Universidad Nacional del Sur

Tesis Doctor en Ingeniería

Estructuras de Procesamiento Neuromórfico de Bajo Consumo para Sistemas de Visión en Internet de las Cosas

Martín Villemur

BAHÍA BLANCA ARGENTINA

2019

Prefacio

Esta tesis se presenta como parte de los requisitos para optar al grado Académico de Doctor en Ingeniería de la Universidad Nacional del Sur y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otra. La misma contiene los resultados obtenidos en investigaciones llevadas a cabo en el ámbito del Instituto de Investigaciones en Ingeniería Eléctrica "Alfredo Desages"(UNS-CONICET) durante el período comprendido entre el 1 de Abril de 2014 y 8 de Marzo de 2019, bajo la dirección del Dr. Pedro Julian, IIIE-CONICET, Departamento de Ingeniería Eléctrica y de Computadoras.

Bahía Blanca, 8 de Marzo de 2019

Ing. Martín Villemur
Departamento de Ingeniería Eléctrica y de Computadoras
UNIVERSIDAD NACIONAL DEL SUR

UNIVERSIDAD NACIONAL DEL SUR
Secretaría General de Posgrado y
Educación Continua

La presente tesis ha sido aprobada el ..../..../...., mereciendo la calificación de .... (.....)

Agradecimientos

A mi familia, amigos y compañeros, que con su tiempo, amor y paciencia me han acompañado en este viaje.

Resumen

Con la reciente popularidad y consecuente aumento en la cantidad de dispositivos electrónicos multimedia interconectados a través de internet, resulta necesario producir sistemas mas eficientes desde el punto de vista energético. Para ello, es fundamental el diseño de dispositivos de bajo consumo con capacidad de procesamiento local que permitan reducir la transferencia de datos a través de la nube. Es por eso que en esta tesis se presenta el desarrollo de arquitecturas digitales energéticamente eficientes para el procesamiento de imágenes.

Los diferentes sistemas se basan en la utilización de estructuras neuronales celulares (CNN) donde el procesamiento es realizado de manera distribuída por un arreglo de celdas idénticas. Cada celda evoluciona conforme a su propio estado y al de sus celdas contiguas utilizando funciones de transferencia lineales a tramos (PWL). Bajo este paradigma, se diseñan y fabrican dos circuitos integrados. El primero, realizado en una tecnología CMOS de 180nm, contiene un arreglo de 56×56 celdas que procesa imágenes binarias. El segundo, fabricado en 55nm, utiliza un vector de 64 celdas para procesar imágenes multibit alojadas en una memoria local.

Posteriormente se presenta un nuevo algoritmo de cómputo utilizando una subclase de funciones lineales a tramos que exhiben cierto tiepo de simetría, lo cual permite expandir el número de celdas de la vecindad y reducir la cantidad de parámetros necesarios para el procesamiento. Se diseñan y se fabrican dos nuevos procesadores de arquitecturas homólogas a las anteriores, donde se utilizan vecindades extendidas de 8 celdas, que implementan la nueva estructura de cálculo PWL simétrica. El primero, que procesa imágenes binarias utilizando un arreglo de 48×48 celdas, fue fabricado en una tecnología de 55nm; mientras que el segundo, de procesamiento multibit, fue fabricado en una tecnología de 130nm.

Finalmente, se muestra el diseño de tres procesadores de alta capacidad de cómputo para el procesamiento no-lineal y lineal de datos, en el marco del desarrollo de un sistema 2.5D muti-chip multi-procesador, fabricado en una tecnología de 55nm, llevado a cabo conjuntamente con la Universidad de Johns Hopkins.

Abstract

With the increasing popularity of multimedia electronic devices interconnected through internet, it is mandatory to build power efficient systems. It is therefore necessary to design low power devices for local processing in order to reduce the data traffic in the cloud. Consequently, this thesis presents the development of highly energy efficient digital architectures for image processing.

The proposed systems are based on cellular neural networks (CNN) structures, which are comprised by an array of dynamical cells with the same behaviour. Each cell computes a multivariate piecewise linear function that involves its own state value and the nearest neighboring cells' state value. Within this paradigm, two integrated circuits were designed and fabricated. The first was designed in a 180nm CMOS technology and implements a 56×56 cell array that process binary images; whereas the second, fabricated in 55nm, processes locally stored grayscale images through a 64-cell vector.

Subsequently, a new algorithm to compute a simplicial piecewise linear function approximation of a symmetric non-linear function is presented, resulting in a reduction of the number of parameter needed for a computation and hence, an increase of the number of elements that make up the neighborhood. Thus, based on the previously proposed architectures, two processors were designed implementing the new symmetric function algorithm scheme in a eight-neighbor configuration. The one that processes binary images was fabricated in 55nm and is comprised by a 48×48 cell array. On the other hand, a vector based chip for muti-bit image processing was taped out in 130nm.

Finally, the design of three high-performance processors for linear and non-linear data processing is shown, in the context of the development of a 2.D multi-module

heterogeneous multi-processor chip, fabricated in 55nm in cooperation with Johns Hopkins University.

Certifico que fueron incluidos los cambios y correcciones sugeridas por los jurados.

Firma del director

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El avance de la tecnología, en particular, la miniaturización de los circuitos integrados ha hecho posible la proliferación masiva de la electrónica. El paradigma de Internet de las cosas (IoT), plantea un horizonte a 10 años [1] donde cada persona contará con aproximadamente 1000 dispositivos inteligentes a su disposición, ya sea de manera directa o indirecta.

Los circuitos integrados para visión, se componen de arreglos de elementos sensibles a la luz, y tienen aplicaciones que incluyen navegación autónoma o semi-autónoma en autos inteligentes [2], procesamiento de datos de imágenes o video de regiones amplias [3], y procesamiento inteligente/aprendizaje en sistemas de Internet de las cosas [4]. El aumento de la capacidad de integración ha hecho posible que se pueda disponer cada vez más de cámaras con mayor resolución. El método convencional de procesamiento, de imágenes o video, se basa en leer un cuadro de la imagen y transmitirlo a otro sistema para su procesamiento. Mas aún, la disponibilidad de cámaras conectadas directamente a la nube, plantea la transmisión de datos desde la cámara hacia la nube. La transmisión de la gran cantidad de datos que origina una cámara de alta definición, a las velocidades requeridas para aplicaciones aún simples como monitoreo plantea un problema de consumo de energía y de saturación del ancho de banda, especialmente en dispositivos móviles, donde ambos factores están limitados.

Una alternativa a estos problemas surge cuando se plantea realizar procesamiento a nivel de píxel. Las estructuras del tipo redes celulares (CNN) permiten procesar

imágenes, de manera similar al procesamiento que realizan las neuronas, donde cada una solo opera compartiendo información con un conjunto limitado de neuronas vecinas. De esta manera, el procesamiento sucede en paralelo, y se reduce la energía necesaria para transmitir información al operar solo con celdas cecanas.

Originalmente el paradigma de CNN fue propuesto por Chua y Yang en el año 1988. Definieron a una red celular neuronal como un arreglo n-dimensional de sistemas dinámicos idénticos, llamados células o celdas, las cuales tienen interacción local en un radio finito r, denominado vecindario [5] [6]. En particular, estas celdas eran analógicas y sus estados eran señales de valores contínuos, cuya función de evolución estaba dado por un circuito analógico no lineal [7]. Los arreglos que posteriormente le siguieron en la literatura fueron en dos dimensiones y los vecindarios eran de radio 1. Particularmente en esta tesis se tratarán CNNs donde la dinámica de evolución del estado está dada por una función lineal a tramos definida en un dominio simplicial [8]. A estas estructuras se las denomina CNN simpliciales o SCNN. Dado que las funciones están definidas por un conjunto de parámetros finitos cuyo número depende de la dimensión del vector de estado de la celda, dichos parámetros pueden ser almacenados en un memoria externa y distribuidos uniformemente al resto de las celdas, lo que le otorga flexibilidad en comparación con la CNN original. Las primeras estructuras del tipo SCNN propuestas fueron de señal mixta [9]. Posteriormente se propusieron estructuras de procesamiento puramente digital, donde la celda además estaba integrada con un sensor de imagen que permitía realizar procesamiento de imágenes distribuido en el plano focal [10] [11] [12] [13].

Si bien se han logrado avances significativos, el estado del arte todavía necesita más trabajo para lograr procesadores con la suficiente capacidad de procesamiento como para sostener aplicaciones inteligentes, con eficiencias de consumo de energía lo suficientemente bajas para ser usadas por dispositivos autónomos durante un período sostenido de tiempo. Esto es cierto tanto para aplicaciones como centros de datos, donde se posee disponibilidad de acceso a energía (centro de Google y chips de Tensor [14]) como para aplicaciones móviles, donde el sistema debe operar con una batería durante al menos un día.

El grueso de las operaciones requeridas por estructuras de tipo neuronales, como redes neuronales tradicionales, redes convolucionales, CNN, y también las requeridas

por operaciones tales como filtrado, correlación o análisis espectral (FFT) se basa en la realización de múltiples acumulaciones de productos. Es por ello, que la optimización de esta operación básica tiene un impacto significativo, más aún, si se tiene en cuenta que cada vez existen más dispositivos con mayor capacidad realizando estas operaciones todo el tiempo. La pregunta que surge entonces es si existen alternativas para implementar estas operaciones, ya sea de manera aproximada o exacta, que permitan reducir la complejidad resultante manteniendo la precisión requerida para la aplicación particular.

En esta tesis se desarrollan arquitecturas de cómputo sobre tecnologías de integración CMOS, basadas en arreglos distribuidos de celdas con conectividad local, que utilizan representaciones lineales a tramos genéricas y/o simétricas. Los objetivos son:

a) Implementar esquemas de procesamiento morfológicas para tareas de procesamiento visual primario.

b) Implementar esquemas de cálculo no-lineal y lineal para redes neuronales y/o redes convolucionales.

c) Implementar sistemas mult-chips (SoC) donde una multiplicidad de cores puedan utilizarse para resolver aplicaciones concretas.

La hipótesis es que la utilización de funciones lineales a tramos, en particular las estructuras (funciones) simétricas, permiten dotar a los arreglos de procesamiento de mayor capacidad de procesamiento con un menor consumo de energía respecto de las estructuras de cómputo convencionales.

La tesis esta organizada de la siguiente forma: el Capítulo 2 introduce nociones generales de procesamiento de imágenes, tales como niveles de abstracción en procesamiento, filtrado lineal y no lineal, operaciones morfológicas en imágenes binarias y en escala de grises, obtenciones de descriptores y clasificación de la información obtenida. Aquí también se introducen conceptos preliminares, en particular se realiza una introducción a las redes celulares no lineales, haciendo foco en los casos particulares de CNN estándar y S-CNN. Se explica en profundidad como funciona un procesador S-CNN, detallando cuales son los pasos a seguir para realizar el cálculo

de un función lineal a tramos dentro de un circuito digital. En el Capítulo 3 se describe el diseño y fabricación de dos sistemas S-CNN parametrizables que implementan funciones no lineales para imágenes binarias (circuito denominado MORPHO1PWL) y en escala de grises (cuyo nombre es MORPHO8PWL). En el caso de MORPHO1PWL su fabricación se realizó en una tecnología CMOS de $180nm$, mientras que el MORPHO8PWL se fabricó en $55nm$ [15]. En el Capítulo 4 se introduce el algoritmo para el cálculo de funciones simétricas, y su impacto en el desarrollo de un procesador digital. Luego se describe el diseño y fabricación de las dos estructuras explicadas en el Capítulo 3, pero utilizando el nuevo método de cómputo. El primer chip, denominado MORPHO1SYM, corresponde a la estructura de procesamiento de imágenes binarias, y fue fabricado en $55nm$ [16]; mientras que el segundo, MORPHO8SYM, corresponde al cómputo de imágenes de hasta 8 bits, su fabricación se realizó en una tecnología CMOS de $130nm$. En ambos capítulos, 3 y 4, se explican las celdas de procesamiento y las estructuras periféricas para su funcionamiento. Se detallan las señales que controlan los circuitos y se brindan ejemplos de configuración y procesamiento. En el Capítulo 5 se presenta un Sistema en Chip (SoC) [17], diseñado en el marco de una colaboración con la Universidad Johns Hopkins. Este sistema se basa en un conjunto de 3 chips dispuestos e interconectados mediante un *"interposer"* de Silicio [18] [19]. El primer chip contiene un arreglo de distintos procesadores de cálculo y una red de comunicación interna de alta velocidad. Los otros dos chips son partes comerciales: uno es una memoria de 8GB y el otro es una FPGA Zynq 7100. En el marco de esta tesis se diseñaron 4 sistemas que se integraron en el primer chip.

La metodología empleada a lo largo de la tesis se basa en la propuesta y simulación de algoritmos de procesamiento, la definición de arquitecturas, su verificación, la síntesis e implementación de CI, su fabricación y posterior testeo.

Los resultados obtenidos en la tesis dieron origen a las siguientes publicaciones: [15] [16] [3] [20] [21].

Capítulo 2

Conceptos preliminares

2.1. Introducción

En este capítulo se introducen conceptos básicos que se tuvieron en consideración y utilizaron para el diseño y desarrollo de las estructuras de procesamiento presentadas en la tesis. La primera sección define que es una imagen digital y presenta las distintas etapas de procesamiento que se le realiza para la extracción de información. Se hace un fuerte énfasis en las etapas de medio y medio-alto nivel, comúnmente conocidas como *Procesamiento y Análisis de imágenes*. Se listan los distintos tipos de operaciones que se llevan a cabo tales como operaciones aritméticas binarias y ordinarias dentro de la imagen y entre ellas, operaciones por histogramas, filtrados lineales y no-lineales, entre otros. Posteriormente, se presentan algoritmos de segmentación y extracción de descriptores visuales, los cuales tienen el fin de obtener y analizar las figuras de interés en una imagen. Por ultimo, se hace una breve introducción a algoritmos y sistemas de procesamiento para el reconocimiento y clasificación de objetos dentro de un sistema de imágenes, y se se presenta, en especial, un tipo de arquitectura denominada *Redes neuronales*. En la segunda sección se describe más detalladamente los sistemas de procesamiento de la información denominadas *Redes celulares no lineales*, las cuales prestan las bases para las arquitecturas planteadas en el trabajo de tesis. Aquí se hace principal hincapié en las *Redes celulares simpliciales* digitales, las cuales permiten la implementación en silicio de sistemas eficientes capaces de realizar procesamiento no-lineal no-lineal de imágenes. Para finalizar el

capítulo, se presenta una breve revisión de distintos circuitos integrados presentes en la literatura distinguidos por su eficiencia a la hora de realizar procesamiento de imágenes en los distintos niveles, utilizando arquitecturas de procesamiento paralelos.

2.2. Procesamiento y análisis de imágenes

El campo de las imágenes digitales es un área tecnológica multidisciplinaria que utiliza conocimientos de distintas otras para producir sistemas utilizados en diferentes áreas de las actividades humanas, como la medicina, biología, ingeniería, automatización industrial, seguridad. La tecnología de imágenes digitales incluye todos los procesos que van desde la adquisición y digitalización de una imagen, su pos-procesamiento, hasta el análisis de la misma, con el fin de realizar una toma de decisiones en base a la información cuantitativa y cualitativa extraída. Un ejemplo típico es el de clasificación y control de transito utilizando arreglos de cámaras de vigilancia aérea [22], en donde los vehículos captados por las imágenes obtenidas son clasificados en base a descriptores (área, perímetro, sentido de movimiento, velocidad, entre otros). La información de entrada en dichos sistemas es una imagen compuesta tomados a través de varias videocámaras, y la información de salida es una lista de posiciones y características de los vehículos detectados. Sin embargo, todo el proceso envuelve tres etapas bien definidas desde la teoría (Fig.2.1):

1. La primer etapa es la *Adquisición de imagen*, la cual incluye la detección de una imagen y su posterior digitalización. Se utilza un dispositivo de estado solido transductor de luz, que convierte intensidad de luz en una variable eléctrica, la cual, a través de un conversor analogco-digital (ADC), se convierte a una imagen digital. Posteriormente, se realiza un procesamiento de corrección y optimización básica, en el cual se ajusta las propiedades físicas de una imagen, como el brillo, contraste, intensidad y corrección de colores (*"dark-frame subtraction"*, *"fixed-pattern noise"*).

2. La segunda etapa es denominada estrictamente *Procesamiento de imagen*, lo cual implica la restauración de la imagen y optimización para su posterior análisis. Aquí se corrigen fallos cometidos durante su captura (mal foco, corrección de errores ópticos, contraste, intensidad, etc.), se realzan los elementos de la imagen que no pueden ser fácilmente distinguibles pero que son de interés, se substrae ruido de censado, entre otras cosas.

3. La última etapa es la conocida como el *Análisis de imagen*, en donde se extrae

la información de útil de la imagen para posterior evaluación y utilización de la misma dentro de diferentes tareas. La principal diferencia con la etapa anterior, de procesamiento de la imagen, es que los resultados del análisis normalmente son numericos. Esto incluye características topográficas (numero de objetos en una imagen y distancia entre ellos), geométricas (área, perímetro, factor de forma, redondez, etc), densidad y de texturas de los objetos identificados en la imagen [23]. Sin embargo, también el resultado puede ser otra imagen en donde se identifiquen otras estructuras como bordes delimitatorios y regiones.

De esta manera, en esta sección se introducen algunas de las operaciones y filtrados que se realizan dentro del procesamiento y análisis de imagen, comenzando por una breve definición de una *imagen digital*. Luego se describen operaciones lineales y no-lineales comunes y mayormente utilizadas dentro del procesamiento, seguido por técnicas de segmentación en las imágenes, para por último, finalizar con metodologías y algoritmos de identificación de objetos de interés.

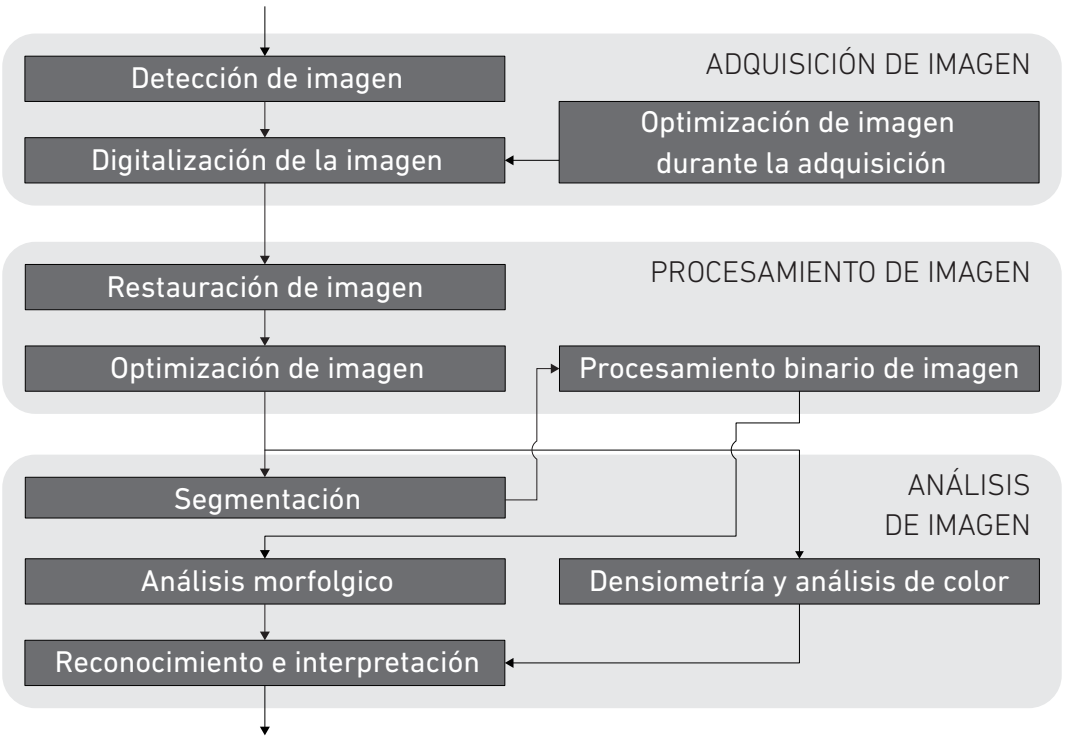


Figura 2.1: Secuencia de funciones típica en el procesamiento de imágenes digital.

Definición de imagen digital y sus propiedades

Previo comenzar a hablar de la manipulación de las imágenes digitales es necesario conocer su definición. Una imagen se define como una función en dos dimensiones, $f(x, y)$, donde x e y son espacios de coordenadas, y la amplitud de f en cualquier par de coordenadas (x, y) es llamado intensidad o nivel de gris de una imagen en ese punto. Si se tiene una imagen multiespectral, $f(x, y)$ es un vector, donde cada componente indica el brillo en el punto (x, y) en cada banda espectral correspondiente.

Cuando los valores de x, y y de las amplitudes asociadas son discretos, entonces se denomina la imagen como *imagen digital*. La misma se compone de arreglo bidimensional finito de N filas y M columnas de elementos denominados píxeles. El valor asignado a cada pixel ubicado en las coordenadas $f[x, y]$, donde $x \in 0, 1, ..., M - 1$ e $y \in 0, 1, ..., N - 1$, se lo re define como $f[x, y]$, el cual resulta, en una imagen en escala de grises, de la intensidad promedio en el píxel redondeado al valor entero mas cercano. El proceso de representar la amplitud de una señal con un valor entero de L diferentes niveles es comúnmente denominado cuantización. Existen valores estándar de los parámetros introducidos previamente. En general es frecuente encontrar a M y N como en valores potencias de dos 2^K, donde K suele ser $6, 7, 8, 9, 10$, entre otros, lo cual facilita la utilización de ciertos algoritmos de procesamiento tales como la transformada rápida de Fourier. Así mismo, los niveles de grises L son usualmente definidos como potencias de dos $L = 2^q$, donde q es la numero de bits en la representación binaria de los niveles de intensidad. Cuando $q > 1$, se habla de imagen en escala de grises, mientras que cuando $q = 1$, se refiere como imagen binaria.

2.2.1. Operaciones en imágenes

Existen una gran variedad de clasificaciones y caracterizaciones de las operaciones sobre imágenes. El objetivo de estas distinciones es entender que tipo de resultados son de esperarse antes cierto tipo de operación, o la complejidad asociada a la misma.

Existen tres tipos de operaciones que pueden ser aplicadas a una imagen digital asociada a su dimensionalidad. La primera se la denomina *puntual* (Fig.2.3(a)), don-

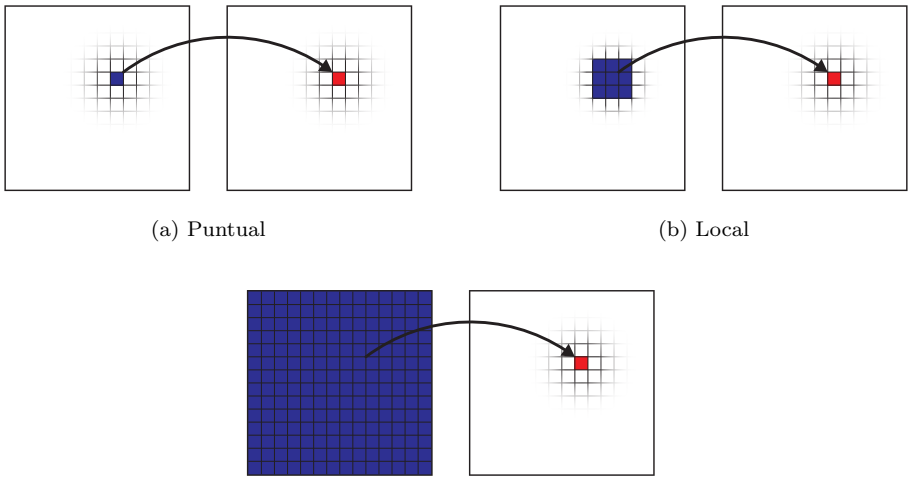
Figura 2.2: Variaciones de una imagen al variar la cantidad de píxeles en el eje vertical (256×256, 128×128 y (64×64), y la cantidad de bits de precisión en el eje horizontal (8, 6 y 3).

de cada píxel en una de salida en una coordenada dada, resultante de una operación, depende única y exclusivamente del píxel de entrada ubicado en la misma posición relativa dentro de la imagen.

El segundo tipo de operación se la conoce como *operación local* (Fig.2.3(b)), en el cual el resultado depende del no solo del píxel de entrada en su misma coordenada, sino que también de su *vecindario*. En este caso resulta importante entender como las imágenes son muestreadas y como este proceso influye en la operatoria por vecindario. A grandes rasgos existen dos tipos, el *muestreo rectangular*, en donde

la imagen es muestreada utilizando una grilla rectangular ortogonal, o el muestreo hexagonal, que como el término lo indica se utiliza una grilla hexagonal. En el presente trabajo solo se utilizaran vecindarios rectangulares, que comunmente están compuestos por 4 u 8 vecinos más el valor central. Dentro de este tipo de operación se puede nombrar el filtrado espacial lineal y no lineal.

Por último, se denominan *operaciones globales* (Fig.2.3(c)) a aquellas operaciones en donde el valor de salida en una coordenada especifica depende de todos los valores de la imagen de entrada, por ejemplo, la Transformada de Fourier de imágenes, o el proceso de generación de un histograma de imagen.



(a) Puntual (b) Local

(c) Global

Figura 2.3: Ilustración de los distintos tipos de operaciones sobre una imagen.

Operaciones matemáticas

Se puede dividir entre operaciones aritméticas binarias y operaciones aritméticas ordinarias. En el caso binario, los valores de intensidad son dos, "0" y "1", en el caso ordinario los valores de intensidad originales son $L = 2^q$, pero de a cuerdo a la operación se pueden generar más niveles.

Las operaciones basadas en aritmética binaria o Booleana son de tipo *puntual* y sirven como base para las operaciones morfológicas, descritas más adelante. Entre

ellas se pueden nombrar:

$$\begin{aligned} NOT &\quad c = \bar{a} \\ OR &\quad c = a + b \\ AND &\quad c = a \bullet b \\ XOR &\quad c = a \oplus b = a \bullet \bar{b} + \bar{a} \bullet b \\ SUB &\quad c = a \backslash b = a - b = a \bullet \bar{b} \end{aligned} \tag{2.1}$$

donde cada operación se hace píxel a píxel, por ejemplo, $c[m,n] = a[m,n] \bullet \bar{b}[m,n],\ \forall m,n.$

En el caso de las operaciones para imágenes con escala de grises se encuentran:

$$\begin{aligned} SUMA &\quad c = a + b \\ RESTA &\quad c = a - b \\ MULT &\quad c = a \bullet b \\ DIVISION &\quad c = a/b \\ INVERSION &\quad c = (2^q - 1) - a \\ OTRAS &\quad c = F(a) \end{aligned} \tag{2.2}$$

Operaciones basada en histograma

Una clase importante de operaciones puntuales son aquellas basadas en la utilización del histograma de una imagen or histograma por regiones. Previamente es necesario definir que es un histograma dentro del procesamiento de imágenes. El histograma de una imagen refriere a un histograma donde se muestra la cantidad píxeles existentes dentro de una imagen que poseen el mismo valor de intensidad. Para una imagen en escala de grises de 8 bits, donde hay 256 posibles intensidades, su histograma muestra la distribución de píxeles a los largo de esa escala de valores posibles.

