implementación deberá ser llevada a cabo de acuerdo a la resolución requerida por el modelo, eligiendo tamaños de palabra tanto de entrada como de salida de la función de acuerdo a nuestras necesidades.

3.3. Conclusiones.

En este capítulo definimos la estructura general que tendrá la neurona digital en base a diversas consideraciones de diseño. Igualmente, definimos el espacio de valores discretos necesarios para la operación correcta de la red. También seleccionamos y describimos de manera general los componentes de la neurona digital, los cuales serán posteriormente descritos en lenguaje VHDL.

Referencias

1. Koren, Israel, “Computer Arithmetic Algorithms”, Ed. Prentice Hall, 1993.
2. Hwang, Kai, “Computer Arithmetic: Principles, architecture and design”, Ed. John Wiley & Sons, Inc., 1979.
3. Cavanagh, Joseph, “Digital Computer Arithmetic: Design and Implementation”, Ed. McGraw-Hill, Inc., 1984.
4. R. Fantacci, M. Forti, M. Marini and Luca Pancani, “Cellular Neural Network Approach to a Class of Communication Problem”, IEEE Trans. Circuits Syst. – 1: Fundamental Theory and App., vol. 46, no. 12, pp. 1457–1467, 1999.

CAPÍTULO 4

Descripción de Red CNN Extendida en Lenguaje VHDL.

Equation Section (Next)

En el capítulo anterior se definieron las características necesarias para cada uno de los módulos de procesamiento que constituirán la neurona digital. Es ahora que necesitamos describir dichos módulos utilizando el lenguaje VHDL, para posteriormente unirlos y determinar sus necesidades de sincronización. En este capítulo describimos los pasos que se siguieron a fin de obtener módulos funcionales en el lenguaje VHDL, se explica el método de prueba que se utilizó para verificar su correcto funcionamiento y posteriormente mostraremos como fueron enlazados dichos componentes a fin de integrar una neurona digital.

Una vez que se obtenga la neurona, explicaremos sus necesidades de sincronización, y cómo se diseñó una máquina de estado que suministrara las señales necesarias. Finalmente, se describirá la forma en que fue descrita la red neuronal completa, replicando el modelo de neurona digital y agregando el módulo de control aquí descrito, para posteriormente hacer algunas pruebas de funcionamiento y eficiencia de la red.

La síntesis y generación de modelos de simulación se llevó a cabo utilizando el programa ISE de Xilinx, en su versión 7.1.04i, y con un dispositivo objetivo FPGA Spartan 2E de Xilinx, modelo xc2s300e. Las simulaciones se hicieron en el programa ModelSim de ModelTech, en su versión 6.0a.

4.1. Traslación de los Módulos de Procesamiento a VHDL.

La descripción de los módulos de procesamiento, componentes de la neurona digital, puede hacerse directamente a partir de su descripción de funcionamiento y características. Gracias a la versatilidad y sencillez del lenguaje VHDL, podemos elegir entre su descripción algorítmica o bien, cuando sea necesario o más eficiente, hacer algunas descripciones RTL o incluso de estructura lógica. La elección entre estos estilos se hará tomando en cuenta las características estructurales y de comportamiento necesarias para cada módulo.

4.1.1. Sumador multioperando.

Se eligió utilizar un sumador diseñado especialmente debido a que la suma directa en lenguaje VHDL requiere la utilización de sumadores de tamaño de palabra igual al producto, con lo que se tendría un gasto innecesario de recursos. Además, las pruebas posteriores mostraron una velocidad semejante entre las dos aproximaciones de diseño. Nuestro sumador tendrá seis entradas, una para cada celda vecina; esto es para una red FPS de 4x4. En caso de una red de tamaño diferente, se tendrá que implementar un sumador con un número diferente de entradas o bien seguir un proceso de suma en serie, dependiendo del compromiso entre área y velocidad al que se llegue. El modelo de nuestro sumador será parametrizable en cuanto al número de bits de las entradas, ya que necesitamos tener la posibilidad de cambiar la resolución de nuestro modelo para resolver cualquier problema de convergencia.

El sumador, para este caso, consta de cuatro unidades sumadoras con almacenamiento de acarreo (CSA) de tres operandos y un sumador con propagación de acarreo de dos operandos. Dado que los componentes de dichas unidades son sumadores completos (FA), o contadores de 3 bits, y un sumador de 5 bits, comenzamos describiendo estos componentes. Consúltese el Apéndice E, sección 1, para el listado de código. Estos componentes se describen como funciones, dado que es más sencillo construir una estructura regular en base a ciclos utilizando funciones como bloques constitutivos. El sumador de 3 bits, función **ctr3**, se define en base a las ecuaciones lógicas características de un sumador completo. El sumador de 5 bits, función **ctr5**, se construye a partir de 3 sumadores de 3 bits en una estructura similar a la que se empleó en la sección 3.2.1 para construir el CSA, por lo que hace uso de la función anterior.

Ahora definimos la unidad básica CSA, función **csa3**, en base a la función **ctr3**. Esta función retorna los bits de suma y los bits de acarreo a partir de la suma de sus 3 operandos. Utilizando un ciclo que llama a la función ctr3 de acuerdo al ancho de palabra de los operandos, se generan los correspondientes sumadores.

Ya que, después de la estructura de árbol constituida por CSAs es necesario introducir un sumador de propagación de acarreo (CPA), definimos este sumador como la función **cpa2**, la cual acepta dos operandos y produce la suma y un bit de acarreo. Esto se hace con otro ciclo que también hace uso de la función **ctr3** para generar los sumadores correspondientes al CPA. Finalmente, el resto de los bits más significativos se generan con la función **ctr5**, utilizando como operandos los acarreos generados por el árbol de CSAs y el acarreo del CPA.

Se puede observar en el listado de código que se define una función para las unidades **ctr3**, **ctr5**, **csa3** y **cpa2**, y posteriormente se describe la estructura general del sumador multioperando utilizando vectores de señales para los resultados intermedios.

4.1.2. Multiplicador.

Debido a las características especiales requeridas para el multiplicador, se descartó el utilizar la multiplicación directa del lenguaje VHDL, y se optó utilizar un multiplicador Baugh-Wooley. Este multiplicador acepta dos operandos en formato de complemento a dos y de diferentes tamaños de palabra, y proporciona a la salida su producto, también en complemento a dos. De acuerdo a lo observado en la sección 3.2.2, la estructura está compuesta solamente por sumadores completos (FA) y algunos medios sumadores (HA), donde se indicó una entrada de valor cero. Así, definimos primero un par de funciones, **fa** y **ha**, como los sumadores completo y medio, respectivamente. Ya que al final de la estructura se necesita un sumador con propagación de acarreo, se define otra función, **cpa**, para que proporcione esta suma.

Siguiendo las reglas obtenidas por el análisis de las reglas en la sección 3.2.2, de las figuras 3.6 y 3.7, y por otros ejemplos similares, se obtuvo un algoritmo que produce el multiplicador requerido de acuerdo a los parámetros suministrados, esto es, los tamaños de palabra de los operandos de entrada. Este algoritmo es mostrado en el Apéndice E, sección 2, y asume que el tamaño de palabra de un operando es mayor que el otro, al menos 2 bits más grande, de otro modo el algoritmo no es operativo. El algoritmo se implemento utilizando ciclos

dentro de los cuales las llamadas a las funciones definidas fueron generando los sumadores que componen la estructura, y se asignaron los resultados a ciertas variables temporales que interconectan los distintos niveles de la estructura.

4.1.3. Función No Lineal de Transferencia SATLIN.

La función lineal de transferencia SATLIN se implementó utilizando un proceso de comparación simple: si el bit de signo es negativo, la salida es cero; de otro modo, si la entrada es mayor que 1 (teniendo en cuenta la posición del punto fraccional), la salida es 1; en cualquier otro caso, la entrada es transferida a la salida, incluyendo obviamente solo los bits de interés. Consulte el Apéndice E, Sección 3, para el código VHDL de la función SATLIN implementada.

4.1.4. Multiplexor 2:1 de 6 canales.

Conjuntando los canales en un vector de entrada, la implementación de este módulo es simple y directa, utilizando programación en estilo RTL. Si el bit de selección es 0, se transfiere un grupo de canales; de lo contrario, se transfiere el otro grupo. Vea el Apéndice E, Sección 4, para la descripción en lenguaje VHDL de éste multiplexor.

4.1.5. Multiplexor 5:1.

Al igual que el caso anterior, la implementación es directa, utilizando una estructura ***with-select-when***. Véase el Apéndice E, Sección 5, para el listado de código del multipliexor.

4.1.6. Sumador de 2 entradas con saturación.

Ya que los datos con los que opera este sumador son de la misma naturaleza y del mismo tamaño de palabra, podemos hacer uso del operador de suma del lenguaje VHDL, como se muestra en el Apéndice E, Sección 6. Primero se realiza la suma sobre las entradas y posteriormente se decide si se utiliza este resultado o el valor máximo (o mínimo) de saturación. Esta selección se realiza de acuerdo a lo explicado en la Sección 3.2.3. Siguiendo estas directivas, la implementación es directa.

4.2. Análisis de Respuesta de los Módulos.

Durante el diseño de los módulos y posteriormente a su implementación se realizaron continuamente simulaciones para verificar su correcto funcionamiento. En esta sección presentamos los resultados de simulación de cada módulo constitutivo de la neurona digital para que se observen sus respuestas y rendimientos. Las simulaciones se llevaron a cabo utilizando el simulador ModelSim versión 6.0a.

4.2.1. Sumador Multioperando.

Se realizaron pruebas de funcionamiento a este sumador durante su diseño para verificar su correcto funcionamiento, alimentando los seis puertos de datos con valores que permitieran comprobar que el modulo proporcionaba respuestas validas. En la Figura 4.1 se oservan resultados para algunos valores.

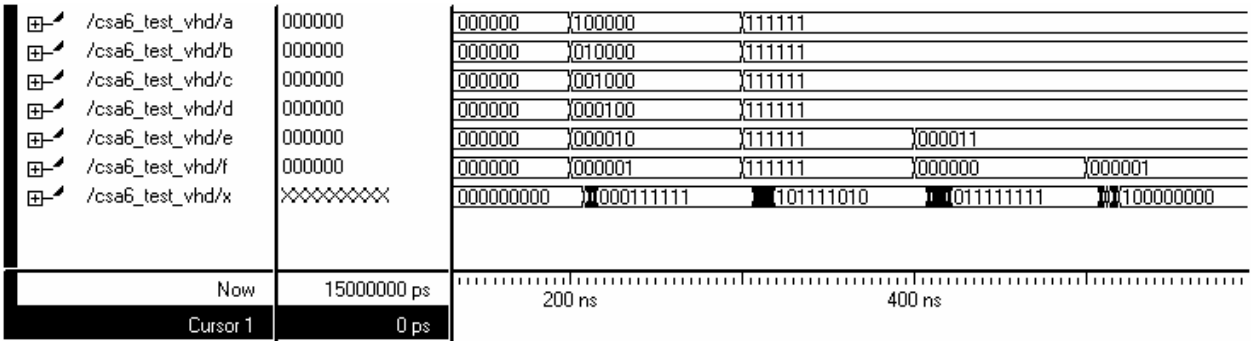


Figura 4.1. *Resultados de simulación para el sumador multioperando.*

A partir de las mismas simulaciones se puede determinar el tiempo de operación del sumador, su tiempo de propagación aproximado. En la figura 4.2 se observa esta medición en el simulador.

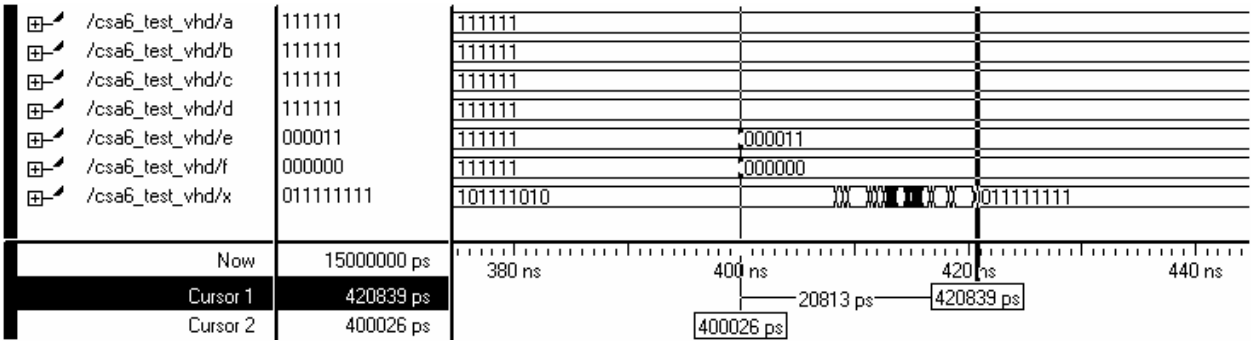


Figura 4.2. *Medición del retardo de propagación del sumador multioperando.*

Se observa que el tiempo aproximado de respuesta es de 21 nanosegundos. Pruebas subsecuentes dan un promedio de 22 nanosegundos.

4.2.2. Multiplicador.

De manera similar, las pruebas para el multiplicador se realizaron con valores representativos para los datos de entrada, tanto positivos como negativos. Se muestra en la Figura 4.3 el resultado de una simulación.

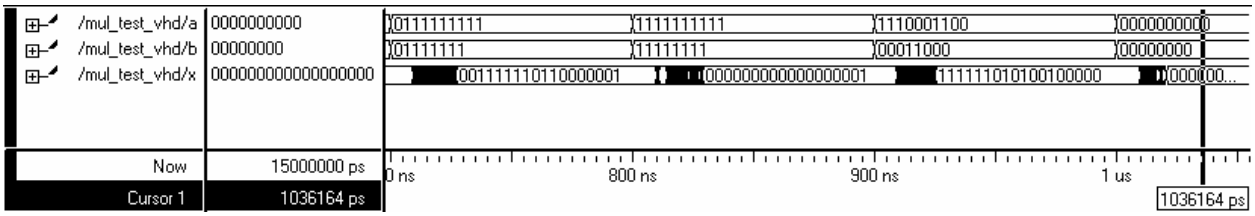


Figura 4.3. *Resultado de simulación del multiplicador Baugh-Wooley de complemento a dos.*

También se realizaron simulaciones para obtener el retardo de propagación del multiplicador. Esto se muestra en la Figura 4.4.

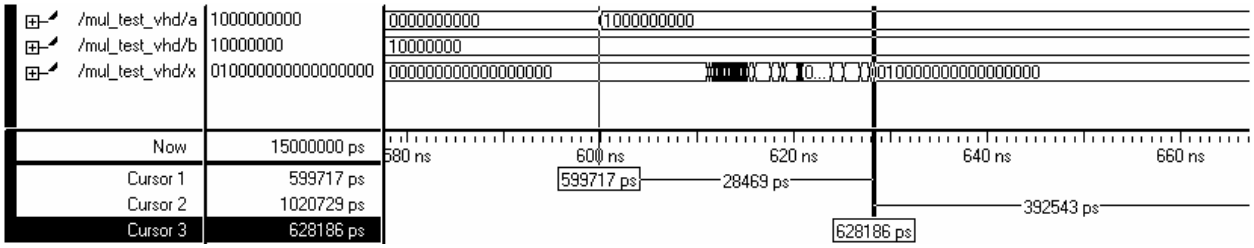


Figura 4.4. *Medición del Tiempo de Propagación para el multiplicador.*

Se observa que el tiempo de propagación aproximado es de 28.5 nanosegundos. Pruebas posteriores mostraron un retardo máximo de alrededor de 30 nanosegundos.

4.2.3. Función No Lineal de Transferencia SATLIN.

La función de transferencia se puede verificar a partir de la gráfica de la Figura 4.5. En ésta se observa que, en la parte superior, los datos con signo de entrada, y en la parte inferior los datos de salida. Se observa que existe la saturación en la parte positiva al llegar al valor correspondiente a 1 (tomando en cuenta que el punto fraccional se encuentra después del bit más significativo), y que en toda la parte negativa se obtiene una saturación al valor de cero. Se observan algunos picos de respuesta transitoria causados por los retrasos de propagación dentro del módulo, pero no afectan la respuesta global.

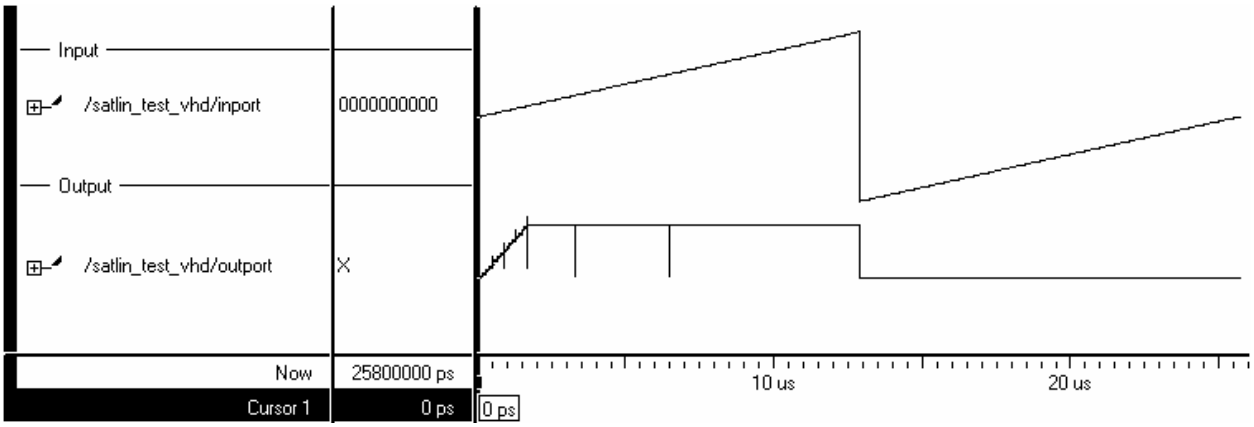


Figura 4.5. *Resultados de simulación de la función de transferencia SATLIN.*

La respuesta en tiempo de este módulo se muestra en la Figura 4.6, donde se observa un retardo de propagación de unos 8 nanosegundos.

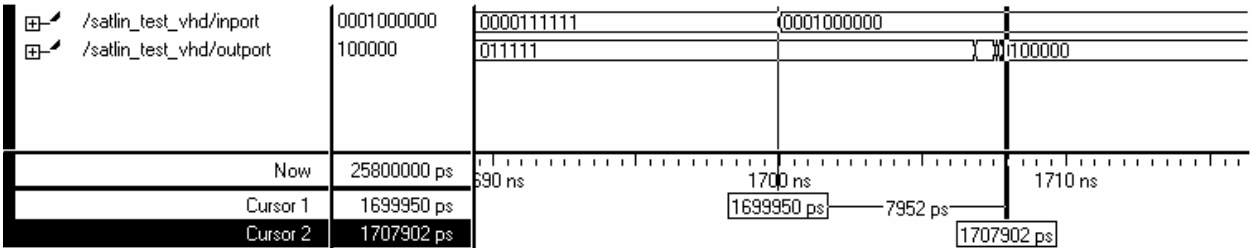


Figura 4.6. *Medición del tiempo de propagación para la función de transferencia SATLIN.*

4.2.4. Multiplexor 2:1 de 6 canales.

En la figura 4.7 se muestran resultados de simulación para este multiplexor. En realidad este es un multiplexor simple 2:1, pero al momento de integrarse en la neurona, los 6 canales consistentes en un vector de bits son agregados en un solo vector más grande, el cual es entonces alimentado a este multiplexor.

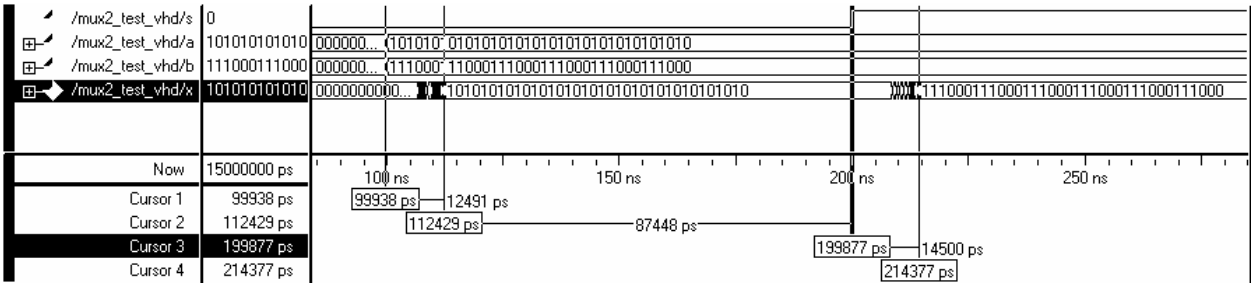


Figura 4.7. *Resultados de simulación para el multiplexor, donde se muestra su funcionamiento y su tiempo de respuesta.*

Se puede observar un retardo de propagación de selección a salida de unos 14.5 nanosegundos, y de entrada a salida de 12.5 nanosegundos.