Un ejemplo de operación basada en histograma es el denominado *estiramiento de contraste*. Frecuentemente, cuando una imagen es tomada, no se hace use todo el rango dinámico de un sensor, por lo que es necesario estirar el histograma a través del rango dinámico disponible. Si la imagen posee potencialmente un rango de brillo

que va desde 0 a $(2^q - 1)$, donde q es la precisión de bits por píxel, la transformación para cada píxel de la imagen A en la coordenada $[m, n]$ es:

$$b[m,n] = \begin{cases} 0 & a[m,n] \leq p_{low}\,\% \\ (2^q - 1)\,\frac{a[m,n] - p_{low}\,\%}{p_{high}\,\% - p_{low}\,\%} & \leq p_{low}\,\% \leq a[m,n] \leq p_{high}\,\% \\ (2^q - 1) & a[m,n] \geq p_{high}\,\% \end{cases} \tag{2.3}$$

en donde los parámetros $p_{low}\,\%$ y $p_{high}\,\%$ son valores dados por el usuario para ajustar el algoritmo y desestimar valores atípicos del histograma.

La segunda operación más importante se realiza cuando se desea "normalizar" el histograma para poder obtener datos de la imagen, como por ejemplo, las distintas texturas que se encuentran presentes en la misma. La técnica de normalización más comúnmente utilizada es la *ecualización de histograma*, donde se aplica una función $b = f(a)$ que idealmente genera una histograma con valores constantes para todos los niveles de brillo [24]. Dicha operación se puede observar en la Fig.2.4 donde se ilustra el antes y después de la imagen y sus respectivos histogramas superpuestos.

Filtrado lineal

El filtrado lineal es la modificación de un píxel en función de los píxeles vecinos utilizando un operador lineal a través de una operación matemática definida como convolución. La integral de convolución en dos dimensiones esta dada por la ecuación:

$$g(x,y) = f * h = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x - m, y - n)\, h(m,n)\, dx\, dy \tag{2.4}$$

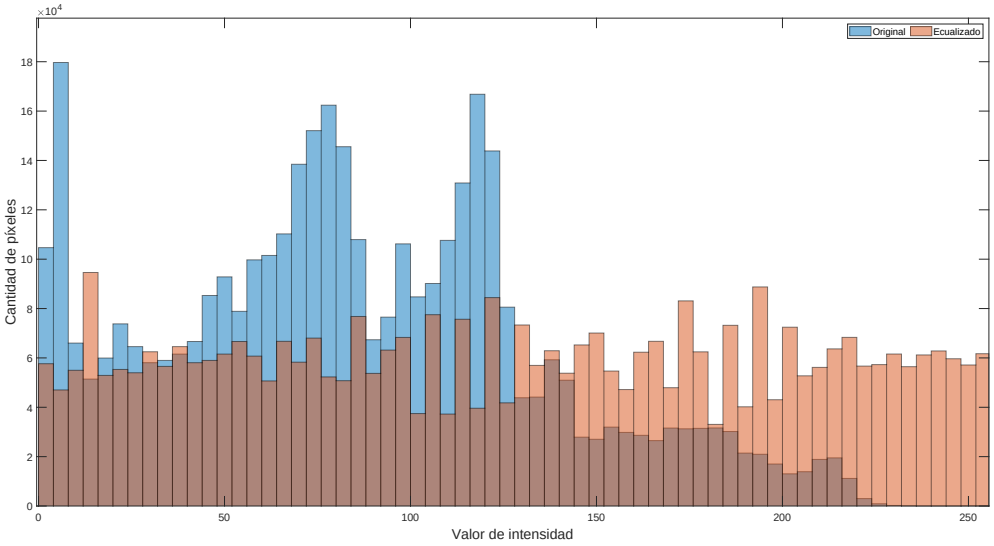
donde la salida g de la "convolución" de una señal de entrada f con la función h. Sin embargo, dado que las imágenes digitales se encuentran en el dominio discreto es necesario reemplazar la integrar por el operador de sumatoria:

$$g(x,y) = f * h = \sum_m \sum_n f(x - m, y - n)\, h(m,n) \tag{2.5}$$

donde f es la imagen de entrada, h es el filtro o *kernel* y de g es la imagen de salida. La sumatoria se da unicamente en el área donde f y h se solapan. Dependiendo el

(a) Original

(b) Ecualizado



(c) Histograma de las dos imágenes.

Figura 2.4: Ejemplo de ecualización de histograma.

tamaño, sus valores y su morfología, los filtros digitales se pueden dividir en:

- Filtros pasa bajos: producen un suavizado en la imagen y reducen los componentes espaciales de alta frecuencia del ruido. Los más comunes son los Gausianos y promediadores [25].

- Filtros pasa altos: realzan las características de bajo contraste cuando son superpuestos sobre un fondo muy obscuro o muy claro. También se utiliza para detectar los bordes dentro de una imagen. En esta categoría se pueden encontrar el filtro de Roberts, Prewitt, Sobel, y los filtros Laplacianos [26], entre otros.

- Filtros pasa banda y de gran énfasis: tienen un efecto más sutil que los anteriores, generalmente tienden a agudizar los bordes y realzar pequeños detalles.

En la Fig2.5(b) se ilustra el filtrado por convolución de una imagen (Fig.2.5(b)) aplicando un filtro pasa bajo promediador de 5×5 pixeles; mientras que en la Fig.2.5(c) se puede observar la misma imagen de entrada pero si se le aplica un filtro pasa altos de Sobel para la detección de bordes horizontales.

(a) Original (b) Filtro promediador (c) Filtro Sobel Horizontal

Figura 2.5: Ejemplo de filtrado lineal.

Filtrado no-lineal

En el caso de los filtros no lineales, el operador es no lineal; esto quiere decir que no cumple la propiedad de superposición, la de proporcionalidad o ninguna de las dos. La salida no será igual por tanto a la convolución de la señal de entrada y la respuesta impulsiva del filtro. Dentro de los filtros no lineales se pueden listar los de *orden estadísticos*, *morfológicos*, *homomórficos*, *polinomiales* y *adaptivos*, entre otros [27] [28]. A continuación se detallarán solo los de orden estadísticos y los morfológicos ya que son de interés para su futura implementación en silicio.

Filtro de orden estadístico Son filtros en el dominio espacio que funcionan ordenando los valores de la vecindad de cada punto de menor a mayor, y de dicha lista ordenada se obtiene un valor. Estos filtros han sido diseñados para conseguir robustez, adaptabilidad a las distribuciones de probabilidad del ruido, preservar la información de los contornos y conservar los detalles de la imagen. Sus mayores ventajas son la simplicidad y rapidez para su cálculo. Además, tienen un buen comportamiento en presencia de ruido blanco Gaussiano y ruido aditivo.

Dada una vecindad de n elementos $x_1, x_2, ..., x_n$, el orden estadístico se puede obtener ordenando el x_i de manera ascendente, lo que produce $x_{(i)}$ tal que:

$$x_{(1)} < x_{(2)} < ... < x_{(n-1)} < x_{(n)} \tag{2.6}$$

Un filtro de orden estadístico (FOE) es un estimador $F(x_i)$ de la media de x que una la combinación lineal del orden estadístico:

$$F(x_i) = \sum_i a_i x_{(i)} \tag{2.7}$$

El principal filtro de orden estadístico es el filtro de mediana, incluso otros filtros dentro de este grupo son modificaciones o extensiones de éste. A continuación, se presentan los distintos tipos:

- El *filtro de mediana* de n elementos $x_i, i = 1, ..., n$ se denota por $med(x_i)$ y está dado por:

$$med(x_i) = \begin{cases} x_{v+1} & n = 2v + 1 \\ \frac{1}{2}\left(x_{(v)} + x_{(v+1)}\right) & n = 2v \end{cases} \tag{2.8}$$

  donde x_i denota el estadístico de orden i. Son útiles para la eliminación de ruido aditivo y ruido impulsivo, sin embargo puede destruir detalles de alta frecuencia y generar sectores de "manchas" dentro de la imagen (Fig.2.6).

(a) Imagen con ruido "sal y pimienta" (b) Filtrado usando un filtro de mediana

Figura 2.6: Ejemplo de filtrado no-lineal.

- Los *filtros híbridos de mediana* combinan un filtro de mediana tradicional con

filtros lineales de respuesta impusliva FIR o IIR. Por ejemplo, en [29], dada una vecindad de 3 elementos, se realiza la mediana entre: la media de los elementos axiales de la vecindad, la media de los elementos diagonales de la misma vecindad, y la mediana de todos los elementos. Esto permite realizar el filtrado del ruido sin perder los detalles espaciales de la imagen. La expresión generalizada es:

$$y_i = med\left[\Phi_1(x_i), ..., \Phi_M(x_i)\right] \tag{2.9}$$

- En los *filtros de orden clasificado* u *orden r*, el parámetro r da el orden del filtro. Si $r$es de valor bajo ($r < v + 1$) se denominan de orden bajo, y logran reproducir una región oscura haciendola más oscura y menos estructurada, donde se suprimen detalles de alta frecuencia brillantes. Si r es de valor alto ($r > v + 1$) se denominan de orden alto, y alargan regiones brillantes. Un filtro clasificado de orden r se puede calcular con el estimador $F(x_i)$ donde sus coeficientes son:

$$a_i = \begin{cases} 1 & i = r \\ 0 & i \neq r \end{cases} \tag{2.10}$$

- Uniendo la media, mediana y los de orden r, se pueden formar los *filtros de media de estado* α, los cuales tienen la siguiente forma:

$$\overline{x}_\alpha = \frac{1}{\alpha} \sum_{i=(n-\alpha+1)/2}^{(n+\alpha+1)/2} x_{(i)} \tag{2.11}$$

donde n e sla cantidad de elementos del vecindario, $x_{(i)}, j = 1, ..., n$ son los estadísticos de ordenes de los elementos x_i, $\alpha = 1, ..., n$ e sel orden , y $\overline{x}_\alpha$ es la salida de la *media de estado* α.

Operaciones, filtrado y procesamiento morfológico El termino *"morfología"* se designa en general al estudio de la forma o formas que presentan los objetos que estudia cualquier ciencia y las variantes que estas formas pueden tener. En particular, dentro del procesamiento de imagen, la morfología es un conjunto de operaciones que permite la modificación de las imágenes basadas en las formas que representan [30].

Si se tiene una imagen binaria como se ilustra en Fig.2.7, se define como *objeto* al

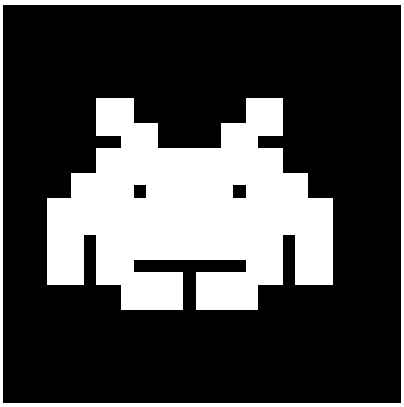


Figura 2.7: Imagen binaria con objeto en blanco.

conjunto de coordenadas o pixeles que comparten una propiedad en común, en ese caso aquellos que son blancos. Así mismo, se define *fondo* como el complemento del *objeto*,o ,para el caso del ejemplo, como $\mathbf{A}^c$.Las operaciones fundamentales asociadas a un objeto son el conjunto de operaciones *unión*, *intersección*, y *complemento* $\cup, \cap, ^c$, más la *traslación* [31]. A partir de estas se definen las operaciones de Minkowski que dan origen las operaciones morfológicas de *dilatación* y *erosión*, detalladas a continuación.

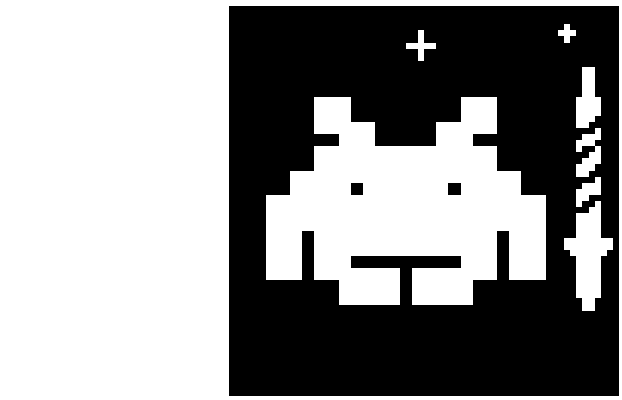
- **Dilatación y erosión:** La dilatación es la primera de las dos operaciones básicas de la morfología matemática y tiene como finalidad aumentar los píxeles en los bordes de los objetos en una imagen X. La dilatación de un conjunto $A \subseteq X$ por un conjunto $B \subseteq X$ esta definida por la suma de Minkowski [32]:

$$\begin{aligned} A \oplus B &= \{a + b \mid a \in A \wedge b \in B\} \\ &= \bigcup_{b \in B} (A + b) \end{aligned} \tag{2.12}$$

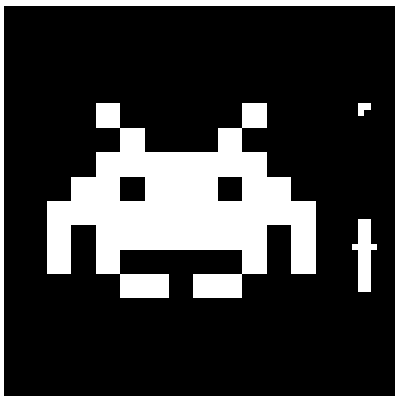
  Por otro lado, la erosión, es la segunda de las operaciones básicas de la morfología matemática y, como bien refiere su nombre, este operador tiene como objetivo erosionar los bordes de los píxeles que pertenecen al objeto en una imagen binaria. De esta manera el área del objeto se reduce, y el área del fondo incrementa su valor. La erosión de un conjunto $A \subseteq X$ por un conjunto

$B \subseteq X$, esta definida por la resta de Minkowski [32]:

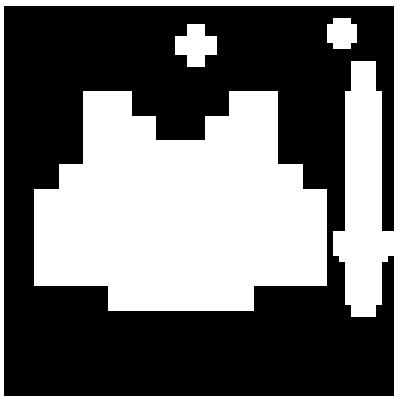
$$\begin{aligned} A \ominus B &= \{x \in X \mid x + b \in A, \forall b \in B\} \\ &= \bigcap_{b \in B} (A + b) \end{aligned} \tag{2.13}$$



(a) Original



(b) Erosión

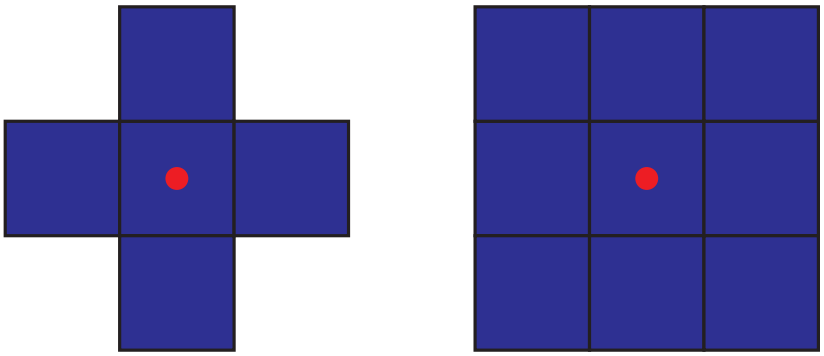


(c) Dilatación

Figura 2.8: Ejemplo de erosión y dilatación en una imagen binaria utilizando un EE de 3×3.

Estas dos operaciones son ilustradas en la Fig.2.8 para los objetos definidos en la Fig.2.8(a). Mientras cualquiera de los dos conjuntos A o B pueden ser considerados como un "objeto imagen", A es considerado la imagen de entrada y B es denominado *elemento estructurante* o EE. El elemento estructurante es a la matemática morfológica lo que el *kernel* de convolución es al filtrado lineal. Los dos elementos estructurantes más comunes (dado una grilla cartesiana), son los conjuntos de 4 y 8, ilustrados en la Fig.2.9.

Para el cálculo practico de la dilatación, a cada píxel de la imagen de entrada se le debe superponer el elemento estructurante B de manera tal que el origen



(a) Elemento estructurante de vecindad 4.

(b) Elemento estructurante de vecindad 8.

Figura 2.9: Ejemplos de elementos estructurante más utilizados. Con un punto rojo se muestra el centro del EE.

del EE coincida con su posición. Si al menos un píxel del elemento estructurante coincide con un píxel perteneciente al objeto A, entonces el píxel de salida, correspondiente a la posición del de entrada, toma el valor de A. Por su parte, para realizar una erosión de una imagen binaria de entrada por un elemento estructurante, se debe iterar sobre cada píxel perteneciente al objeto de la siguiente manera. Por cada píxel de entrada se superpone el elemento estructurante de manera tal que su origen coincida con la coordenada del píxel del entrada. Si alguno de los píxeles del elemento estructurante se solapa con un píxel de fondo de la imagen de entrada, entonces el resultado para ese píxel entrada es el valor de fondo. Caso contrario, se mantiene como píxel perteneciente a un objeto.

- **Apertura y clausura:** En la práctica rara vez se opere con dilataciones y erosiones de manera asilada. Aprovechando que ambas operaciones no son inversas entres, a partir de de sus implementaciones secuenciales se generan dos nuevas operaciones fundamentales de la morfología bajo el nombre de *apertura* y *clausura*. La *apertura* es la dilatación de la erosión de una conjunto A por un elemento estructurante B y se denota como:

$$A \circ B = (A \ominus B) \oplus B, \tag{2.14}$$

donde $\ominus$ y $\oplus$ son los operadores de erosión y dilatación respectivamente. Tiene

como utilidad, entre otras cosas, eliminar el ruido sin deformar ni alterar el tamaño de los objetos imagen más grandes dependiendo de las características del EE. En la Fig.2.10 se puede observar el efecto de la apertura en una imagen binaria utilizando un elemento estructurante de 3×3; la cantidad de píxeles pertenecientes al fondo aumenta sin "perder" el objeto de interés, y se suavizan los bordes del mismo. Sin embargo, puede suceder que las zonas del objeto de menor "anchura" que el EE se destruyan. Por otro lado, la *clausura* se realiza operando con una dilatación seguido de una erosión:

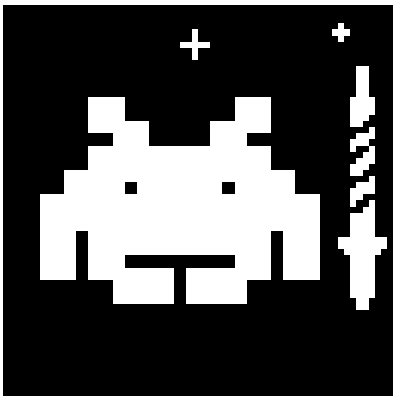
$$A \bullet B = (A \oplus B) \ominus B \tag{2.15}$$

En contrapartida, la *clausura* tiene por finalidad rellenar los huecos de un objeto, o realizar conexiones con objetos próximos entre si. En la Fig.2.10(c) se puede observar el efecto de la clausura utilizando un elemento estructurante de 4 elementos conectados. Se puede ver que los huecos menores al tamaño del EE se han cerrado, y como el objeto se conectó con otro que se encontraba próximo. Para realizar el rellenado completo de ciertas regiones existen otro algoritmos y transformaciones ejemplificadas en [26].

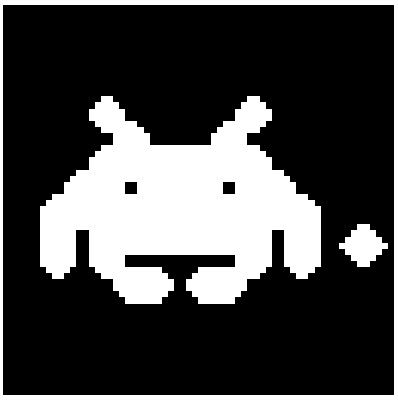
- **Transformada "Hit or miss":** La operación de "hit or miss" ("acierta o falla" en ingles) es utilizada para detectar cierto "patrón" dentro de una imagen binaria implementando la *erosión* con un conjunto de elementos estructurantes disjuntos. Dada una imagen A y dos elementos estructurantes B_1 y B_2, tal que $B_1 \cap B_2 = \emptyset$ se denota:

$$A \circledast B = (A \ominus B_1) \cap (A^c \ominus B_2)\,, \tag{2.16}$$

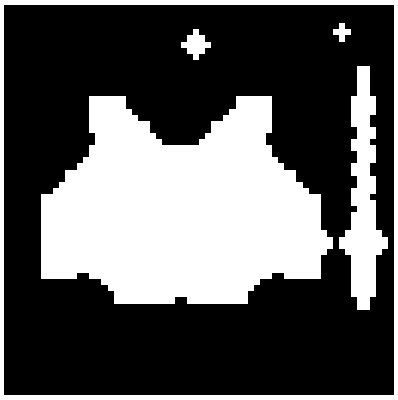
donde $B = (B_1, B_2)$, B_1 es el conjunto de elementos de B asociados a un objeto, y B_2 es el conjunto de elementos de B que corresponden al fondo. En otras palabras, se puede pensar a $A \circledast B$ como todos los puntos que simultáneamente el EE B_1 "acertó" en A y el EE B_2 "acertó" en A^c. De manera directa, se utiliza para detectar puntos aislados, píxeles finales de un objeto y sus los píxeles de contorno.



(a) Original



(b) Apertura



(c) Clausura

Figura 2.10: Ejemplo de apertura y clausura en una imagen binaria utilizando un EE de 4 elementos conectados.

- **Adelgazamiento y ensanchamiento** Ya definida la transformación "hit and miss", a partir de ella se desprende la transformación *adelgazamiento*, la cual tiene como objetivo "afinar" el objeto. El *adelgazamiento* de una imagen A por un elemento estructurante B:

$$Thin(A, B) = A \cap (A \circledast B) \tag{2.17}$$

A diferencia de la erosión, la transformada de *adelgazamiento* no permite que desaparezca el objeto. De manera dual, se define la transformación de *ensanchamiento* de una imagen A con un EE B como :

$$Thick(A, B) = A \cup (A \circledast B)^c \tag{2.18}$$

Tiene como objetivo hacer crecer regiones pertenecientes a objetos, sin que que se produzcan clausuras dentro del mismo objeto ni uniones con otros. A pesar

de que los efectos de estas transformaciones han sido definidas para cuando se realizan una sola vez, normalmente se aplican repetidas veces con grandes conjuntos de EE, hasta que el resultado imagen converge, como en el caso de la *esqueletonización* [33].

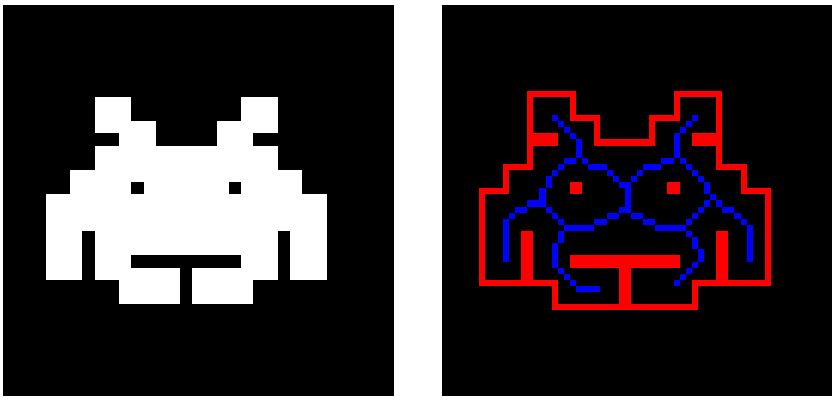
- **Esqueletonización:** Informalmente, el esqueleto de un objeto se puede definir como una linea que represente un objeto y que tenga las siguientes caracterizaras:

  - Un píxel de ancho.
  - Atraviese el punto "medio" del objeto.
  - Mantenga la topología del objeto.

  Por lo tanto, *esqueletonización* es el procesao por el cual se reduce las regiones de un objeto dentro de una imagen binaria hasta llegar a su esqueleto, el cual preserve su extensión y conectividad. En la Fig.2.11 se muestra una imagen objeto con su esqueleto y su borde. Como se menciono anteriormente, uno de los tanto algoritmos utilizados para lograr un esqueleto es aplicando sucesivamente la transformacion de *adelgazamiento*. Sin embargo, el resultado puede contener irregularidadaes que no representen fielmente el objeto madre, por lo cual existe la transformación de *podado* [26], la cual permite cortar las ramificaciones no deseadas del esqueleto. Si se sigue cortando el esqueleto, se puede llegar a un punto, el cual se puede tomar como el centro de gravedad del objeto. A este proceso se lo llama *granulación*, con el cual se puede calcula la posición del objeto dentro de la imagen.

- **Morfología en escala de grises:** Las técnicas de filtrado morfológico pueden ser extendidas a imágenes en escalas de grises. En este caso el elemento estructurante B puede ser plano o también tener valores no binarios. En este caso, se reemplaza la utilización de los operadores de Minkowski por los de máximo y mínimo, lo que supone pasar de operar con conjuntos a operar con funciones. Se define la dilatación en escala de grises como:

$$A \oplus_G B = \max_{[j,k] \in B} \{a[m+j, n+] + b[j,k]\} \tag{2.19}$$



(a) Original (b) Esqueleto y contorno

Figura 2.11: Resultado de esqueletonización de un objeto (en azul) con su respectivo contorno (en rojo).

Es decir, para una salida coordenada de salida $[m, n]$ dada, el elemento estructurante se suma a una versión desplazada de la imagen, y se calcula el máximo dentro del dominio $J \times K$ de B como resultado. Para el caso de de la erosión se utiliza el mínimo, y el EE se resta:

$$A \ominus_G B = \min_{[j,k] \in B} \{a[m + j, n+] - b[j, k]\} \tag{2.20}$$

En el caso de que los valores de los elementos de B sean cero, las expresiones anterior se reducen a:

$$A \oplus_G B = \max_B(A) \tag{2.21}$$

$$A \ominus_G B = \min_B(A) \tag{2.22}$$

para la dilatación y erosión en imágenes en escala de grises, respectivamente. La dilatación si el EE posee elementos positivos genera una imagen de salida mas brillante que la entrada, y a nivel general reduce o elimina detalles oscuros, dependiendo de la característica del EE. En cambio, la erosión realza los oscuros y genera que los detalles con brillo reduzcan su área o desaparezcan. En la Fig.2.12 se observa el efecto de la dilatación y erosión de un elemento estructurante de plano sobre una imagen. El resto de las operaciones, como la apertura y clausura se desprenden naturalmente de las anteriores sin modificaciones.

(a) Original

(b) Erosión

(c) Dilatación

(d) Apertura

(e) Clausura

Figura 2.12: Ejemplo de erosión, dilatación, apertura y clausura, para una imagen en escala de grises utilizando un elemento estructurante plano cuadrado de 5×5.

2.2.2. Segmentación

En el análisis de objetos dentro de imágenes es esencial poder distinguir entre el objeto de interés y el "resto", o lo que comúnmente se denomina fondo. Existen varias técnicas utilizadas, bajo el nombre de *tecnicas de segmentación*, algunas de las cuales se detallan en esta sección. Cabe aclarar que no existe una técnica aplicable que universalmente aceptada que sirva para todos los casos, y que ninguna técnica

de segmentación es perfecta en cuanto a sus resultados.

Segmentación por umbral o binarización

En este caso un parámetro θ, llamado *umbral de brillo*, es utilizado y aplicado a un píxel imagen $a[m,n]$ de la siguiente manera:

$$b[m,n] = \begin{cases} 1 & a[m,n] > \theta \\ 0 & a[m,n] \leq \theta \end{cases} \tag{2.23}$$

suponiendo que el objeto es más claro que el fondo. La salida es etiquetada como "objeto" o "fondo", la cual puede ser representada por una variable booleana "1" o "0". Queda definir el valor de θ óptimo para cada imagen. La manera más simple y menos precisa es utilizar un *umbral fijo* independientemente de la información de la imagen, lo cual puede ser útil para imágenes de gran contraste entre objeto y fondo, donde el último sea homogéneo. El segundo grupo de métodos deriva de la información que brinda el histograma de una imagen. Existen varias técnicas que permiten elegir automáticamente el umbral a partir de un histograma a escala de grises, $\{h[b]|b = 0, 1, \ldots, 2^q–1\}$. Previamente a cualquier algoritmo, se debe realizar un *suavizado de histograma*, el cual remueve las pequeñas fluctuaciones que pueda tener el mismo, sin desviar las posiciones de los picos significativos:

$$h_{suavizado}[b] = \frac{1}{W} \sum_{w=-(W-1)/2}^{(W-1)/2} h[b-w], \tag{2.24}$$

donde el coeficiente W es un número impar que típicamente toma valores ente 3 y 5. Entre los algoritmos más utilizados se encuentran el *algoritmo de isodata* [34] y el *algoritmo de triangulo* [35].

Segmentación por búsqueda de bordes

Una alternativa a la técnica de segmentación por umbral es la de encontrar aquellos píxeles que pertenezcan a los bordes de los objetos de interés. La utilización de procedimientos basados en *busqueda de bordes*, tiene como objetivo encontrar procedimientos que produzcan contorno cerrados alrededor de objeto de interés, y

que a su vez mantengan su morfología. La primer técnica utilizada es aplicar un filtro lineal pasa-altos, como el de Sobel, y a eso aplicarle una binarización, como por ejemplo de *isodata*. A este método se lo conoce como *basado en gradientes*, y consiste en realizar simplemente una detección de bordes, para luego segmentar.

La segunda técnica, basada en filtro lineales con *derivada-cero* en el cruce, tiene mejores resultados con imágenes de alto ruido. El procedimiento [36] consta en aplicar un filtro que tenga derivada cero en el centro, filtrar el resultado con un detector de cruce por cero [37], realzar los bordes y finalmente se realizar una binarización. Esta técnica se denomina de, y consta en aplicar un filtro gaussiano de σ acorde a la aplicación, seguido de un filtro laplacciano para la primer etapa. Alternativamente, surgen filtros especializados como el filtro *LoG* [38], o el filtro *PLUS* [39], que generan mejores resultados, pero son mas costosos computacionalemente.

Segmentación basada en morfología matemática en escala de grises

Para el caso de imágenes de escala de grises, el procesamiento morfológico provee de ciertas transformaciones que puede ser útiles para la segmentación. La transformada *"top-hat"* permite aislar objetos convexos claros en escala de grises [40] a través de la resta entre la imagen de entrada A y la apertura en escala de grises de A con el elemento estructurante B :

$$TopHat(A, B) = A - (A \circ B) = A - \max_{B}(\min_{B}(A)) \tag{2.25}$$

donde B debe ser elegido mas grande que los objetos de interés. En el caso que se quiera resaltar objetos oscuros se utiliza la relación dual de los operadores y la ecuación resulta:

$$TopHat(A, B) = (A \bullet B) - A = A - \min_{B}(\max_{B}(A)) \tag{2.26}$$

Esto genera saliencias en la imagen que luego pueden ser binarizadas un cualquiera de los métodos presentados anteriormente. Sin embargo otra posibilidad de umbralizado proviene de la morfología, generando un binarizado parcial utilizando la dilatación

y la erosión con un EE B que modela la espacialidad de la transformación:

$$\theta[m,n] = \frac{1}{2}(\max_B(A) + \min_B(A)) \tag{2.27}$$

Por último, la transformación de *"watershed"* permite, bajo una interpretación topográfica de la imagen, realzar píxeles brillantes como si fuesen zonas con mayor "altura". El algoritmo más conocido es el de por *inundación* [41] y consta en sembrar "semillas" en los mínimos locales de la zona de interés ("valles"), "inundar" las zonas (dilatación morfológica), y a los resultados parciales compararlos con barreras que delimitan el crecimiento de las regiones. De esta manera los distintos valles se van conectando y formando lineas de unión que representarán las fronteras de regiones homogéneas que constituyen el resultado de la segmentación.

Segmentación basada en morfología matemática binaria

Una vez sucedida la binarización de una imagen existen grandes probabilidades de que el resultado no sea el deseado, ya que el objeto se puede encontrar con ruido, sobre-segmentado, o que la imagen tenga ruido que no permita aplicar algoritmos de detección o etiquetado de objetos. Para eliminar ruido *sal y pimienta*, se pueden aplicar operadores de dilatación y erosión condicional, utilizando EE de 4 u 8 elementos según convenga. Posterior "limpieza" de la imagen, se pueden separar los distintos objetos utilizando algoritmos llamados *Crecimiento de regiones*, en donde se "siembra" una semilla que pertenecen a una región de iteres, y se dilata condicionalmente de manera iterativa, hasta que el algoritmo converge y la región ya fue identificada. El problema ahora radica en la selección de la semilla. Si se desea que el sembrado no sea manual, entonces se puede aislar los objetos generando esqueletos especiales que solo prevalezcan aquellos que tengan ciertas características, y de allí generar las semillas.

Segmentación basada en movimiento

La utilización de movimiento para la extracción de objetos y regiones se basa en el desplazamiento relativo entre el sistema de sensado y una escena u objeto observado. La aproximación más sencilla es detectar los cambios que sucedieron entre

dos imágenes $f(x, y, t_i)$ y $f(x, y, t_j)$ tomadas en los instantes t_i y t_j, respectivamente, y compararlas píxel a píxel. Una forma de realizar dicha comparación es generar una imagen de diferencias. Si se tiene una imagen de referencia que contenga solo componentes estacionarios, se la puede comparar con una subsiguiente de la misma escena, pero que incluya un objeto en movimiento, y le resultado de la diferencia entre las dos imágenes cancela los elementos estacionarios, y deja con valores a aquellos píxeles correspondientes a los componentes de la imagen que se encuentran en movimiento.

La imagen diferencia resultante de dos imágenes tomadas en los tiempos t_i y t_j se define como

$$d_{ij} = \begin{cases} 1 & si|f(x, y, t_i) - f(x, y, t_j)| > \theta \\ 0 & \text{otro} \end{cases} \tag{2.28}$$

donde θ es un umbral específico. Todos aquellos píxeles en $d_{ij}(x, y)$ que poseen el valor 1 son considerados el resultado de un objeto en movimiento. Sin embargo este método solo es posible si las dos imágenes están encuadradas dentro de los mismo bordes y si la iluminación no cambia. En la práctica, píxeles con valor en 1 pueden aparecer como resultado del ruido entre las mismas. Para mejorar la segmentación se desarrollaron técnicas que se encargan de estimar el fondo de la escena bajo cambios de iluminación y no estables en movimiento, y restarlo en los distintos cuadros para generar la imagen diferencia.

2.2.3. Descriptores visuales

Luego de haber realizado algún tipo de segmentación y filtrado de la imagen, se procede a calcular los denominados *descriptores visuales*. Tienen el objetivo de representar, a través de distintos tipos de mediciones de parámetros, los objetos en un imagen de tal manera que posteriormente sea posible describirlos, distinguirlos y clasificaros del resto. En general se esperan que estos no sean sensibles a los cambios en tamaño, traslación ni rotación, y que sus propiedades sean extensibles a cualquier imagen. El primer grupo es tipo general, son descriptores de bajo nivel que obtienen una medida relacionada con una propiedad básica del objeto como el color, la forma, cantidad de regiones del mismo, su textura y movimiento:

- Descriptores de bordes: dan información a partir de mediciones relacionadas al borde del objeto. Entre los más simples existe el largo del borde (puede ser la cantidad de píxeles que lo compone), el diámetro del borde (la máxima distancia entre dos puntos que componen el borde), excentricidad (relación entre el lago del eje más largo contenido dentro de la figura y el eje más largo perpendicular al anterior), y la curvatura (aproximada por la diferencia entre las pendientes de los segmentos lineales en que se descompone el borde). Otros descriptores más complejos son los de *numero de forma n*, para el cual se necesita implementar el algoritmo de *codigo cadena*, *descriptores de Fourier*, y los *momentos estadísticos*.

- Descriptores de región: entre ellos se encuentra el área A (cantidad de píxeles dentro de los objetos), el perímetro P (longitud de su borde exterior delimitatorio), compactación (P^2/A), circularidad (relación entre A y el área de un circulo con igual perímetro), elongación(relación entre el ancho y alto del menor alguno que contiene el objeto), extensión (evalúa que tan rectangular es la forma), relación de aspecto, convexidad, entre otras. También se toma en cuenta, la media, mediana, el mínimo y máximo, y la distribución de las intensidades de los píxeles que la integran.

    - Descriptores topológicos: refiere a las propiedades de una forma que no cambian con su tamaño. El mas utilizado es el *número de Euler* (E), el cual consiste en la cantidad de componentes conectadas C menos la cantidad de agujeros que existen H ($E = C - H$).

    - Descriptores de textura: definen si una superficie es suave, rugosa, regular, irregular, etc. Para ello se utilizan medidas estadísticas de primer orden provenientes del sector de la imagen que contiene al objeto con su histograma de escala de grises normalizado. Entre ellas se encuentra el promedio, desvió estándar, contraste en escala de grises, la uniformidad, su entropía y los momentos de segundo y tercer orden. Entre mediciones estadísticas de segundo orden se encuentra la *matrix de co-ocurrencia en niveles de grises* (GLMC), el cual consigue mediciones estadísticas locales por vecindarios.

- Momentos invariantes: el *momento* en dos dimensiones de orden $(p+q)$ de una imagen digital $f(x,y)$ de tamaño $M \times N$ es :

$$m_{pq} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} x^p y^q f(x,y), \tag{2.29}$$

donde $p = 0, 1, 2...$ y $q = 0, 1, 2, ...$ son enteros. Utilizando esta definición y sus derivadas se pueden calcular un conjuntos de siete *momentos invariantes* con respecto a la traslación, rotación y escalamiento de un imagen u objeto.

2.2.4. Reconocimiento de objetos

Se define el *reconocimiento de objetos* a la tarea de encontrar un determinado objeto o patrón en un imagen dada sus características particulares. Se define como *patrón* a un conjunto de descriptores, como los definidos anteriormente. Una *clase patrón* w_i es una familia de patrones que tienen propiedades en común. El reconocimiento de patrones en una imagen consiste entonces, en técnicas para asignar patrones a sus correspondientes clases de forma automática, o con la menor intervención humana posible.

El primer tipo de aproximación es conocida como decisión teórica, el cual consta en seleccionar un vector de descriptores cuantitativos $\mathbf{x} = (x_1, x_2, ..., x_n)^T$, como largo, área y textura, que puedan describir la diversidad de patrones que existen. Otro tipo de clasificación es utilizando relaciones estructurales, en el cual se utilizan descriptores cualitativos de forma anidada para la detección de un patrón. Por ejemplo, si se tiene una imagen satélite de un área con un centro comercial rodeado por áreas residenciales, se utiliza una estructura de tipo árbol donde la imagen se encuentra en el tope del mismo. El segundo nivel esta compuesto por dos estructuras, área comercial y área residencial. En un tercer nivel, el área residencial se encuentra compuesta por casas, mercados y rutas; y por su parte, el área comercial esta compuesto por edificios y rutas. De esta manera se puede continuar subdividiendo hasta el punto de poder definir las diferentes regiones que existen en la imagen. La dewcripción de que cada uno re aliza definiendo elementos primitivos a, b, c,etc. que se utilizan

para definir patrones como la sucesión de dichos elementos ($w = ...abcbbca...$). Algunos ejemplos de métodos de clasificación son los clasificadores mediante árboles de decisión, algoritmos genéticos, clasificador de Bayes, redes neuronales, análisis por componentes principales, análisis regresivo, modelos de Markov, entre otros.

Redes neuronales

Las redes neuronales artificiales (RNA) son sistemas de procesamiento de datos cuya estructura y funcionamiento están inspirados en las redes neuronales biológicas [42]. Consisten en un gran número de elementos simples de procesamiento llamados nodos o neuronas que están organizados en capas. Cada neurona está conectada con otras neuronas (*sinapsis*) mediante enlaces de comunicación (en la biología llamado *axón*), cada uno de los cuales tiene asociado un peso. Los pesos representan la información que será usada por la red neuronal para resolver un problema determinado.Así, las RNA son sistemas adaptativos que aprenden de la experiencia, esto es, aprenden a llevar acabo ciertas tareas mediante un entrenamiento con ejemplos ilustrativos. Con este entrenamiento o aprendizaje, las RNA crean su propia representación interna del problema, por tal motivo se dice que son auto-organizadas. Posteriormente, pueden responder adecuadamente cuando se les presentan situaciones a las que no habían sido expuestas anteriormente, es decir, las RNA son capaces de generalizar de casos anteriores a casos nuevos. La Fig.2.13 ilustra el ejemplo de una red neuronal de varias capas.

De acuerdo a la función que las neuronas implementan, a la topología de conexión y a su mecanismo de aprendizaje, existen distintos tipos de redes neuronales artificiales. En la RNA *feedfoward* [43] los datos de entrada viajan en una sola dirección hacia la salida, sin realimentación. En contraposición, la RNA *recurrente* [44] posee una capa de entrada de datos, un conjunto de capas intermedias que se realimentan entre ellas, y una capa de salida. Una red de gran uso en la actualidad son las *redes convolucionales* [45], las cuales son redes *feedfoward* en donde a cada capa de neuronas se le aplica un conjunto de filtros lineales por convolución seguido por una función no-lineal simple. En la Sección 2.3 se explicará un tipo de red llamada *red neuronal celular no-lineal*, en donde todos los elementos de procesamiento realizan la misma función no-lineal en paralelo.

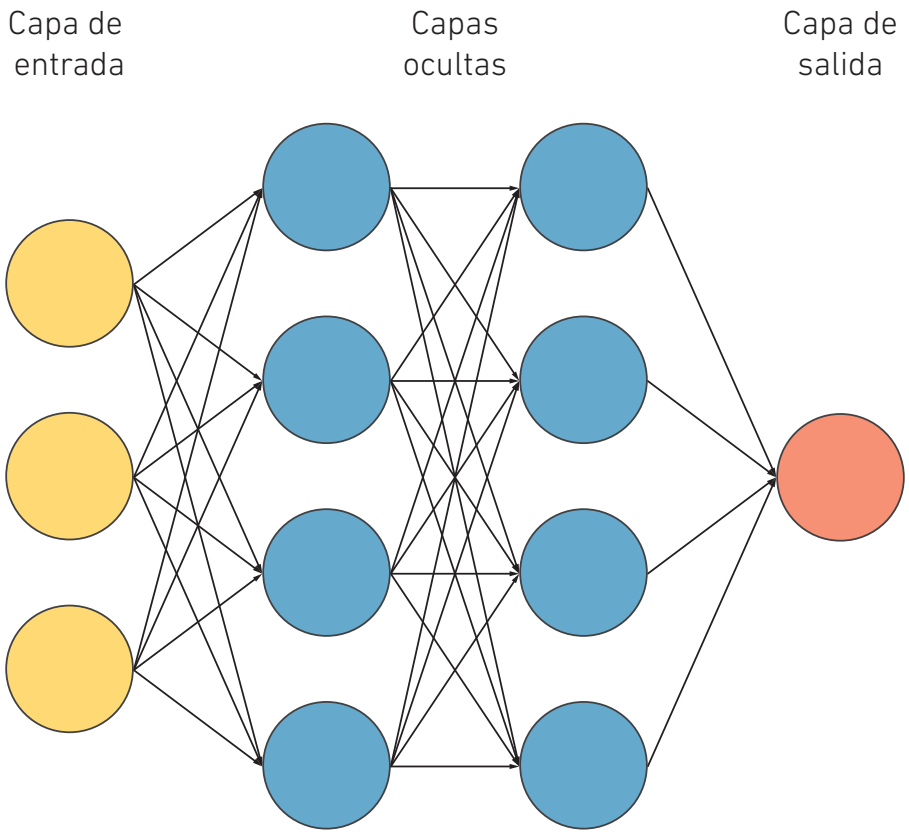
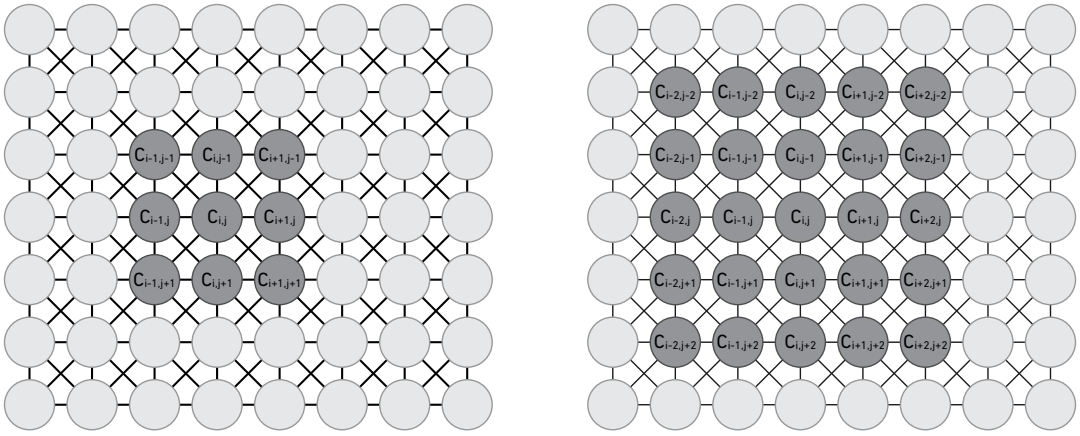


Figura 2.13: Ejemplo ilustrativo de red neuronal de 4 capas. La primer capa de dimensión 3 es por donde ingresan los datos de entrada. Las dos capas oculta están compuestas por 4 neuronas cada una, y la ultima capa de salida esta compuesta por una sola neurona. Dado que los datos solo se mueven a la siguiente capa es una sola dirección sin realimentación, se puede considerar la misma como una RNA *feedfoward*.

2.3. Redes celulares no lineales

La Red Celular No-lineal o "Cellular Neural Network" (CNN) es una arquitectura de RNA introducida por Chua [46] [47], la cual consta de un arreglo de elementos dinámicos (células o celdas) con acoplamiento local que procesan información en paralelo. Las células pueden ser organizada con diversas configuraciones, sin embargo la CNN más popular es de dos dimensiones organizada es una grilla rectangular con vecindad o conectividad de 8 elementos. Dada la similaridad de dicha estructura con la disposición de píxeles en una imagen, resulta atractiva para el procesamiento de imágenes de bajo, medio y alto nivel.



(a) Red con conectividad de 8 elementos (b) Red con conectividad de 24 elementos

Figura 2.14: Ejemplo de redes neuronales celulares con conectividad de $r = 1$ y $r = 2$.

Cada celda es un sistema dinámico compuesto por una entrada, un estado y una salida, e interactúa solo con las células dentro del radio de vecindad igual r. En general, el estado de cada celda, y por lo tanto su salida, depende solo de la entrada y la salida del vecindario, y del estado inicial de la red. La descripción matemática de una celda C ubicada en la posición (i, j), donde $1 \leq i \leq M$ y $1 \leq j \leq N$, en un arreglo de tamaño $M \times N$, está dada por la ecuación de evolución del estado $\dot{x}(t)$ y la ecuación de salida $y_{i,j}(t)$:

$$C_{i,j} = \begin{cases} x_{i,j}(t) = F(x_{i,j}(t), z_{i,j}(t); \mathbf{y}_{S_{i,j}}(t), \mathbf{u}_{S_{i,j}}(t)) \\ y_{i,j}(t) = G(x_{i,j}(t)) \end{cases} \tag{2.30}$$

donde $F : \Re^m \to \Re^1$ es la función que determina la dinámica celda y $G : \Re^1 \to [0, 1]$,

es la función de salida, los vectores $\mathbf{u}_S(t) \in \Re^n$ y $\mathbf{y}_S(t) \in \Re^n$, son los vectores formados por las entradas y salidas dentro del vecindario de interés, $S_{i,j}$ refiere al conjunto de las n celdas involucradas en la función de evolución de estados, conocido como *esfera de influencia*, y $m = 2n$. Si el radio de vecindad es r es igual a 1 (Fig.2.14(a)), donde la distancia se mide con respecto a la celda central $C_{i,j}$ utilizando la norma infinito, entonces la esfera de influencia queda definida como:

$$S_{i,j} = \{C_{i-1,j-1}, C_{i,j-1}, ..., C_{i-1,j+1}, C_{i-1,j}, C_{i,j}\} \tag{2.31}$$

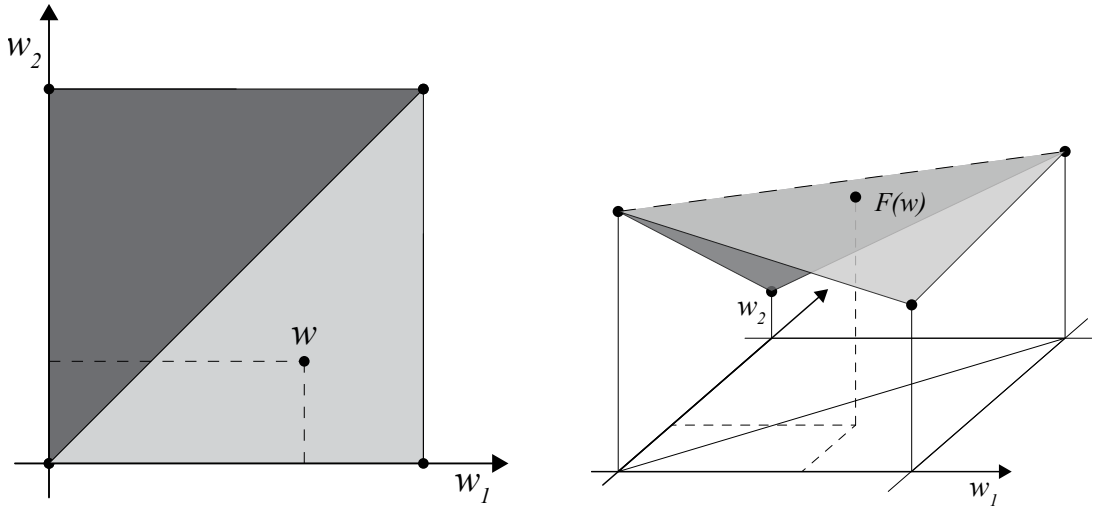
Se puede observar que la esfera de influencia para aquellas celdas ubicadas en los bordes del arreglo no se encuentra totalmente definida. Por lo tanto, es necesario especificar *condiciones de borde* de manera tal que la ecuación Ec.2.30 quede completamente definida. En el modelo estándar la condiciones pueden ser:

- Fijas (o Dirichlet): los estados x de las celdas C cuya ubicación es por fuera del arreglo se le asigna una valor constante.
- Flujo cero (o Neumann): los estados x toman el valor de los estados de las celdas vecinas mas cercanas que se encuentran perpendiculares dentro del arreglo.
- Periódico (Toroidal): los estados de las celdas fuera del arreglo toman el valor del de las celdas del borde opuesto.

Dada la expresión general de una celda perteneciente a un arreglo CNN y su esfera de influencia, cabe destacar que las funciones $F()$ y $G()$ definen la estructura de la CNN y sus capacidades de computo. En el trabajo [48] se puede observar la expresión de la estructura CNN estándar, ampliamente estudiada y utilizada en la literatura. Sin embargo, es de interés para la presente tesis introducir la red neuronal celular simplicial (cuyo acrónimo en ingles es SCNN), la cual es la base estructural de los trabajo aquí presentados.

2.3.1. Redes celulares simpliciales SCNN

Una Red Neuronal Celular Simplicial [49] es una CNN en donde la ecuación de evolución de estado $F(\cdot)$ en las celdas es una función lineal a tramos (PWL),



(a) (b)

Figura 2.15: (a)Interpolación lineal del rango de la función PWL; (b) Interpolación lineal de la función PWL .

y $G(\cdot)$ es una función lineal de derivada igual a uno. A continuación se introduce su expresión en tiempo discreto para un $r = 1$, la cual resulta conveniente para su implementación digital:

$$C_{i,j} = \begin{cases} x_{i,j}(k+1) = F(x_{i,j}(k), z_{i,j}(k); \mathbf{y}_{S_{i,j}}(k), \mathbf{u}_{S_{i,j}}(k)) \\ y_{i,j}(k) = x_{i,j}(k) \end{cases} \tag{2.32}$$

donde $F : \Re^m \to \Re^1$ es una función PWL, $\mathbf{y}_{S_{i,j}}(k) \in \Re^n$ y $\mathbf{u}_{S_{i,j}}(k) \in \Re^n$ los vectores de salida y entrada pertenecientes a la esfera de influencia $S_{i,j}$, $x_{i,j}(k) \in [0, 1]$ y $y_{i,j}(k) \in [0, 1]$ son el estado y salida de la celda ubicada en la coordenada (i, j), n la cantidad de celdas dentro de $S_{i,j}$, y $m = 2n$.