4.2.5. Multiplexor 5:1.

El resultado de la simulación para el multiplexor 5:1 se muestra en la Figura 4.8.

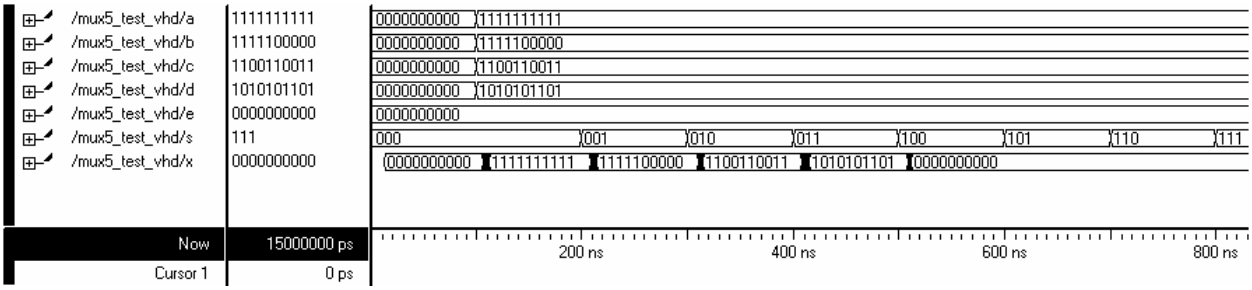


Figura 4.8. *Simulación de funcionamiento del multiplexor 5:1.*

La medición del tiempo de respuesta se muestra en la Figura 4.9.

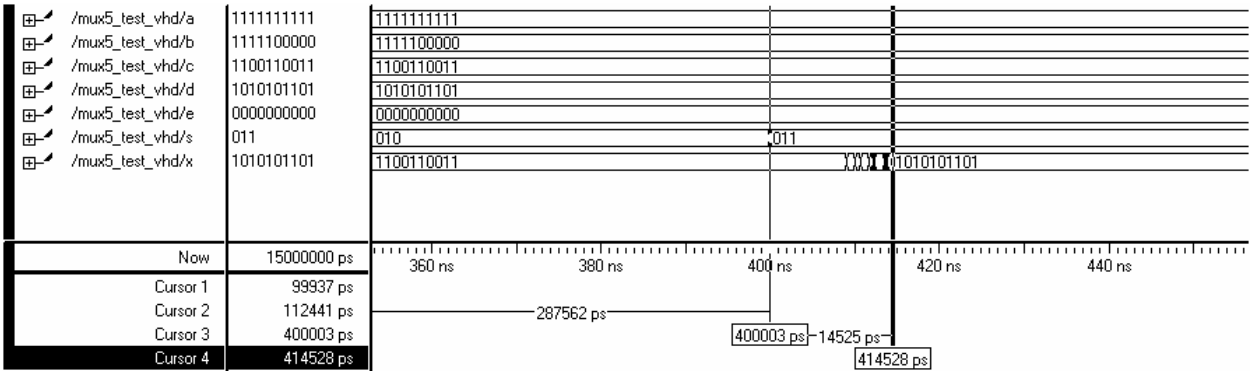


Figura 4.9. *Medición del retardo de propagación de selección a salida para el multiplexor 5:1*

Se observa un retardo de propagación de selección a salida de 14.5 nanosegundos, y también se midió un tiempo de respuesta de entrada a salida de 12.5 nanosegundos.

4.2.6. Sumador de 2 entradas con saturación.

El sumador saturado se simuló con varios datos de muestra que incluyeran valores tanto positivos como negativos, y con valores cercanos a la saturación para observar sus efectos. Los resultados de algunos de estos datos se muestran en la Figura 4.10.

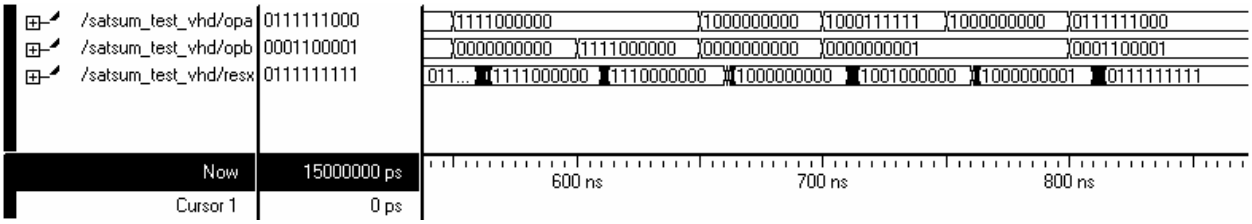


Figura 4.10. *Resultados de simulación del sumador saturado.*

La respuesta en tiempo de este sumador se muestra en la Figura 4.11.

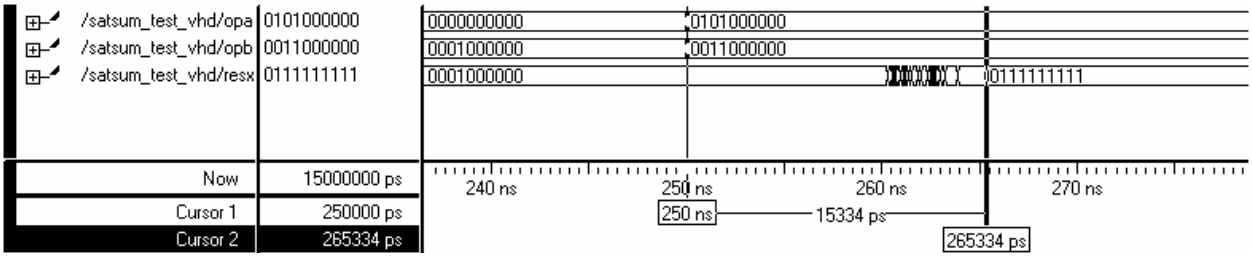


Figura 4.11. *Medición del tiempo de respuesta del sumador saturado.*

Se observó un tiempo de respuesta de aproximadamente 15.5 nanosegundos.

Podemos resumir los tiempos de respuesta de los módulos en la siguiente tabla:

Módulo	Tiempo de Respuesta
Sumador Multioperando	22 ns
Multiplicador	30 ns
Función de Transferencia	8 ns
Multiplexor 2:1 de 6 Canales	15 ns
Multiplexor 5:1	15 ns
Sumador Saturado	16 ns

Tabla 4.1. *Resumen de los tiempos de respuesta de los módulos constitutivos de la neurona digital.*

4.3. Respuesta de la Neurona Digital.

4.3.1. Diseño de la Neurona Digital.

Una vez que han sido diseñados y probados todos los módulos constitutivos de la neurona digital, es necesario integrarlos en una sola estructura. Siguiendo la estructura de la Figura 3.1 y las consideraciones de la sección 3.1.1, se describió la neurona digital. Véase el Apéndice E, Sección 7, para el listado de código en VHDL que describe la neurona digital.

Después de que se instanciaron los componentes descritos en la sección anterior, fueron conectados por medio de señales declaradas en este módulo. Se agregaron los diferentes registros necesarios (almacenamiento del parámetro de prioridad, registro de acumulación, registro de estado) describiéndolos directamente en este módulo y se incluyó la sección de truncamiento describiendola simplemente como una selección de un vector de bits. La sección de control consta solamente de un par de compuertas que permite inicializar la neurona si es seleccionada por medio de las líneas de selección de columna y renglón.

Al obtenerse el modelo general de la neurona digital podemos determinar, de acuerdo a la estructura de la Figura 3.1 y los resultados de la Tabla 4.1, los requerimientos generales de temporizado y control de la neurona, a fin de poder realizar una simulación que nos permita verificar el funcionamiento de la neurona. Posteriormente trasladaremos estos requisitos al diseño del módulo de control, que se determinará en la siguiente sección.

El proceso que debe seguirse para operar la neurona es el siguiente:

1. Se cargan el estado inicial y el registro de prioridad por medio de la línea de control 1. Esto se hace manejandola a nivel bajo y luego regresando a nivel alto. Se despeja el acumulador en forma similar.
2. Para cada entrada y elemento de mascarilla, se selecciona por medio de los multiplexores la entrada apropiada (usando las líneas de control 4 a la 7) y se coloca en la entrada del multiplicador el valor de la mascarilla correspondiente.
3. Después de esperar el tiempo apropiado que toma la propagación de señales por el multiplicador y el sumador saturado, se acumula el resultado, elevando el nivel en la línea de control 2.

4. Una vez que se hizo lo anterior para todos los elementos, se transfiere el valor del acumulador al registro de estado, activando la línea de control 0.
5. Este ciclo de solución se repite hasta que la red obtenga la convergencia.

4.3.2. Respuesta de la Neurona Digital.

Siguiendo los pasos mencionados, se escribió un programa de prueba para probar una neurona aislada, suministrándole datos obtenidos de las simulaciones realizadas en el Capítulo 2 como entradas de las respuestas de las neuronas vecinas y utilizando el mismo conjunto de mascarillas. Los resultados de simulación se muestran en la Figura 4.12.

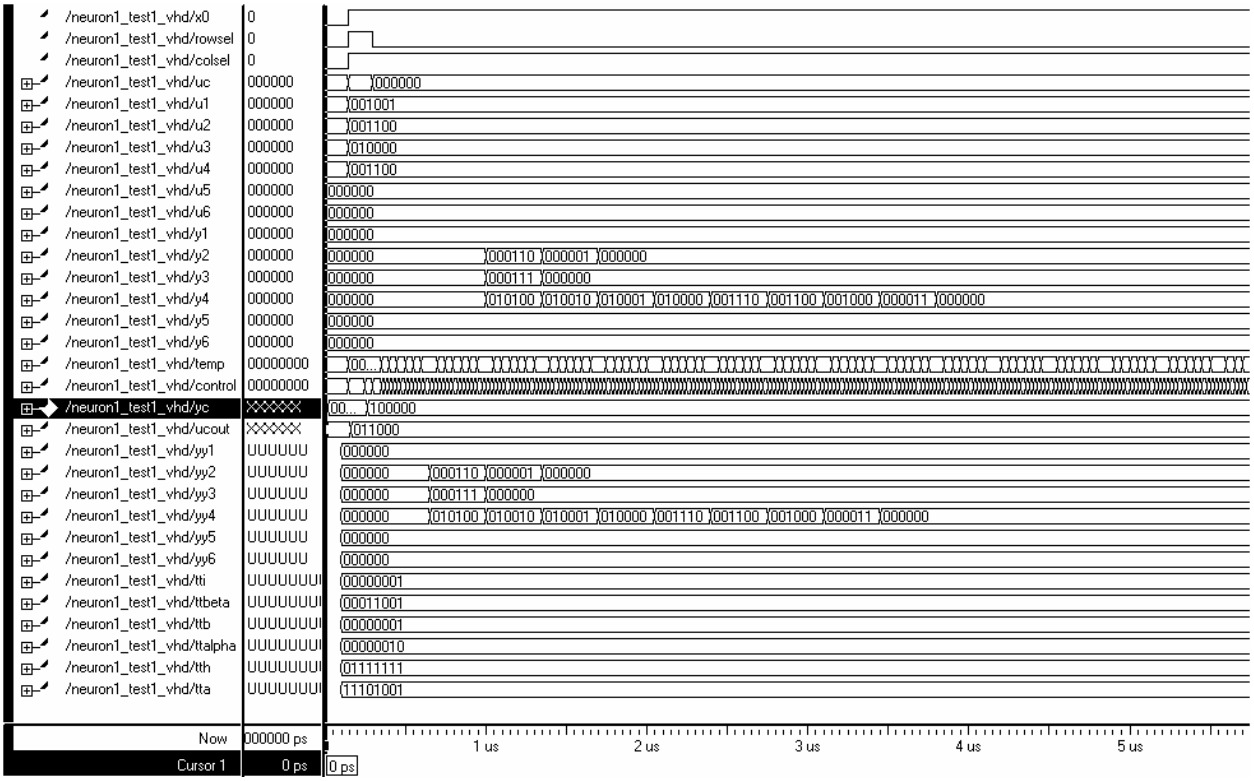


Figura 4.12. *Resultados de simulación para una neurona aislada.*

Se puede observar en el renglón marcado que, como era de esperarse de acuerdo a los datos obtenidos anteriormente, la neurona permanece en el valor de 1. Se realizaron pruebas similares con otras neuronas, dando resultados satisfactorios.

4.4. Diseño del Módulo de Control.

A fin de proprcionar la señalización adecuada para el control de la neurona, se diseñó el módulo de control, el cual proporciona las señales requeridas para toda la red neuonal. En base a los pasos definidos en la sección anterior se describió dicho módulo en lenguaje VHDL. Previamente se diseñaron un par de componentes necesarios para la simplificación de las

funciones del módulo. Dichos componentes son una memoria secuencial, la cual proporciona los valores de las mascarillas, y un contador selector de filas y renglones, a fin de automatizar el proceso de carga de los estados iniciales y los parámetros de prioridad a las neuronas en la red.

4.4.1. Memoria Secuencial.

Esta memoria esta constituida por un contador y un arreglo de memoria. Permite leer y escribir datos en locaciones de memoria consecutivas al aplicar los pulsos de reloj a su entrada de conteo. La descripción en VHDL se da en el Apéndice E, Sección 8.

4.4.2. Contador Selector.

El contador selector permite cargar en forma consecutiva los estados iniciales de las neuronas al aplicar pulsos de reloj en la entrada de conteo. Genera señales de selección de columna y renglón las cuales, al ser detectadas por la sección de control de la neurona, permiten el paso de la señal de carga del estado inicial y del parámetro de prioridad.

Este módulo está constituido por un contador y un par de decodificadores. Su listado de código se muestra en el Apéndice E, Sección 9.

4.4.3. Módulo de Control.

El módulo de control consta de los dos componentes anteriores y de una máquina de estado que genera señalización para toda la red. También incluye un contador que lleva el conteo de las iteraciones de la red y permite detener la máquina de estado si se alcanza una cuenta máxima predeterminada, en caso de que no se obtenga convergencia.

La máquina de estados es tipo Moore, consta de 19 estados y es muy simple en su operación. A partir de su estado inicial, comienza a ejecutar los ciclos de solución cuando se activa la señal externa de inicio. A partir de esto circula por todos sus estados consecutivamente, y se detiene al llegar a la cuenta máxima de iteraciones. Se puede observar el diagrama de estados en la Figura 4.13. Por simplicidad no se incluye ninguna salida en el diagrama. Consúltese el Apéndice E, Sección 10, para el listado de código del módulo de control.

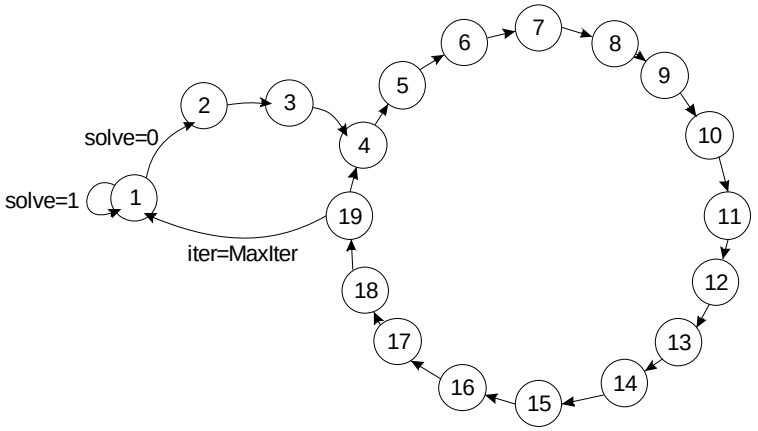


Figura 4.13. *Máquina de estados para el módulo de control.*

4.4.4. Respuesta del Módulo de Control.

En la figura 4.14 se muestra la salida típica del módulo de control, durante una simulación.

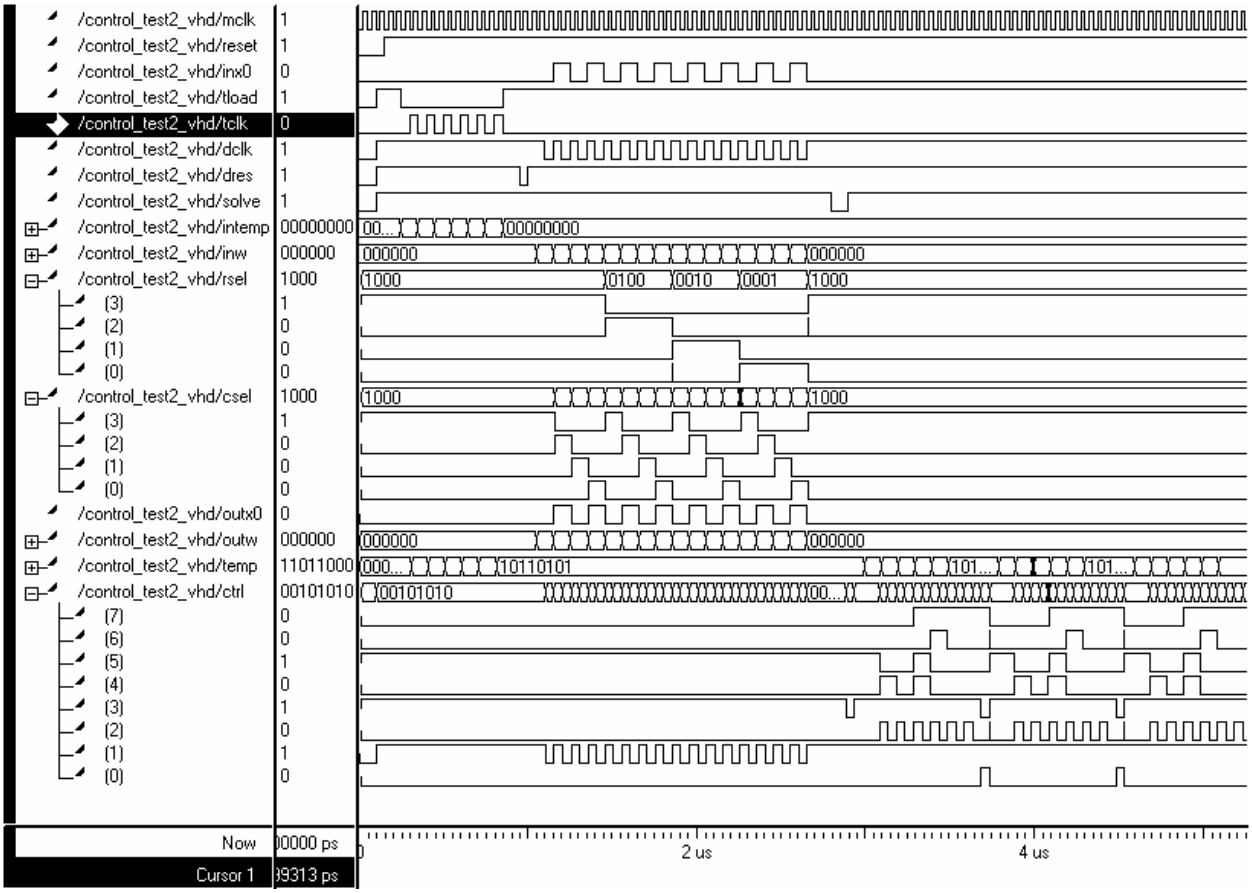


Figura 4.14. *Formas de onda de las salidas del módulo de control.*

4.5. Red Neuronal Digital Extendida.

4.5.1. Diseño de la Red.

El paso final en la construcción de nuestra red neuronal celular extendida digital consiste en instanciar el módulo de la neurona, **neuron1**, tantas veces como neuronas existan en la red, e incluir una instancia del módulo de control. En el Apéndice E, Sección 11, se lista la descripción en lenguaje VHDL de la red neuronal completa. Fueron definidas diversas señales para interconectar las neuronas entre sí y con el módulo de control. Tanto la estructura como el código resultan sumamente regulares, gracias a las características de las CNNs, por lo que la implementación es directa y comprensible.

La salida de la red se forma tomando solamente el bit más significativo de la salida de cada neurona. Es éste bit el que interesa finalmente.

4.5.2. Respuesta de la Red.

Una vez que se ha descrito, sintetizado y obtenido el modelo de nuestra red, solo resta hacer algunas pruebas operativas para verificar su funcionamiento. Se escribió un módulo de prueba que utiliza archivos de texto como fuentes de señales, a fin de hacer actualizaciones más rápidamente a los parámetros de prueba. Estos archivos proveen los pesos de prioridad, estados iniciales y elementos de mascarillas.

Como ejemplo, mostramos en la Figura 4.15 el resultado de una simulación con los datos ejemplo de la sección 2.1, utilizando los estados iniciales de la Ec. (2.1) y la matriz de prioridades de la Ec. (2.4).

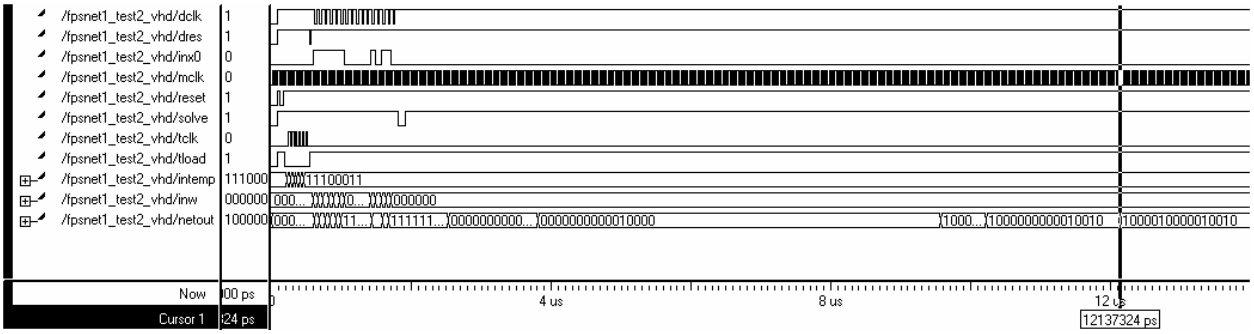


Figura 4.15. *Resultado de Simulación de la Red Neuronal Celular Digital Extendida.*

Se observa que el resultado lo da la variable **netout**.Este es un vector que contiene las filas de la matriz de solución concatenadas en orden creciente, formando una matriz que corresponde a la solución óptima P_2 en la Ec. (2.2). La solución se obtiene en unos 12 microsegundos utilizando los valores de las mascarillas mostradas en la Ec. (2.11). De acuerdo con el reporte de la herramienta de síntesis, se utilizó en aproximadamente 80% el dispositivo elegido para la implementación.

4.6. Conclusiones.

En este capítulo hemos descrito el procedimiento que se siguió para la descripción de la red neuronal celular digital extendida en lenguaje VHDL. Corroboramos que la estructura propuesta en el capítulo anterior es funcional y que la elección de los componentes de la red fue válida, al verificar que el funcionamiento de la red fue el correcto. En el siguiente capítulo se discutirá el análisis global de resultados, incluyendo la evaluación de la red finalizada.

CAPÍTULO 5
Conclusiones.

5.1. Análisis Global de Resultados.

En la introducción a este trabajo de tesis se señaló que nuestro objetivo era diseñar e implementar una red neuronal celular modificada en tecnología digital, y cuyas características debían ser como las indicadas en la Sección 2.1. Este objetivo se ha cumplido y se ha llegado a el a través de los pasos mencionados en los capítulos 3 y 4. Aunque, en efecto, nuestra red muestra ser funcional y entrega resultados aceptables, cabe hacer ciertas observaciones sobre su desempeño.

De acuerdo a lo publicado en la referencia [2] del capítulo 2, la red neuronal análoga diseñada por dichos autores converge en menos de 1 microsegundo, mientras que nuestra red lo hace en unos 12 microsegundos. Si bien resulta más rápida, la implementación analógica de dicha red en VLSI Custom resulta menos económica, mientras que la programación de un dispositivo FPGA es más económica para muchos casos. De nuevo, se tienen que hacer consideraciones sobre el compromiso entre costo y eficiencia que se plantee.

Veremos ahora algunas optimizaciones que tendrían lugar para hacer más eficiente nuestra red.

5.2. Optimización.

A fin de hacer más eficiente nuestra red, tenemos que optimizar muchos de los módulos componentes y realizar algunas optimizaciones a los valores de mascarilla. Sería conveniente paralelizar algunas de las operaciones internas conforme aparezcan nuevos dispositivos en el mercado, con mayores capacidades y prestaciones. En esta misma línea, cabe mencionar que no se hizo uso explícito de algunos recursos que los nuevos dispositivos poseen, como multiplicadores en hardware. Aunque no son recursos específicamente diseñados para nuestra necesidad, podrían ser adaptados para la tarea.

La interface entre nuestra red y el sistema que la utilice deberá ser mejorada, ya que la carga de parámetros y datos se realiza en forma serial. Igualmente, dado el carácter experimental de nuestra red, se debe de proveer de algun mecanismo que permita informar al sistema en cuanto exista la convergencia de la red, en vez de esperar un número fijo de iteraciones.

Ya que se ha observado que la red entrega resultados correctos, sería deseable experimentar la reducción en el número de bits que se emplean en los módulos, y observar si se siguen manteniendo los resultados. De esta forma, obtendremos una reducción de recursos utilizados.

Sería también interesante implementar la red con métodos de discretización de mayor orden, aun a costa de aumentar el área utilizada, con el fin de aumentar la eficiencia de la red.

5.3. Trabajo Futuro.

Se han discutido en la anterior sección algunas optimizaciones que podrían darse en un trabajo a futuro. Es claro que las posibles optimizaciones son numerosas, y que el campo de estudio de las redes neuronales digitales es amplio y con bastantes oportunidades para continuar investigando.

APÉNDICES

Apéndice A. Ejemplos de Optimización de Mascarillas.

Shadowing (SH)

El conjunto de mascarillas propuesto para esta aplicación es:

$$A = \begin{bmatrix} 0 & 2.1 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 0.9 \end{bmatrix} \quad I = 1.$$

El vector **p** de parámetros es:

$$\mathbf{p} = \begin{bmatrix} 2.1 & 2 & 0.9 & 1 \end{bmatrix},$$

donde se listan todos los parámetros de las mascarillas, iniciando con a_c, finalizando con I y los demás parámetros en orden arbitrario. Definimos ahora el nuevo vector $\tilde{\mathbf{p}}$, en el cual a_c es reemplazado por $a_c - 1$ (i.e., $\tilde{\mathbf{p}} = \mathbf{p} - \mathbf{e}_1$, donde $\mathbf{e}_1$ es un vector unitario en la dirección creciente de a_c). Esto equivale a una traslación del espacio de las mascarillas, a fin de convertir el conjunto de desigualdades en un sistema homogéneo. Esto resulta en:

$$\tilde{\mathbf{p}} = \begin{bmatrix} 1.1 & 2 & 0.9 & 1 \end{bmatrix}.$$

Definimos m como el número de parámetros de las mascarillas diferentes de cero. En este caso, $m = 4$. También definimos n como el número de posibles combinaciones que se pueden presentar de salidas y entradas de las neuronas vecinas, esto es, el número de constelaciones posibles. En el caso de que el parámetro I, el umbral, sea diferente de cero, $n = 2^{m-1}$; de otro modo, $n = 2^m$. Esto se debe a que el umbral no se multiplica por ningún valor dependiente de alguna salida o entrada. Para nuestro caso, $n = 8$.

Definiremos ahora una matriz $\mathbf{v} \in \{-1,1\}^{n \times m}$ donde cada renglón es una constelación posible. En base a esta matriz **v**, definimos una matriz $\mathbf{K} \in \{-1,1\}^{n \times m}$ definida por:

$$\mathbf{K} = \begin{bmatrix} \operatorname{sgn}\left(\mathbf{v}_1^T \tilde{\mathbf{p}}\right) \cdot \mathbf{v}_1^T \\ \operatorname{sgn}\left(\mathbf{v}_2^T \tilde{\mathbf{p}}\right) \cdot \mathbf{v}_2^T \\ \vdots \\ \operatorname{sgn}\left(\mathbf{v}_n^T \tilde{\mathbf{p}}\right) \cdot \mathbf{v}_n^T \end{bmatrix},$$

donde $\mathbf{v}_i$ es cada renglón de la matriz **v**, y para una función signo ligeramente modificada, definida por:

$$\operatorname{sgn}(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}.$$

Para nuestro caso, las matrices **v** y **K** son:

$$\mathbf{v} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & -1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix} \quad \mathbf{K} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \end{bmatrix}.$$

La matriz K contiene ahora el sistema homogéneo de desigualdades

$$\left(\mathbf{K}\tilde{\mathbf{p}}\right)_i > 0 \quad \forall 1 \leq i \leq n$$

que caracterizan a la CNN y definen un subespacio $\mathcal{R} \subset \mathbb{R}^m$ dentro del cual la mascarilla funciona correctamente. Sin embargo, algunas de las desigualdades pueden ser redundantes, i.e., pueden ser removidas sin afectar el espacio de solución. Se determina si una desigualdad es redundante si el vector correspondiente es una combinación lineal positiva de algunos otros.

Tras eliminar las desigualdades redundantes, terminamos con un sistema no redundante más pequeño en una matriz $\tilde{\mathbf{K}} \in \mathbb{R}^{\hat{n} \times m}$. Para nuestro caso, el sistema no redundante es

$$\hat{K} = \begin{bmatrix} -1 & 1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & 1 \end{bmatrix},$$

y, por tanto, $\hat{n} = 4$.

$\left(\hat{\mathbf{K}}\tilde{\mathbf{p}}\right)_i$ es una medida de la robustez absoluta del conjunto de mascarillas. Podemos obtener la robustez relativa por medio de

$$D(\tilde{\mathbf{p}}) = \frac{\min_i \left(\hat{\mathbf{K}}\tilde{\mathbf{p}}\right)_i}{\|\mathsf{T}\|_1},$$

donde $\|\mathcal{T}\|_1 = \|\tilde{\mathbf{p}}\|_1 + 1$. La robustez de las mascarillas originales es

$$D(\tilde{\mathbf{p}}) = \frac{\min\{0.8, 1.0, 1.2, 1.0\}}{(1.1 + 2.0 + 0.9 + 1) + 1} = \frac{0.8}{6} = 13.33\%.$$

Si todas las $\left(\hat{\mathbf{K}}\tilde{\mathbf{p}}\right)_i$ son iguales, las mascarillas son óptimamente robustas para una norma L_1 dada. Por tanto, tenemos que resolver $\hat{\mathbf{K}}\mathbf{p} = \gamma \mathbf{1}^m$, donde γ es una constante positiva y

$$\mathbf{1}^m = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \in \mathbb{R}^m.$$

Para nuestro caso, la matriz $\hat{\mathbf{K}}$ es directamente invertible, y por tanto, el conjunto óptimo es:

$$\tilde{\mathbf{p}}_{opt}(\gamma) = \hat{\mathbf{K}}^{-1}\gamma\mathbf{1}^4 = \gamma\begin{bmatrix} 1 & 2 & 1 & 1 \end{bmatrix}^T = \begin{bmatrix} \gamma \\ 2\gamma \\ \gamma \\ \gamma \end{bmatrix}.$$

Así, nuestras mascarillas óptimas para la operación de *shadowing* son

$$A = \begin{bmatrix} 0 & 1+\gamma & 2\gamma \end{bmatrix} \quad B = \begin{bmatrix} \gamma \end{bmatrix} \quad I = \gamma.$$

Dependiendo de la elección del parámetro γ, la robustez relativa del conjunto de mascarillas será

$$D\left(\tilde{\mathbf{p}}_{opt}(\gamma)\right) = \frac{1}{\left\|\tilde{\mathbf{p}}_{opt}(1)\right\|_1 + \frac{1}{\gamma}},$$

y el máximo teórico será cuando $\gamma \to \infty$. Para nuestro caso, el máximo es de 20%.

Detector de Líneas Horizontales (HLD)

Parámetros propuestos:

$$A=\begin{bmatrix}1 & 2 & 1\end{bmatrix} \quad B=\begin{bmatrix}0\end{bmatrix} \quad I=-1$$
$$\mathbf{p}=\begin{bmatrix}2 & 1 & 1 & -1\end{bmatrix} \quad ,$$
$$\tilde{\mathbf{p}}=\begin{bmatrix}1 & 1 & 1 & -1\end{bmatrix}$$

donde

$$m=4 \quad n=2^{4-1}=8\,.$$

Matrices **v** y **K**:

$$\mathbf{v}=\begin{bmatrix}1 & -1 & -1 & 1\\ -1 & 1 & -1 & 1\\ 1 & 1 & -1 & 1\\ -1 & -1 & 1 & 1\\ 1 & -1 & 1 & 1\\ -1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1\\ -1 & -1 & -1 & 1\end{bmatrix} \quad \mathbf{K}=\begin{bmatrix}-1 & 1 & 1 & -1\\ 1 & -1 & 1 & -1\\ 1 & 1 & -1 & 1\\ 1 & 1 & -1 & -1\\ 1 & -1 & 1 & 1\\ -1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1\\ 1 & 1 & 1 & -1\end{bmatrix}.$$

El sistema no redundante $\hat{\mathbf{K}}$ es:

$$\hat{\mathbf{K}}=\begin{bmatrix}-1 & 1 & 1 & -1\\ 1 & -1 & 1 & -1\\ 1 & 1 & -1 & 1\\ 1 & 1 & -1 & -1\\ 1 & -1 & 1 & 1\\ -1 & 1 & 1 & 1\end{bmatrix},$$

donde $\hat{n}=6$. La robustez actual es de:

$$D\left(\tilde{\mathbf{p}}\right)=\frac{\min\left\{2,2,0,2,0,0\right\}}{\left(1+1+1+1\right)+1}=\frac{0}{5}=0\%\,.$$

Resolvemos $\hat{\mathbf{K}}\mathbf{p}=\gamma\mathbf{1}^m$. Como $\hat{\mathbf{K}}$ no es directamente invertible, utilizaremos solución por mínimos cuadrados:

$$\tilde{\mathbf{p}}_{opt}(\gamma) = \left(\hat{\mathbf{K}}^T \hat{\mathbf{K}}\right)^{-1} \hat{\mathbf{K}}^T \gamma \mathbf{1}^{\hat{n}}.$$

La solución es $\tilde{\mathbf{p}}_{opt}(\gamma) = \begin{bmatrix} \gamma+1 & \gamma & \gamma & 0 \end{bmatrix}$ y por tanto el conjunto de mascarillas es:

$$A = \begin{bmatrix} \gamma & \gamma+1 & \gamma \end{bmatrix} \quad B = \begin{bmatrix} 0 \end{bmatrix} \quad I = 0,$$

con una robustez relativa teórica máxima del 33.33%.

Detector de Componentes Conectados (CCD)

Parámetros propuestos:

$$A = \begin{bmatrix} 2 & 3.3 & -3.2 \end{bmatrix} \quad B = \begin{bmatrix} 0 \end{bmatrix} \quad I = 0$$
$$\mathbf{p} = \begin{bmatrix} 3.3 & 2 & -3.2 \end{bmatrix}$$
$$\tilde{\mathbf{p}} = \begin{bmatrix} 2.3 & 2 & -3.2 \end{bmatrix}$$

donde

$$m = 3 \quad n = 2^m = 8,$$

matrices **v** y **K**:

$$\mathbf{v} = \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ 1 & 1 & -1 \\ -1 & -1 & 1 \\ 1 & -1 & 1 \\ -1 & 1 & 1 \\ 1 & 1 & 1 \\ -1 & -1 & -1 \end{bmatrix} \quad \mathbf{K} = \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & -1 \\ -1 & 1 & -1 \\ 1 & -1 & -1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Los renglones 4 a 7 claramente están duplicados y el primero es una combinación lineal positiva. La matriz $\hat{\mathbf{K}}$ no redundante resulta en:

$$\hat{\mathbf{K}} = \begin{bmatrix} -1 & 1 & -1 \\ 1 & -1 & -1 \\ 1 & 1 & 1 \end{bmatrix},$$

donde $\hat{n} = 3$. La robustez relativa actual es de 12.94% y la absoluta es de 36.67%. Como la matriz $\hat{\mathbf{K}}$ es cuadrada y directamente invertible, obtenemos:

$$\tilde{\mathbf{p}}_{opt}(\gamma) = \hat{\mathbf{K}}^{-1}\gamma\mathbf{1}^{\hat{n}} = \begin{bmatrix}\gamma & \gamma & -\gamma\end{bmatrix}^T,$$

así que las mascarillas optimas son:

$$A = \begin{bmatrix}\gamma & \gamma+1 & -\gamma\end{bmatrix} \quad B = \begin{bmatrix}0\end{bmatrix} \quad I = 0,$$

con una robustez relativa teórica máxima del 33.33%.

Rellenado de Huecos (HF)

Los parámetros son:

$$A = \begin{bmatrix} 0 & 1.0 & 0 \\ 1.0 & 1.5 & 1.0 \\ 0 & 1.0 & 0 \end{bmatrix} \quad B = \begin{bmatrix}4.0\end{bmatrix} \quad I = 0.5$$

$$\mathbf{p} = \begin{bmatrix}1.5 & 1.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.5\end{bmatrix}$$

$$\tilde{\mathbf{p}} = \begin{bmatrix}0.5 & 1.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.5\end{bmatrix}$$

donde $m = 7$ y $n = 2^6 = 64$. Por razón de espacio no se muestran las matrices **v** y **K**, ya que sus dimensiones son de 64 x 7. La matriz no redundante $\hat{\mathbf{K}}$ es:

$$\hat{\mathbf{K}} = \begin{bmatrix}
1 & -1 & -1 & -1 & 1 & 1 & -1 \\
-1 & -1 & -1 & -1 & 1 & 1 & -1 \\
1 & -1 & -1 & 1 & -1 & 1 & -1 \\
-1 & -1 & -1 & 1 & -1 & 1 & -1 \\
1 & -1 & 1 & -1 & -1 & 1 & -1 \\
-1 & -1 & 1 & -1 & -1 & 1 & -1 \\
1 & 1 & -1 & -1 & -1 & 1 & -1 \\
-1 & 1 & -1 & -1 & -1 & 1 & -1 \\
-1 & 1 & 1 & 1 & 1 & -1 & 1 \\
1 & 1 & 1 & 1 & 1 & -1 & 1 \\
-1 & -1 & -1 & -1 & -1 & 1 & 1 \\
1 & -1 & -1 & -1 & -1 & 1 & 1
\end{bmatrix},$$

donde $\hat{n} = 12$. La robustez actual es de 0%. Por medio de LSQ obtenemos la solución:

$$\tilde{\mathbf{p}}_{opt}(\gamma) = \begin{bmatrix} 0 & 1.0 & 1.0 & 1.0 & 1.0 & 4.0 & 1.0 \end{bmatrix},$$

así que las mascarillas óptimas son:

$$A = \begin{bmatrix} 0 & \gamma & 0 \\ \gamma & 1 & \gamma \\ 0 & \gamma & 0 \end{bmatrix} \quad B = \begin{bmatrix} 4\gamma \end{bmatrix} \quad I = \gamma,$$

con una robustez máxima teórica del 11.11%.

Extracción de Bordes (EE)

Los parámetros son:

$$A = \begin{bmatrix} 4 \end{bmatrix} \quad B = \begin{bmatrix} 0 & -2 & 0 \\ -2 & 5 & -2 \\ 0 & -2 & 0 \end{bmatrix} \quad I = -1$$

$$\mathbf{p} = \begin{bmatrix} 4 & -2 & -2 & 5 & -2 & -2 & -1 \end{bmatrix}$$

$$\tilde{\mathbf{p}} = \begin{bmatrix} 3 & -2 & -2 & 5 & -2 & -2 & -1 \end{bmatrix}$$

donde $m = 7$ y $n = 2^6 = 64$. Por razones de espacio no se transcriben las matrices **v** y **K**, ya que son de dimensiones 64 x 7. La matriz $\hat{\mathbf{K}}$ no redundante es:

$$\hat{\mathbf{K}} = \begin{bmatrix} 1 & 1 & -1 & -1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 & -1 & -1 & 1 \\ -1 & 1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 & 1 & -1 & 1 \\ 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ -1 & 1 & -1 & 1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 & -1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 & -1 \end{bmatrix},$$

con $\hat{n} = 12$. La robustez relativa actual es de 5.56% y la absoluta de 14.29%. La solución, por mínimos cuadrados (LSQ) es:

$$\tilde{\mathbf{p}}_{opt}(\gamma) = \begin{bmatrix} 3\gamma & -2\gamma & -2\gamma & 5\gamma & -2\gamma & -2\gamma & -\gamma \end{bmatrix},$$

así que las mascarillas óptimas son:

$$A = \begin{bmatrix} 3\gamma + 1 \end{bmatrix} \quad B = \begin{bmatrix} 0 & -2\gamma & 0 \\ -2\gamma & 5\gamma & -2\gamma \\ 0 & -2\gamma & 0 \end{bmatrix} \quad I = -\gamma,$$

con una robustez teórica máxima relativa de 5.88%.