Como se puede observar de la Ec.2.32, la función F es una función PWL definida sobre una partición del dominio simplicial del espacio de entrada-salida de la CNN. Así mismo, un dominio simplicial es un dominio subdividido en símplices, donde un símplice es una generalización en n-dimensiones de un triangulo en $\Re^2$ y puede ser descrito por la combinación convexa de sus $(n + 1)$ vértices. Los vértices son puntos dentro del dominio definidos por las intersecciones de dimensión cero de la partición(Fig.2.15(a)). En cada símplice la función lineal a tramos F es un a función lineal afín la cual puede ser representada por un hiperplano, unicamente valido solo en ese símplice. De este modo, el hiperplano queda definido por la combinación

convexa de los $(n+1)$ valores de la función definida en cada vértice del símplice. Si se toma una partición de grillado unitario, el conjunto de vértices definidos para una espacio de n dimensiones se denota:

$$V_D = \{\mathbf{v} \in \Re^n : v_i \in \{0,1\}\,,\ i = 1, ..., n\}, \tag{2.33}$$

y los símplices asociados a dichos vértices son

$$\begin{aligned}\tilde{V}_D = \Big\{ & \tilde{V}_D^k = \{\mathbf{v}^1, ..., \mathbf{v}^i, ..., \mathbf{v}^{n+1}\} : k = 1, ..., n!\,, \\ & v_j^1 = 0, j = 1, ..., n; \quad v_j^{n+1} = 1, j = 1, ..., n; \\ & \|\mathbf{v}^{i+1} - \mathbf{v}^i\|_\infty = 1, i = 1, ..., n \Big\}.\end{aligned} \tag{2.34}$$

Una manera alternativa de representación de una función lineal a tramos sobre un dominio simplicial es presentada por Julian en el trabajo [50]; la función PWL se expresa en un espacio Hilbert de dimensión $q = 2^n$ para una partición unitaria. Utilizando la base propuesta en [51], la función descriptora F queda expresada como:

$$F(w) = c^T \Lambda(w), \tag{2.35}$$

donde $w \in \Re^n$ es un punto en el espacio entrada-salida, $c \in \Re^q$ es un vector de coeficientes, y $\Lambda : \Re^n \to \Re^q$ es un vector de funciones base $\Lambda(\cdot) = [\alpha_1(\cdot), ..., \alpha_q(\cdot)]$. Cada elemento $\alpha_i : \Re^n \to \Re^1$ de la base es igual a uno sobre un vértice y cero en los restantes:

$$\alpha(\mathbf{v}_j) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases} \qquad \mathbf{v}_j \in V_D, \quad i, j = 1, ..., q. \tag{2.36}$$

En la expresión (2.35) el vector de coeficientes c coincide con los valores de la función en cada vértice. De esta manera, los parámetros de cálculo necesarios pueden almacenarse en una memoria talque la dirección de lectura este asociada a la coordenada del vértice, y el valor almacenado sea el valor de la función en ese vértice. Vale destacar, que con esta representación dado un punto de entrada perteneciente a un símplice dado, para el cálculo de la función en ese punto solo se necesita evaluar $(n+1)$ coeficientes. Queda resolver como se identifica el símplice al

cual el punto de entrada pertenece.

2.3.2. Computo de una SCNN

Para el cálculo de la función F se utiliza la información almacenada en una tabla indexada o memoria y un algoritmo que interpola el valor final, el cual consta de los siguientes tres pasos:

Primer : dado un vector de entrada w el símplice que lo contiene debe ser identificado, y de esta manera, el conjunto de vértices $\tilde{V}_D^k$ que lo identifican.

Segundo : una vez identificado el símplice, se debe descomponer la entrada w en una combinación convexa de los vértices:

$$w = \sum_{l=1}^{n+1} \mu_l v_l^k, \quad v_l^k \in \tilde{V}_D^k, \tag{2.37}$$

donde $\mu_l \in \Re^1$, $l = 1, ..., (n+1)$ y $\sum_{l=1}^{n+1} \mu_l = 1$.

Tercer : una vez obtenidos los vértices $\tilde{V}_D^k$ y calculados los pesos μ_l de la combinación convexa, resta calcular el valor de la función en el punto de entrada. Para ello es necesario deben leer los coeficientes $\{c_l, l = 1, ..., (n+1)\}$ asociados a los vértices identificados v_l^k, y realizar la sumatoria ponderada del paso anterior:

$$F(w) = \sum_{l=1}^{n+1} \mu_l c_l. \tag{2.38}$$

En el trabajo [52] se presenta una forma de implementar el algoritmo, y en [53] se propone una versión modificada para el procesamiento en señal mixta. Sin embargo, es de interés la implementación digital presentada por Mandolesi en [54], la cual se cita a continuación.

Implementación digital

Dado que la implementación es digital, todas la variables intervinientes son discretas, inclusive el tiempo; las componentes de la señal de entrada w y los coeficientes

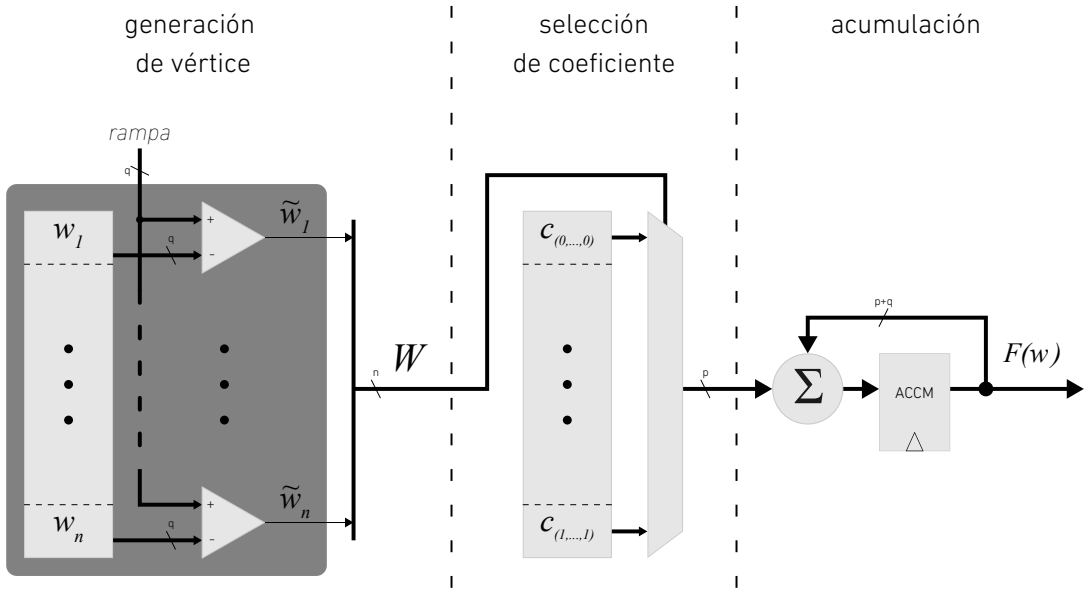


Figura 2.16: Diagrama en bloques de una estructura digital de calculo simplicial

c_l se almacenan en una memoria con precisión fija, p y q respectivamente. El computo del algoritmo para el cálculo de la función se realiza en sincronismo con una señal de rampa digital de p bits que recorre todos los valores posibles del dominio de manera monotonamente ascendente con el tiempo. Las entradas digitales se comparan individualmente con la rampa y se generan n señales de 1 bit $\{\tilde{w}_i,\ i = 1, ..., n\}$, las cuales codifican en tiempo el valor de los w_i. Las $\tilde{w}_i$ se concatenan formando el vector W, el cual identifica los distintos vértices del símplice que contiene a w conforme la rampa avanza en el tiempo. Por otro lado, el valor de los pesos μ_l de la combinación convexa (Eq.2.37) se ven representados por el tiempo en que W apunta a cada vértice. De esta manera, el grupo de comparadores llevan a cabo los dos primeros pasos de procedimiento de computo. El tercer paso es realizado entre la memoria de almacenamiento de coeficientes y un acumulador. El vector W de indexación de vértices ingresa a la memoria como dirección, y los correspondientes c_l leídos son acumulados en cada paso de rampa. Al final del ciclo de rampa, el valor de la función $F(w)$ queda almacenado en el acumulador. En la Fig.2.17 se ilustra la evolución de la rampa, las comparaciones y el vector de identificación de vértices para un sistema con tres entradas. La Fig.2.16 muestra el diagrama en bloques de la arquitectura anteriormente descrita. Esta estructura digital resulta beneficiosa al ser implementada en todas las celdas dentro del arreglo, ya que, como se menciono anteriormente, las células computan la misma función en paralelo y recursos pueden

ser compartidos, reduciendo el área total del arreglo.

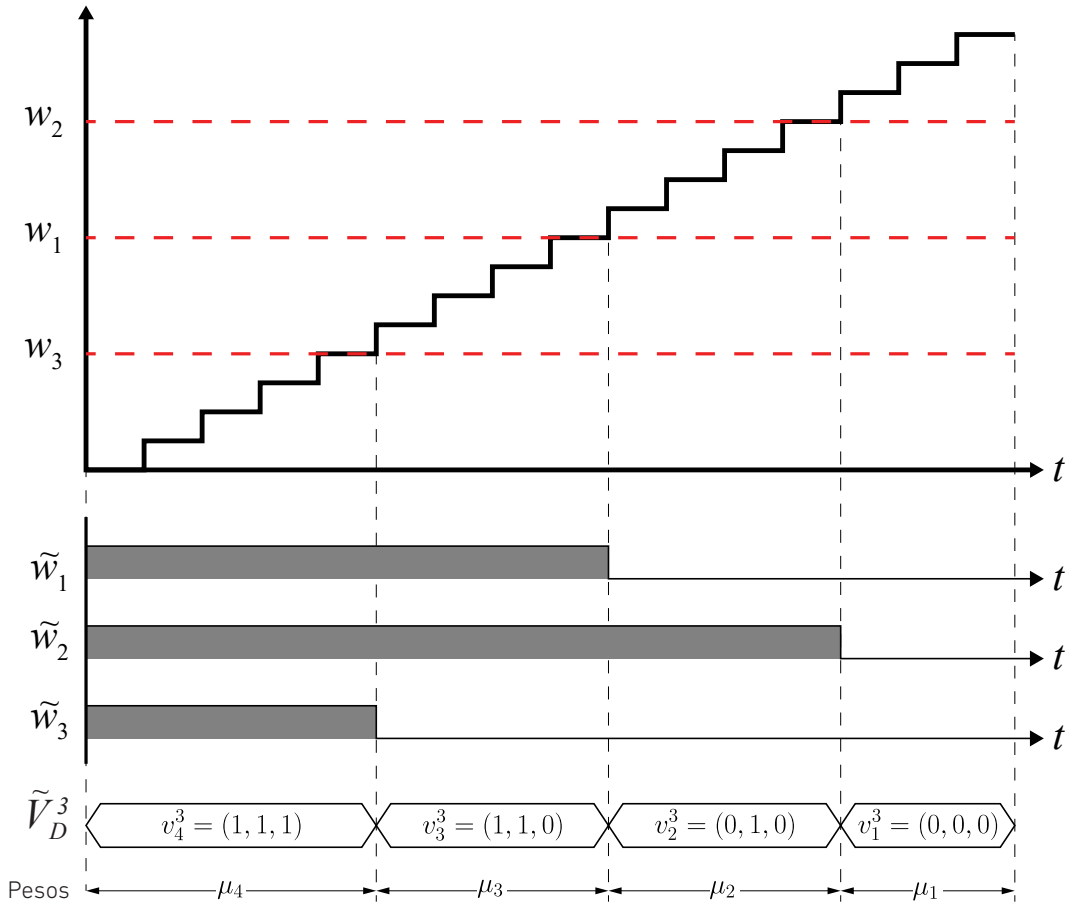


Figura 2.17: Ejemplo de codificación de vértices utilizando comparaciones con una rampa genérica y tres entradas.

2.4. Resultados existentes de procesadores de imágenes

En la literatura existen varios sistemas con diferentes enfoques de arquitecturas para el procesamiento masivo de señales, en especial de imágenes. La primer distinción que se puede hacer es cuando la adquisición de la imagen se realiza en el mismo chip donde ocurre el procesamiento. En un primer caso los elementos de computo (PE) pueden estar embebidos en el arreglo de fotodiodos, integrados en el mismo píxel, lo que permite procesar la imagen en paralelo aprovechando la espacialidad del arreglo. Todos los elementos de procesamiento operan simultáneamente aplicando la misma función utilizando los correspondiente valores de la vecindad, lo cual resulta beneficioso para operaciones puntuales y locales. La principal ventaja de este tipo de sistema es la velocidad de procesamiento y el bajo consumo de energía, ya que no se necesitan grandes desplazamientos de datos, sin embargo, el tamaño del píxel incrementa al añadir las nuevas estructuras y se pierde densidad de píxeles en área de silicio. Otra desventaja deviene de la necesidad de diseñar manualmente de las celdas para una tecnología dada, lo que dificulta la potabilidad del diseño hacia otros nodos.

En el segundo caso, los PE están dispuestos en un vector separado del bloque de captura de imagen. Estos ejecutan instrucciones comunes (SIMD) provistas por una memoria de programa principal en donde los operandos son datos leídos por a nivel fila o columna. Si bien estas arquitecturas pueden realizar operaciones puntuales y locales, son espaciales en procesamiento global que involucran gran cantidad de datos. Por otro lado, si los datos son ingresados desde el exterior del chip, se reemplaza el arreglo de sensores por memorias locales que almacenan imágenes. En estas estructuras los PE que previamente estaban embebidos con los sensores, se organizan en arreglos bidimensionales que emulan espacialmente el arreglo de fotosensores, pero que toman las imágenes de entradas desde la memoria local.

Un segundo grupo de chips surgen directamente de la implementación de redes neuronales. En estos se diseñan estructuras especialmente diseñas para implementar distintas topologías de redes y algoritmos que existen en tiempo real. Generalmente poseen varias jerarquías de memorias para un acceso rápido de estados y parámetros

de las distintas capas, y elementos de procesamiento que emulan de diferentes maneras el comportamiento de una neurona, los cuales poseen acumuladores de precisión fija (funciones lineales) y elementos de procesamiento no-lineal. A continuación se describen algunos de los chips más destacados que se encuentran en la literatura para el procesamiento de imágenes de bajo, medio y alto nivel.

VAE En el trabajo [55] Seungjin Lee, Minsu Kim, Kwanho Kim, Joo-Young Kim, y Hoi-Jun Yoo presentan el acelerador **VAE** basado en la utilización de una red neuronal celular (CNN) que implementa el algoritmo de *atención visual* [56] (VA) para el reconocimiento de objetos. La topología utilizada para la red es la denominada TMPE (*"time-multiplexed processing element"* o en español *elementos de procesamiento multiplexados en tiempo*), en donde se utiliza una cantidad reducida de elementos de procesamiento (PE) para emular el funcionamiento de una red mayor cantidad de celdas.

La arquitectura esta compuesta por un arreglo de 80×60 celdas con interconectividad local, organizadas en 4 grupos de 20×60. Entre el primer grupo y el segundo, y el tercero y el cuarto, se instancian 2 vectores de 60 elementos PE. Las celdas tienen como objetivo el almacenamiento de las variables (4 registros de 8 bits en SRAM) y la comunicación intra-celular de la CNN (registro de desplazamiento de 8 bits conectados con las 4 celdas vecinas). Cada PE lee dos valores de la fila correspondiente y los opera en su ALU generando dos salidas seleccionables, una para cada grupo de arreglo de celdas. La ALU posee dos hilos de procesamiento: el primero es una MAC seguido de una función no-lineal de saturación, el segundo implementa la diferencia, máximo y mínimo.Por otro lado, un controlador administra las señales de las celdas y los procesadores a través de la lectura y codificación de instrucciones de 16 bits de ancho almacenadas en una memoria local de 2 KB.

El chip **VAE** fue fabricado en una tecnológica CMOS de 130nm en un área de $4,5\text{mm}^2$ y reportó una eficiencia de 261 GOPs/W a 200MHz de frecuencia de reloj y $1,2$V de tensión de alimentación.

Zhang2011 En el trabajo [57] de Wancheng Zhang, Qiuyu Fu, y Nan-Jian Wu, se presenta un sistema de visión en chip con múltiples niveles de paralelismo para el

procesamiento imágenes. Un arreglo de 32×128 elementos de procesamiento (PE) SIMD de 1 bit de computo, permite llevar a cabo algoritmos de filtrado espacial lineales y no lineales, operadores morfológicos y operaciones entre imágenes que requieran alto nivel de paralelismo. Cada PE posee 64 bits de almacenamiento local en grupos de 8, y se encuentra conectado con los 4 vecinos en forma de cruz, mas el los PE a distancia 2 y 4, de la izquierda y derecha. Conectado al arreglo de PEs a nivel fila, un vector de 32 procesadores SIMD interconetados localmente, permite computar operaciones de mayor precisión, que requieran información de toda la imagen, como ecualización por histograma, FFT, DCT, y operaciones estadística, entre otras. El mayor nivel de abstracción de procesamiento es dado gracias a la presencia de un procesador de propósito general 8051, el cual también se encarga de administrar las señales e instrucciones de los distintos niveles de SIMD. Por último, un arreglo de 128×128 fotodiodos APS y un vector de 128 conversores analógico-digital (ADC), permite capturar imágenes de 8 bits de precisión a una velocidad máxima de más de 1000 cuadros por segundo. El chip fue fabricado en una tecnología CMOS de 180nm, y registró una eficiencia de $97,9$GOPs/W operancon con una frecuencia de reloj de 100MHz a una tensión de $1,8$V.

Carey2013 En el trabajo [58] Stephen J. Carey, Alexey Lopich, David R.W. Barr, Bin Wang y Piotr Dudek presentan un sistema adquisidor de imágenes embebido en un arreglo de 256×256 procesadores SIMD analógicos, capaz de muestrear 100 mil cuadros por segundo. Cada celda contiene 14 registros DRAM y 7 analógicos acoplados a un fotodiodo, una unidad lógica aritmética, y un modulo de conexión de vecindario asíncrono analógico que habilita la conexión con sus 4 vecinos en forma de cruz.. Esta estructura le da la capacidad de leer la imagen en formato digital binario y en formato analógico. El chip se fabricó en 180nm, ocupa una área de $10 \times 10\text{mm}^2$, y con una tensión de $1,8$V para los circuitos digitales y $1,5$V para los analógicos, reportó una eficiencia de 655GOPs/W con una frecuencia de reloj de 10MHz.

SCDVP y 3DDife En el trabajo [59] de Martin Di Federico, Pedro Julián, y Pablo S. Mandolesi se presenta un chip de adquisición y procesamiento de imágenes

de bajo y medio nivel utilizando una red neuronal celular simplicial (S-CNN) denominado **SCDVP**. La red esta conformada por arreglo bidimensional de 64×64 celdas con vecindad de 4 elementos reconfigurable, basadas en las celdas publicadas en [60] y [54] realizadas por el mismo de trabajo. Cada celda esta compuesta por un sensor APS y un conversor AD para la captura de imagen, un banco de 4 registros de 6-bits y un motor de procesamiento PWL, que permite computar hasta 2 funciones lineales a tramos de 5 entradas de 6-bits utilizando parámetros binarios. Dicha estructura le da la funcionalidad al procesador de realizar operaciones lógicas, aritméticas y morfológicas entre píxeles e imágenes, aplicar filtros no lineales de detección de borde, mediana, binarización, entre otros; realizar extracciones de lóbulos, esqueletonización, calculo centroide y "optical flow" [61]. El chip fue fabricado en la tecnología 90nm, y reporta una eficiencia total de $817,8$GOPs/W para una condición de operación de $1,1$V y 133MHz.

Posteriormente, el mismo grupo presenta un trabajo [62] en donde se muestra un sistema para la adquisición, procesamiento y análisis de imágenes en el plano focal. La fabricación del chip se realizó utilizando la tecnología 3D de Tezzaron Semiconductors en el nodo 130nm, en donde se divide la estructura en dos pisos para mayor densidad de pixeles y procesamiento. La arquitectura esta compuesta por una arreglo de 48×32 celdas y un conjunto de periféricos como un procesador 8051, un módulo de comunicación serie, una unidad aritmética de 16bits, integrador y memoria, entre otros. Cada celda posee el sensor APS y su correspondiente ADC, junto con registros de almacenamiento local y un contador, pero en este caso, el motor PWL se encuentra compartido por fila. Tiene la capacidad de procesar hasta 2 funciones lineales a tramos de 5 entradas de 7 bits de precisión con parámetros de 1 bit.

Schmitz2016 En el trabajo [63] de Joseph A. Schmitz, Mahir K. Gharzai, Sina Balkır y otros, se presenta la arquitectura de una chip que implementa una red celular jerárquica de dos niveles para el procesamiento de bajo, medio y alto nivel de imágenes. El primer nivel es una unidad denominada NP compuesta por un procesador SIMD personalizado RISC de 8 bits, embebido en un arreglo de 8×8 sensores DPS con sus respectivos ADCs en el mismo plano focal. El segundo nivel es

un arreglo de 8×10 de NPs, los cuales se interconectan localmente con vecindad de 4 elementos. De esta manera el arreglo puede procesar en distintas escalas imágenes de resolución de 64×80 píxeles. Para el control del arreglo implementan un controlador principal que programa las celdas y envía las instrucciones a los procesadores. Otra maquina de estado se implementa para generar las señales globales para la conversión dentro de los sensores. El chip se fabricó en una tecnología CMOS de 130nm, y se reportó una eficiencia máxima de 44GOps/W para una frecuencia de reloj de 20MHz y 1,2V de tensión de alimentación.

YodaNN En el trabajo [64] de Renzo Andri, Lukas Cavigelli, Davide Rossi and Luca Benini se detalla el acelerador **YodaNN**, una arquitectura digital "data-flow" flexible sintetizada en la tecnología CMOS de 65nm, que permite computar redes convolucionales (CNN) de imágenes. El sistema utiliza 1024 procesadores de multiplicación y acumulación (MAC) de 12bits, para implementar capas de CNN de 32 entradas y 32 salidas completamente conectadas en paralelo utilizando filtros lineales 2D de 7×7 elementos de 1-bit. Además la arquitectura cuenta con dos niveles de memoria ($10,5$kB y $2,29$kB) para almacenar y procesar hasta 12 conjuntos de entradas de precisión de 8 bits, y un banco de filtros de acceso rapido de $6,125$kB. Si bien el sistema no fue fabricado, simulaciones post-layout reportan una eficiencia máxima de computo teórica de $61,2$TOPs/W en el núcleo y $0,98$TOPs/W del dispositivo entero, para una tensión de $0,6$V.

XNORBIN En el trabajo [65] de Andrawes Al Bahou, Geethan Karunaratne, Renzo Andri, Lukas Cavigelli y Luca Benini, se presenta una arquitectura digital reconfigurable para la implementación de redes convolucionales binarias o BNN [66] denominada **XNORBIN**. El Sistema XNORBIN posee un arreglo de 7 *unidades básicas de procesamiento* o BPUs, las cuales realizan en conjunto la suma de la XOR punto a punto entre la ventana de entrada y un filtro binario de 7×7 elementos. Este resultado ingresa a una unidad de computo CU, el cual acumula los resultados parciales de acuerdo a la programación y realiza el proceso de binarización o activación, cuyo resultado es leído por un DMA que lo guarda en memoria. El XNORBIN posee tres niveles de memorias jerárquicas para evitar el acceso reiterado

y costoso hace fuera del chip. El primero almacena los mapas de descriptores, las sumas parciales, y las configuraciones de cada capa como los pesos de los filtros y sus tamaños, los valores de binarizacion, entre otros. El segundo nivel es un banco de memoria que sirve como almacenamiento temporal de las imágenes de entrada leídas de la memoria principal. Por último, el tercer nivel de jerarquía es un "cross-bar" que conecta la memoria dentro de las BPUs con los valores almacenados en el segundo nivel, para reconfigurar la ventada de entrada. El chip fue sintetizado en ls tecnología de 65nm, y simulaciones poss-layout reportan una eficiencia máxima del núcleo de $95,2$TOPs/W y de 23TOPs/W para el dispositivo entero, con una tensión de $0,8$V de alimentación.

Eyeriss En el trabajo [67] de Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, y Vivienne Sze, se presenta un sistema basado en un acelerador de redes neuronales convolucionales (CNN) para el procesamiento de imágenes denominado **Eyeriss**. El sistema esta compuesto por un arreglo reconfigurable de 12×14 elementos de procesamientos SIMD (PE) con comunicación local, una memoria local de 108 KB, un modulo ReLU, y una unidad de compresión de datos que de salida que detecta ceros para mejorar la eficiencia en energía. El almacenamiento principal esta dado por una memoria DRAM fuera del chip que funciona en un segundo demonio de reloj conectado al sistema a través de una FIFO. Cada elemento de procesamiento esta compuesto por tres tipos de memoria, un multiplicador de dos etapas y un acumulador. La primer memoria, es un banco de registros de $12 \times 16b$ que almacena el dato de entrada que va a ser filtrado. La segunda es de tipo SRAM de $224 \times 16b$ para el almacenamiento de los filtros. La tercer memoria es un bando de registros de $24 \times 16b$ para el almacenamiento de sumas parciales.

El chip fue fabricado en una tecnología CMOS de 65nm en un área de $3,5mm \times 3,5mm$, y reportó, durante la ejecución de una red AlexNet [68], una eficiencia de $1228,8$GMACs/W, con una tensión de alimentación de $0,82$V.

TrueNorth En el trabajo [69] de Filipp Akopyan, Jun Sawada, Andrew Cassidy y otros, miembros de el grupo de investigación de IBM, se presenta el diseño, desarrollo y fabricación de un chip neuromórfico, llamado *TrueNorth*, capaz de modelar

el funcionamiento neuronas programables. La arquitectura esta conformada por un arreglo bidimensional de 4096 núcleos neuro-sinápticos, que almacenan 1 millon de neuronas con una interconectividad de 256 millones de sinapsis. En particular el chip no ejecuta instrucciones de programa seriales, propio de una arquitectura convencional Von-Neumann, sino que el comportamiento de las neuronas es programado y la evolución de sus estados esta dado por la interacción entre ellas a través de "spikes" [70] que son transmitidos utilizando una red de comunicación (NoC) con protocolo asíncrono basado en eventos.

Funcionalmente cada núcleo posee una *crossbar* sináptica, o matriz de conexiones, donde las lineas horizontales representan los *axones* de las neuronas, los verticales las *dendritas*, y las conexiones entre ellas las *sinapsis*. Al final de cada linea vertical se halla un una unidad que se alimenta de la *crossbar* y emula el comportamiento de una neurona que integra las entradas y actualiza su *potencial de membrana* implementando una función multivariable no lineal compleja [71]. Cuando dicho potencial excede un umbral programable, genera un *spike* hacia otra neurona del mismo núcleo u otro dentro de la NoC. La configuración y las variables de estado de cada neurona se almacenan localmente en una memoria SRAM de 256 filas (una por neurona) y 410 columnas que representa los 410 bits necesarios para el funcionamiento de cada neurona.

El chip fue fabricado una tecnología CMOS de 28nm en un área aproximada de $4,3\ \text{cm}^2$, y consume 65mW en aplicación visuales típicas de detección de objetos [72] y clasificación de los mismos [73].