Removedor de Ruido (NR)

Los parámetros son:

$$A = \begin{bmatrix} 0 & 2 & 0 \\ 2 & 2.1 & 2 \\ 0 & 2 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 3.9 \end{bmatrix} \quad I = 0$$

$$\mathbf{p} = \begin{bmatrix} 2.1 & 2 & 2 & 2 & 2 & 3.9 \end{bmatrix}$$

$$\tilde{\mathbf{p}} = \begin{bmatrix} 1.1 & 2 & 2 & 2 & 2 & 3.9 \end{bmatrix}$$

donde $m = 6$ y $n = 2^6 = 64$. Se omiten las matrices **v** y **K** ya que sus dimensiones son 64 x 6. La matriz no redundante $\hat{\mathbf{K}}$ es:

$$\hat{\mathbf{K}} = \begin{bmatrix} -1 & 1 & 1 & 1 & 1 & -1 \\ 1 & -1 & 1 & 1 & 1 & -1 \\ 1 & 1 & -1 & -1 & -1 & 1 \\ 1 & 1 & -1 & 1 & 1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 \\ -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & -1 & 1 & 1 \\ -1 & 1 & -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & -1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \end{bmatrix},$$

donde $\hat{n} = 15$. La robustez relativa actual es de 7.14% y la absoluta de 16.67%. La solución por LSQ es:

$$\tilde{\mathbf{p}}_{opt}(\gamma) = [\gamma \quad \gamma \quad \gamma \quad \gamma \quad \gamma \quad 2\gamma],$$

por lo que las mascarillas óptimas son:

$$A = \begin{bmatrix} 0 & \gamma & 0 \\ \gamma & 1+\gamma & \gamma \\ 0 & \gamma & 0 \end{bmatrix} \quad B = [2\gamma] \quad I = 0,$$

con una robustez relativa máxima teórica de 14.29%.

Detector de Conectividad Global (GCD)

Los parámetros son:

$$A = \begin{bmatrix} 0 & 4.4 & 0 \\ 4.4 & 3.6 & 4.4 \\ 0 & 4.4 & 0 \end{bmatrix} \quad B = [10.7] \quad I = 7$$

$$\mathbf{p} = [3.6 \quad 4.4 \quad 4.4 \quad 4.4 \quad 4.4 \quad 10.7 \quad 7]$$

$$\tilde{\mathbf{p}} = [2.6 \quad 4.4 \quad 4.4 \quad 4.4 \quad 4.4 \quad 10.7 \quad 7]$$

donde $m = 7$ y $n = 2^6 = 64$. Se omiten las matrices **v** y **K** ya que sus dimensiones son 64 x 6. La matriz no redundante $\hat{\mathbf{K}}$ es:

$$\hat{\mathbf{K}} = \begin{bmatrix} -1 & -1 & -1 & 1 & 1 & 1 & -1 \\ -1 & -1 & 1 & -1 & 1 & 1 & -1 \\ -1 & 1 & -1 & -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & 1 & -1 & -1 & 1 \\ -1 & -1 & 1 & 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ -1 & 1 & 1 & -1 & 1 & -1 & 1 \\ -1 & 1 & 1 & -1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & -1 & -1 & 1 & 1 \end{bmatrix},$$

con $\hat{n} = 12$. La robustez relativa actual es de 2.83% y la absoluta es de 15.71%. La solución por mínimos cuadrados es:

$$\tilde{\mathbf{p}}_{opt}(\gamma) = [\gamma \quad 2\gamma \quad 2\gamma \quad 2\gamma \quad 2\gamma \quad 5\gamma \quad 3\gamma],$$

por lo que las mascarillas óptimas son:

$$A = \begin{bmatrix} 0 & 2\gamma & 0 \\ 2\gamma & 1+\gamma & 2\gamma \\ 0 & 2\gamma & 0 \end{bmatrix} \quad B = [5\gamma] \quad I = 3\gamma,$$

con una robustez relativa máxima teórica de 5.88%.

Apéndice B. Ejemplos de Diseño Óptimo de Mascarillas.

Shadowing (SH).

La operación de *shadowing* consiste en las siguientes reglas locales:

- Un píxel negro tendrá salida siempre negra.
- Los pixeles blancos serán siempre blancos excepto si existe un píxel negro a su izquierda. En tal caso, la salida cambiará a negro.

Dado que sólo el píxel de la izquierda afecta al píxel, y que se trata de una operación que requiere propagación, se proponen las siguientes mascarillas:

$$A = \begin{bmatrix} a & a_0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} b_0 \end{bmatrix} \quad I = z\,.$$

Se ha supuesto que la condición inicial es $x_{ij}(0) = u_{ij}$. Las reglas locales se trasladan a las siguientes desigualdades:

Entrada	**Salida**	**Desigualdad**	**Comentario**
-1	-1	$-a_0 - b_0 - a + z \le 0$	El píxel es blanco siempre que el píxel de la izquierda sea blanco.
-1	1	$-a_0 - b_0 + a + z > 0$	El píxel blanco cambia a negro si el píxel de la izquierda es negro.
1	1	$a_0 + b_0 + a + z \ge 0$	Los píxeles negros siempre son negros.
		$a_0 + b_0 - a + z \ge 0$	
Estabilidad		$a_0 > 0$	Condición de estabilidad.

Esto resulta en la siguiente matriz con las reglas de diseño:

$$\mathbf{K} = \begin{bmatrix} 1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Eliminando los renglones redundantes:

$$\hat{\mathbf{K}} = \begin{bmatrix} 1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Como la matriz es directamente invertible, podemos obtener las mascarillas óptimas por:

$$\tilde{\mathbf{p}}_{opt}(\gamma) = \hat{\mathbf{K}}^{-1}\gamma\mathbf{1}^4 = \begin{bmatrix}\gamma & 0 & \gamma & \gamma\end{bmatrix}^T,$$

por lo que, para la operación de *shadowing*, las mascarillas que realizan la tarea de forma óptima son:

$$A = \begin{bmatrix}\gamma & 1+\gamma & 0\end{bmatrix} \quad B = \begin{bmatrix}0\end{bmatrix} \quad I = \gamma,$$

con una robustez máxima relativa teórica del 33.33%. Hay que repetir que la condición inicial es igual a la entrada, por lo cual b_0 es 0.

Detector de Líneas Horizontales (HLD).

Proponemos mascarillas de la siguiente forma:

$$A = \begin{bmatrix}a_0\end{bmatrix} \quad B = \begin{bmatrix}b & b_0 & b\end{bmatrix} \quad I = z,$$

y las siguientes reglas de diseño:

Entrada	**Salida**	**Desigualdad**	**Comentario**
1	-1	$a_0 + b_0 - 2b + z < 0$	El píxel negro se convierte en blanco si sus dos vecinos son blancos.
1	1	$a_0 + b_0 - b + z \geq 0$	El píxel negro permanece negro si sus vecinos son de color opuesto o si son negros.
		$a_0 + b_0 + 2b + z \geq 0$	
-1	-1	$-a_0 - b_0 - 2b + z \leq 0$	El píxel blanco siempre es blanco. Estas condiciones incluyen $-2b$, 0, y $2b$.
		$-a_0 - b_0 + 2b + z \leq 0$	
Estabilidad		$a_0 > 0$	Condición de estabilidad.

Estas reglas de diseño se resumen en la matriz **K**:

$$\mathbf{K} = \begin{bmatrix} -1 & -1 & 2 & -1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 2 & -1 \\ 1 & 1 & -2 & -1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

La matriz de desigualdades no redundantes es:

$$\hat{\mathbf{K}} = \begin{bmatrix} -1 & -1 & 2 & -1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & -2 & -1 \\ 1 & 0 & 0 & 0 \end{bmatrix},$$

y, como es cuadrada, es invertible. La solución es:

$$\tilde{\mathbf{p}}_{opt}(\gamma) = \begin{bmatrix} \gamma & 3\gamma & 2\gamma & -\gamma \end{bmatrix},$$

y las mascarillas óptimas son:

$$A = \begin{bmatrix} 1+\gamma \end{bmatrix} \quad B = \begin{bmatrix} 2\gamma & 3\gamma & 2\gamma \end{bmatrix} \quad I = -\gamma,$$

con una robustez relativa máxima teórica de 14.29%.

Detector de Componentes Conectados (CCD).

Las mascarillas propuestas son:

$$A = \begin{bmatrix} s & a_0 & q \end{bmatrix} \quad B = \begin{bmatrix} b_0 \end{bmatrix} \quad I = z.$$

Las reglas locales expresadas en desigualdades son:

Entrada	Salida	Desigualdad	Comentario
-1	1	$-a_0 - b_0 + s - q + z > 0$	Un píxel blanco se convierte en negro sólo si tiene un negro a la izquierda y un blanco a la derecha.
1	-1	$a_0 + b_0 - s + q + z < 0$	Un píxel negro se convierte en blanco sólo si tiene un blanco a la izquierda y un negro a la derecha.
-1	-1	$-a_0 - b_0 - s + q + z \le 0$	Píxel blanco permanece blanco, excepto para la regla de arriba.
		$-a_0 - b_0 + s + q + z \le 0$	
		$-a_0 - b_0 - s - q + z \le 0$	
1	1	$a_0 + b_0 + s + q + z \ge 0$	Píxel negro permanece negro, excepto para la regla de arriba.
		$a_0 + b_0 + s - q + z \ge 0$	
		$a_0 + b_0 - s - q + z \ge 0$	
Estabilidad		$a_0 > 0$	Condición de estabilidad.

lo que produce la matriz **K**:

$$\mathbf{K} = \begin{bmatrix} -1 & -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & -1 & -1 \\ 1 & 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

La matriz no redundante es:

$$\hat{\mathbf{K}} = \begin{bmatrix} -1 & -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

La solución por LSQ es:

$$\tilde{\mathbf{p}}_{opt}(\gamma) = [\gamma \quad 0 \quad \gamma \quad -\gamma \quad 0],$$

por lo que las mascarillas óptimas para esta operación son:

$$A = [\gamma \quad 1+\gamma \quad -\gamma] \quad B = [0] \quad I = 0,$$

con una robustez relativa teórica máxima del 33.33%.

Removedor de Ruido (NR)

Las mascarillas propuestas para esta operación son:

$$A = \begin{bmatrix} 0 & a & 0 \\ a & a_0 & a \\ 0 & a & 0 \end{bmatrix} \quad B = [b_0] \quad I = z.$$

Las reglas de diseño y desigualdades son:

Entrada	**Salida**	**Desigualdad**	**Comentario**
1	-1	$a_0 - 4a + b_0 + z < 0$	Un píxel negro rodeado por cuatro blancos se vuelve blanco.
1	1	$a_0 - 2a + b_0 + z \geq 0$	Píxel negro permanece negro, excepto en la regla de arriba. Incluye las condiciones para $2a$, 0 y $-2a$.
		$a_0 + 2a + b_0 + z \geq 0$	
-1	-1	$-a_0 - 2a - b_0 + z \leq 0$	Píxel blanco permanece blanco, excepto en la regla de abajo. Incluye las condiciones para $2a$, o y $-2a$.
		$-a_0 + 2a - b_0 + z \leq 0$	
-1	1	$-a_0 + 4a - b_0 + z > 0$	Un píxel blanco rodeado de cuatro negros se vuelve negro.
Estabilidad		$a_0 > 0$	Condición de estabilidad.

La matriz **K** es:

$$K = \begin{bmatrix} -1 & 4 & -1 & -1 \\ 1 & -2 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 2 & 1 & -1 \\ 1 & -2 & 1 & -1 \\ -1 & 4 & -1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

La matriz no redundante de desigualdades es:

$$\hat{\mathbf{K}} = \begin{bmatrix} -1 & 4 & -1 & -1 \\ 1 & -2 & 1 & 1 \\ 1 & -2 & 1 & -1 \\ -1 & 4 & -1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

La solución por LSQ es:

$$\tilde{\mathbf{p}}_{opt}(\gamma) = [\gamma \quad \gamma \quad 2\gamma \quad 0],$$

por lo que el conjunto de mascarillas optimizado para esta aplicación es:

$$A = \begin{bmatrix} 0 & \gamma & 0 \\ \gamma & 1+\gamma & \gamma \\ 0 & \gamma & 0 \end{bmatrix} \quad B = [2\gamma] \quad I = 0.$$

Como se esperaba, $I = 0$, puesto que funciona para ruido blanco y negro. También es posible obtener la mascarilla no acoplada, esto es, la matriz A es cero excepto en su elemento central.

Detector de Conectividad Global (GCD).

Las mascarillas propuestas son:

$$A = \begin{bmatrix} 0 & a & 0 \\ a & a_0 & a \\ 0 & a & 0 \end{bmatrix} \quad B = [b_0] \quad I = z.$$

Las reglas de diseño:

Entrada	**Salida**	**Desigualdad**	**Comentario**
-1	1	$-a_0 - b_0 + 4a + z > 0$	Un píxel blanco rodeado por tres negros se vuelve negro.
		$-a_0 - b_0 + 2a + z > 0$	
-1	-1	$-a_0 - b_0 + z \le 0$	Los píxeles blancos permanecen blancos, excepto para la regla anterior.
		$-a_0 - b_0 - 2a + z \le 0$	
1	1	$a_0 + b_0 + 4a + z \ge 0$	Los píxeles negros permanecen

		$a_0 + b_0 - 4a + z \geq 0$	negros.
Estabilidad		$a_0 > 0$	Condición de estabilidad.

La matriz **K** es:

$$\mathbf{K} = \begin{bmatrix} -1 & -1 & 4 & 1 \\ -1 & -1 & 2 & 1 \\ 1 & 1 & 0 & -1 \\ 1 & 1 & 2 & -1 \\ 1 & 1 & 4 & 1 \\ 1 & 1 & -4 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

La matriz no redundante es:

$$\hat{\mathbf{K}} = \begin{bmatrix} -1 & -1 & 2 & 1 \\ 1 & 1 & 0 & -1 \\ 1 & 1 & -4 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix},$$

y la solución es:

$$\tilde{\mathbf{p}}_{opt}(\gamma) = \begin{bmatrix} \gamma & 2\gamma & 1 & 2\gamma \end{bmatrix},$$

por lo que las mascarillas optimas son:

$$A = \begin{bmatrix} 0 & \gamma & 0 \\ \gamma & 1+\gamma & \gamma \\ 0 & \gamma & 0 \end{bmatrix} \quad B = \begin{bmatrix} 2\gamma \end{bmatrix} \quad I = 2\gamma\,,$$

con una robustez relativa teórica máxima de 16.67%

Apéndice C. Programas de Optimización de Mascarillas en MATLAB.

Optimizador.

robustify.m

```
%ROBUSTIFY Optimizes a template

%Written by J.L.Ochoa. CINVESTAV IPN

% Modify This ****************************
% Template entry. ac first.
p = [3.6 4.4 4.4 4.4 4.4 10.7 7]';
% Set bias = 1 if bias exists and is
% included in the template. Otherwise, set
% to 0.
bias = 1;
% Set acinc = 1 if the first template
% element is a5, or center element from
% matrix A. Otherwise (ac is not included),
% set to 0.
acinc = 1;
% End of modifiable parameters ************

% Begin ----------------------------------+
% Display Entries
disp('Template:')
disp(p)
if bias
    disp('Bias Included')
end

% Make Homogeneous
pcar = p;
if acinc
    pcar(1) = pcar(1)-1;
end

% Dimensions
m = numel(p);
n = 2^(m-bias);

outs = sprintf('Analyzing %d combinations...',n);
disp(outs)

% Create all constelations
v = vmatrix(n,m,bias);

% Create K matrix w/modified sign function
K = ones(n,m);
for i = 1:n
    K(i,:) = sgn(v(i,:)*pcar)*v(i,:);
end

% Redundancy Elimination
Kcar = redrem(K);
ncar = size(Kcar,1);

% Determine current robustness
sset = ones(ncar,1);
for i = 1:ncar
    sset(i) = Kcar(i,:)*pcar;
end
ssetmin = min(sset);
TRobD = ssetmin/(norm(pcar,1)+1) * 100;
TRobe = ssetmin/m * 100;

% Display Robustness
```

```
disp('Template Robustness')
outs = sprintf('D: %0.2f%%  e: %0.2f%%',TRobD, TRobe);
disp(outs)

% Obtain optimized template
poptcar = sslv(Kcar,m);

% Display Optimized Template
if (poptcar == 0)
    disp('No solution!')
else
    disp('Optimized Template:')
    disp(poptcar)

    % Show optimized template maximum theoretical robustness
    TRobMax = 1/norm(poptcar,1) * 100;
    outs = sprintf('Max Theoretical Relative Robustness Dmax: %0.2f%%',TRobMax);
    disp(outs)
end
```

Diseño Robusto.

robdes.m

```
%ROBDES Obtains an optimized template from design rules

% Written by: J.L.Ochoa - CINVESTAV IPN

disp('Design inequalities:')

% BEGIN MODIFIABLE PARAMETERS ------+

% Inequality System Entry
K = [-1 -1 1 -1 1;
    -1 -1 1 -1 -1;
    1 1 1 -1 -1;
    1 1 -1 -1 -1;
    1 1 1 1 -1;
    1 1 1 1 1;
    1 1 1 -1 1;
    1 1 -1 -1 1;
    1 0 0 0 0]

% END MODIFIABLE PARAMETERS --------+

% Remove redundant inequalites.
Kcar = redrem(K);
m = size(Kcar,2);

% Obtain optimized template
poptcar = sslv(Kcar, m);

% Display Optimized Template
if (poptcar == 0)
    disp('No solution!')
else
    disp('Optimized Template:')
    disp(poptcar)

    % Show optimized template maximum theoretical robustness
    TRobMax = 1/norm(poptcar,1) * 100;
    outs = sprintf('Max Theoretical Relative Robustness Dmax: %0.2f%%',TRobMax);
    disp(outs)
end
```

Subprogramas.

vmatrix.m

```
function y = vmatrix(n, m, bias)
%VMATRIX Creates the v matrix
%  v-matrix contains all the possible constelations
%  for a given template definiton.

% Written by: J.L.Ochoa - CINVESTAV IPN

tsw = 0;
tmp = ones(n, m);
for j = 1:(m-bias)
    tctr = 0;
    for i = 1:n
        tctr = tctr+1;
        if tctr == 2^(j-1)
            tsw = not(tsw);
            tctr = 0;
        end
        tmp(i,j)=tmp(i,j)*(-1)^(2^tsw);
    end
end

        y = tmp;
```

sgn.m

```
function y=sgn(x)
%SGN Modified Sign Function
%
%             1   if x >= 0
% SGN(x) = {
%            -1   if x < 0
%
% To be used in CNN optimizer

% Written by: J.L. Ochoa - CINVESTAV IPN

if x>=0
    y = 1;
else
    y=-1;
end
```

redrem.m

```
function Y = redrem(X)
%REDREM Removes redundant inequalities
%  REDREM Removes the redundant inequalities from the
%  constellations matrix.
%  A redundant inequality is a positive linear combination
%  of any others.

% Written by: J.L. Ochoa - CINVESTAV IPN

% Size of Matrix
tmpn = size(X,1);
tmpm = size(X,2);
tmpncar = tmpn;
tmpi = 1;
```

```
15
16  tmpY = X;
17
18  % Check each combination
19  while tmpi <= tmpncar
20      tmpB = tmpY(tmpi,:);
21      tmpA = tmpY;
22      tmpA(tmpi,:) = [];
23      % NonNegative Least Squares
24      tmpS = lsqnonneg(tmpA',tmpB');
25      % Residual Difference between norms
26      tmpR = abs(norm(tmpA'*tmpS) - norm(tmpB'));
27      if tmpR < 0.01
28          % Is a positive linear combination
29          tmpY = tmpA;
30          tmpncar = tmpncar - 1;
31      else
32          % Independent
33          tmpi = tmpi + 1;
34      end
35  end
36
37  Y = tmpY;
```