Loihi En el articulo [74] el grupo de investigación de Intel liderado por Mike Davies presenta el chip **Loihi** de procesamiento de datos utilizando modelos de redes de neuronas por disparo o "spike" [75](SNN del ingles *"spiking neural network"*) con capacidad de aprendizaje. Su arquitectura contiene una NoC asincrónica mallada que comunica un arreglo 128 procesadores neuromórficos (PN) y tres CPU de 32 bits. Un grupo de interfaces de comunicación permiten conectar el sistema con otros chips similares, y así, expandir el arreglo de procesamiento SNN. En la NoC se implementa un protocolo que soporta pedidos y respuestas de escritura y lectura entre los núcleos por partes de los CPU para administración de los mismos, mensajes

de disparo para el computo de tipo SNN, y mensajes de sincronización entre los PN. Para la comunicación con el exterior, los mensajes son empaquetados con jerarquías y administrados por los CPUs. De esta manera el protocolo permite escalar a 4096 la cantidad de PN por chip, y hasta 16,384 chips. Cada unidad neuromófica es capaz de implementar hasta 1024 unidades neuronales multiplexadas en tiempo. La estructura esa compuesta por una memoria de 2Mb de datos donde se almacenan los parámetros y estados de las neuronas modeladas, y una unidad sináptica que procesa los "spikes" que ingresan al PN y generan las salidas hacia otros PN. El chip fue fabricado en una tecnología FinFET de 14nm de Intel en un área de 60mm^2, con capacidad de almacenar más de $2,1$ millones de variables sinápticas por mm^2.

2.5. Conclusión

Como se puede observar el universo de arquitecturas que se implementan para el procesamiento de imágenes es de lo más variado, y en general se intenta que los módulos que se encargan de las distintas etapas de adquisición, procesamiento y análisis sean lo más flexibles y autocontenidos posibles para poder implementar los diversos algoritmos y operaciones anteriormente vistos, con la menor cantidad de traslado de la información entre ellos, y de este modo lograr una mayor eficiencia energética. Los procesadores siempre tienen una componente de procesamiento espacial paralela, módulos de procesamiento no-lineal, otros que realizan funciones más globales, y por último una procesador de más alto nivel que se encarga de la administración de los distinto recursos y de la toma de decisiones que afectan al algoritmo implementado. Dentro de este contexto, la utilización de RNA, y en especial las redes celulares simpliciales, es de gran conveniencia ya que implementan procesamiento paralelo distribuido con funciones no-lineales embebidas, lo cual presentan una solución natural a las primeras capas del procesamiento de imágenes.

Capítulo 3

Procesadores morfológicos basados en funciones lineales a tramos

3.1. Introducción

En este capítulo se presentan dos arquitecturas de procesamiento SCNN, una para imágenes binarias denominada MORPHO1PWL, y otra para escalas de grises denominada MORPHO8PWL. El objetivo es diseñar estructuras digitales totalmente parametrizables y flexibles, que implementen funciones no-lineales con eficiencia comparable a los diseños reportados en la literatura que poseen funcionalidades similares.

Los elementos de procesamiento (PU) utilizados implementan dos funciones lineales a tramos en una vecindad configurable de 4 celdas, basados en celdas de trabajos previos([9] [10] [11]). La cantidad de parámetros necesarios por función es 32 y dependiendo de la arquitectura la precisión es 1 o varios bits. Aprovechando que todas las celdas tienen el mismo comportamiento, la memoria de parámetros es diseñada en la periferia con el objetivo de aumentar la densidad de computo [12] [13].

El capítulo se encuentra organizado de la siguiente manera. Cada sección refiere a una arquitectura diferente. Dentro de las mismas se describe el diseño y funcionamiento de la celda de procesamiento elemental, y luego se detalla su instanciación dentro de un arreglo de tipo CNN y los controladores diseñados para el manejo de las señales de procesamiento y configuración del sistema. Una vez descrita la arqui-

tectura, se explican los métodos de configuración del sistema y se ilustran algunas de las tareas básicas que puede ejecutar el circuito. Por último, se muestra el desarrollo del circuito en silicio, donde se muestran detalles de implementación y resultados experimentales para poder evaluar el rendimiento. Al final del capítulo se resaltan las características más importantes de los sistemas propuestos.

3.2. Procesador energéticamente eficiente para tareas visuales primarias de un bit

En este capítulo se describe un arreglo CNN de 64x 64 celdas, donde cada una está interconectada con 5 vecinos y es capaz de implementar funciones no-lineales con 1 bit de precisión para sus variables y parámetros. El objetivo principal del arreglo es implementar operaciones morfológicas sobre imágenes binarias

El diseño denominado MORPHO1PWL posee una arquitectura compuesta del arreglo de 64 x 64 elementos de procesamiento (PE), y módulos periféricos que configuran y manejan las señales pertinentes al procesamiento. Para su configuración y transferencia de datos desde el exterior del módulo, se implementa un protocolo de comunicación basado en AMBA AHB-Lite de ARM®. El diseño fue fabricado en una tecnología de 180nm.

3.2.1. Arquitectura

La arquitectura del sistema está compuesta por un arreglo de 64×64 PE, un módulo de configuración y control con una memoria de programa de 64 instrucciones de profundidad, y una memoria de funciones capaz de almacenar hasta 16 conjuntos de 32 parámetros, tal como se muestra en la Fig.3.1.

Elemento de Procesamiento

Cada uno de los elementos de procesamiento (PE) tiene una estructura como la ilustrada en la Fig. 3.2. Hay dos registros de estados de un bit denominados X y U, que almacenan el estado y la entrada de la celda . La celda implementa una composición programable de dos funciones $F \circ G$, donde la función F se aplica sobre

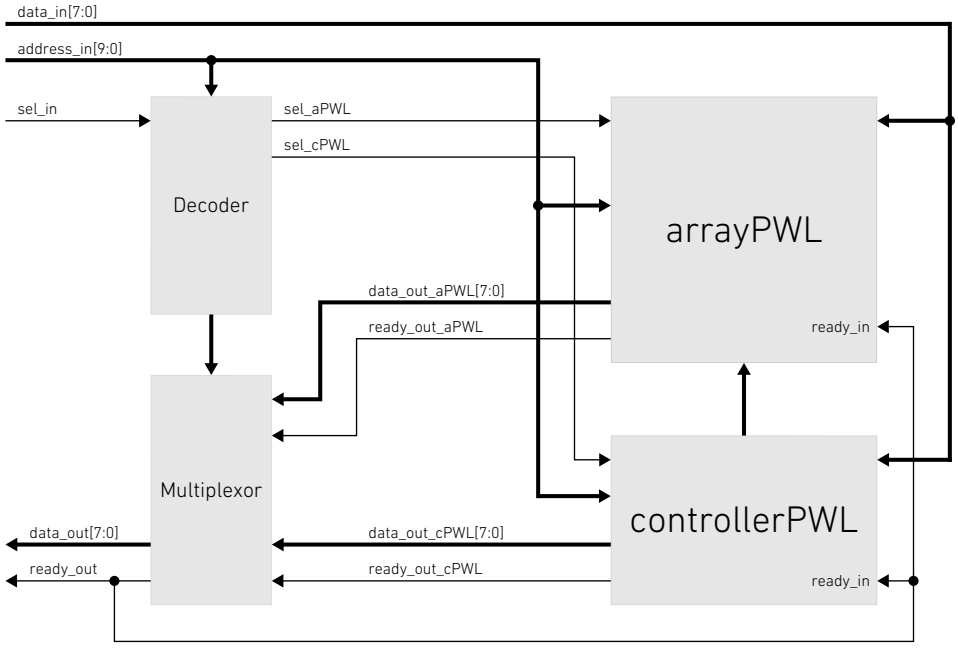


Figura 3.1: Arquitectura del procesador MORPHO1PWL.

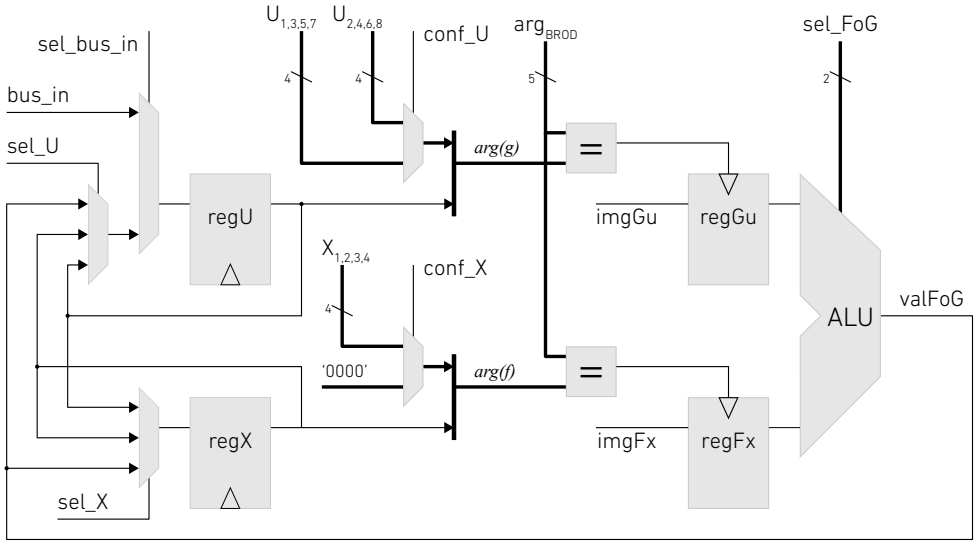


Figura 3.2: Esquema del elemento de procesamiento.

el conjunto de estados X de la propia celda y 4 vecinos, y la función G se aplica sobre el conjunto de argumento U de la propia celda y 4 vecinos:

$$\arg(f) = \{X_4, X_3, X_2, X_1, X_0\}\, \arg(g) = \{U_4, U_3, U_2, U_1, U_0\} \tag{3.1}$$

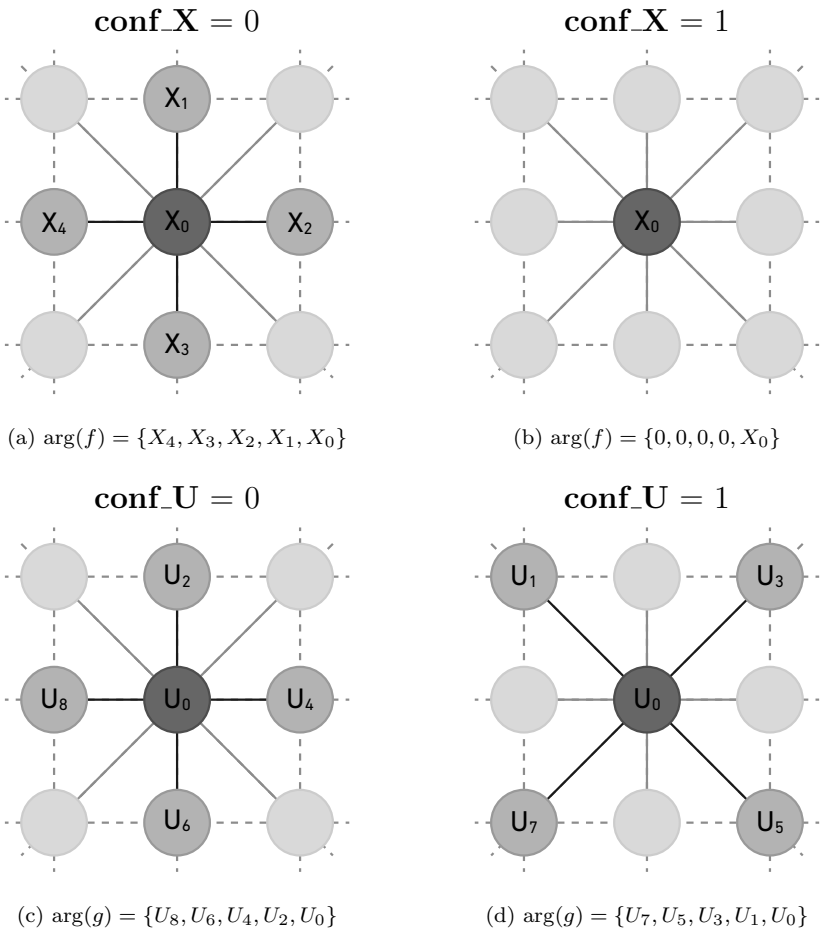
donde $arg(f)$ y $arg(g)$ son de 5 bits y pueden tomar 32 valores posibles desde 0

a 31.

Para evitar guardar la información correspondiente a las funciones dentro de cada celda, que consiste en 32 bits para F y otros 32 para G, esta información se almacena fuera del arreglo y se distribuye al arreglo, tal como se explicó en la Sección X.x. En efecto, durante la fase de procesamiento, que dura 32 ciclos de reloj, el argumento de cada función es comparado con la señal de rampa global arg_{BROD} el cual es común a todos los PE. Cuando ambas son iguales, el valor de la función f_i, proveniente de la señal global $imgFx$, es almacenado en el registro $regF_x$. De igual manera, el valor de U junto con sus correspondientes valores vecinos forman $arg(g)$, para que luego g_i sea registrada en $regG_u$. Como se observa en la Fig. 3.3, la variable $conf\_X$, $arg(f)$ permite enmascarar las señales de celdas vecinas. Por otro lado, la señal de un bit $conf\_U$ selecciona la configuración de $arg(g)$ (Fig.3.3(c)), 3.3(d)) para seleccionar una vecindad en "x" o "+".

En la fase de procesamiento, que dura 32 ciclos de reloj, el argumento de la función es comparado con la señal de rampa global arg_{BROD} el cual es común a todos los PE. Cuando ambas son iguales, el valor de la función f_i, proveniente de la señal global $imgFx$, es almacenado en el registro $regF_x$. De igual manera, el valor de U junto con sus correspondientes valores vecinos forman $arg(g)$, para que luego g_i sea registrada en $regG_u$. Como se observa en la Fig.3.3, dependiendo del valor de $conf\_X$, $arg(f)$ también puede ser formado unicamente el valor de X local más un agregado de ceros, brindando la opción de enmáscaramiento.

El valor almacenado en $regG_u$ a junto con el de $regF_x$ ingresan a una unidad aritmética lógica (ALU) donde se operan para generar el resultado final val_FoG. El valor resultante puede guardarse en el registro X una vez concluido el ciclo de procesamiento. La ALU puede implementar 4 funciones lógicas AND, XOR, OR y NOR dependiendo de la señal global de selección sel_FoG (Tab.3.1). La variable sel_X (Tab.3.2) define el valor que se almacenará en el registro de estado durante el siguiente ciclo de funcionamiento. Las tres opciones son: val_FoG, U o su propio valor. El registro U, por su parte, dependiendo sel_U puede almacenar val_FoG, X o U (esta última opción también puede utilizarse si se utiliza el registro X para enmascarar). El registro U es el único que puede ser cargado directamente a través del bus externo bus_in, si las señales de decodificación $sel\_bus\_in\_col$ y $sel\_bus\_in\_row$



(a) $\arg(f) = \{X_4, X_3, X_2, X_1, X_0\}$

(b) $\arg(f) = \{0, 0, 0, 0, X_0\}$

(c) $\arg(g) = \{U_8, U_6, U_4, U_2, U_0\}$

(d) $\arg(g) = \{U_7, U_5, U_3, U_1, U_0\}$

Figura 3.3: Configuraciones posibles para la formación del argumento en el cálculo de las funciones G_u y F_x. En especial cuando **conf_X**=1 el procesador entra en un modo de enmascaramiento en el cual si el valor de X es igual a 1, el estado U permanece inalterado al final del procesamiento.

(sintetizadas en *sel_bus_in*) son igual a uno (Tab.3.2).

Tabla 3.1: Operaciones de la ALU.

sel_FoG	Función
0	AND
1	XOR
2	OR
3	NOR

Arreglo de procesadores

Los elementos de procesamiento se disponen en un arreglo de dos dimensiones de 64 por 64 elementos (*arrayPWL*), interconectados locamente a través de las señales

Tabla 3.2: Selección del próximo valor de *regU*.

Próximo *regU*	Condición
bus_in	Si $sel\_bus\_in = 1$
U	Si $sel\_U = 1$, o si $conf\_X = 1$ y $X = 1$
val_FoG	Si $sel\_U = 1$
X	Si $sel\_U = 2$ o $sel\_U = 3$

Tabla 3.3: Selección del próximo valor de *regX*.

Próximo *regX*	Condición
X	Si $sel\_X = 0$ o $sel\_X = 3$
val_FoG	Si $sel\_X = 1$
U	Si $sel\_X = 2$

U y X, tal como aparece en la Fig.3.4. Dado que el arreglo es finito, los PE de las fronteras no tienen vecinos y la esfera de influencia se extiende por fuera del arreglo. Para ello, se implementa la señal *sel_border* de dos bit que permite elegir como condiciones de borde los siguientes tres valores: '0', '1', o el valor igual a la de su vecino dentro del arreglo, tanto U como para X. Junto con esta señal, ingresan también las señales globales de procesamiento, configuración y control necesarias para el funcionamiento de las celdas.

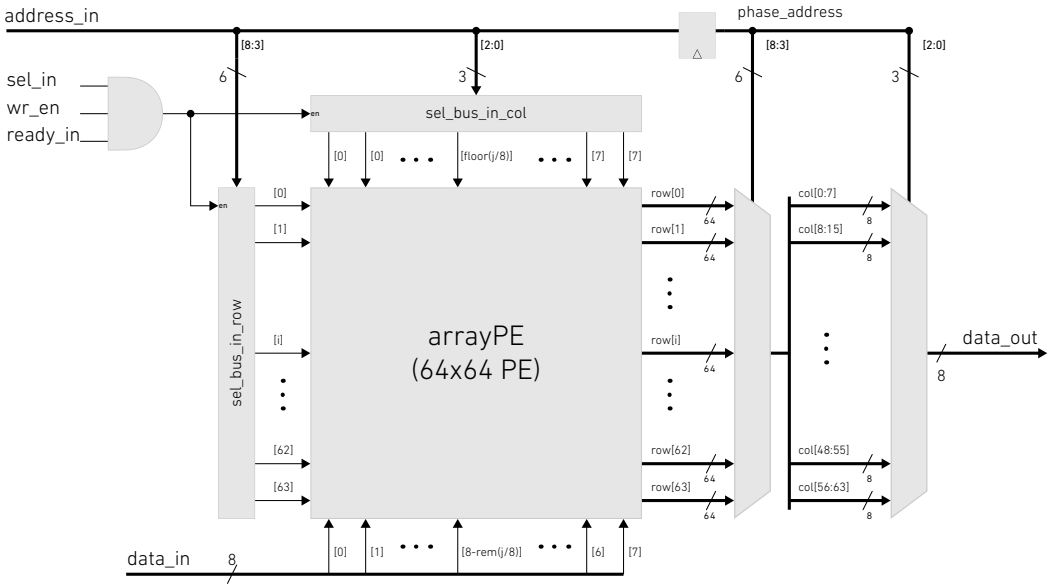


Figura 3.4: módulo donde se instancia el arreglo de celdas de procesamientos y circuitos periféricos para su es escritura y lectura.

Para la lectura y escritura delos buses de datos de entrada y salida se utiliza

el protocolo AHBL simplificado de 8 bits. Para el bus de direcciones *address_in*, el cual se destina únicamente para el acceso de los registros *regU* de los procesadores, se utilizan 9 bits. El bloque se encuentra ubicado en las primeras 512 direcciones lógicas de memoria del sistema completo, desde 0 hasta 511.

Durante un proceso de escritura los primeros tres bits de *address_in* ingresan a un decodificador que genera 8 señales habilitación *sel_bus_in_col* en el eje horizontal H. Esto se debe a que tanto en la escritura como en la lectura, los procesadores no son accedidos individualmente sino en grupos de ocho, agrupados a lo largo del eje horizontal, utilizando completamente los buses de datos. Por lo tanto, la primer señal de habilitación *sel_bus_in_col[0]* ingresa en las primeras ocho columnas, *sel_bus_in_col[1]* en el segundo grupo de ocho columnas, y así sucesivamente. Los 6 bits restantes de direcciones generan la señal de decodificación *sel_bus_in_row* de 64 bits correspondiente al eje vertical en el arreglo. Las salidas de ambos decodificadores son registradas y únicamente son habilitadas cuando el módulo interno de manejo del protocolo de comunicación lo permite.

Para la lectura se implementan dos multiplexores anidados, uno por cada eje, cuyas señales de selección provienen de *phase_address_in* (version *buffereada* de *address_in*) manteniendo la misma lógica de mapeo de direcciones. Por ejemplo, el PE ubicado en la columna 10 y la fila 2 debe ser accedido seteando el valor de direcciones $address\_in = \mathbf{2} \times 2^3 + \mathsf{floor}(\mathbf{10}/8) = 17$, y corresponderá al bit número $8 - \mathsf{rem}(\mathbf{10}/8) = 3$ de los buses de datos. Se debe aclarar que como la escritura y lectura se realizan por paquetes de 8 bits, los 7 procesadores restantes, desde la columna 8 a la 15, son accedidos al mismo tiempo. Si se quiere acceder al arreglo mientras se encuentra procesando, el valor de la señal IO *ready_out* es cero, lo cual no permite que la transacción se complete. Es necesario, entonces, esperar que el procesamiento termine para que *ready_out* vaya a uno, y así finalizar el acceso.

Periféricos de configuración y control

El control de las señales que interviene en el procesamiento dentro del arreglo de PE, es llevado a cabo por un módulo controlador principal, *controllerPWL*. El mismo está conformado por una máquina de estado de control (denominado CFSM en Fig.3.5) y un conjunto de memorias y registros que permiten su configuración.

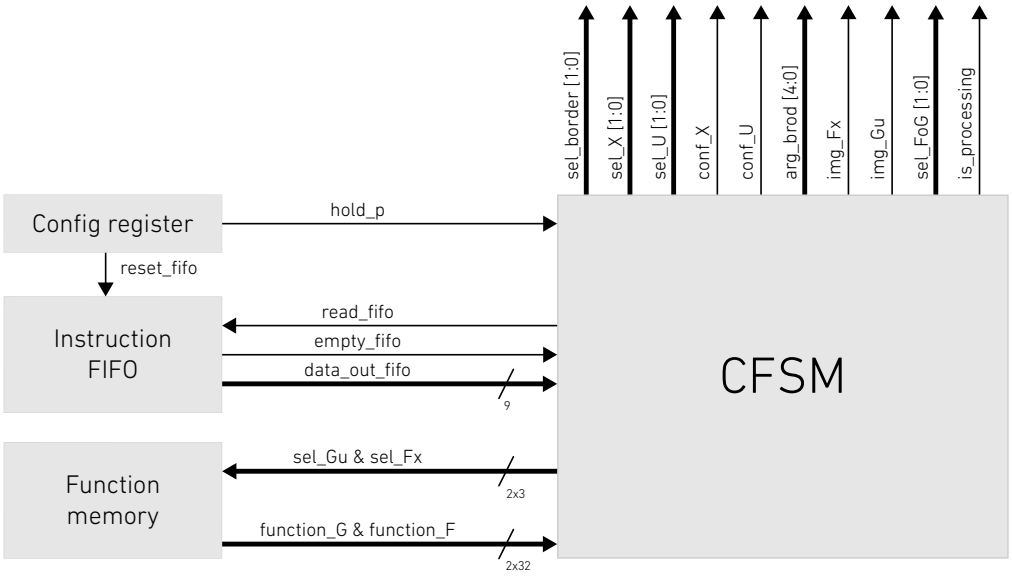


Figura 3.5: Diagrama en bloques del módulo de control *controllerPWL* resaltando la conectividad de los diferentes bloques que lo integran.

El CFSM ejecuta instrucciones almacenadas en una memoria de tipo FIFO (*'First-In-First-Out'*) de 9 bits de ancho y profundidad igual a 64. Para cargar instrucciones nuevas, se accede externamente escribiendo entre las direcciones lógicas 544 y 545. El bit menos significativo del bus de direcciones se concatena con el bus de datos de entrada para formar la palabra de instrucción. La FIFO implementa las señales de bandera *full_flag* y *empty_flag* para indicar si se encuentra llena o vacía, respectivamente. La bandera *empty_flag* le indica a la CFSM que existe una instrucción nueva que debe ser leída y ejecutada. Cada vez que se desee cargar desde el inicio la FIFO, es necesario reiniciar sus punteros internos a través del registro $reset\_fifo$, que se ubica en el bit 1 de la dirección 546.

Una segunda memoria sirve para almacenar los parámetros de las funciones que se desean utilizar. Esta se encuentra conformada por un grupo de registros, localizados entre las direcciones lógicas 512 y 543, que permiten guardar hasta tres funciones de 32 parámentos de un bit para el cómputo de G_u (G_u_0, G_u_1 y G_u_2), y otras tres de iguales características para F_x (F_x_0, F_x_1 y F_x_2). Cada parámetro corresponde a una dirección lógica individual, es decir, para escribir el parámetro 0, se debe apuntar a la dirección 512 e ingresar el valor en el bus de datos de entrada:

$$data\_in = \{0, 0, G_u\_2[0], G_u\_1[0], G_u\_0[0], F_x\_2[0], F_x\_1[0], F_x\_0[0]\} \tag{3.2}$$

A su vez, la memoria de funciones ofrece conjuntos predeterminados de parámetros, comunes para G y F (Tabla 3.4): ZEROS (todos los parámetros son cero, dando como resultado siempre cero), MIN (mínimo dentro de la esfera de influencia), MAX (máximo dentro de la esfera de influencia), MEDIAN (cálculo de la mediana), y COPY (copia del valor de la celda central, o de posición cero dentro del elemento estructurante).

Tabla 3.4: Funciones disponibles en memoria.

sel_Gu / *sel_Fx*	Función
0	ZEROS
1	MIN
2	MAX
3	MEDIAN
4	COPY
5	Gu_0 / Fx_0
6	Gu_1 / Fx_1
7	Gu_2 / Fx_2

Por otro lado, a medida que la CFSM lee y ejecuta las instrucciones de programa, va alterando los valores de un conjunto de registros, los cuales manejan y configuran directamente las señales que van al *arrayPWL*. Estos registros se describen en Tab.B.1 del Apéndice de Tablas.

Finalmente, para permitir el inicio del funcionamiento de la máquina de estado del controlador, se debe setear el registro *hold_proc* a 0, que se se localiza en el bit 0 de la dirección 546 del sistema.

Máquina de estados

El controlador PWL implementa una máquina de dos estados que lee las instrucciones ingresadas a la FIFO de programa y setea registros internos que manejan las señales dentro del arreglo de procesadores PWL. A continuación se detallan los estados participantes en la máquina, cuyo diagrama de flujo se encuentra ilustrado

en la Fig.3.6.