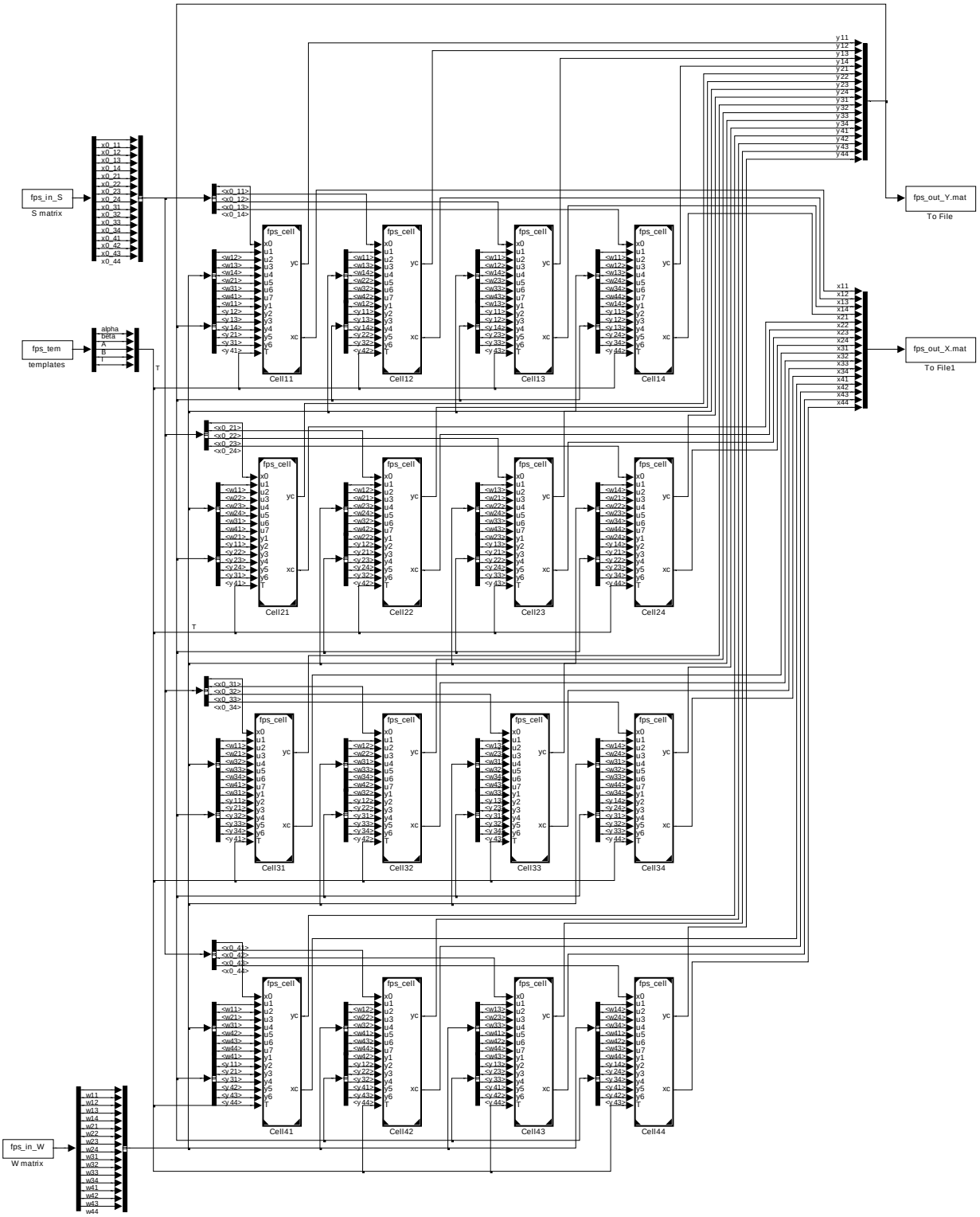
sslv.m

```
 1  function y = sslv(X, m)
 2  %SSLV Solve system depending on matrix rank.
 3  %  X is the non-redundant K matrix.
 4  %  m - number of template parameters.
 5
 6  % Written by: J.L. Ochoa CINVESTAV IPN
 7
 8  % Get Matrix Rank & Number of Inequalities
 9  tmpRank = rank(X);
10  tmpmcar = size(X,1);
11  outs = sprintf('Solving for Rank %d, %d ineq´s & %d params...', tmpRank, tmpmcar, m);
12  disp(outs)
13
14  oo = ones(size(X,1),1);
15
16  % Solve on case
17
18  if (tmpRank == m)&(m == tmpmcar) %  Rank(K) = m = m^
19      disp('Exact Solution')
20      y = inv(X)*oo;
21  elseif (tmpRank < m)&(m == tmpmcar) % Rank(K) < m = m^
22      tmpd = m - tmpRank;
23      outs = sprintf('Solution with dimension %d', tmpd);
24      disp(outs)
25      disp('Solving with QR')
26      [tmpQ,tmpR] = qr(X); % Decompose
27      tmpt = Q'*oo;
28      y = R\tmpt;
29  elseif (tmpRank == m)&(m < tmpmcar) % Rank(K) = m < m^
30      disp('LSQ Solution.')
31      disp('If solution, most robust one is presented.')
32      y = inv(X'*X)*X'*oo;
33  elseif (tmpRank < m)&(m < tmpmcar) % Rank(K) < m < m^
34      disp('Please reduce the number of parameters!')
35      y = 0;
36  elseif (tmpRank <= tmpmcar)&(tmpmcar < m) % Rank(K) <= m^ < m
37      tmpd = m - tmpRank;
38      outs = sprintf('Solution with dimension %d\.Not implemented.', tmpd);
39      disp(outs)
40      y = 0;
41  else
42      disp('Error. (?)')
```

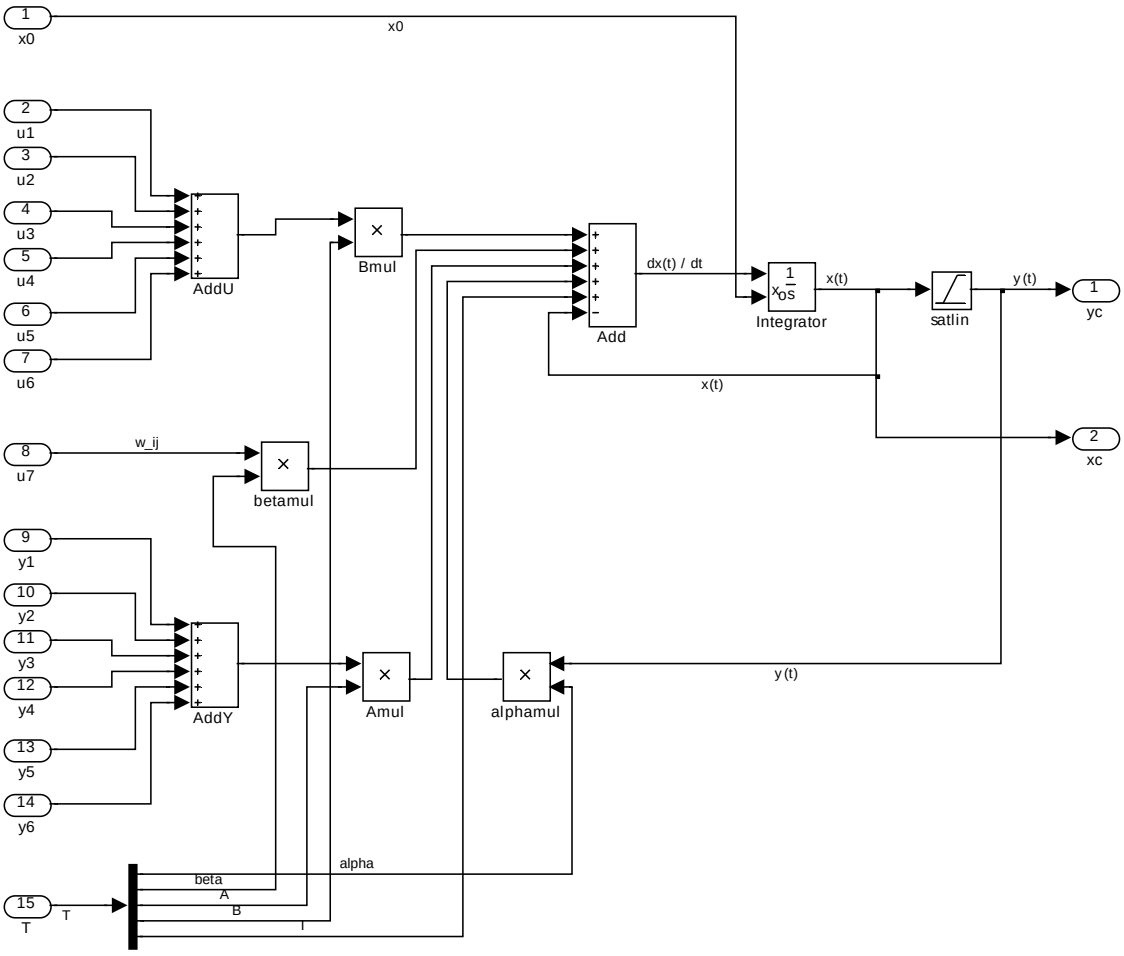
```
 43      y = 0;
 44  end
```

Apéndice D. Diagramas de simulación en Simulink®.

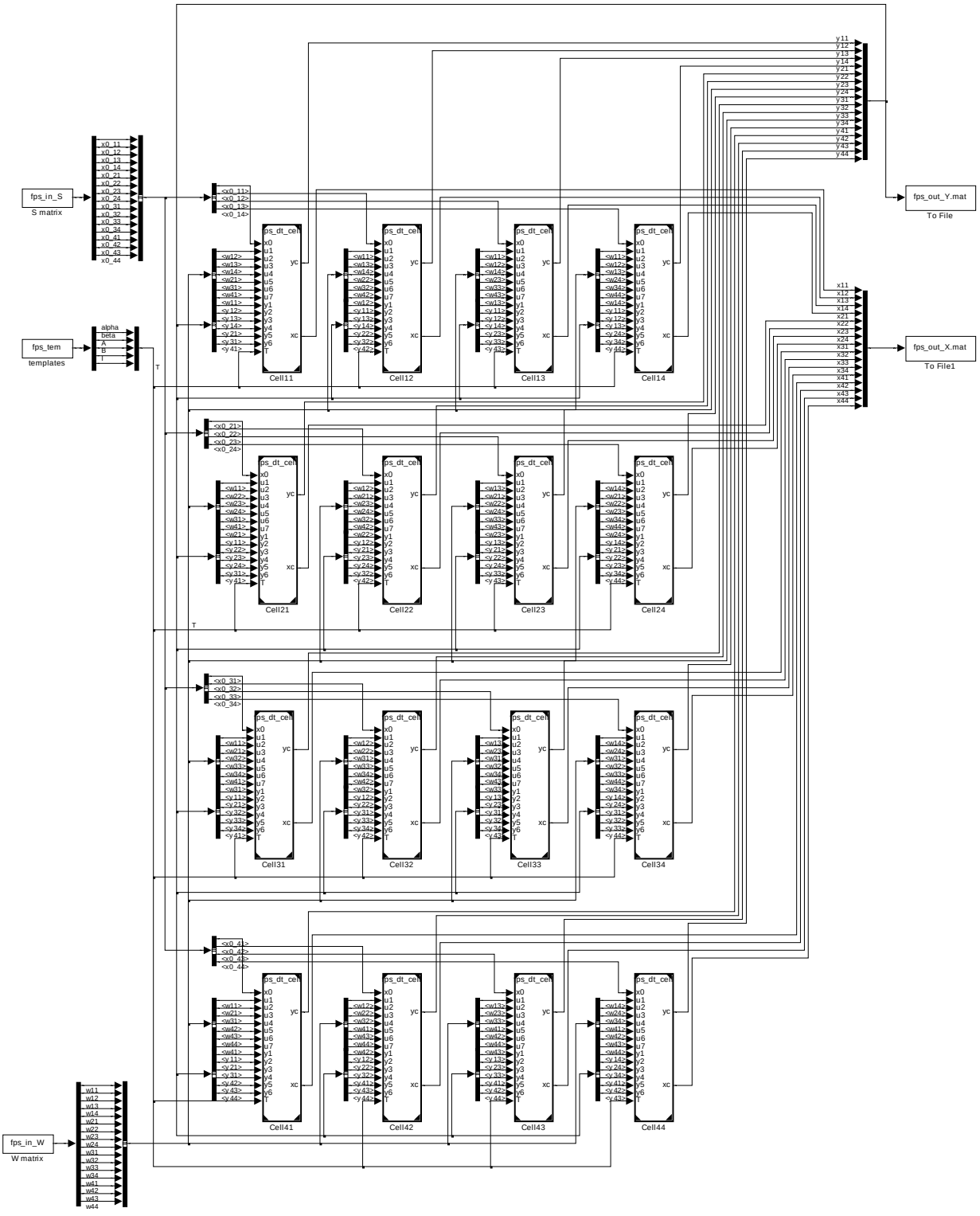
1. Modelo de Simulación de CNN en Tiempo Continuo.



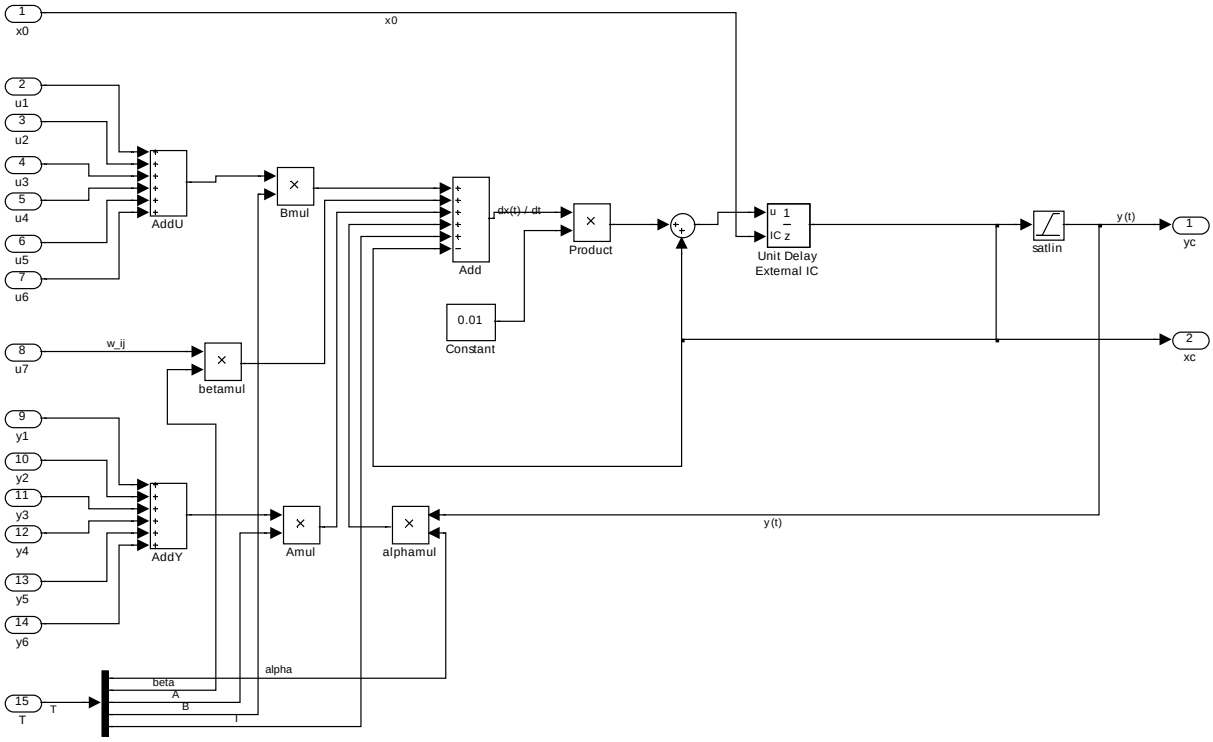
2. Estructura Neuronal del Modelo de CNN de Tiempo Continuo.



3. Modelo de Simulación de CNN en Tiempo Discreto.



4. Estructura Neuronal del Modelo de CNN de Tiempo Discreto.



Apéndice E. Listados de Código en Lenguaje VHDL.

1. Sumador de 6 operandos (csa6.vhd)

```
 1  library IEEE;
 2  use IEEE.std_logic_1164.ALL;
 3  use IEEE.std_logic_ARITH.ALL;
 4  use IEEE.std_logic_UNSIGNED.ALL;
 5
 6  entity csa6 is
 7         Generic ( iosize: integer := 6);
 8         Port ( A : in std_logic_vector(iosize-1 downto 0);
 9             B : in std_logic_vector(iosize-1 downto 0);
10             C : in std_logic_vector(iosize-1 downto 0);
11             D : in std_logic_vector(iosize-1 downto 0);
12             E : in std_logic_vector(iosize-1 downto 0);
13             F : in std_logic_vector(iosize-1 downto 0);
14                 -- Output is iosize+log2(6) =~ iosize+3
15             X : out std_logic_vector(iosize+2 downto 0));
16 end csa6;
17
18 architecture arch_csa6 of csa6 is
19
20 -- 3-bit "counter" function.
21 function ctr3(inA, inB, inC: in std_logic) return std_logic_vector is
22 variable c,s: std_logic;
23 begin
24         s := inA xor inB xor inC;
25         c := (inA and inB) or (inC and (inA or inB));
26         return c&s;
27 end;
28
29 -- 5-bit "counter" function.
30 -- Implemented with 3 ctr3 modules.
31 function ctr5(inA, inB, inC, inD, inE: in std_logic) return std_logic_vector is
32 variable x, y, z: std_logic;
33 variable L1, L2, L3: std_logic_vector(1 downto 0);
34 begin
35
36         L1 := ctr3(inC,inD,inE);
37         L2 := ctr3(inA,inB,L1(0));
38         L3 := ctr3('0',L1(1),L2(1));
39         x := L3(1);
40         y := L3(0);
41         z := L2(0);
42
43         return x&y&z;
44 end;
45
46 -- 3-input iosize-bit carry save adder.
47 function csa3(inA, inB, inC: in std_logic_vector(iosize-1 downto 0)) return
48 std_logic_vector is
49 variable rt: std_logic_vector(iosize*2-1 downto 0);
50 variable tmp: std_logic_vector(1 downto 0);
51 begin
52         for idx in 0 to iosize-1 loop
53                 tmp := ctr3(inA(idx), inB(idx), inC(idx));
54                 rt(idx+iosize) := tmp(1);
55                 rt(idx) := tmp(0);
56         end loop;
57         return rt;
58 end;
59
60 -- (iosize-1)-stage(bit) 2-input carry propagate adder.
61 function cpa2(inA, inB: in std_logic_vector(iosize-2 downto 0)) return
62 std_logic_vector is
63 variable c, s: std_logic_vector(iosize-2 downto 0);
64 variable tmp: std_logic_vector(1 downto 0);
65 begin
```

```
66          tmp := ctr3(inA(0), inB(0), '0');
67          c(0) := tmp(1);
68          s(0) := tmp(0);
69          for idx in 1 to iosize-2 loop
70                  tmp := ctr3(inA(idx), inB(idx), c(idx-1));
71                  c(idx) := tmp(1);
72                  s(idx) := tmp(0);
73          end loop;
74          return c(iosize-2)&s;
75  end;
76
77  -- Signals...!
78
79  signal L1, L2, L3, L4: std_logic_vector(iosize*2-1 downto 0);
80  signal L5: std_logic_vector(iosize-1 downto 0);
81
82  -- CSA6 Architecture Begins...!
83  begin
84
85          L1 <= csa3(A, B, C);
86          L2 <= csa3(D, E, F);
87          L3 <= csa3(L1(iosize-1 downto 0), L2(iosize*2-2 downto iosize)&'0',
88  L2(iosize-1 downto 0));
89          L4 <= csa3(L1(iosize*2-2 downto iosize)&'0', L3(iosize*2-2 downto
90  iosize)&'0', L3(iosize-1 downto 0));
91          X(0) <= L4(0);
92          L5 <= cpa2(L4(iosize*2-2 downto iosize), L4(iosize-1 downto 1));
93          X(iosize-1 downto 1) <= L5(iosize-2 downto 0);
94          X(iosize+2 downto iosize) <= ctr5(L1(iosize*2-1), L2(iosize*2-1),
95  L3(iosize*2-1), L4(iosize*2-1), L5(iosize-1));
96
97  end arch_csa6;
```