- IDLE_ST: representa el estado inicial. Si la señal *hold_proc* es cero, se evalúa si la memoria de programa tiene una instrucción disponible($empty\_flag = 0$). De ser así, la memoria es leída, la instrucción se evalúa, y se avanza al siguiente estado según corresponda (Tab. 3.5). En *IDLE_ST*, la bandera *is_processing* siempre es cero y por lo tanto el arreglo es accesible para su escritura o lectura.

Tabla 3.5: Descripción de las instrucciones de programa que se lee durante el estado *IDLE_ST*.

Nombre	Bits									Descripción
	8	7	6	5	4	3	2	1	0	
CONFIG	0	sel_border		conf_U	conf_X	sel_U		sel_X		Se configura la condición de borde, las esferas de influencias, y el valor que toma el registro U y X en cada PE en el siguiente ciclo de reloj. Se mantiene el estado *IDLE_ST* para leer la siguiente instrucción.
START_P	1	sel_Gu			sel_Fx			sel_FoG		Se elijen las funciones a computar para G y F, y la función FoG, se lleva a cero el contador de *arg_broad*, y se comienza a procesar pasando al estado PROCESSING_ST.

- PROCESSING_ST: representa el estado de procesamiento al cual se llega luego de que la instrucción *START_P* es evaluada. Si la señal *hold_proc* es cero, incrementa el indice en uno y setea la señal de bandera *is_processing* en uno. Una vez que *arg_broad* llega al valor 31, todos los parámetros de las funciones seleccionadas ya han sido distribuidos, y se vuelve al estado *IDLE_ST*.

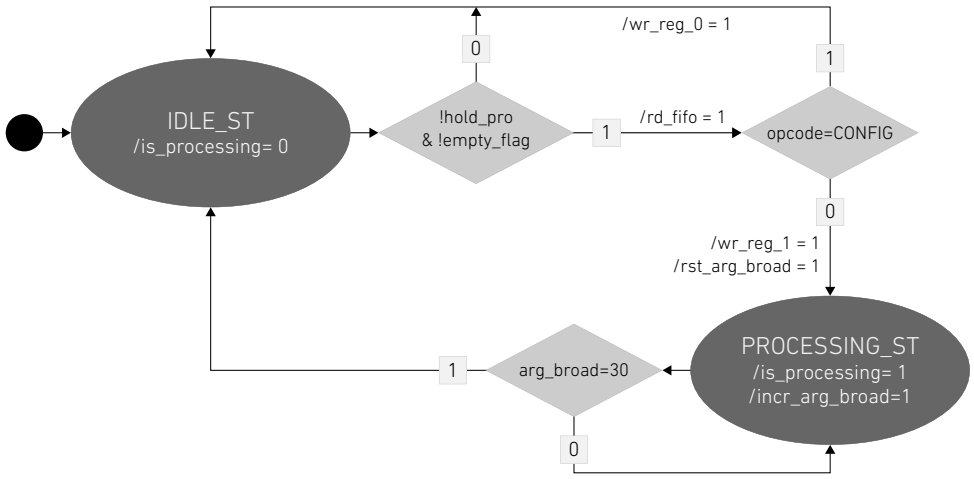


Figura 3.6: Diagrama de estados del controlador de procesamiento utilizado en la arquitectura MORPHO1PWL.

3.2.2. Funcionamiento

A continuación se explican las distintas tareas realizables por el sistema ilustrando sus respectivos pasos a seguir. Para ello es necesario considerar el mapeo lógico de las distintas unidades dentro del sistema presentado en la Tabla 3.6.

Tabla 3.6: Mapeo lógico de las direcciones para escritura y lectura.

Dirección	Sentido	Descripción
0-511	Escritura/ Lectura	Se accede a el registro *regU* de las celdas del arreglo.
512-543	Escritura/ Lectura	Memoria de funciones. Ocupa 32 espacios de memoria correspondiente a los 32 parámetros por función.
544-546	Escritura	Memoria de instrucciones. La dirección 544 corresponde a la instrucción *CONFIG*, mientras que la 545 es *START_P*.
546	Escritura	Se escribe al registro de configuración.
546	Lectura	Se lee una palabra de estado de 8 bits compuesta por cuatro ceros en los 4 bits mas significativos, las banderas de estado de la FIFO de instrucciones, y los dos bits menos significativos del registro de configuración ($data\_out[7:0] = \{0,0,0,0,empty\_flag, full\_flag, hold\_proc, is\_processing\}$).

Previo a realizar cualquier tarea, se debe pulsar la señal de reinicio del sistema *reset*, para reiniciar la máquina de estado de control a sus valores por defecto. Cuando se reinicia, también lo hace los registros de configuración de procesamiento, permitiendo que los PE almacenen, de manera estable, los valores ingresados desde el exterior. A continuación se detalla la configuración de las tareas básicas para la

escritura, lectura, y procesamiento de una imagen de entrada binaria.

Para almacenar una imagen binaria de 64×64 en el arreglo, en la primera fase de reloj se deben habilitar las señales de selección *sel_in* y la de escritura *wr_en_in*, y direccionar el grupo de celda internas deseado con el bus de direcciones *address_in*. Si la señal de respuesta *ready_out* es alta, en la segunda fase se procede a presentar el dato de entrada en el bus *data_in*. Por ejemplo, si se quieren cargar los valores $0, 1, 1, 1, 0, 0, 0, 1$ en 8 celdas ubicadas en la fila 5 y las columnas desde 16 a 23, entonces la dirección a ingresar es $address\_in = \mathbf{5} \times 2^3 + \mathbf{2} \times 2^0 = 42$ y el dato a presentar es $data\_in[7:0]$ = "10001110". En la Fig.3.7 se muestra la secuencia de señales para la escritura de una imagen completa.

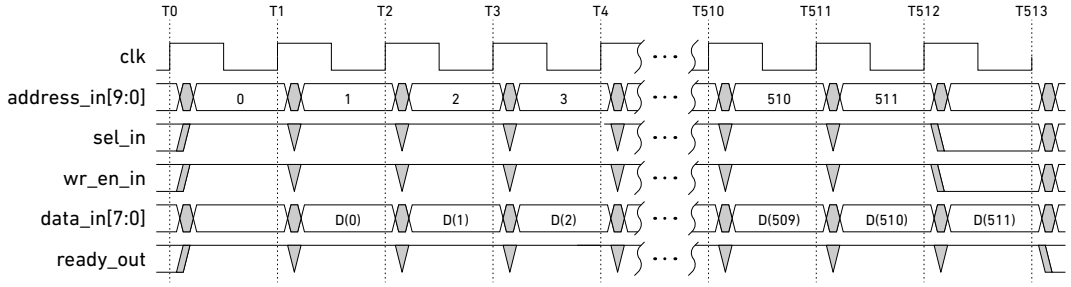


Figura 3.7: módulo donde se instancia el arreglo de celdas de procesamientos y circuitos periféricos para su es escritura y lectura.

Para la lectura de las celdas, el procedimiento es similar al de la escritura pero con la señal *wr_en_in* en 0. Durante la primera fase se presenta la dirección, y , si la señal de respuesta *ready_out*, se espera en la segunda fase el dato de salida a través de *data_out*. En la Fig.3.8 se puede observar el diagrama de señales para una lectura de todo el arreglo de procesadores.

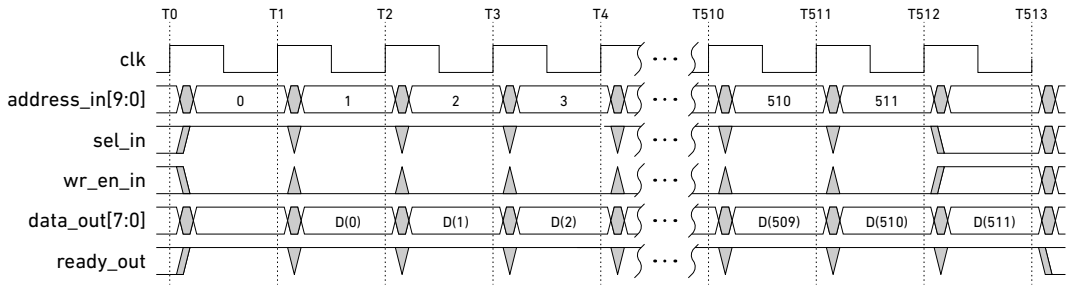


Figura 3.8: módulo donde se instancia el arreglo de celdas de procesamientos y circuitos periféricos para su es escritura y lectura.

Para procesar una imagen es preciso previamente haber ingresado las funciones

que se quieren computar (si no son las ya almacenas en la memoria), reiniciar la FIFO de programa , y luego escribir las instrucciones pertinentes (detalladas anteriormente) que configuran los PEs, teniendo en cuenta que el orden en que las operaciones son escritas definen el orden en que van a ser ejecutadas. A continuación se explica con más detalle los pasos para realizar estas tareas:

- **Carga de funciones**: Si no se desea utilizar la funciones ya almacenadas, se deben cargar aquellas que quieran ser utilizadas. Para ello es necesario escribir los 32 parámetros en las direcciones mostradas en Tab.3.6.

- **Reinicio de la memoria de programa y controlador** : Se escribe en la dirección 546, el registro de configuración para reiniciar la FIFO ingresando un 1 en el bit 1 (*reset_fifo*) como dato de entrada; y luego un 0 en el bit 1 nuevamente. Si así se desea, se puede poner en pausa el controlador seteando el bit 0 (*hold_proc*) en 1, y luego de programar la memoria de instrucciones, volver a guardar un 0 en el bit 0 para reanudarlo.

- **Carga de instrucciones**: La primera debe ser la instrucción de tipo *CONFIG* (ver Tab.3.5), donde se configuran las condiciones de borde *sel_border* y las vecindades con *conf_U* y *conf_X*. Sin embargo, *sel_U* y *sel_X* deben ser seteados en 0 para que no pierdan sus estados iniciales. Seguidamente una instrucción *START_P* debe ser ingresada, con el fin de seleccionar las funciones y dar comienzo al procesamiento. Por último, se debe realizar un pulso en *sel_U* y *sel_X*, para que los registros internos tomen los nuevos datos. Para ello se escriben dos instrucciones *CONFIG* una seguida de la otra, para que los pulsos internos solo sean de un ciclo de reloj. Una vez finalizada la escritura del programa se vuelve a setear el registro *hold_proc* en 0 para habilitar la lectura de la memoria de programa. En la Fig.3.9 se muestra un ejemplo de escritura de la memoria de instrucciones para computar el máximo de la vecindad de U (en morfología equivale a una dilatación con un elemento estructurante de forma +), con una condición de borde igual a 0; el resultado se guarda en X.

Una vez iniciado el procesamiento, le lleva al procesador 1 ciclo de reloj procesar una instrucción de tipo *CONFIG*, y 32 ciclos de reloj una de tipo *START_P* ya que

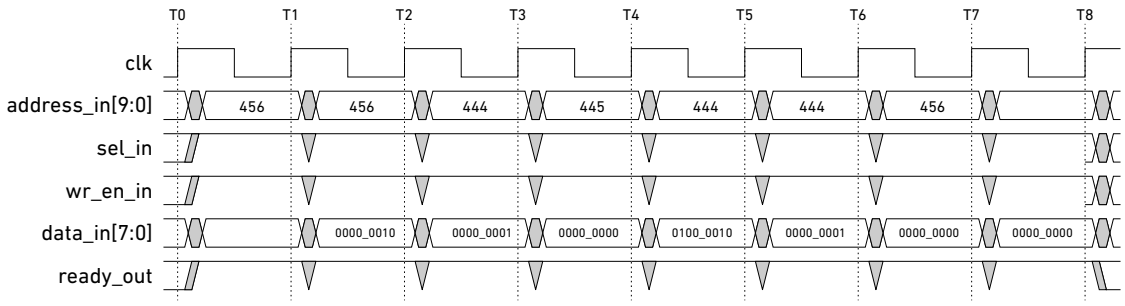


Figura 3.9: Diagrama de tiempo de las señales IO para ejemplo de carga de instrucciones para el sistema MORPHO1PWL.

en vez de propagar todos los parámetros de la función en un mismo ciclo, se hace uno por uno, reduciendo la cantidad de señales físicas que llegan a los procesadores, y por lo tanto, reduciendo el área de implementación.

3.2.3. Desarrollo del circuito

Para el diseño y la implementación se elije el flujo de tipo *"Top-Down"*, donde se realiza una descripción funcional de alto nivel del circuito en el lenguaje VHDL, y este luego se simula, verifica, sintetiza, mapea y rutea de acuerdo a lo explicado en el apéndice. La tecnología utilizada para su implementación es 180 manómetros de GlobalFoundries© donde se utilizan la librería de compuertas y pads brindadas por ARM©. En la Fig.3.10 se muestra la máscara del chip enviado, donde el tamaño del arreglo es 56×56 en una área total igual a 4mm^2. La reducción en la cantidad de celdas se debió a una limitación en el área de silicio; sin embargo, dado que el diseño es parametrizable, dicha modificación se pudo llevar a cabo sin cambiar el diseño. En la Tab.3.7 se listan las características del chip fabricado.

Para evaluar y testear el chip fabricado, se utilizó un placa personalizada montada a una de desarrollo OpalKelly XEM6010, con la cual, a través de la FPGA Spartan 6 que tiene integrada, se generan los vectores de excitación necesarios para testear la funcionalidad y el rendimiento del chip. Con una tensión de core de $V_{DD} = 1{,}8\text{V}$ y una frecuencia de reloj igual a $F_{core} = 100\text{MHz}$, el chip consume $15{,}3\text{mW}$.

Si se define una operación básica como una función lógica de dos entradas de 1 bit y una salida de 1 bit, la celda realiza 641 operaciones por ciclo de procesamiento de 33

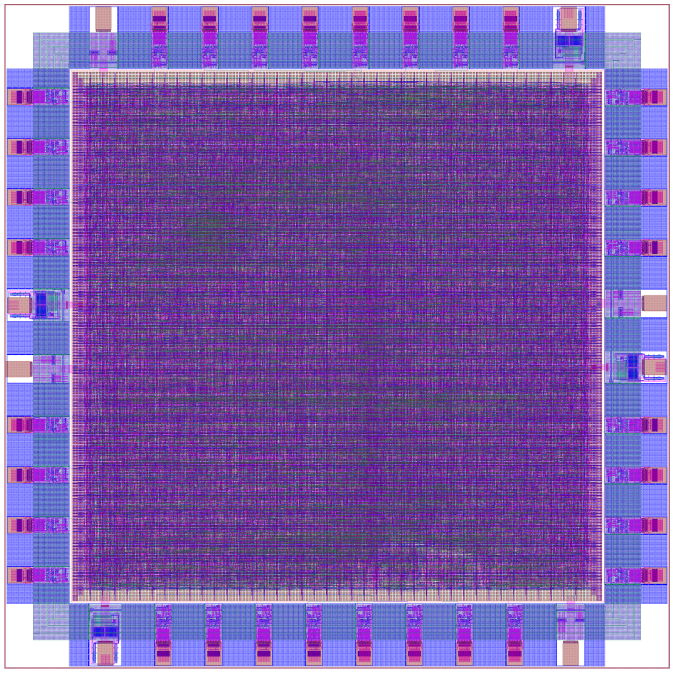


Figura 3.10: Máscara del chip implementado en 180nm de diseño MORPHO1PWL.

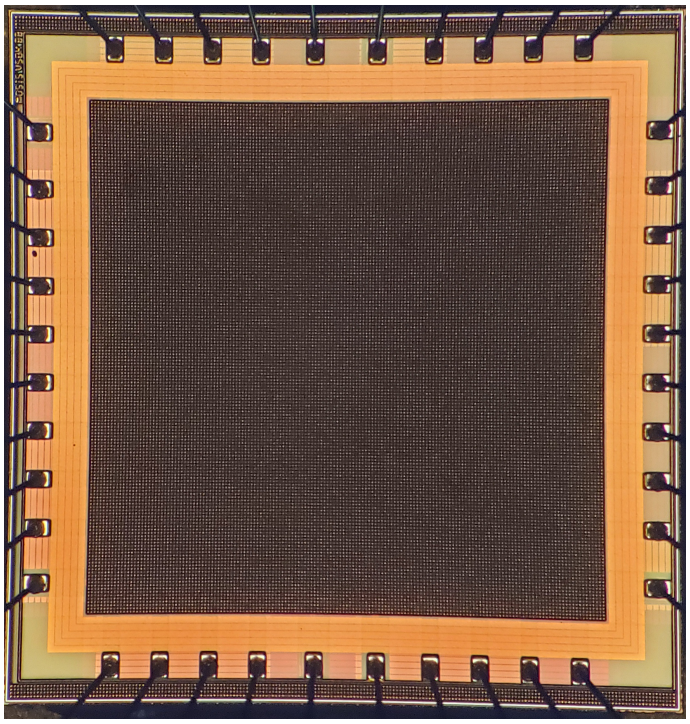


Figura 3.11: Foto del chip fabricado MORPHO1PWL.

ciclos de reloj. Se realizan 10 operaciones para la comparación de cada argumento (5 XOR y 5 AND), la cual se repite por 32 ciclos de reloj. Además se le suma una operación a la relizada en la ALU, lo que da un total de 641 operaciones por ciclo de computo. Esto equivale a un rendimiento del Sistema MORPHO1PWL de

Tabla 3.7: Características del chip MORPHO1PWL en 180nm.

Tecnología	180nm de GlobalFoundries GF7RF
Tamaño del arreglo	56×56
Área de chip	2×2 mm^2
Área de core	$1{,}7 \times 1{,}7$ mm^2
Cantidad de transistores	1,731M
Precisión de entrada	1bit
Precisión de parámetro	1bit
Memoria de imágenes	2
Algoritmo	Linear a tramo
Máxima dimensión de función	5
Velocidad de trabajo	100 MHz
Cantidad de ciclos de reloj por procesamiento	33
Operaciones por segundo	$6,091$ TOPs/s

$641 \times 56 \times 56 \times 100 \times 10^6 \div 33 = 6,091$ TOPs/s, logrando una eficiencia de $406,096$ TOPs/W.

3.3. Procesador morfológico en escala de grises

En esta sección se describe la arquitectura e implementación de un vector de procesadores CNN fabricado en una tecnología de 55nm de bajo consumo, el cual se denomina MORPHO8PWL. Con el objetivo de procesar imágenes en escalas de grises, las celdas implementan la siguiente función:

$$y(k+1) = F\left(\mathbf{x}(k), \mathbf{c}\right) \circ G\left(\mathbf{u}(k), \mathbf{d}\right) \tag{3.3}$$

donde $F, G : \mathbb{R}^5 \longrightarrow \mathbb{R}^1$ son funciones simpliciales lineales a tramos, $\mathbf{u}$ y $\mathbf{x} \in \mathbb{R}^5$ son de entradas de 8 bits que conforman el vector de estados correspondiente a la esfera de influencia (Fig.3.14), $\mathbf{c}$ y $\mathbf{d} \in \mathbb{R}^{32}$ son vectores de 32 parámetros de 3 bits, $\circ$ es una función digital programable, y k es el índice de tiempo discreto.

La Fig.3.12 ilustra el esquemático básico del sistema. El diseño consta de un vector lineal de procesadores *vectorPWL*, una memoria cache de imágenes (*cache_UX*) donde se almacenan U y X, una unidad de configuración *confUnit* con una memoria de funciones donde se alojan F y G, un controlador de procesamiento *controllerPWL* el cual dirige las señales a *vectorPWL*, y un controlador de lectura y otro de escritura, *cacheRD_ctrl* y *cacheWR_ctrl*, que obran de árbitro entre el mundo exterior y *controllerPWL*, para el acceso a la memoria de imágenes. La comunicación I/O es llevada a cabo utilizando un protocolo basado en AMBA AHB-Lite de ARM®, para el cual se implementan otros dos módulos menores, *Decoder* y *Multiplexor*.

3.3.1. Arquitectura

El sistema completo consta de un vector lineal de 64 celdas, el cual procesa por columna las imágenes U y X alojadas en el arreglo de memorias. A medida que las columnas son procesadas, los correspondientes resultados son guardados en el mismo arreglo de memoria, preservando su ubicación relativa en la imagen. Cada celda o elemento de procesamiento PE contiene dos grupos de tres registros de 8 bits, los cuales almacenan temporalmente el valor del pixel de la columna central, más el de la izquierda y la derecha de U y X. El vector de estados se completa tomando los datos correspondientes a las celdas ubicadas arriba y abajo. La región de vecindad

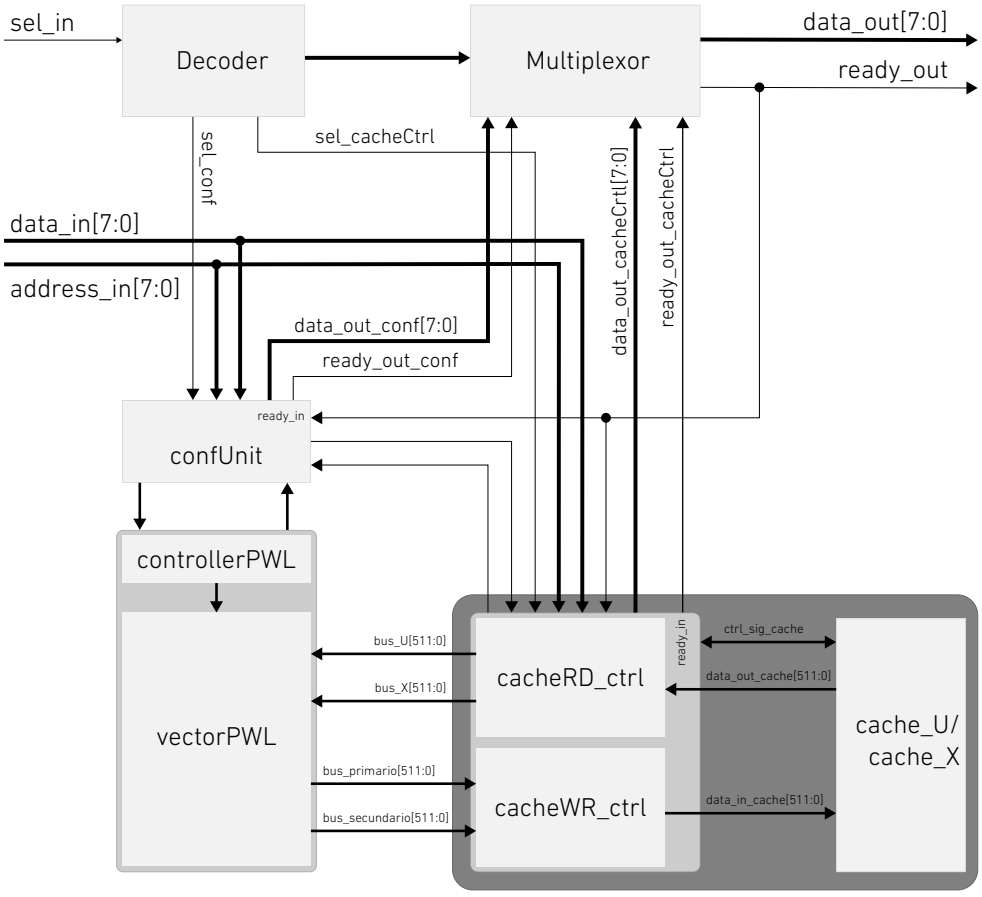


Figura 3.12: Esquemático básico del sistema.

en cada procesador es de cinco elementos, pudiendo ser configurado en forma de equis ($\times$) o ($+$) al igual que el MORPHO1PWL. Las funciones F y G son elegidas de una memoria la cual puede ser cargada externamente. Como las funciones son comunes a todas las celdas, aquellas seleccionadas son alojadas en el controlador, y distribuidas a lo largo del arreglo, evitando así su almacenamiento local en cada PE. Por último, una ALU de ocho funciones es implementada en cada procesador con el fin de computar $F \circ G$. Las unidades *cacheRD_ctrl* y *cacheWR_ctrl* arbitran entre el controlador de procesamiento y el acceso externo. Para la escritura externa se utiliza un registro de desplazamiento el cual se carga en serie con el valor de una columna completa, y luego esta se escribe en la memoria deseada. Por otra parte, para la lectura de una imagen, otro registro de desplazamiento se carga con una columna y se lee externamente en serie.

A continuación se detalla la arquitectura del sistema empezando por una descripción estructural y de funcionamiento de un PE. Luego se describe el vector que los agrupa, para pasar, por último, a las unidades de configuración y control de todo el procesador.

La estructura de un PE es mostrada en la Fig.3.13. Cada procesador computa dos funciones usando cinco valores de 8 bit de $\mathbf{u}$ para G, y otros cinco de $\mathbf{x}$ para F, ambos provenientes de la memoria *cache*; los cuales conforman la esfera de influencia. Para empezar, los elementos de la izquierda, centro y derecha, son obtenidos por cada PE a través de la lectura de tres columnas, para luego ser almacenados localmente en los registros *regU_L*, *regU_C* y *regU_R*, y *regX_L*, *regX_C* y *regX_R* de las respectivas unidades de codificación temporal (TEU). Cada grupo de tres registros ingresa a un set de tres comparadores donde se comparan con una rampa digital, generando tres señales de un bit donde el valor esta codificado en tiempo (señal PWM). Estas tres señales junto con otras dos provenientes de los registros centrales de los procesadores de arriba y abajo, terminan de conformar una palabra de 5 bits, arg, que identifica en tiempo los vértices del símplice dado por una esfera de influencia de cinco elementos en forma de ($+$). Si se quiere un elemento estructurante en equis, entonces se toman las señales PWM del de la izquierda y derecha de los PE lindantes (Fig.3.14).

Como se ejemplifica en Fig. 3.15, los 5 elementos $\{x_4, x_3, x_2, x_1, x_0\}$ del vecindario se comparan contra una rampa común generando el vector $arg(f) = \{\tilde{x}_4, \tilde{x}_3, \tilde{x}_2, \tilde{x}_1, \tilde{x}_0\}$.

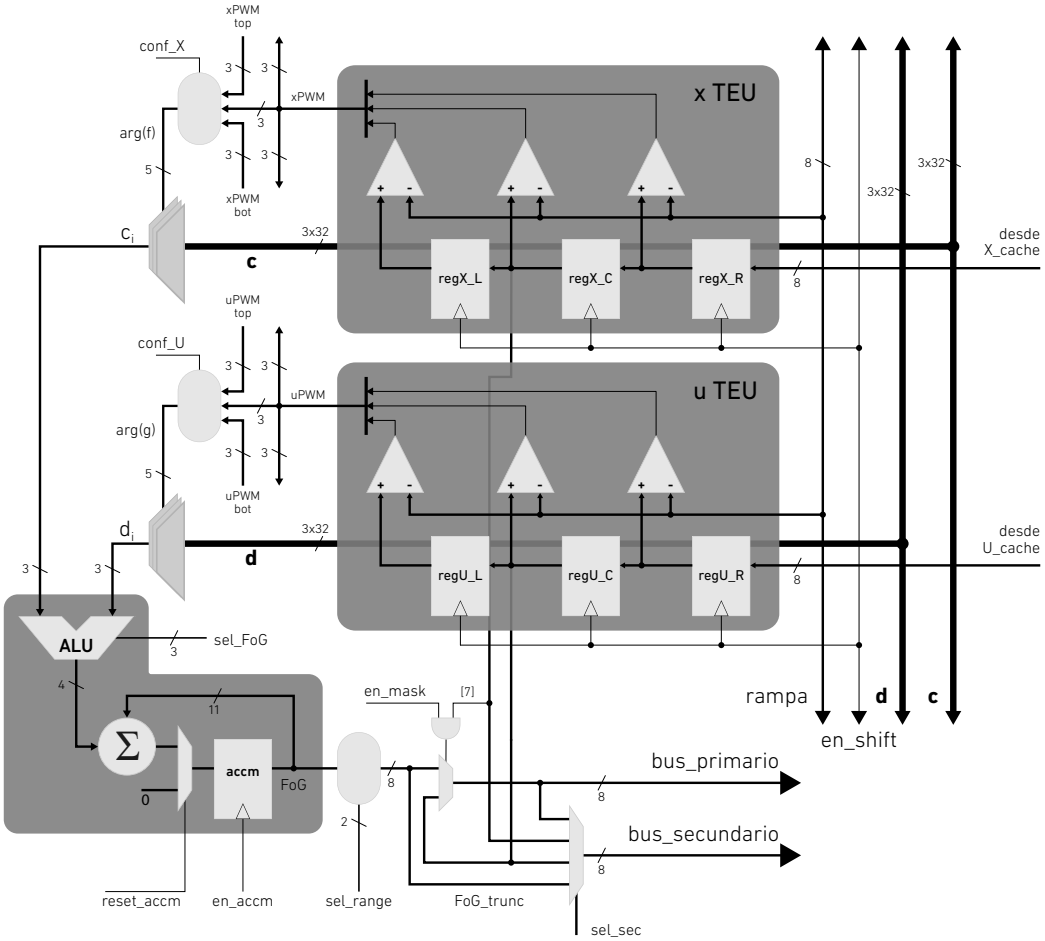


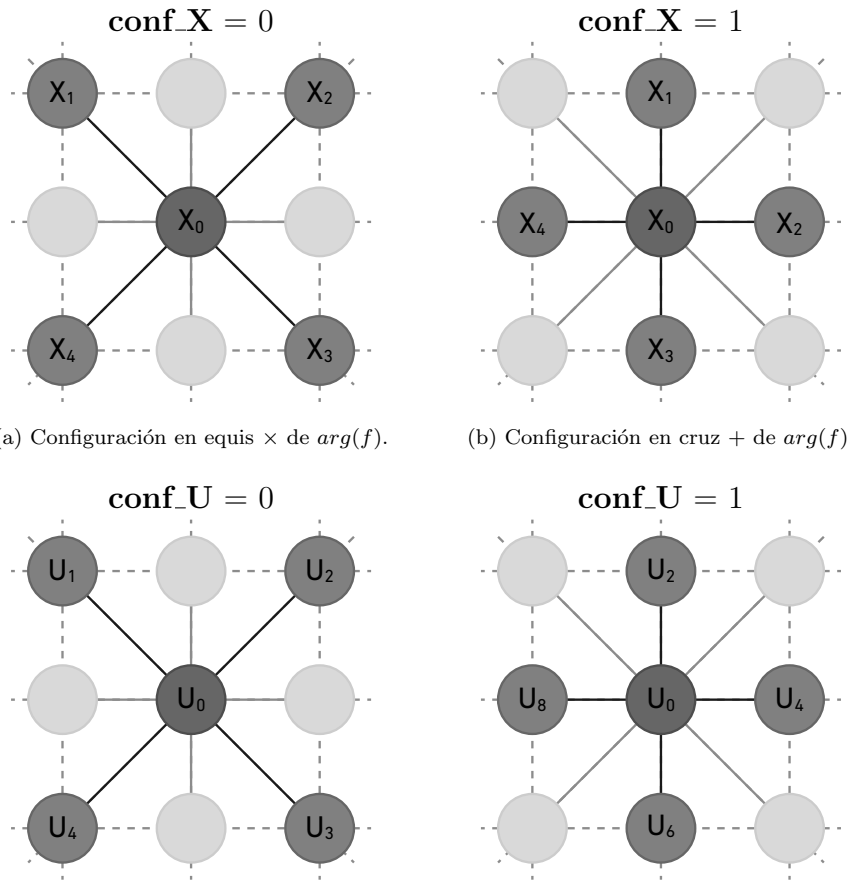
Figura 3.13: Esquema del elemento de procesamiento.

Al inicio de un ciclo de procesamiento, $arg(f)$ es igual a 11111, y sus bits van cambiando a 0 medida que la rampa incrementa y es superior a x_i. Una vez que la rampa llega a su fin, $arg(f)$ se encuentra en el estado 00000, y la función F es calculada:

$$F\left(\mathbf{x}(k), \mathbf{c}\right) = \sum_{i=0}^{5} \mu_i \hat{c}_i \tag{3.4}$$

siendo $\{\mu_0, \mu_1, \mu_2, \mu_3, \mu_4, \mu_5\} = \{\hat{x}_0, \hat{x}_1 - \hat{x}_0, \hat{x}_2 - \hat{x}_1, \hat{x}_3 - \hat{x}_2, \hat{x}_4 - \hat{x}_3, 1 - \hat{x}_4\}$, $\{\hat{x}_0, \hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4\} = sort\{x_0, x_1, x_2, x_3, x_4\}$ el vector de estado ordenado, y los $\hat{c}_i$ los parámetros direccionados por $arg(f)$. Los valores iniciales y finales de la rampa, así como el valor de su incremento, pueden también ser programados externamente sobre el controlador, habilitando funcionalidades al procesador.

La señal *arg*, a través de un multiplexor, selecciona uno de los 32 parámetros de 3 bits correspondiente a la función elegida. La salida de los dos multiplexores,



(a) Configuración en equis × de $arg(f)$.

(b) Configuración en cruz + de $arg(f)$.

(c) Configuración en equis × de $arg(g)$.

(d) Configuración en cruz + de $arg(g)$.

Figura 3.14: Configuraciones posibles para la formación del argumento en el calculo de las funciones G y F.

c_i y d_i, ingresan una unidad lógica aritmética (ALU) donde se operan produciendo una salida de 4 bits FoG_i con los posibles siguientes valores: $c_i + d_i$, $\mathsf{MAX}(c_i, d_i)$, $\mathsf{MIN}(c_i, d_i)$, $|c_i - d_i|$, o las funciones binarias $\mathsf{AND}(c_i, d_i)$, $\mathsf{OR}(c_i, d_i)$, $\mathsf{NOR}(c_i, d_i)$ y $\mathsf{XOR}(c_i, d_i)$ (Tab.3.8). Posteriormente, FoG_i es acumulado en un registro de 11 bits $ACCM$ que, una vez finalizada la rampa, constituye el resultado de la función compuesta FoG. Con la señal *sel_range* se procede a truncar el valor FoG, y la señal resultante de 8 bits, *FoG_trunc*, es leída y guardada (Tab.3.9).

Cada PE posee dos buses de salida denominados *bus_primario* y *bus_secundario*, permitiendo colocar varios procesadores en cascada. El *bus_primario* puede tomar el valor del resultado de la función truncado, o el valor central $\mathsf{U_0}$ si el modo de enmascaramiento esta habilitado y el valor bit mas significativo del valor central $\mathsf{X_0}$ es 1 ($\mathsf{AND}(en\_mask, X_0[7])$). El *bus_secundario* puede tomar el valor de los dos registros del centro, *FoG_trunc* o copiar el *bus_primario* (Tab.3.10). La ubicación en

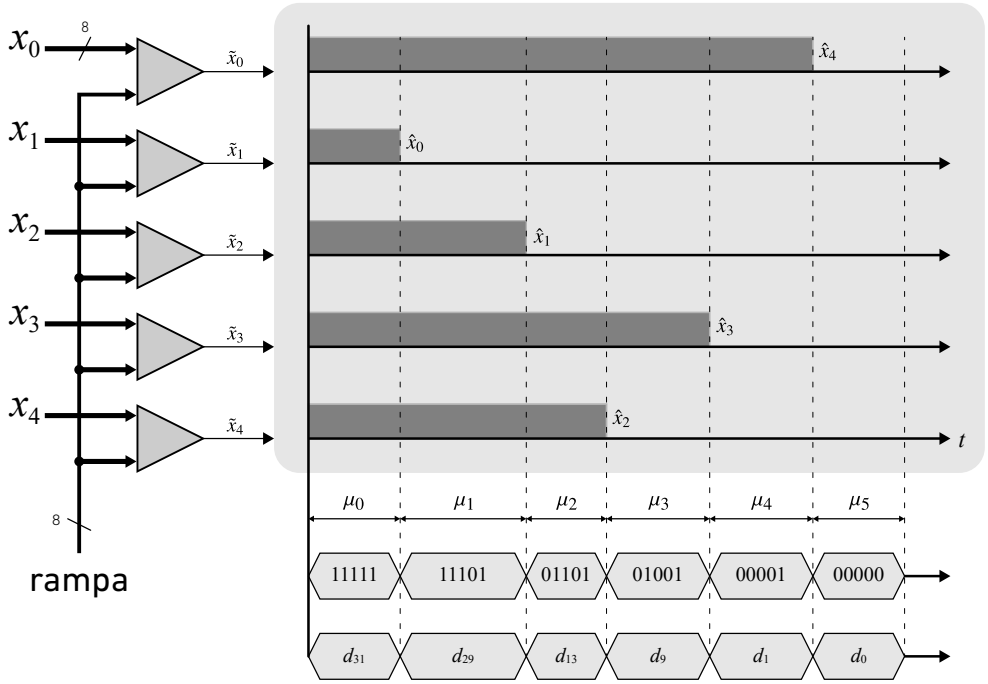


Figura 3.15: Actividad en un ciclo de procesamiento, mostrando el ordenamiento en tiempo del vector de estados y el direccionamiento de los parámetros. En este ejemplo

Tabla 3.8: Selección de función en la ALU.

sel_FoG	Función
0	AND(c_i, d_i)
1	XOR(c_i, d_i)
2	OR(c_i, d_i)
3	NOR(c_i, d_i)
4	MAX(c_i, d_i)
5	MIN(c_i, d_i)
6	$c_i + d_i$
7	$\lvert c_i - d_i \rvert$

Tabla 3.9: Selección rango de *FoG* con *sel_range*.

sel_range	*FoG_trunc*
0	FoG[7:0]
1	FoG[8:1]
2	FoG[9:2]
3	FoG[10:3]

memoria de los valores a escribir corresponde la columna central procesada. Como último paso para iniciar el siguiente cálculo, se desplazan los valores de los registros

de entrada hacia la izquierda y se leen las dos columnas nuevas en los registros de la derecha.

Tabla 3.10: Selección de bus secundario.

sel_sec	bus secundario
0	valor de *regX_C*
1	valor de *regU_C*
2	*FoG_trunc*
3	bus primario

Cada procesador se instancia en un arreglo lineal con interconectividad local de distancia uno, conformando de esta manera el vector de procesamiento *vectorPWL*, tal como se muestra en la Fig.3.16. Las señales de entrada *rst_accm*, *en_accm*, $\boldsymbol{c}$, $\boldsymbol{d}$, *rampa*, *sel_FoG*, *conf_U*, *conf_X*, *sel_sec*, *sel_range*, *en_mask* provenientes del modulo de control, ingresan y se distribuyen globalmente a todos los procesadores para su control y configuración. Los vectores de datos *vector_bus_U* y *vector_bus_X* de $8 \times 64 = 512$ bits de ancho ingresan en palabras de 8 bits a los 64 procesadores como datos de entradas provenientes de las memorias cache. Por su parte, los buses de salida primario y secundario de cada procesador se concatenan formando dos vectores de tamaño $8 \times 64 = 512$ bits, que son escritos en *U_cache* y *X_cache* respectivamente. Para resolver las condiciones de borde en el primer procesador y en el último, las señales *sel_border* completan las vecindades seleccionando los valores 0, 1 o una copia de la señal PWM del procesador de frontera (Tab.3.11).

Tabla 3.11: Selección de condición de borde para los procesadores de frontera.

sel_border	señal PWM
0	'000'
1	'111'
2/3	copia

El procesamiento llevado a cabo dentro del vector de procesadores se configura a través de la unidad *confUnit*, la cual se encarga de establecer los parámetros de funcionamiento del controlador primario *controllerPWL*, y los controladores de escritura y lectura de memoria cache. La mencionada unidad se encuentra compuesta

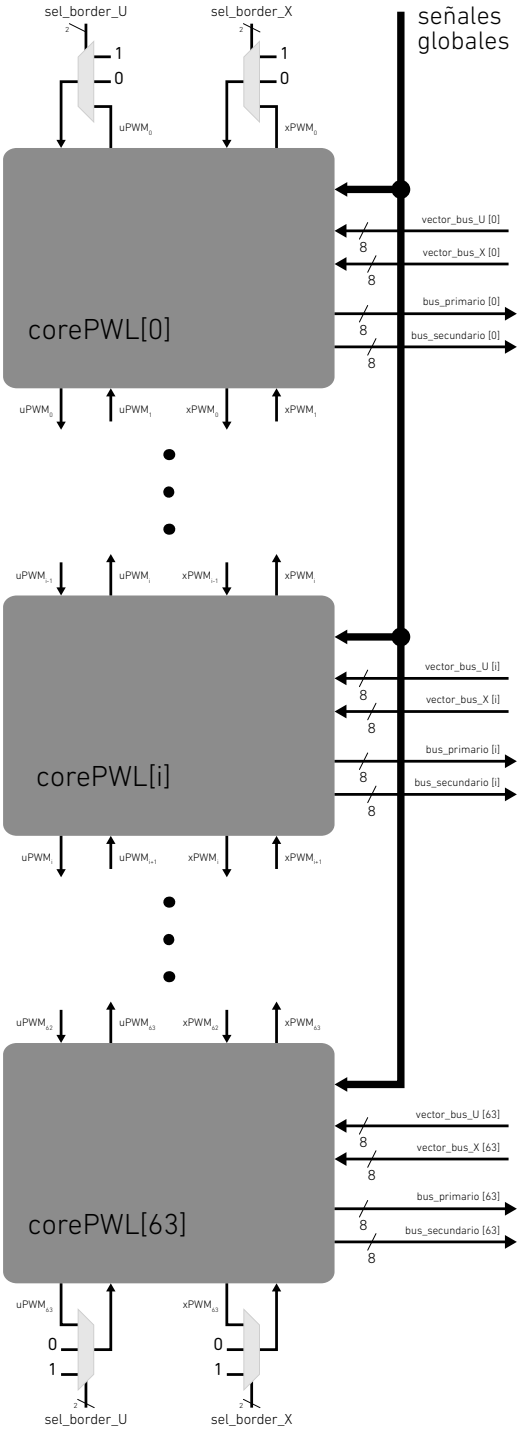


Figura 3.16: Diagrama del vector de procesadores *vectorPWL*.

por una memoria de parámetros de funciones y un arreglo de registros los cuales se detallan a continuación.

El primer grupo de registros constituye la memoria de funciones. Esta permite el almacenamiento de dos conjuntos de 32 parámetros de 3 bits $\mathbf{c}$, c_0 y c_1, para el

computo de F_x; y dos conjuntos de 32 parámetros **d**, d_0 y d_1, también de 3 bits, para el computo de G_u. Para cargar las funciones c_0 y d_0 se deben escribir sus parámetros entre las direcciones 128 y 159 con un dato de entrada con el siguiente formato:

$$data\_in = \{0, 0, d_0[2], d_0[1], d_0[0], c_0[2], c_0[1], c_0[0]\} \tag{3.5}$$

Para almacenar c_1 y d_1 se escribe en las direcciones 160 y 191 con el siguiente formato:

$$data\_in = \{0, 0, d_1[2], d_1[1], d_1[0], c_1[2], c_1[1], c_1[0]\} \tag{3.6}$$

Además, la memoria ofrece 6 funciones de fábrica comunes para **c** y **d** (Tab.3.12): ZEROS (todos los parámetros son 0 dando como resultado la función cero), COPY (copia el valor central de la celda, o de posición cero dentro del elemento estructurante), NOT (genera el valor complementario al valor central de la celda), MIN (copia el valor mínimo de la vecindad), MAX (calcula el máximo de la vecindad), MEDIAN (calcula la mediana de la vecindad).

Tabla 3.12: Funciones disponibles en memoria.

sel_Gu / *sel_Fx*	Función
0	ZEROS
1	COPY
2	NOT
3	MIN
4	MAX
5	MEDIAN
6	Gu_0 / Fx_0
7	Gu_1 / Fx_1

El segundo grupo de registros se encarga de la configuración y control de procesamiento (Tab.B.2), y se ubica entre las direcciones de memoria 192 y 203 (Tab.B.3). Si el sistema se encuentra ocupado procesando y se quiere escribir en cualquiera de las direcciones excepto la 192, la señal de respuesta *ready_out* es 0, obligando al usuario esperar o parar previamente su funcionamiento. Cuando se lee la dirección 192, los dos bits mas significativos, 7 y 6 representan señal de bandera *is_working_PWL* y *is_working_cache* respectivamente. Una vez introducidos los registros que modifican el comportamiento del sistema se pasa a describir el controlador de los elementos de

procesamiento.

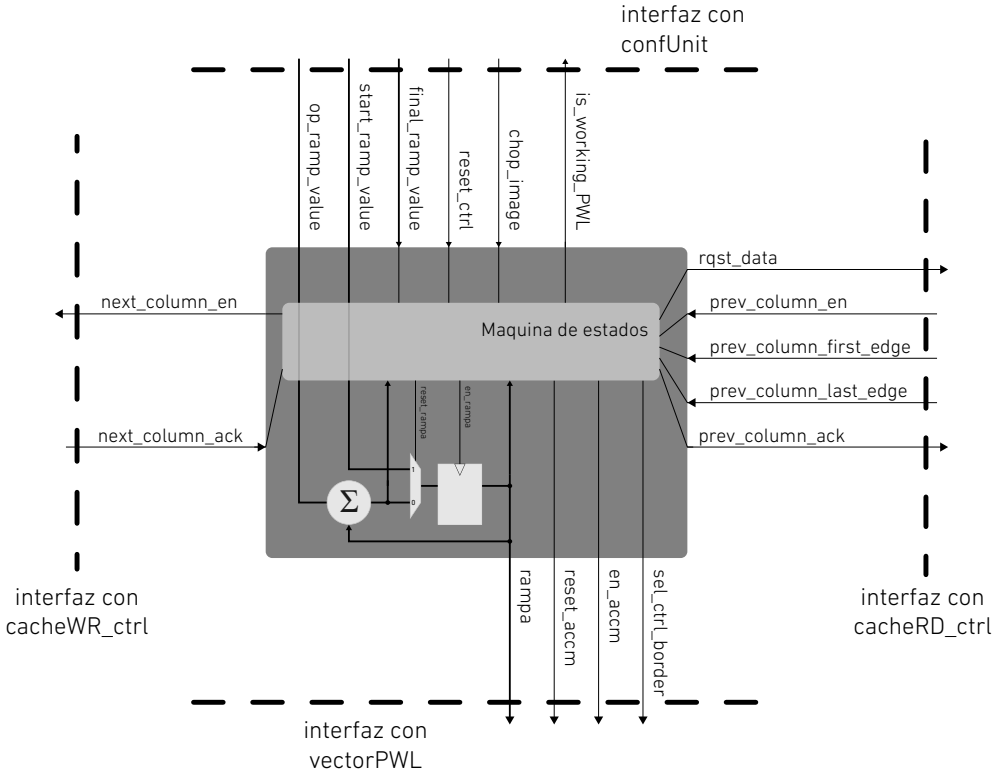


Figura 3.17: Esquemático simple del controlador PWL y sus conexiones con los controladores de la memoria cache, y la unidad de configuración.

En la Fig.3.18 se muestra un diagrama de estados de la máquina implementada en el controlador *controllerPWL*, para el manejo y temporizado de la señales que ingresan al arreglo de procesadores. El controlador utiliza una comunicación asíncrona con los controladores de memoria para la carga de los datos de entrada y el almacenamiento de los vectores de resultados, tal como se ilustra en Fig.3.17. Por su parte, recibe de la unidad de configuración los parámetros de rampa y de opción para rellenado de la imagen (*chop_image*). A continuación se describe brevemente su flujo de funcionamiento y las señales más importantes intervinientes:

Cuando el registro *rst_ctrl* es pulsado externamente, la máquina de estados se reinicia en el estado **WAITING_PREV_COL**. Sin embargo, para dar comienzo a todo el sistema es necesario setear *start_reg* en uno, de modo tal que los controladores *cacheRD_ctrl* y *cacheWR_ctrl* ingresen al modo procesamiento, y particularmente el de lectura, lea la primer columna.

Inicialmente *cacheRD_ctrl* le comunica a *controllerPWL*, con *prev_column_en* en

1, que existe una columna nueva de datos que han sido leídos y deben ser procesados. El controlador le envía una señal de lectura (*prev_column_ack*=1) como respuesta y carga los datos nuevos al arreglo de celdas con *en_shift*. Si el dato cargado es la primer columna (*prev_column_first_edge*=1), con la señal *rqst_data* le pide al controlador de lectura una nueva columna para cargar en el registro del medio, y se pasa un estado especial denominado **LOAD_MID_COL**. En **LOAD_MID_COL** se carga la columna del medio, se genera un nuevo pedido y se regresa a **WAITING_PREV_COL** para esperar el dato columna que se alojará en el registro derecho de los *corePWL*.

Al recibir la tercer columna, se da reinicio al contador de la señal *rampa* que se propaga a las celdas, dando lugar al estado **PROCESSING**, donde la rampa incrementa su valor por cada ciclo de reloj con un paso igual a *op_ramp_value*. Cuando la rampa llega a su fin, *ramp_end*=1 (Ecu. 3.7), el vector de resultados se encuentra listo, y se genera una petición de guardado a *cacheWR_ctrl* llevando la señal *next_column_en* a 1, y otro pedido de una nueva columna de datos de entrada a *cacheRD_ctrl* a través de *rqst_data*. Al ciclo siguiente la máquina regresa a **WAITING_PREV_COL** esperando ser informado de que el resultado ha podido ser almacenado en la memoria cache (*next_column_ack*=1); para luego volver a la etapa inicial donde se esperan datos nuevos.

$$ramp\_end \leftarrow (rampa \leq final\_ramp\_value) \cap (rampa + op\_ramp\_value > final\_ramp\_value) \quad (3.7)$$

Dicho ciclo se repite hasta que el controlador recibe *prev_column_last_edge* en uno, en señal de que la ultima columna de datos ha sido leída. Se procede a registrar la nueva entrada en los registros de la derecha dentro de los procesadores y se inicia un nuevo procesamiento. El resultado luego es almacenado siguiendo el protocolo anteriormente detallado, y de esta manera finaliza el procesamiento de teniendo como entrada las dos imágenes alojadas en *cache_UX*.

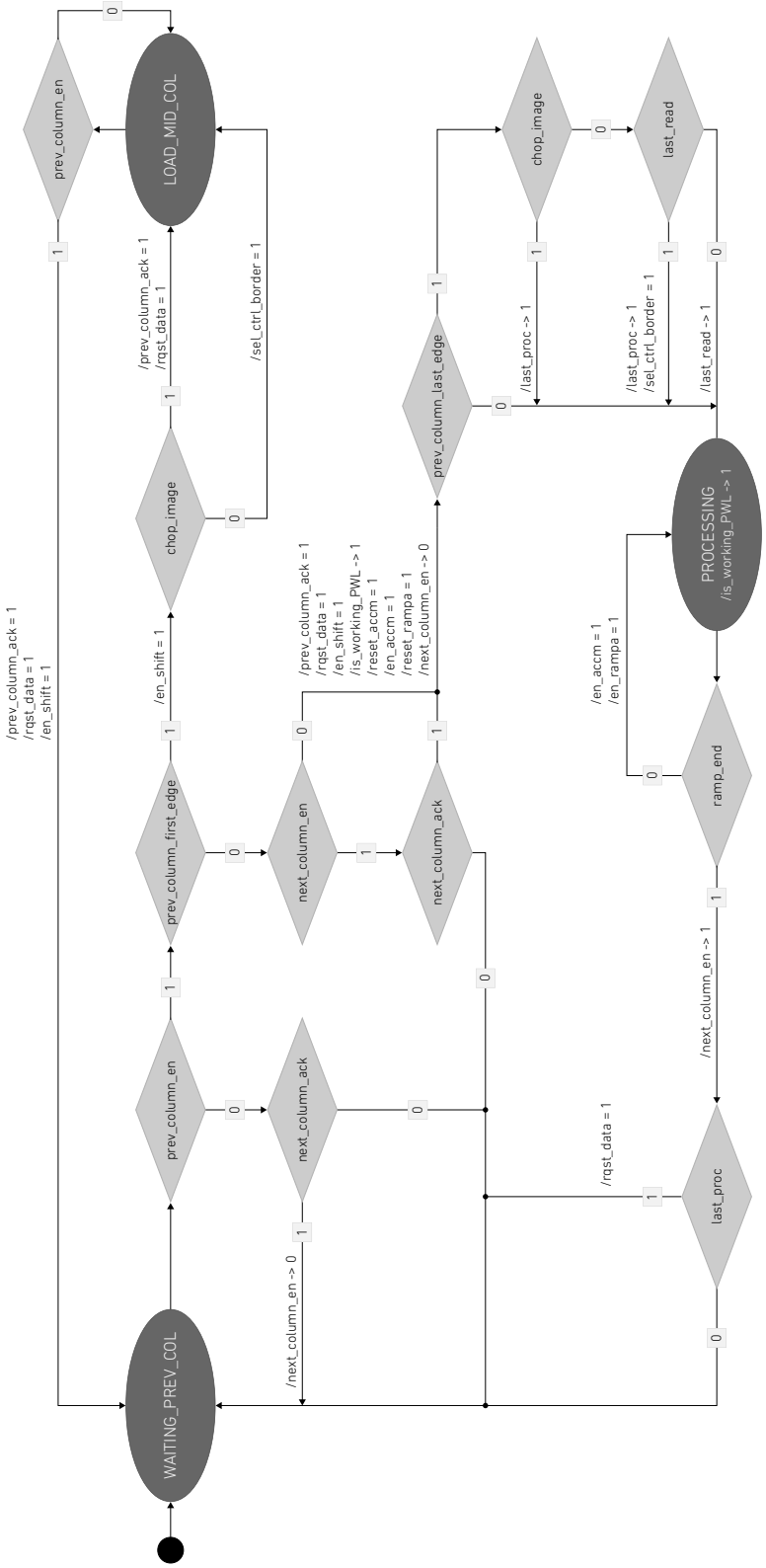


Figura 3.18: Diagrama de estados de la máquina implementada en *controllerPWL*. Las elipses representan los estados, los rombos evaluaciones condicionales, los cuadrados las distintas condiciones, y las flechas transiciones. En el controlador se implementa dos registros auxiliares, *last_read* y *last_proc*, como señales bandera de la ultima columna de datos de entrada y el último procesamiento realizado respectivamente.