2. Multiplicador (`mul.vhd`)

```
 1  library IEEE;
 2  use IEEE.std_logic_1164.ALL;
 3  use IEEE.std_logic_ARITH.ALL;
 4  use IEEE.std_logic_UNSIGNED.ALL;
 5
 6  entity mul is
 7          Generic ( ssize: integer := 10;
 8                    tsize: integer := 8);
 9          Port ( A : in std_logic_vector(ssize-1 downto 0);
10                 B : in std_logic_vector(tsize-1 downto 0);
11                 X : out std_logic_vector(ssize+tsize-1 downto 0));
12  end mul;
13
14  architecture arch_mul of mul is
15
16  -- full adder function.
17  function fa(inA, inB, inC: in std_logic) return std_logic_vector is
18  variable c, s: std_logic;
19  begin
20          s := inA xor inB xor inC;
21          c := (inA and inB) or (inC and (inA or inB));
22          return c&s;
23  end;
24
25  -- half adder function.
26  function ha(inA, inB: in std_logic) return std_logic_vector is
27  variable c, s: std_logic;
28  begin
29          s := inA xor inB;
30          c := inA and inB;
31          return c&s;
32  end;
33
34  -- carry propagate adder. size is adequate for our multiplier.
```

```
 35  function cpa(inA, inB: in std_logic_vector(ssize-1 downto 0); inCarry: in std_logic)
 36  return std_logic_vector is
 37  variable c, s: std_logic_vector(ssize-1 downto 0);
 38  variable tmp: std_logic_vector(1 downto 0);
 39  begin
 40          -- First adder.
 41          tmp := fa(inA(0), inB(0), inCarry);
 42          c(0) := tmp(1);
 43          s(0) := tmp(0);
 44          -- Remaining n-1 adders.
 45          for idx in 1 to ssize-1 loop
 46                  tmp := fa(inA(idx), inB(idx), c(idx-1));
 47                  c(idx) := tmp(1);
 48                  s(idx) := tmp(0);
 49          end loop;
 50          -- Leftmost bit is carry out.
 51          return c(ssize-1)&s;
 52  end;
 53
 54
 55  -- Multiplier Architecture Begins!
 56
 57  -- Levels outputs structure.
 58  type struct is array(tsize-2 downto 0) of std_logic_vector(2*ssize-3 downto 0);
 59
 60
 61  -- Row levels outputs.
 62  signal L: struct;
 63
 64  -- Last level outputs.
 65  signal LL: std_logic_vector(tsize*2-1 downto 0);
 66
 67  -- CPA adder outputs.
 68  signal LC: std_logic_vector(ssize downto 0);
 69
 70  -- CPA adder inputs.
 71  signal AC, BC: std_logic_vector (ssize-1 downto 0);
 72
 73  -- ...and an extra signal!
 74  signal extra: std_logic;
 75
 76  begin
 77
 78  process (A, B)
 79          variable tmp: std_logic_vector(1 downto 0);
 80          begin
 81
 82          -- LSB of product.
 83          X(0) <= A(0) and B(0);
 84
 85          -- First row.
 86          for idx in 0 to ssize-2 loop
 87
 88                  -- Select adder type depending on position.
 89                  if (idx = ssize -2) then
 90                          -- Add MSB of A.
 91                          tmp := ha(A(ssize-1), A(ssize-2) and B(1));
 92                  elsif (idx = tsize-2) then
 93                          -- Add MSB of B.
 94                          tmp := fa(A(idx+1) and B(0), A(idx) and B(1), B(tsize-1));
 95                  else
 96                          -- Add sequenced terms.
 97                          tmp := ha(A(idx+1) and B(0), A(idx) and B(1));
 98                  end if;
 99
100                  -- Assign terms from temporary variable.
101                  L(0)(idx*2) <= tmp(0);
102                  L(0)(idx*2+1) <= tmp(1);
103
104          end loop;
105
```

```
106         -- Main body.
107         for idy in 1 to tsize-2 loop
108
109                 -- Main chain.
110                 for idx in 0 to ssize-3 loop
111
112                         -- Check second-last-row exception.
113                         if (idy = tsize-2) then
114                                 tmp := fa(not A(idx) and B(idy+1), L(idy-
115 1)((idx*2)+1), L(idy-1)((idx*2)+2));
116                         else
117                                 tmp := fa(A(idx) and B(idy+1), L(idy-1)((idx*2)+1),
118 L(idy-1)((idx*2)+2));
119                         end if;
120                         L(idy)(idx*2) <= tmp(0);
121                         L(idy)(idx*2+1) <= tmp(1);
122
123                 end loop;
124
125                 -- Row-end half-adder.
126                 -- Check second-last-row exception.
127                 if (idy = tsize-2) then
128                         tmp := ha(not A(ssize-2) and B(idy+1), L(idy-1)((ssize-
129 2)*2+1));
130                 else
131                         tmp := ha(A(ssize-2) and B(idy+1), L(idy-1)((ssize-2)*2+1));
132                 end if;
133                 L(idy)((ssize-2)*2) <= tmp(0);
134                 L(idy)((ssize-2)*2+1) <= tmp(1);
135
136         end loop;
137
138         -- Last row.
139         for idx in 0 to tsize-2 loop
140
141                 tmp := fa(A(ssize-1) and not B(idx), L(tsize-2)((idx+ssize-tsize)*2),
142 L(tsize-2)((idx+ssize-tsize)*2-1));
143                 LL(idx*2) <= tmp(0);
144                 LL(idx*2+1) <= tmp(1);
145
146         end loop;
147
148         -- Last-row's row-end half-adder.
149         tmp := ha(A(ssize-1) and B(tsize-1), L(tsize-2)(2*ssize-3));
150         LL(2*tsize-2) <= tmp(0);
151         LL(2*tsize-1) <= tmp(1);
152
153
154         -- Carry-propagate adder.
155
156         -- Construct input terms first...
157         for idx in 0 to ssize-tsize-2 loop
158                 AC(idx) <= L(tsize-2)(2*idx+2);
159                 BC(idx) <= L(tsize-2)(2*idx+1);
160         end loop;
161         AC(ssize-tsize-1) <= LL(0);
162         BC(ssize-tsize-1) <= '0';
163         for idx in ssize-tsize to ssize-2 loop
164                 AC(idx) <= LL((idx-ssize+tsize+1)*2);
165                 BC(idx) <= LL((idx-ssize+tsize+1)*2-1);
166         end loop;
167         AC(ssize-1) <= '1';
168         BC(ssize-1) <= LL(tsize*2-1);
169
170         -- Now add...!!!
171         LC <= cpa(AC, BC, '0');
172
173
174         -- Assign bits of product.
175         for idx in 0 to tsize-2 loop
176                 X(idx+1) <= L(idx)(0);
```

```
177         end loop;
178
179         for idx in tsize to ssize+tsize-3 loop
180                 X(idx) <= LC(idx-tsize);
181         end loop;
182
183         tmp := fa(not A(ssize-1), not B(tsize-1), LC(ssize-2));
184         X(ssize+tsize-2) <= tmp(0);
185         extra <= tmp(1);
186         X(ssize+tsize-1) <= extra xor LC(ssize-1);
187
188 end process;
189
190 end arch_mul;
```

3. Función No Lineal de Transferencia SATLIN (satlin.vhd).

```
 1 library IEEE;
 2 use IEEE.STD_LOGIC_1164.ALL;
 3 use IEEE.STD_LOGIC_ARITH.ALL;
 4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
 5
 6 entity satlin is
 7         Generic ( ssize: integer := 10;
 8                           iosize: integer := 6;
 9                           point: integer := 6);
10         Port ( inPort : in std_logic_vector(ssize - 1 downto 0);
11                     outPort : out std_logic_vector(iosize - 1 downto 0));
12 end satlin;
13
14 architecture arch_satlin of satlin is
15
16 begin
17
18         -- Comparation & assignment process...
19         process (inPort)
20                 begin
21
22                 if (inPort(ssize - 1) = '1') then -- Negative input...
23                         outPort <= (others => '0');
24                 elsif (inPort >= (2 ** point)) then -- 1 or bigger input...
25                         outPort(iosize - 2 downto 0) <= (others => '0');
26                         outPort(iosize - 1) <= '1';
27                 else -- 0 to 1 input...
28                         outPort <= inPort(point downto point-iosize+1);
29                 end if;
30
31         end process;
32
33 end arch_satlin;
```

4. Multiplexor 2:1 de 6 canales (mux2.vhd).

```
 1 library IEEE;
 2 use IEEE.STD_LOGIC_1164.ALL;
 3 use IEEE.STD_LOGIC_ARITH.ALL;
 4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
 5
 6 entity mux2 is
 7         Generic ( N: integer := 6;
 8                           iosize: integer := 6);
 9     Port (  A, B : in std_logic_vector(N*iosize-1 downto 0);
10                         S : in std_logic;
11                         X : out std_logic_vector(N*iosize-1 downto 0));
12 end mux2;
13
14 architecture arch_mux2 of mux2 is
```

```
15
16  begin
17
18  process (A, B, S)
19  begin
20
21          if s = '0' then
22                  X <= A;
23          else
24                  X <= B;
25          end if;
26
27  end process;
28
29  end arch_mux2;
```

5. Multiplexor 5:1 (mux5.vhd).

```
 1  library IEEE;
 2  use IEEE.STD_LOGIC_1164.ALL;
 3  use IEEE.STD_LOGIC_ARITH.ALL;
 4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
 5
 6  entity mux5 is
 7          Generic ( ssize: integer := 10);
 8          Port ( A : in std_logic_vector(ssize-1 downto 0);
 9             B : in std_logic_vector(ssize-1 downto 0);
10             C : in std_logic_vector(ssize-1 downto 0);
11             D : in std_logic_vector(ssize-1 downto 0);
12             E : in std_logic_vector(ssize-1 downto 0);
13             S : in std_logic_vector(2 downto 0);
14             X : out std_logic_vector(ssize-1 downto 0));
15  end mux5;
16
17  architecture arch_mux5 of mux5 is
18
19  begin
20
21          with S select
22                  X <= A when "000",
23                          B when "001",
24                          C when "010",
25                          D when "011",
26                          E when others;
27
28  end arch_mux5;
```

6. Sumador de 2 entradas con Saturación (satsum.vhd).

```
 1  library IEEE;
 2  use IEEE.STD_LOGIC_1164.ALL;
 3  use IEEE.STD_LOGIC_ARITH.ALL;
 4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
 5
 6  entity satsum is
 7          Generic ( ssize: integer := 10);
 8          Port ( opA : in std_logic_vector(ssize-1 downto 0);
 9                     opB : in std_logic_vector(ssize-1 downto 0);
10                     resX : out std_logic_vector(ssize-1 downto 0));
11  end satsum;
12
13  architecture arch_satsum of satsum is
14  signal tSum: std_logic_vector(ssize-1 downto 0);
15  begin
16
17          tSum <= opA + opB;
18
```

```
19  process (opA, opB, tSum)
20  begin
21
22          if (tSum(ssize-1) /= opA(ssize-1)) and ((opA(ssize-1) xnor opB(ssize-1))='1')
23  then
24                  resX(ssize-1) <= opA(ssize-1);
25                  resX(ssize-2 downto 0) <= (others => not opA(ssize-1));
26          else
27                  resX <= tSum;
28          end if;
29
30  end process;
31
32  end arch_satsum;
```

7. Neurona Digital (neuron1.vhd).

```
 1  library IEEE;
 2  use IEEE.std_logic_1164.ALL;
 3  use IEEE.std_logic_ARITH.ALL;
 4  use IEEE.std_logic_UNSIGNED.ALL;
 5  use work.npack.all;
 6
 7  entity neuron1 is
 8          Generic ( neighborhood: integer := 6;
 9                          iosize: integer := 6;
10                          ssize: integer := 10;
11                          tsize: integer := 8;
12                          point: integer := 6);
13          Port ( uc: in std_logic_vector(iosize-1 downto 0);  -- self-input: priority
14  weight
15                  u1, u2, u3, u4, u5, u6: in std_logic_vector(iosize-1 downto 0);  -
16  - neighborhood inputs
17                  y1, y2, y3, y4, y5, y6: in std_logic_vector(iosize-1 downto 0);  -
18  - neighborhood outputs
19                  x0: in std_logic;  -- initial state
20                  temp: in std_logic_vector(tsize-1 downto 0);  -- template entry
21                  rowsel, colsel: in std_logic;  -- row & column select
22                  control : in std_logic_vector(7 downto 0);  -- control lines
23                  yc : out std_logic_vector(iosize-1 downto 0);  -- cell output
24                  ucout : out std_logic_vector(iosize-1 downto 0));
25  end neuron1;
26
27  architecture arch_neuron of neuron1 is
28
29  -- ===== Signals =====
30
31  -- Input Buses
32  signal inBus1, inBus2, sel6i: std_logic_vector(neighborhood*iosize-1 downto 0);  --
33  Input & selected buses.
34  signal out6Add: std_logic_vector(iosize+2 downto 0);  -- iosize+log2(6) =~ iosize+3
35  signal inSel: std_logic;  -- 2-input 6-channel bus selector
36
37  -- Inputs to 6-Mux
38  signal mFeed0: std_logic_vector(ssize-1 downto 0);
39  signal mFeed1: std_logic_vector(ssize-1 downto 0);
40  signal mFeed2: std_logic_vector(ssize-1 downto 0);
41  signal mFeed3: std_logic_vector(ssize-1 downto 0);
42  signal mulSel: std_logic_vector(2 downto 0);
43
44  -- Arithmetics
45  signal inMul1: std_logic_vector(ssize-1 downto 0);
46  signal inMul2: std_logic_vector(tsize-1 downto 0);
47  signal outMul: std_logic_vector(ssize+tsize-1 downto 0);
48
49  signal outClip: std_logic_vector(ssize-1 downto 0);
50  signal outSum2: std_logic_vector(ssize-1 downto 0);
51
52  -- Internal input storage register
```

```
signal ucreg: std_logic_vector(iosize-1 downto 0);

-- Accumulator & State registers signals
signal acc: std_logic_vector(ssize-1 downto 0);
signal state: std_logic_vector(ssize-1 downto 0);
signal accXfer, stateXfer, stateReset, accReset: std_logic;

signal cellOut: std_logic_vector(iosize-1 downto 0);


-- Architecture begins

begin

        -- *** Neuron structure ***

        -- Cells Inputs & Outputs
        -- Create input bus (6-bus).
        inBus1 <= u1&u2&u3&u4&u5&u6;
        inBus2 <= y1&y2&y3&y4&y5&y6;

        -- 2-Mux form bus to 6-input adder.
        uInMux2: mux2 generic map (iosize=>iosize) port map (A => inBus1, B =>
inBus2, S => inSel, X => sel6i);
        uIn6Add: csa6 generic map (iosize=>iosize) port map (A => sel6i(iosize*6-1
downto iosize*5), B => sel6i(iosize*5-1 downto iosize*4), C => sel6i(iosize*4-1
downto iosize*3), D => sel6i(iosize*3-1 downto iosize*2), E => sel6i(iosize*2-1
downto iosize), F => sel6i(iosize-1 downto 0), X => out6Add);

        -- Prepare inputs to 5-Mux
        mFeed0 <= '0' & out6Add(iosize+1 downto 0) & '0';
        mFeed1 <= "000" & ucreg & '0';
        mFeed2 <= "0001000000";
        mFeed3 <= "000" & cellOut & '0';

        uInMux5: mux5 generic map (ssize=>ssize) port map (A => mFeed0, B => mFeed1,
C => mFeed2, D => mFeed3, E => state, S => mulSel, X => inMul1);

        -- Multiplier
        uMul: mul generic map (ssize=>ssize, tsize=>tsize) port map (A=>inMul1,
B=>inMul2, X=>outMul);

        -- Accumulator: Saturated Adder. (hissy, hissy, little snakey! ~~~~)
        uSum2: satsum generic map (ssize=>ssize) port map (opA => outClip, opB =>
acc, resX => outSum2);

        -- satlin xfer function
        uFunc: satlin generic map (ssize=>ssize, iosize=>iosize, point=>point) port
map (inPort => state, outport => cellOut);

        -- Output
        yc <= cellOut;
        ucout <= ucreg;

        -- Clip (Truncate)
        outClip(ssize-1) <= outMul(ssize+tsize-1);
        outclip(ssize-2 downto 0) <= outMul(15 downto 7);  -- <= Calculate this,
based on fpoint.


inMul2 <= temp;

        -- *** Control Section ***

inSel <= control(7);
mulSel <= control(6 downto 4);

accXfer <= control(2);
stateXfer <= control(0);
accReset <= control(3);

```

```
124  -- Cell selection process
125  process (control(1), rowsel, colsel)
126  begin
127         if ((rowsel and colsel) = '1') then
128                 stateReset <= control(1);
129         else
130                 stateReset <= '1';
131         end if;
132  end process;
133
134  -- Internal Priority Weight Register
135  -- Shares control signaling with state register
136  process (stateReset)
137  begin
138
139         if (stateReset'event and stateReset='0') then
140                 ucreg <= uc;
141         end if;
142
143  end process;
144
145  -- Accumulator Register
146  process (accXfer, accReset)
147  begin
148
149         if (accReset = '0') then
150                 acc <= (others => '0');
151         elsif (accXfer'event and accXfer='1') then
152                 acc <= outSum2(ssize-1 downto 0);
153         end if;
154
155  end process;
156
157  -- State Register (CUSTOM)
158  process (stateXfer, stateReset)
159  begin
160
161         if (stateReset = '0') then
162                 state <= (6 => x0, others => '0');  -- <= Calculate this, based on
163  fpoint.
164         elsif (stateXfer'event and stateXfer='1') then
165                 state <= acc;
166         end if;
167
168  end process;
169
170  end arch_neuron;
```

8. Memoria Secuencial (seqmem.vhd).

```
 1  library IEEE;
 2  use IEEE.std_logic_1164.ALL;
 3  use IEEE.std_logic_ARITH.ALL;
 4  use IEEE.std_logic_UNSIGNED.ALL;
 5
 6  entity seqmem is
 7         Generic ( size: integer := 6;
 8                   width: integer := 8);
 9      Port ( d : in std_logic_vector(width-1 downto 0);
10             w, clk, reset : in std_logic;
11             q : out std_logic_vector(width-1 downto 0));
12  end seqmem;
13
14  architecture arch_seqmem of seqmem is
15
16  -- RAM Signals
17  type ram_type is array (size-1 downto 0) of std_logic_vector (width-1 downto 0);
18  signal memory: ram_type;
19  signal mem_read: integer range 0 to size-1;
```

```
20  -- Counter Signals
21  signal mAdd: integer range 0 to size-1;
22
23  begin
24
25  -- RAM Structure
26  RAM: process (clk, w)
27  begin
28          if (clk'event and clk = '1') then
29                  if (w='0') then
30                          memory(conv_integer(mAdd)) <= d;
31                  end if;
32                  mem_read <= conv_integer(mAdd);
33          end if;
34          q <= memory(mem_read);
35  end process;
36
37  -- Counter Structure
38  CTR: process (clk, reset)
39  begin
40          if (reset='0') then
41                  mAdd <= 0;
42          elsif (clk'event and clk = '1') then
43                  if mAdd < (size-1) then
44                          mAdd <= mAdd+1;
45                  else
46                          mAdd <= 0;
47                  end if;
48          end if;
49  end process;
50
51  end arch_seqmem;
```

9. Contador Selector de Columnas y Filas (`crsel.vhd`).

```
 1  library IEEE;
 2  use IEEE.std_logic_1164.ALL;
 3  use IEEE.std_logic_ARITH.ALL;
 4  use IEEE.std_logic_UNSIGNED.ALL;
 5
 6  entity crsel is
 7          Port ( clk, reset : in std_logic;
 8                      cSel, rSel : out std_logic_vector(3 downto 0));
 9  end crsel;
10
11  architecture arch_crsel of crsel is
12
13  signal rcCtr: std_logic_vector (3 downto 0);
14
15  begin
16
17  -- Cell selector counter
18  COUNTER: process (clk, reset)
19  begin
20          if (reset = '0') then
21                  rcCtr <= (others=>'0');
22          elsif (clk'event and clk = '1') then
23                  rcCtr <= rcCtr + 1;
24          end if;
25  end process;
26
27  -- DECOS for cell selection
28  DECOS: process (rcCtr)
29  begin
30          case rcCtr(3 downto 2) is  -- rows
31                  when "00" =>
32                          rSel <= "1000";
33                  when "01" =>
34                          rSel <= "0100";
```

```
35                when "10" =>
36                       rSel <= "0010";
37                when "11" =>
38                       rSel <= "0001";
39                when others =>
40                       rSel <= "0000";
41        end case;
42        case rcCtr(1 downto 0) is  -- cols
43                when "00" =>
44                       cSel <= "1000";
45                when "01" =>
46                       cSel <= "0100";
47                when "10" =>
48                       cSel <= "0010";
49                when "11" =>
50                       cSel <= "0001";
51                when others =>
52                       cSel <= "0000";
53        end case;
54  end process;
55
56  end arch_crsel;
```

10. Modulo de Control (`control.vhd`).

```
 1  library IEEE;
 2  use IEEE.std_logic_1164.ALL;
 3  use IEEE.std_logic_ARITH.ALL;
 4  use IEEE.std_logic_UNSIGNED.ALL;
 5  use work.cpack.all;
 6
 7  entity control is
 8        Generic ( maxIter: integer := 100;  -- Maxiumum iterations per solution cycle
 9                        numTemp: integer := 6;  -- Number of templates
10                        tempSize: integer := 8 );  -- Template word size
11        Port ( mClk : in std_logic;  -- Master clock
12                  reset : in std_logic;  -- Master reset
13                  inTemp : in std_logic_vector(7 downto 0);  -- Template programming
14  input
15                  inX0 : in std_logic;  -- Input of initial states
16                  inW : in std_logic_vector(5 downto 0);  -- Input of priority
17  weights
18                  tLoad : in std_logic;  -- Template load enable
19                  tClk : in std_logic;  -- Template load clock
20                  dClk : in std_logic;  -- Data load clock
21                  dRes : in std_logic;  -- Data Selector Reset
22                  solve: in std_logic;  -- Solve Start
23                -- OUTPUTS --
24                  rSel : out std_logic_vector(3 downto 0);  -- Row selection
25                  cSel : out std_logic_vector(3 downto 0);  -- Column Selection
26                  outX0 : out std_logic;  -- Initial States output to cells
27                  outW : out std_logic_vector(5 downto 0);  -- Weights output to
28  cells
29                  temp : out std_logic_vector(7 downto 0);  -- Templates out
30                  ctrl : out std_logic_vector(7 downto 0));  -- Control to all cells
31  end control;
32
33  architecture arch_control of control is
34
35  -- Main Control States & Signals
36  type sState_Type is (st_0Idle,
37        st_PreH,
38        st_PreL,
39        st_InitH,
40        st_InitL,
41        st_T1H,
42        st_T1L,
43        st_T2H,
44        st_T2L,
```

```
45          st_T3H,
46          st_T3L,
47          st_T4H,
48          st_T4L,
49          st_T5H,
50          st_T5L,
51          st_T6H,
52          st_T6L,
53          st_XferH,
54          st_XferL);
55  signal state, next_state: sState_Type;
56
57  signal ramExtRes: std_logic;  -- RAM reset from machine
58  signal sClk: std_logic;  -- RAM Clock from machine
59
60  -- RAM Signals
61  signal ramRes, ramMemClk: std_logic;  -- Memory clock input
62  -- ColRow Selector Signals
63  signal crRes: std_logic;  -- ColRow Selector Reset
64
65  -- Iteration Counter
66  signal iReset, iClk, iExtRes: std_logic;
67  signal iter: integer range 0 to maxIter;
68
69
70  begin
71
72  -- RAM Reset
73  ramRes <= reset and ramExtRes;
74  -- RAM Clock from LOAD CYCLE or from STATE MACHINE
75  ramMemClk <= (tClk and not tLoad) or sClk;
76  -- Sequential RAM
77  uRAM: seqmem generic map ( size=>numTemp, width=>tempSize ) port map ( d=>inTemp,
78  w=>tLoad, clk=>ramMemClk, reset=>ramRes, q=>temp );
79
80
81  -- ColRow Selector Reset
82  crRes <= reset and dRes;
83  -- ColRow Selector
84  uSEL: crsel port map ( clk=>dClk, reset=>crRes, cSel=>cSel, rSel=>rSel );
85
86
87  -- Iteration Counter
88  iReset <= reset and iExtRes;
89  ITER_CTR: process (iClk, iReset)
90  begin
91          if (iReset='0') then
92                  iter <= 0;
93          elsif (iClk'event and iClk='1') then
94                  iter <= iter+1;
95          end if;
96  end process;
97
98  -- Main Control Machine
99
100 -- Synchronization process
101 CTRL_SYNC: process (mClk, reset)
102 begin
103         if (reset='0') then
104                 state <= st_0Idle;
105         elsif (mClk'event and mClk='1') then
106                 state <= next_state;
107         end if;
108 end process;
109
110 -- Outputs decoder
111 CTRL_DECO: process (state, mClk)
112 begin
113
114         case state is
115                 when st_0Idle =>
```

```
116                     -- Initial State
117                     ramExtRes <= '1';  -- Ram Reset out
118                     iExtRes <= '1';  -- Iter Counter Reset out
119                     iClk <= '0';  -- Iter Counter Clock out
120                     sClk <= '0';  -- Ram Clock out
121                     ctrl(7) <= '0';  -- Select inBus1
122                     ctrl(6 downto 4) <= "010";  -- Select input 2 - "1" for I
123                     ctrl(3) <= '1';  -- Acc Reset out
124                     ctrl(2) <= '0';  -- Acc Xfer out
125                     ctrl(0) <= '0';  -- Acc to state Xfer out
126
127                 when st_PreH =>
128                     ramExtRes <= '0';  -- Reset RAM
129                     iExtRes <= '0';  -- Reset Iter Counter
130
131                     ctrl(7 downto 2) <= "001010";
132                     ctrl(0) <= '0';
133
134                 when st_PreL =>
135                     ramExtRes <= '1';  -- RAM Reset out
136                     ctrl(3) <= '0';  -- Acc Reset
137                     iExtRes <= '1';  -- Iter Counter Reset out
138
139                     ctrl(7 downto 4) <= "0010";
140                     ctrl(2) <= '0';
141                     ctrl(0) <= '0';
142
143                 when st_InitH =>
144                     ramExtRes <= '1';
145                     ctrl(7) <= '0';  -- Select inBus1 - un
146                     ctrl(6 downto 4) <= "010";  -- Select input 2 - "1" for I
147                     ctrl(3) <= '1';  -- Acc Reset out
148                     ctrl(0) <= '0';  -- Acc to state Xfer pulse out
149                     sClk <= '0';
150
151                     ctrl(2) <= '0';
152
153                 when st_InitL =>
154                     ramExtRes <= '1';
155                     sClk <= '1';  -- Read Mem Loc 0 (I)
156
157                     ctrl(7 downto 2) <= "001010";
158                     ctrl(0) <= '0';
159
160                 when st_T1H =>
161                     ramExtRes <= '1';
162                     ctrl(2) <= '0';  -- Clear Xfer to acc
163                     sClk <= '0';  -- Clear mem clk
164
165                     ctrl(7 downto 3) <= "00101";
166                     ctrl(0) <= '0';
167
168                 when st_T1L =>
169                     ramExtRes <= '1';
170                     ctrl(2) <= '1';  -- Xfer to acc
171                     sClk <= '1';  -- Read Mem Loc 1 (beta)
172                     ctrl(6 downto 4) <= "001";  -- Select input 1 - uc for beta
173
174                     ctrl(7) <= '0';
175                     ctrl(3) <= '1';
176                     ctrl(0) <= '0';
177
178                 when st_T2H =>
179                     ramExtRes <= '1';
180                     ctrl(2) <= '0';
181                     sClk <= '0';
182
183                     ctrl(7 downto 3) <= "00011";
184                     ctrl(0) <= '0';
185
186                 when st_T2L =>
```

```
                        ramExtRes <= '1';
                        ctrl(2) <= '1';  -- Xfer to acc
                        sClk <= '1';  -- Read Mem Loc 2 (B)
                        ctrl(6 downto 4) <= "000";  -- Select input 0 - un for B

                        ctrl(7) <= '0';
                        ctrl(3) <= '1';
                        ctrl(0) <= '0';

                when st_T3H =>
                        ramExtRes <= '1';
                        ctrl(2) <= '0';
                        sClk <= '0';

                        ctrl(7 downto 3) <= "00001";
                        ctrl(0) <= '0';

                when st_T3L =>
                        ramExtRes <= '1';
                        ctrl(2) <= '1';  -- Xfer to acc
                        sClk <= '1';  -- Read Mem Loc 3 (alpha)
                        ctrl(6 downto 4) <= "011";  -- Select input 3 - yc for alpha
                        ctrl(7) <= '1';  -- Select inBus2 - yn

                        ctrl(3) <= '1';
                        ctrl(0) <= '0';

                when st_T4H =>
                        ramExtRes <= '1';
                        ctrl(2) <= '0';
                        sClk <= '0';

                        ctrl(7 downto 3) <= "10111";
                        ctrl(0) <= '0';

                when st_T4L =>
                        ramExtRes <= '1';
                        ctrl(2) <= '1';  -- Xfer to acc
                        sClk <= '1';  -- Read Mem Loc 4 (1-h)
                        ctrl(6 downto 4) <= "100";  -- Select input 4 - state for 1-h

                        ctrl(7) <= '1';
                        ctrl(3) <= '1';
                        ctrl(0) <= '0';

                when st_T5H =>
                        ramExtRes <= '1';
                        ctrl(2) <= '0';
                        sClk <= '0';

                        ctrl(7 downto 3) <= "11001";
                        ctrl(0) <= '0';

                when st_T5L =>
                        ramExtRes <= '1';
                        ctrl(2) <= '1';  -- Xfer to acc
                        sClk <= '1';  -- Read Mem Loc 5 (A)
                        ctrl(6 downto 4) <= "000";  -- Select input 0 - yn for A

                        ctrl(7) <= '1';
                        ctrl(3) <= '1';
                        ctrl(0) <= '0';

                when st_T6H =>
                        ramExtRes <= '1';
                        ctrl(2) <= '0';
                        sClk <= '0';

                        ctrl(7 downto 3) <= "10001";
                        ctrl(0) <= '0';

```

```
258                 when st_T6L =>
259                         ctrl(2) <= '1';  -- Xfer to acc
260                         ramExtRes <= '0';  -- Reset RAM Counter
261
262                         ctrl(7 downto 3) <= "10001";
263                         ctrl(0) <= '0';
264
265                 when st_XferH =>
266                         ctrl(2) <= '0';
267                         ramExtRes <= '1';  -- RAM Counter Reset Out
268                         iClk <= '1';  -- Iter++
269
270                         ctrl(7 downto 3) <= "10001";
271                         ctrl(0) <= '0';
272
273                 when st_XferL =>
274                         ramExtRes <= '1';
275                         ctrl(0) <= '1';  -- Xfer acc to state
276                         ctrl(3) <= '0';  -- Reset Acc
277                         iClk <= '0';
278
279                         ctrl(7 downto 4) <= "1000";
280                         ctrl(2) <= '0';
281
282         end case;
283 end process;
284
285 -- Next state selector
286 CTRL_NEXT: process (state)
287 begin
288         next_state <= state;
289         case state is
290                 when st_0Idle =>
291                         if solve = '0' then
292                                 next_state <= st_PreH;
293                         end if;
294                 when st_PreH =>
295                         next_state <= st_PreL;
296                 when st_PreL =>
297                         next_state <= st_InitH;
298                 when st_InitH =>
299                         next_state <= st_InitL;
300                 when st_InitL =>
301                         next_state <= st_T1H;
302                 when st_T1H =>
303                         next_state <= st_T1L;
304                 when st_T1L =>
305                         next_state <= st_T2H;
306                 when st_T2H =>
307                         next_state <= st_T2L;
308                 when st_T2L =>
309                         next_state <= st_T3H;
310                 when st_T3H =>
311                         next_state <= st_T3L;
312                 when st_T3L =>
313                         next_state <= st_T4H;
314                 when st_T4H =>
315                         next_state <= st_T4L;
316                 when st_T4L =>
317                         next_state <= st_T5H;
318                 when st_T5H =>
319                         next_state <= st_T5L;
320                 when st_T5L =>
321                         next_state <= st_T6H;
322                 when st_T6H =>
323                         next_state <= st_T6L;
324                 when st_T6L =>
325                         next_state <= st_XferH;
326                 when st_XferH =>
327                         next_state <= st_XferL;
328                 when st_XferL =>
```

```
329                         if iter = maxIter then
330                                 next_state <= st_0Idle;
331                         else
332                                 next_state <= st_InitH;
333                         end if;
334         end case;
335 end process;
336
337 -- Control Signals Assignment
338 ctrl(1) <= dClk;
339
340 -- Direct assignment - In-to-out initial states and weights
341 outX0 <= inX0;
342 outW <= inW;
343
344
345 end arch_control;
```

11. Red Neuronal Extendida FPS (`fpsnet.vhd`).

```
  1 library IEEE;
  2 use IEEE.std_logic_1164.ALL;
  3 use IEEE.std_logic_ARITH.ALL;
  4 use IEEE.std_logic_UNSIGNED.ALL;
  5
  6 entity fpsnet1 is
  7    port ( dClk   : in    std_logic;
  8                 dRes   : in    std_logic;
  9                 inTemp : in    std_logic_vector (7 downto 0);
 10                 inW    : in    std_logic_vector (5 downto 0);
 11                 inX0   : in    std_logic;
 12                 mClk   : in    std_logic;
 13                 reset  : in    std_logic;
 14                 solve  : in    std_logic;
 15                 tClk   : in    std_logic;
 16                 tLoad  : in    std_logic;
 17                 netout : out   std_logic_vector (15 downto 0));
 18 end fpsnet1;
 19
 20 architecture BEHAVIORAL of fpsnet1 is
 21    signal cSel   : std_logic_vector (3 downto 0);
 22    signal CTRL   : std_logic_vector (7 downto 0);
 23    signal OUTU11 : std_logic_vector (5 downto 0);
 24    signal OUTU12 : std_logic_vector (5 downto 0);
 25    signal OUTU13 : std_logic_vector (5 downto 0);
 26    signal OUTU14 : std_logic_vector (5 downto 0);
 27    signal OUTU21 : std_logic_vector (5 downto 0);
 28    signal OUTU22 : std_logic_vector (5 downto 0);
 29    signal OUTU23 : std_logic_vector (5 downto 0);
 30    signal OUTU24 : std_logic_vector (5 downto 0);
 31    signal OUTU31 : std_logic_vector (5 downto 0);
 32    signal OUTU32 : std_logic_vector (5 downto 0);
 33    signal OUTU33 : std_logic_vector (5 downto 0);
 34    signal OUTU34 : std_logic_vector (5 downto 0);
 35    signal OUTU41 : std_logic_vector (5 downto 0);
 36    signal OUTU42 : std_logic_vector (5 downto 0);
 37    signal OUTU43 : std_logic_vector (5 downto 0);
 38    signal OUTU44 : std_logic_vector (5 downto 0);
 39    signal outW   : std_logic_vector (5 downto 0);
 40    signal outX0  : std_logic;
 41    signal OUTY11 : std_logic_vector (5 downto 0);
 42    signal OUTY12 : std_logic_vector (5 downto 0);
 43    signal OUTY13 : std_logic_vector (5 downto 0);
 44    signal OUTY14 : std_logic_vector (5 downto 0);
 45    signal OUTY21 : std_logic_vector (5 downto 0);
 46    signal OUTY22 : std_logic_vector (5 downto 0);
 47    signal OUTY23 : std_logic_vector (5 downto 0);
 48    signal OUTY24 : std_logic_vector (5 downto 0);
 49    signal OUTY31 : std_logic_vector (5 downto 0);
```

```
   signal OUTY32 : std_logic_vector (5 downto 0);
   signal OUTY33 : std_logic_vector (5 downto 0);
   signal OUTY34 : std_logic_vector (5 downto 0);
   signal OUTY41 : std_logic_vector (5 downto 0);
   signal OUTY42 : std_logic_vector (5 downto 0);
   signal OUTY43 : std_logic_vector (5 downto 0);
   signal OUTY44 : std_logic_vector (5 downto 0);
   signal rSel   : std_logic_vector (3 downto 0);
   signal TEMP   : std_logic_vector (7 downto 0);

   component neuron1
      port ( x0      : in    std_logic;
             rowsel  : in    std_logic;
             colsel  : in    std_logic;
             uc      : in    std_logic_vector (5 downto 0);
             u1      : in    std_logic_vector (5 downto 0);
             u2      : in    std_logic_vector (5 downto 0);
             u3      : in    std_logic_vector (5 downto 0);
             u4      : in    std_logic_vector (5 downto 0);
             u5      : in    std_logic_vector (5 downto 0);
             u6      : in    std_logic_vector (5 downto 0);
             y1      : in    std_logic_vector (5 downto 0);
             y2      : in    std_logic_vector (5 downto 0);
             y3      : in    std_logic_vector (5 downto 0);
             y4      : in    std_logic_vector (5 downto 0);
             y5      : in    std_logic_vector (5 downto 0);
             y6      : in    std_logic_vector (5 downto 0);
             temp    : in    std_logic_vector (7 downto 0);
             control : in    std_logic_vector (7 downto 0);
             yc      : out   std_logic_vector (5 downto 0);
             ucout   : out   std_logic_vector (5 downto 0));
   end component;

   component control
      port ( mClk   : in    std_logic;
             reset  : in    std_logic;
             inX0   : in    std_logic;
             tLoad  : in    std_logic;
             tClk   : in    std_logic;
             dClk   : in    std_logic;
             dRes   : in    std_logic;
             solve  : in    std_logic;
             inTemp : in    std_logic_vector (7 downto 0);
             inW    : in    std_logic_vector (5 downto 0);
             outX0  : out   std_logic;
             rSel   : out   std_logic_vector (3 downto 0);
             cSel   : out   std_logic_vector (3 downto 0);
             outW   : out   std_logic_vector (5 downto 0);
             temp   : out   std_logic_vector (7 downto 0);
             ctrl   : out   std_logic_vector (7 downto 0));
   end component;

begin

   cell11 : neuron1
      port map (colsel=>cSel(3),
                control(7 downto 0)=>CTRL(7 downto 0),
                rowsel=>rSel(3),
                temp(7 downto 0)=>TEMP(7 downto 0),
                uc(5 downto 0)=>outW(5 downto 0),
                u1(5 downto 0)=>OUTU21(5 downto 0),
                u2(5 downto 0)=>OUTU31(5 downto 0),
                u3(5 downto 0)=>OUTU41(5 downto 0),
                u4(5 downto 0)=>OUTU12(5 downto 0),
                u5(5 downto 0)=>OUTU13(5 downto 0),
                u6(5 downto 0)=>OUTU14(5 downto 0),
                x0=>outX0,
                y1(5 downto 0)=>OUTY21(5 downto 0),
                y2(5 downto 0)=>OUTY31(5 downto 0),
                y3(5 downto 0)=>OUTY41(5 downto 0),
                y4(5 downto 0)=>OUTY12(5 downto 0),
```

```
121                y5(5 downto 0)=>OUTY13(5 downto 0),
122                y6(5 downto 0)=>OUTY14(5 downto 0),
123                ucout(5 downto 0)=>OUTU11(5 downto 0),
124                yc(5 downto 0)=>OUTY11(5 downto 0));
125
126     cell12 : neuron1
127        port map (colsel=>cSel(2),
128                control(7 downto 0)=>CTRL(7 downto 0),
129                rowsel=>rSel(3),
130                temp(7 downto 0)=>TEMP(7 downto 0),
131                uc(5 downto 0)=>outW(5 downto 0),
132                u1(5 downto 0)=>OUTU11(5 downto 0),
133                u2(5 downto 0)=>OUTU13(5 downto 0),
134                u3(5 downto 0)=>OUTU14(5 downto 0),
135                u4(5 downto 0)=>OUTU22(5 downto 0),
136                u5(5 downto 0)=>OUTU32(5 downto 0),
137                u6(5 downto 0)=>OUTU42(5 downto 0),
138                x0=>outX0,
139                y1(5 downto 0)=>OUTY11(5 downto 0),
140                y2(5 downto 0)=>OUTY13(5 downto 0),
141                y3(5 downto 0)=>OUTY14(5 downto 0),
142                y4(5 downto 0)=>OUTY22(5 downto 0),
143                y5(5 downto 0)=>OUTY32(5 downto 0),
144                y6(5 downto 0)=>OUTY42(5 downto 0),
145                ucout(5 downto 0)=>OUTU12(5 downto 0),
146                yc(5 downto 0)=>OUTY12(5 downto 0));
147
148     cell13 : neuron1
149        port map (colsel=>cSel(1),
150                control(7 downto 0)=>CTRL(7 downto 0),
151                rowsel=>rSel(3),
152                temp(7 downto 0)=>TEMP(7 downto 0),
153                uc(5 downto 0)=>outW(5 downto 0),
154                u1(5 downto 0)=>OUTU11(5 downto 0),
155                u2(5 downto 0)=>OUTU12(5 downto 0),
156                u3(5 downto 0)=>OUTU14(5 downto 0),
157                u4(5 downto 0)=>OUTU23(5 downto 0),
158                u5(5 downto 0)=>OUTU33(5 downto 0),
159                u6(5 downto 0)=>OUTU43(5 downto 0),
160                x0=>outX0,
161                y1(5 downto 0)=>OUTY11(5 downto 0),
162                y2(5 downto 0)=>OUTY12(5 downto 0),
163                y3(5 downto 0)=>OUTY14(5 downto 0),
164                y4(5 downto 0)=>OUTY23(5 downto 0),
165                y5(5 downto 0)=>OUTY33(5 downto 0),
166                y6(5 downto 0)=>OUTY43(5 downto 0),
167                ucout(5 downto 0)=>OUTU13(5 downto 0),
168                yc(5 downto 0)=>OUTY13(5 downto 0));
169
170     cell14 : neuron1
171        port map (colsel=>cSel(0),
172                control(7 downto 0)=>CTRL(7 downto 0),
173                rowsel=>rSel(3),
174                temp(7 downto 0)=>TEMP(7 downto 0),
175                uc(5 downto 0)=>outW(5 downto 0),
176                u1(5 downto 0)=>OUTU11(5 downto 0),
177                u2(5 downto 0)=>OUTU12(5 downto 0),
178                u3(5 downto 0)=>OUTU13(5 downto 0),
179                u4(5 downto 0)=>OUTU24(5 downto 0),
180                u5(5 downto 0)=>OUTU34(5 downto 0),
181                u6(5 downto 0)=>OUTU44(5 downto 0),
182                x0=>outX0,
183                y1(5 downto 0)=>OUTY11(5 downto 0),
184                y2(5 downto 0)=>OUTY12(5 downto 0),
185                y3(5 downto 0)=>OUTY13(5 downto 0),
186                y4(5 downto 0)=>OUTY24(5 downto 0),
187                y5(5 downto 0)=>OUTY34(5 downto 0),
188                y6(5 downto 0)=>OUTY44(5 downto 0),
189                ucout(5 downto 0)=>OUTU14(5 downto 0),
190                yc(5 downto 0)=>OUTY14(5 downto 0));
191
```

```
192     cell21 : neuron1
193        port map (colsel=>cSel(3),
194                  control(7 downto 0)=>CTRL(7 downto 0),
195                  rowsel=>rSel(2),
196                  temp(7 downto 0)=>TEMP(7 downto 0),
197                  uc(5 downto 0)=>outW(5 downto 0),
198                  u1(5 downto 0)=>OUTU11(5 downto 0),
199                  u2(5 downto 0)=>OUTU31(5 downto 0),
200                  u3(5 downto 0)=>OUTU41(5 downto 0),
201                  u4(5 downto 0)=>OUTU22(5 downto 0),
202                  u5(5 downto 0)=>OUTU23(5 downto 0),
203                  u6(5 downto 0)=>OUTU24(5 downto 0),
204                  x0=>outX0,
205                  y1(5 downto 0)=>OUTY11(5 downto 0),
206                  y2(5 downto 0)=>OUTY31(5 downto 0),
207                  y3(5 downto 0)=>OUTY41(5 downto 0),
208                  y4(5 downto 0)=>OUTY22(5 downto 0),
209                  y5(5 downto 0)=>OUTY23(5 downto 0),
210                  y6(5 downto 0)=>OUTY24(5 downto 0),
211                  ucout(5 downto 0)=>OUTU21(5 downto 0),
212                  yc(5 downto 0)=>OUTY21(5 downto 0));
213
214     cell22 : neuron1
215        port map (colsel=>cSel(2),
216                  control(7 downto 0)=>CTRL(7 downto 0),
217                  rowsel=>rSel(2),
218                  temp(7 downto 0)=>TEMP(7 downto 0),
219                  uc(5 downto 0)=>outW(5 downto 0),
220                  u1(5 downto 0)=>OUTU21(5 downto 0),
221                  u2(5 downto 0)=>OUTU23(5 downto 0),
222                  u3(5 downto 0)=>OUTU24(5 downto 0),
223                  u4(5 downto 0)=>OUTU12(5 downto 0),
224                  u5(5 downto 0)=>OUTU32(5 downto 0),
225                  u6(5 downto 0)=>OUTU42(5 downto 0),
226                  x0=>outX0,
227                  y1(5 downto 0)=>OUTY21(5 downto 0),
228                  y2(5 downto 0)=>OUTY23(5 downto 0),
229                  y3(5 downto 0)=>OUTY24(5 downto 0),
230                  y4(5 downto 0)=>OUTY12(5 downto 0),
231                  y5(5 downto 0)=>OUTY32(5 downto 0),
232                  y6(5 downto 0)=>OUTY42(5 downto 0),
233                  ucout(5 downto 0)=>OUTU22(5 downto 0),
234                  yc(5 downto 0)=>OUTY22(5 downto 0));
235
236     cell23 : neuron1
237        port map (colsel=>cSel(1),
238                  control(7 downto 0)=>CTRL(7 downto 0),
239                  rowsel=>rSel(2),
240                  temp(7 downto 0)=>TEMP(7 downto 0),
241                  uc(5 downto 0)=>outW(5 downto 0),
242                  u1(5 downto 0)=>OUTU21(5 downto 0),
243                  u2(5 downto 0)=>OUTU22(5 downto 0),
244                  u3(5 downto 0)=>OUTU24(5 downto 0),
245                  u4(5 downto 0)=>OUTU13(5 downto 0),
246                  u5(5 downto 0)=>OUTU33(5 downto 0),
247                  u6(5 downto 0)=>OUTU43(5 downto 0),
248                  x0=>outX0,
249                  y1(5 downto 0)=>OUTY21(5 downto 0),
250                  y2(5 downto 0)=>OUTY22(5 downto 0),
251                  y3(5 downto 0)=>OUTY24(5 downto 0),
252                  y4(5 downto 0)=>OUTY13(5 downto 0),
253                  y5(5 downto 0)=>OUTY33(5 downto 0),
254                  y6(5 downto 0)=>OUTY43(5 downto 0),
255                  ucout(5 downto 0)=>OUTU23(5 downto 0),
256                  yc(5 downto 0)=>OUTY23(5 downto 0));
257
258     cell24 : neuron1
259        port map (colsel=>cSel(0),
260                  control(7 downto 0)=>CTRL(7 downto 0),
261                  rowsel=>rSel(2),
262                  temp(7 downto 0)=>TEMP(7 downto 0),
```

```
263                uc(5 downto 0)=>outW(5 downto 0),
264                u1(5 downto 0)=>OUTU21(5 downto 0),
265                u2(5 downto 0)=>OUTU22(5 downto 0),
266                u3(5 downto 0)=>OUTU23(5 downto 0),
267                u4(5 downto 0)=>OUTU14(5 downto 0),
268                u5(5 downto 0)=>OUTU34(5 downto 0),
269                u6(5 downto 0)=>OUTU44(5 downto 0),
270                x0=>outX0,
271                y1(5 downto 0)=>OUTY21(5 downto 0),
272                y2(5 downto 0)=>OUTY22(5 downto 0),
273                y3(5 downto 0)=>OUTY23(5 downto 0),
274                y4(5 downto 0)=>OUTY14(5 downto 0),
275                y5(5 downto 0)=>OUTY34(5 downto 0),
276                y6(5 downto 0)=>OUTY44(5 downto 0),
277                ucout(5 downto 0)=>OUTU24(5 downto 0),
278                yc(5 downto 0)=>OUTY24(5 downto 0));
279
280     cell31 : neuron1
281        port map (colsel=>cSel(3),
282                control(7 downto 0)=>CTRL(7 downto 0),
283                rowsel=>rSel(1),
284                temp(7 downto 0)=>TEMP(7 downto 0),
285                uc(5 downto 0)=>outW(5 downto 0),
286                u1(5 downto 0)=>OUTU32(5 downto 0),
287                u2(5 downto 0)=>OUTU33(5 downto 0),
288                u3(5 downto 0)=>OUTU34(5 downto 0),
289                u4(5 downto 0)=>OUTU11(5 downto 0),
290                u5(5 downto 0)=>OUTU12(5 downto 0),
291                u6(5 downto 0)=>OUTU14(5 downto 0),
292                x0=>outX0,
293                y1(5 downto 0)=>OUTY32(5 downto 0),
294                y2(5 downto 0)=>OUTY33(5 downto 0),
295                y3(5 downto 0)=>OUTY34(5 downto 0),
296                y4(5 downto 0)=>OUTY11(5 downto 0),
297                y5(5 downto 0)=>OUTY12(5 downto 0),
298                y6(5 downto 0)=>OUTY14(5 downto 0),
299                ucout(5 downto 0)=>OUTU31(5 downto 0),
300                yc(5 downto 0)=>OUTY31(5 downto 0));
301
302     cell32 : neuron1
303        port map (colsel=>cSel(2),
304                control(7 downto 0)=>CTRL(7 downto 0),
305                rowsel=>rSel(1),
306                temp(7 downto 0)=>TEMP(7 downto 0),
307                uc(5 downto 0)=>outW(5 downto 0),
308                u1(5 downto 0)=>OUTU31(5 downto 0),
309                u2(5 downto 0)=>OUTU33(5 downto 0),
310                u3(5 downto 0)=>OUTU34(5 downto 0),
311                u4(5 downto 0)=>OUTU12(5 downto 0),
312                u5(5 downto 0)=>OUTU22(5 downto 0),
313                u6(5 downto 0)=>OUTU42(5 downto 0),
314                x0=>outX0,
315                y1(5 downto 0)=>OUTY31(5 downto 0),
316                y2(5 downto 0)=>OUTY33(5 downto 0),
317                y3(5 downto 0)=>OUTY34(5 downto 0),
318                y4(5 downto 0)=>OUTY12(5 downto 0),
319                y5(5 downto 0)=>OUTY22(5 downto 0),
320                y6(5 downto 0)=>OUTY42(5 downto 0),
321                ucout(5 downto 0)=>OUTU32(5 downto 0),
322                yc(5 downto 0)=>OUTY32(5 downto 0));
323
324     cell33 : neuron1
325        port map (colsel=>cSel(1),
326                control(7 downto 0)=>CTRL(7 downto 0),
327                rowsel=>rSel(1),
328                temp(7 downto 0)=>TEMP(7 downto 0),
329                uc(5 downto 0)=>outW(5 downto 0),
330                u1(5 downto 0)=>OUTU31(5 downto 0),
331                u2(5 downto 0)=>OUTU32(5 downto 0),
332                u3(5 downto 0)=>OUTU34(5 downto 0),
333                u4(5 downto 0)=>OUTU13(5 downto 0),
```

```
                u5(5 downto 0)=>OUTU23(5 downto 0),
                u6(5 downto 0)=>OUTU43(5 downto 0),
                x0=>outX0,
                y1(5 downto 0)=>OUTY31(5 downto 0),
                y2(5 downto 0)=>OUTY32(5 downto 0),
                y3(5 downto 0)=>OUTY34(5 downto 0),
                y4(5 downto 0)=>OUTY13(5 downto 0),
                y5(5 downto 0)=>OUTY23(5 downto 0),
                y6(5 downto 0)=>OUTY43(5 downto 0),
                ucout(5 downto 0)=>OUTU33(5 downto 0),
                yc(5 downto 0)=>OUTY33(5 downto 0));

    cell34 : neuron1
       port map (colsel=>cSel(0),
                control(7 downto 0)=>CTRL(7 downto 0),
                rowsel=>rSel(1),
                temp(7 downto 0)=>TEMP(7 downto 0),
                uc(5 downto 0)=>outW(5 downto 0),
                u1(5 downto 0)=>OUTU31(5 downto 0),
                u2(5 downto 0)=>OUTU32(5 downto 0),
                u3(5 downto 0)=>OUTU33(5 downto 0),
                u4(5 downto 0)=>OUTU14(5 downto 0),
                u5(5 downto 0)=>OUTU24(5 downto 0),
                u6(5 downto 0)=>OUTU44(5 downto 0),
                x0=>outX0,
                y1(5 downto 0)=>OUTY31(5 downto 0),
                y2(5 downto 0)=>OUTY32(5 downto 0),
                y3(5 downto 0)=>OUTY33(5 downto 0),
                y4(5 downto 0)=>OUTY14(5 downto 0),
                y5(5 downto 0)=>OUTY24(5 downto 0),
                y6(5 downto 0)=>OUTY44(5 downto 0),
                ucout(5 downto 0)=>OUTU34(5 downto 0),
                yc(5 downto 0)=>OUTY34(5 downto 0));

    cell41 : neuron1
       port map (colsel=>cSel(3),
                control(7 downto 0)=>CTRL(7 downto 0),
                rowsel=>rSel(0),
                temp(7 downto 0)=>TEMP(7 downto 0),
                uc(5 downto 0)=>outW(5 downto 0),
                u1(5 downto 0)=>OUTU42(5 downto 0),
                u2(5 downto 0)=>OUTU43(5 downto 0),
                u3(5 downto 0)=>OUTU44(5 downto 0),
                u4(5 downto 0)=>OUTU14(5 downto 0),
                u5(5 downto 0)=>OUTU24(5 downto 0),
                u6(5 downto 0)=>OUTU34(5 downto 0),
                x0=>outX0,
                y1(5 downto 0)=>OUTY42(5 downto 0),
                y2(5 downto 0)=>OUTY43(5 downto 0),
                y3(5 downto 0)=>OUTY44(5 downto 0),
                y4(5 downto 0)=>OUTY14(5 downto 0),
                y5(5 downto 0)=>OUTY24(5 downto 0),
                y6(5 downto 0)=>OUTY34(5 downto 0),
                ucout(5 downto 0)=>OUTU41(5 downto 0),
                yc(5 downto 0)=>OUTY41(5 downto 0));

    cell42 : neuron1
       port map (colsel=>cSel(2),
                control(7 downto 0)=>CTRL(7 downto 0),
                rowsel=>rSel(0),
                temp(7 downto 0)=>TEMP(7 downto 0),
                uc(5 downto 0)=>outW(5 downto 0),
                u1(5 downto 0)=>OUTU41(5 downto 0),
                u2(5 downto 0)=>OUTU43(5 downto 0),
                u3(5 downto 0)=>OUTU44(5 downto 0),
                u4(5 downto 0)=>OUTU12(5 downto 0),
                u5(5 downto 0)=>OUTU22(5 downto 0),
                u6(5 downto 0)=>OUTU32(5 downto 0),
                x0=>outX0,
                y1(5 downto 0)=>OUTY41(5 downto 0),
                y2(5 downto 0)=>OUTY43(5 downto 0),
```

```
405                 y3(5 downto 0)=>OUTY44(5 downto 0),
406                 y4(5 downto 0)=>OUTY12(5 downto 0),
407                 y5(5 downto 0)=>OUTY22(5 downto 0),
408                 y6(5 downto 0)=>OUTY32(5 downto 0),
409                 ucout(5 downto 0)=>OUTU42(5 downto 0),
410                 yc(5 downto 0)=>OUTY42(5 downto 0));
411
412     cell43 : neuron1
413        port map (colsel=>cSel(1),
414                  control(7 downto 0)=>CTRL(7 downto 0),
415                  rowsel=>rSel(0),
416                  temp(7 downto 0)=>TEMP(7 downto 0),
417                  uc(5 downto 0)=>outW(5 downto 0),
418                  u1(5 downto 0)=>OUTU41(5 downto 0),
419                  u2(5 downto 0)=>OUTU42(5 downto 0),
420                  u3(5 downto 0)=>OUTU44(5 downto 0),
421                  u4(5 downto 0)=>OUTU13(5 downto 0),
422                  u5(5 downto 0)=>OUTU23(5 downto 0),
423                  u6(5 downto 0)=>OUTU33(5 downto 0),
424                  x0=>outX0,
425                  y1(5 downto 0)=>OUTY41(5 downto 0),
426                  y2(5 downto 0)=>OUTY42(5 downto 0),
427                  y3(5 downto 0)=>OUTY44(5 downto 0),
428                  y4(5 downto 0)=>OUTY13(5 downto 0),
429                  y5(5 downto 0)=>OUTY23(5 downto 0),
430                  y6(5 downto 0)=>OUTY33(5 downto 0),
431                  ucout(5 downto 0)=>OUTU43(5 downto 0),
432                  yc(5 downto 0)=>OUTY43(5 downto 0));
433
434     cell44 : neuron1
435        port map (colsel=>cSel(0),
436                  control(7 downto 0)=>CTRL(7 downto 0),
437                  rowsel=>rSel(0),
438                  temp(7 downto 0)=>TEMP(7 downto 0),
439                  uc(5 downto 0)=>outW(5 downto 0),
440                  u1(5 downto 0)=>OUTU41(5 downto 0),
441                  u2(5 downto 0)=>OUTU42(5 downto 0),
442                  u3(5 downto 0)=>OUTU43(5 downto 0),
443                  u4(5 downto 0)=>OUTU14(5 downto 0),
444                  u5(5 downto 0)=>OUTU24(5 downto 0),
445                  u6(5 downto 0)=>OUTU34(5 downto 0),
446                  x0=>outX0,
447                  y1(5 downto 0)=>OUTY41(5 downto 0),
448                  y2(5 downto 0)=>OUTY42(5 downto 0),
449                  y3(5 downto 0)=>OUTY43(5 downto 0),
450                  y4(5 downto 0)=>OUTY14(5 downto 0),
451                  y5(5 downto 0)=>OUTY24(5 downto 0),
452                  y6(5 downto 0)=>OUTY34(5 downto 0),
453                  ucout(5 downto 0)=>OUTU44(5 downto 0),
454                  yc(5 downto 0)=>OUTY44(5 downto 0));
455
456     CTRL_UNIT : control
457        port map (dClk=>dClk,
458                  dRes=>dRes,
459                  inTemp(7 downto 0)=>inTemp(7 downto 0),
460                  inW(5 downto 0)=>inW(5 downto 0),
461                  inX0=>inX0,
462                  mClk=>mClk,
463                  reset=>reset,
464                  solve=>solve,
465                  tClk=>tClk,
466                  tLoad=>tLoad,
467                  cSel(3 downto 0)=>cSel(3 downto 0),
468                  ctrl(7 downto 0)=>CTRL(7 downto 0),
469                  outW(5 downto 0)=>outW(5 downto 0),
470                  outX0=>outX0,
471                  rSel(3 downto 0)=>rSel(3 downto 0),
472                  temp(7 downto 0)=>TEMP(7 downto 0));
473
474  netout(15) <= OUTY11(5);
475  netout(14) <= OUTY12(5);
```

```
476  netout(13) <= OUTY13(5);
477  netout(12) <= OUTY14(5);
478  netout(11) <= OUTY21(5);
479  netout(10) <= OUTY22(5);
480  netout(9) <= OUTY23(5);
481  netout(8) <= OUTY24(5);
482  netout(7) <= OUTY31(5);
483  netout(6) <= OUTY32(5);
484  netout(5) <= OUTY33(5);
485  netout(4) <= OUTY34(5);
486  netout(3) <= OUTY41(5);
487  netout(2) <= OUTY42(5);
488  netout(1) <= OUTY43(5);
489  netout(0) <= OUTY44(5);
490
491  end BEHAVIORAL;
```

12. Paquete de Componentes para la Neurona Digital (npack.vhd).

```
  1  -- NeuronPack - Package for neuron1, a FPSNET neuron.
  2
  3  library IEEE;
  4  use IEEE.STD_LOGIC_1164.all;
  5
  6  package npack is
  7
  8  component mux2
  9         Generic ( N: integer := 6;
 10                   iosize: integer);
 11         Port (  A, B : in std_logic_vector(N*iosize-1 downto 0);
 12                 S : in std_logic;
 13                 X : out std_logic_vector(N*iosize-1 downto 0));
 14  end component;
 15
 16  component csa6
 17         Generic ( iosize: integer);
 18         Port ( A : in std_logic_vector(iosize-1 downto 0);
 19             B : in std_logic_vector(iosize-1 downto 0);
 20             C : in std_logic_vector(iosize-1 downto 0);
 21             D : in std_logic_vector(iosize-1 downto 0);
 22             E : in std_logic_vector(iosize-1 downto 0);
 23             F : in std_logic_vector(iosize-1 downto 0);
 24                -- Output is iosize+log2(6) =~ iosize+3
 25             X : out std_logic_vector(iosize+2 downto 0));
 26  end component;
 27
 28  component mux5
 29         Generic ( ssize: integer);
 30         Port ( A : in std_logic_vector(ssize-1 downto 0);
 31             B : in std_logic_vector(ssize-1 downto 0);
 32             C : in std_logic_vector(ssize-1 downto 0);
 33             D : in std_logic_vector(ssize-1 downto 0);
 34             E : in std_logic_vector(ssize-1 downto 0);
 35             S : in std_logic_vector(2 downto 0);
 36             X : out std_logic_vector(ssize-1 downto 0));
 37  end component;
 38
 39  component mul
 40         Generic ( ssize: integer;
 41                   tsize: integer);
 42         Port ( A : in std_logic_vector(ssize-1 downto 0);
 43                 B : in std_logic_vector(tsize-1 downto 0);
 44                 X : out std_logic_vector(ssize+tsize-1 downto 0));
 45  end component;
 46
 47  component sum2
 48         Generic ( ssize: integer);
 49         Port ( A : in std_logic_vector(ssize-1 downto 0);
 50             B : in std_logic_vector(ssize-1 downto 0);
```

```
51            X : out std_logic_vector(ssize downto 0));
52  end component;
53
54  component satlin
55          Generic ( ssize: integer;
56                            iosize: integer;
57                            point: integer);
58          Port ( inPort : in std_logic_vector(ssize - 1 downto 0);
59                        outPort : out std_logic_vector(iosize - 1 downto 0));
60  end component;
61
62  component satsum
63          Generic ( ssize: integer);
64          Port ( opA, opB : in std_logic_vector(ssize-1 downto 0);
65                        resX : out std_logic_vector(ssize-1 downto 0));
66  end component;
67
68  end npack;
```

13. Paquete de Componentes para el Módulo de Control (cpack.vhd).

```
 1  -- Control Pack - Package for control, FPSNET network controller
 2
 3  library IEEE;
 4  use IEEE.STD_LOGIC_1164.all;
 5
 6  package cpack is
 7
 8  component seqmem
 9          Generic ( size: integer;
10                            width: integer);
11          Port ( d : in std_logic_vector(width-1 downto 0);
12                        w, clk, reset : in std_logic;
13                        q : out std_logic_vector(width-1 downto 0));
14  end component;
15
16  component crsel is
17          Port ( clk, reset : in std_logic;
18                        cSel, rSel : out std_logic_vector(3 downto 0));
19  end component;
20
21  end cpack;
```

This page has been left blank intentionally…
and we DO NOT pretend to remove it…