Memoria de imágenes y sus controladores

Para el almacenamiento de los datos de entrada y de salida, se implementó una memoria de tamaño 32Kb y 512 bits de bus de datos, que permite guardar dos imágenes, U y X, de 64×64 pixeles de 8 bits de precisión. La memoria utiliza como protocolo de comunicación interno uno de dos fases, semejante el AMBA AHB-Lite, donde en el primer ciclo de reloj se presenta la dirección, y en el segundo el dato de entrada o de salida.

Las imágenes se acceden de a columnas de 64 pixeles a través de registros de desplazamiento manejados por los controladores de escritura y de lectura, que administran su acceso desde el exterior y durante los ciclos de procesamiento. La imagen U se carga externamente escribiendo 64 datos en serie a la dirección de la columna deseada, mapeadas entre las direcciones 0 a 63. Para su lectura, se direcciona nuevamente la columna y se lee los 64 datos en serie. Por su parte, las columnas de X se encuentran lógicamente ubicadas entre las direcciones 64 y 127.

Para que ambos controladores puedan tener acceso a la memoria se implementa la unidad denominada *cache arbiter* en Fig.3.19, que multiplexa la señales de control de cada uno evitando colisiones, dando siempre prioridad al controlador de escritura. Es decir, cuando *cacheWR_ctrl* escribe en *cache_UX*, la señales de disponibilidad *ready_cacheRD* se va a 0 indicando que el controlador de lectura no tiene el bus disponible para acceder a la memoria. A continuación se describen ambos controladores, su interacción entre ellos, con la memoria, el exterior del sistema y el *controllerPWL*.

El controlador de escritura de la memoria caché está compuesto por una máquina de 6 estados y un contador generador de direcciones, y un registro de desplazamiento de 8 bits de ancho y 64 de palabras de profundidad (512 bits en total). En la Fig.3.20 se muestra el flujo de funcionamiento de la máquina de estado. Si se desea ingresar una columna desde el exterior, el controlador entra en un modo en el cual se escriben los datos en serie en el registro de desplazamiento, y una vez que se cargó todo el vector, el controlador ejecuta un proceso de escritura en la memoria. El otro modo corresponde al de procesamiento; se inicia con el registro *start_reg* en uno, ahi se reinicia un contador de direcciones en el valor alojado en *initial_address_wr*,

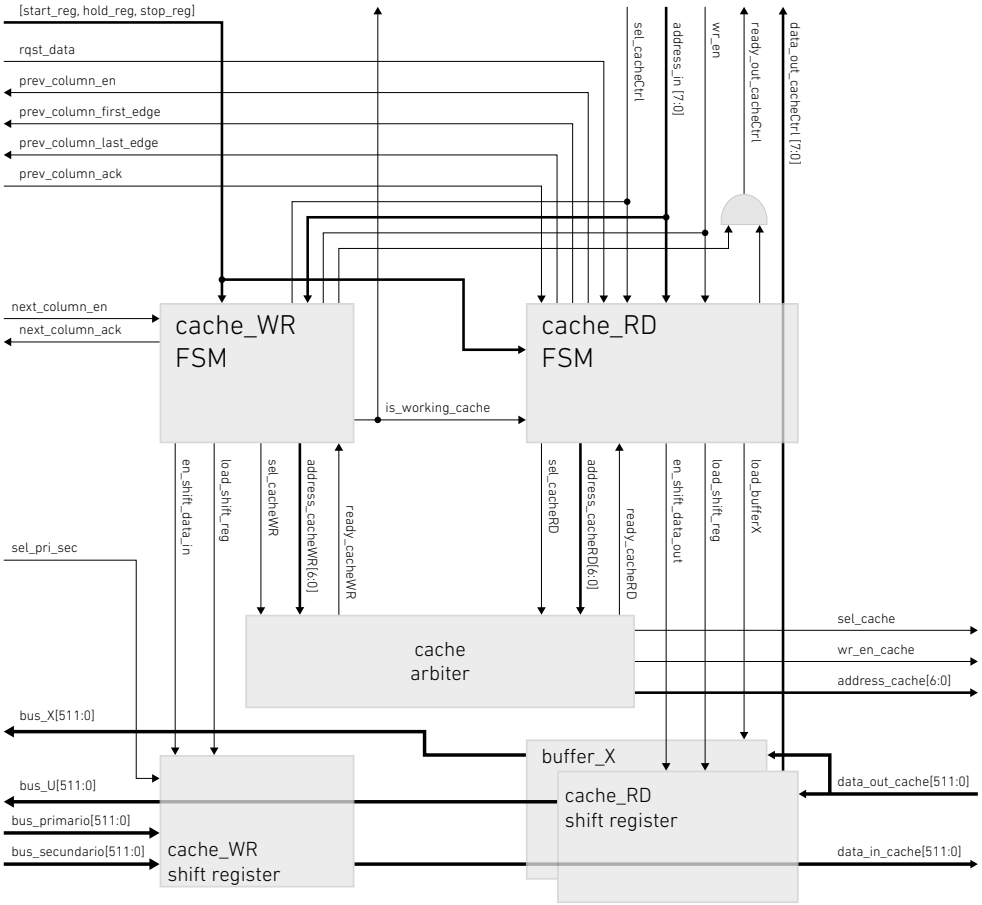


Figura 3.19: Esquemático del módulo del controlador de la memoria de imágenes.

y cada vez que recibe la señal *next_column_en* de *controllerPWL*, carga el vector de resultados proveniente de los procesadores en el registro de desplazamiento, sea *bus_primario* o *bus_secundario* según *sel_pri_sec* indique. Luego lo escribe en memoria con la dirección dada por la concatenación del contador y *sel_cacheXU* en el bit mas significativo, e incrementa la dirección con el valor *step_address_wr*. A continuación se detalla los distintos estados intervinientes en cada modo:

- **IDLE_ST**: estado inicial en el cual se inicia la máquina luego de que se realiza un reinicio general del sistema, o que de que *stop_reg* sea llevado a uno. Si la memoria esta siendo accedida externamente para su escritura, *sel_cache* y la señal *wr_en_in* en uno, se pasa al estado **SHIFT_DATA_ST** y se indica que en el próximo ciclo de reloj se debe empezar a cargar en serie el registro de desplazamiento llevando la señal *en_shift_data_in* a 1. En el caso que se inicie el modo procesamiento, se reinicia el contador de direcciones,

se pone *ready_out* en 0 evitando el acceso externo, y se transiciona al estado **RECEIVE_RESULT_ST**.

- **SHIFT_DATA_ST**: si se mantiene la condición de escritura iniciada en **IDLE_ST**, se mantiene *en_shift_data_in* en 1. Caso contrario, se supone que el usuario ya cargo todo el vector y se pasa al estado **WRITE_DATA_IN_ST**.

- **WRITE_DATA_IN_ST**: se selecciona la memoria y se inicia un proceso de escritura, utilizando como dirección el último valor ingresado a través de *address_in* durante la carga del registro de desplazamiento. Por defecto se pasa **IDLE_ST**, pero si se da nuevamente una condición de escritura como en **IDLE_ST**, *en_shift_data_in* va a 1, y **SHIFTING_ST** será el siguiente estado.

- **RECEIVE_RESULT_ST**: se informa al *controllerPWL* que se esta en modo procesamiento, y se espera por la señal *next_column_en* para habilitar el guardado del vector de resultados. De ser así, la señal *load_shift_reg* se lleva a uno, se responde al controlador PWL con *next_column_ack*, y si la memoria está disponible (ready_in_mem igual a uno), se inicia el proceso de escritura de la misma, poniendo en el bus de dirección el resultado de la concatenación del contador y *sel_cacheXU*. Como último paso se incrementa el contador y se transiciona a **DATA_ST**. En el caso que la memoria no esté lista, se espera que lo esté pasando al estado **ADDRESS_ST**.

- **ADDRESS_ST**: se espera que la memoria este disponible para iniciar el proceso de escritura al igual que en **RECEIVE_RESULT_ST**, y se avanza a **DATA_ST**.

- **DATA_ST**: representa la fase de datos del protocolo de acceso a la memoria, y si esta se encuentra disponible, se vuelve a **RECEIVE_RESULT_ST**.

Por su parte, el controlador de lectura de la memoria caché se encuentra compuesto por una máquina de 7 estados y un contador generador de direcciones, y un registro de desplazamiento y un set de registros ambos de 8 bits de ancho y 64 de palabras de profundidad (512 bits en total). Si se desea leer una columna desde el exterior, el controlador accede al espacio en memoria seleccionado por el bus de

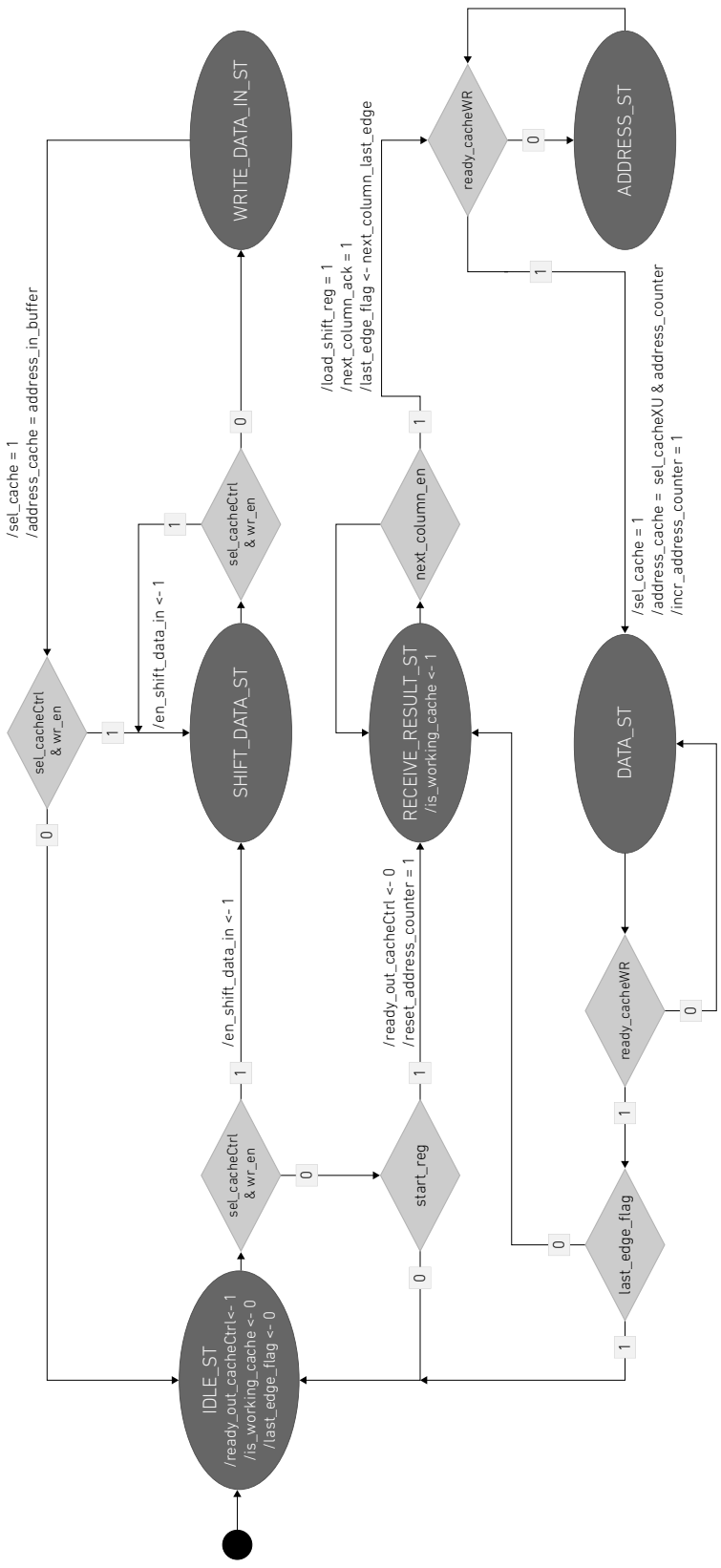


Figura 3.20: Diagrama de estados de la máquina implementada en *cacheWR_ctrl*.

direcciones, carga el dato leído en el registro de desplazamiento, y este es leído en serie sucesivamente. Al igual que el de escritura, el controlador de lectura también posee un modo de procesamiento, el cual se inicia con el registro *start_reg* en uno. Se reinicia el contador en *initial_address_rd*, y cada vez que se recibe la señal de peticion de datos *rqst_data* por parte de *controllerPWL*, lee el la columna correspondiente a la imagen U y su respectiva en X, e incrementa la dirección en *step_address_rd*. Una vez que se la dirección dada llega a el valor dado por *final_address_rd*, se lee el set de columnas, y se informa al controlador de procesamiento que es el último dato a través de *prev_column_last_edge*. A continuación se detalla los distintos estados intervinientes en cada modo, los cuales se muestran en la Fig.3.21:

- **IDLE_ST**: estado inicial en el cual se inicia la máquina luego de que se realiza un reinicio general del sistema, o que de que *stop˙reg* sea llevado a uno. Si la memoria esta siendo accedida externamente para su lectura, *sel_cache* y la señal de escritura *wr_en_in* en 0, se selecciona la memoria y se inicia un proceso de lectura, utilizando como dirección el valor de *address_in*. Si la memoria esta disponible, se pasa al estado **READ_DATA_OUT_ST**, y de no ser asi se lleva el *ready_out_cacheCtrl* a cero y se permanece en el estado. En el caso de que la memoria no es accedida, pero *start_reg* esta en uno, se reinicia el contador de direcciones y se pasa al estado **WAIT_RQST_ST**.

- **READ_DATA_OUT_ST**: si la memoria se encuentra disponible, el dato leído es cargado en el registro de desplazamiento llevando *load_shift_reg* a 1. Si el controlador sigue siendo accedido para su lectura, se pasa al estado **SHIFT_DATA_ST**.

- **SHIFT_DATA_ST**: mientras el controlador es accedido para su lectura, se lee en serie el registro de desplazamiento, y una vez terminado el acceso, se retorna al estado **IDLE_ST**.

- **WAIT_RQST_ST**: si existe una petición de datos de entradas por parte del controlador de procesamiento, se accede a la memoria para la lectura de U. Si se encuentra disponible, pasa al estado **ADDR_X_ST** a la espera de la columna U y accediendo a la de X; caso contrario, el próximo estado es **ADDR_U_ST**

donde se renueva el acceso al sector de U ya que la memoria se encontraba ocupada.

- **ADDR_U_ST**: estado en el que se accede al sector U de la memoria. Si la cache se encuentra disponible, se pasa al estado **ADDR_X_ST**.
- **ADDR_X_ST**: se accede al sector de X de la memoria y se espera el dato de la columna U. Si la cache se encuentra disponible, se registra el dato de salida y se pasa al estado **DATA_X_ST**.
- **DATA_X_ST**: si la memoria se está disponible, se registra la columna X en el set de registros *buffer_columnX*, se le avisa a *controllerPWL* que el próximo ciclo de reloj los nuevos datos de entradas van a estar disponibles seteando la señal *prev_column_en* en uno, y se vuelve al estado **WAIT_RQST_ST**.

Además en paralelo, durante los estados de inicio de lectura del par de columnas, **READ_DATA_OUT_ST** y **ADDR_U_ST**, se evalúa si *prev_column_en* se encuentra en alto y si el *controllerPWL* respondió en reciprocidad con *prev_column_ack* en uno. En caso positivo, se lleva *prev_column_en* a cero, y se copia el valor del contador de direcciones en un registro de 6 bits denominado *address_buffer*. Este se compara *initial_address_rd* y *final_address_rd* para la generación de *prev_column_first_edge* y *prev_column_last_edge* respectivamente.

3.3.2. Funcionamiento del sistema

A continuación se describe el funcionamiento del sistema e ilustrando como se realizan tareas básicas de escritura/lectura de la imágenes y configuración de los controladores para su procesamiento.

Previo a realizar cualquier tarea, se debe pulsar la señal de *reset* para el reinicio de las máquinas de estado. A continuación se explica los procedimientos de escritura y lectura de las imágenes, y de configuración de los controladores para el inicio del procesamiento.

El almacenamiento de una imagen de 8 bits de 64×64 píxeles en el sector U o X de la memoria cache, se realiza a través de la carga de sus 64 columnas, sin que necesariamente estas deban ser escritas en orden. Para ingresar un vector

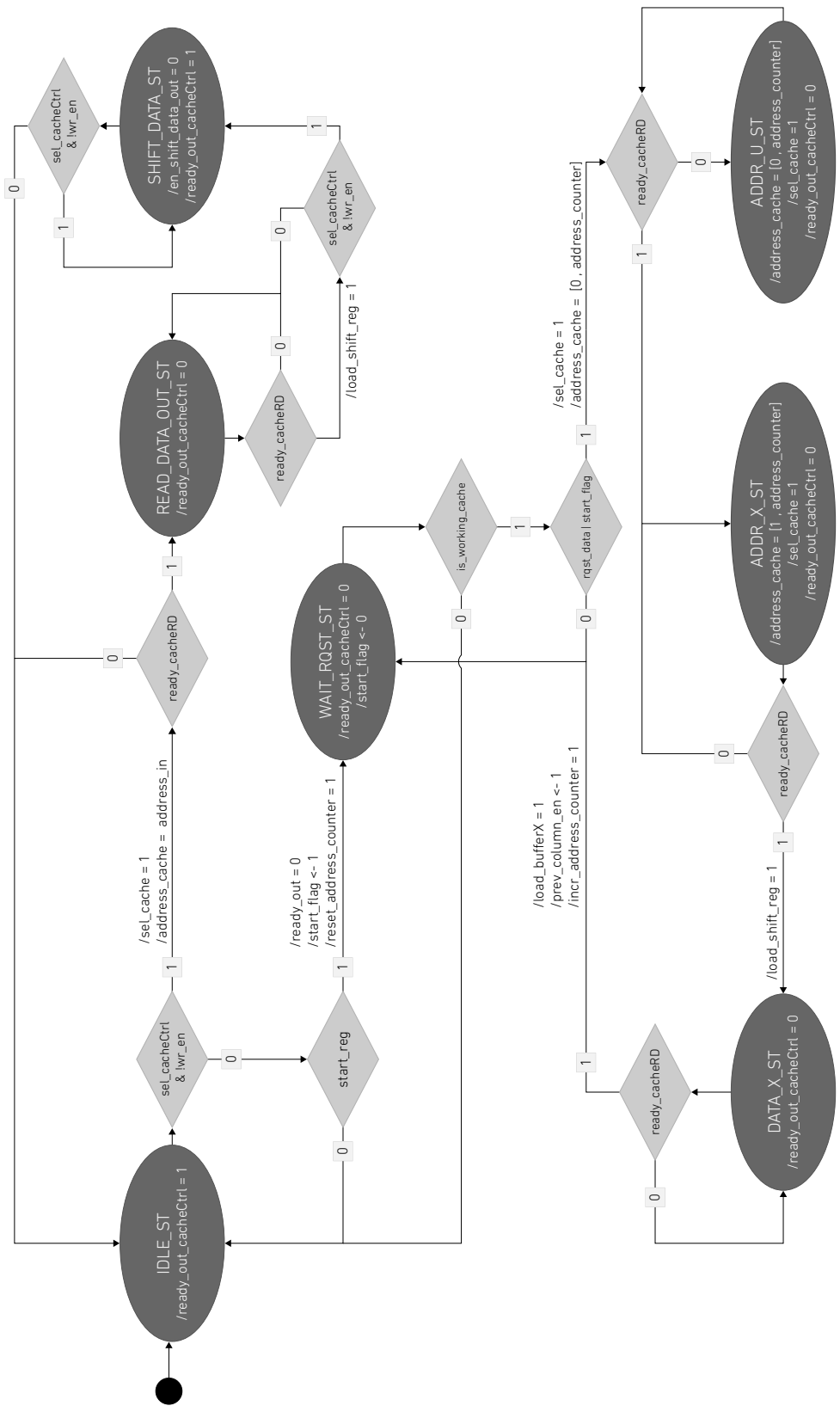


Figura 3.21: Diagrama de estados de la máquina implementada en *cacheRD_ctrl*.

Tabla 3.13: Mapeo lógico de las direcciones para escritura y lectura.

Dirección	Sentido	Descripción
0-63	Escritura/ Lectura	Acceso serial de las columnas de la imagen U en la memoria cache.
64-127	Escritura/ Lectura	Acceso serial de las columnas de la imagen X en la memoria cache.
128-191	Escritura/ Lectura	Memoria de funciones. En el rango 128-159 se accede a los 32 parámetros que conforman las funciones *Gu_0* y *Fx_0*. En el rango 160-191 se accede a los 32 parámetros que conforman las funciones *Gu_1* y *Fx_1*.
192-203	Escritura/ Lectura	Acceso los registros de controlo y procesamiento.

columna de datos primero se debe habilitar las señal *sel_in* con *wr_en_in* en uno, y direccionar con *address_in* la ubicación de la columna que se desea cargar. Si la señal de *ready_out* se encuentra en alto, se cargan sucesivamente los valores de las filas en orden ascendente (Fig.3.22).

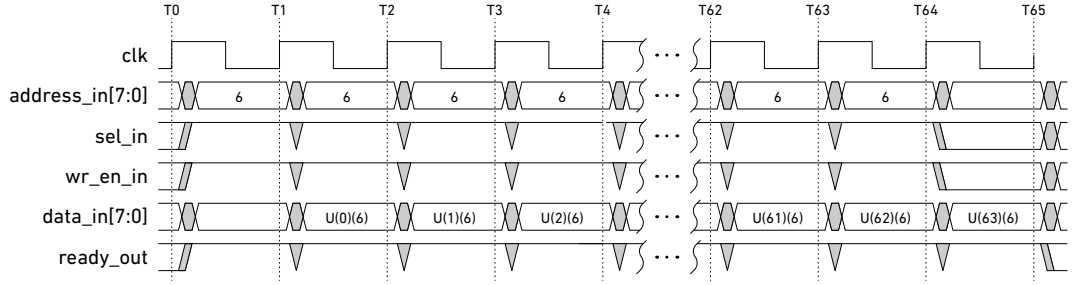


Figura 3.22: Temporizado de las señales del protocolo de comunicación del sistema para escribir un vector de datos en la columna 6 de la imagen U.

Así mismo, la lectura de las imágenes también se realiza a través de la lectura de sus columnas. Para ello, se empieza habilitando *sel_in*, con *wr_en_in* en cero y el valor de la columna que se desea acceder en *address_in*. Si el acceso es valido, entonces se procede a la lectura en serie de las 64 filas, tal como se ejemplifica en la Fig.3.23 para la columna 59 de la imagen X.

Para realiza un ciclo completo de procesamiento utilizando las dos imágenes, U y X, es necesario cargar nuevas funciones, si así se desea (Fig.3.24), y escribir los registros de configuración. Por último, se debe pulsar *rst_ctrl* para reiniciar *controllerPWL*, y setear *start_reg* en 1 comenzar a procesar.

En la Fig.3.25(a) se muestra el canal verde de una imagen RGB de 8 bits guar-

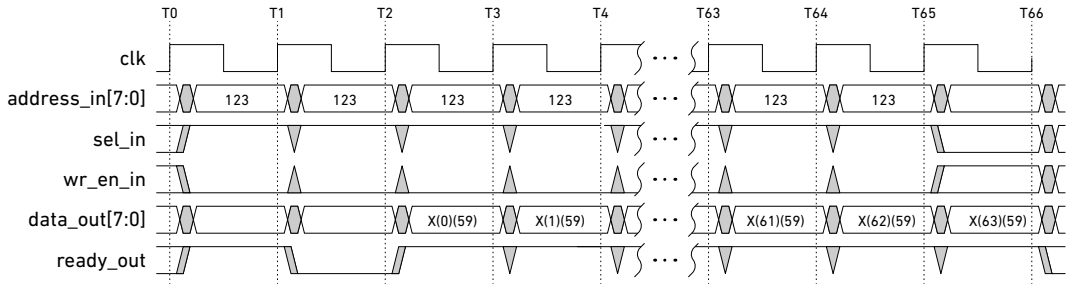


Figura 3.23: Temporizado de las señales del protocolo de comunicación del sistema para leer un vector de datos en la columna 59 de la imagen X.

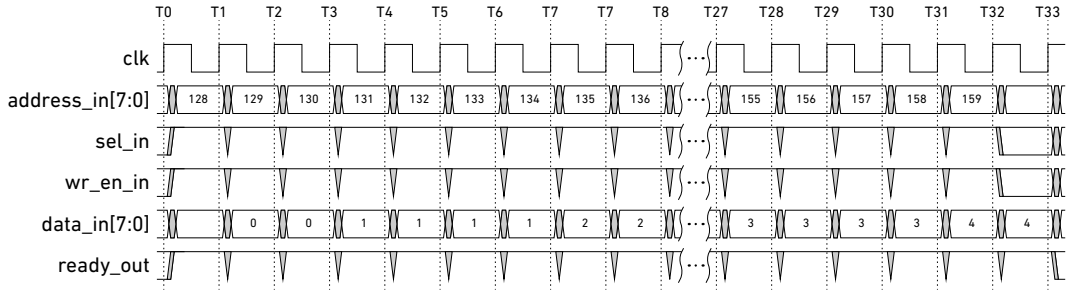


Figura 3.24: Temporizado de las señales del protocolo de comunicación del sistema cargar en los parametros de las funciones *suma* y *zero* en *Gu_0* y *Fx_0* respectivamente.

dada en U, a la cual se le deben reconstruir aquellos pixeles cuyas posiciones estan indicadas por 1 en la figura patrón Fig.3.25(b) alojada en X, la cual se usa como mascara. Para ello se puede interpolar en U; o se puede implementar la mediana del vecindario. La primer solución, representa una función lineal, la cual requiere almacenar un nuevo set de parámetros para implementar la suma de cuatro elementos (Tab.3.14), y su división seteando $sel\_range = 2$. Por su parte, para la segunda opción implementa función no lineal *median*, la cual se selecciona con *sel_Gu* en 5. La configuración completa del procesador para ambos casos se presenta en la Tab.3.15. En la Fig.3.26 se ilustra el diagrama en tiempo de las señales durante el configurado de *confUnit* para la realización de ambos filtrados.

Tabla 3.14: Parámetros de las funciones utilizadas en el ejemplo para realizar la media y el promedio.

Función	Parámetros (32x3 bits)
suma	0 0 1 1 1 1 2 2 1 1 2 2 2 2 3 3 1 1 2 2 2 2 3 3 2 2 3 3 3 3 4 4
median	0 0 0 0 0 1 0 0 0 1 0 1 1 1 0 0 0 1 0 1 1 1 0 1 0 1 1 1 1 1 1 1
zero	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

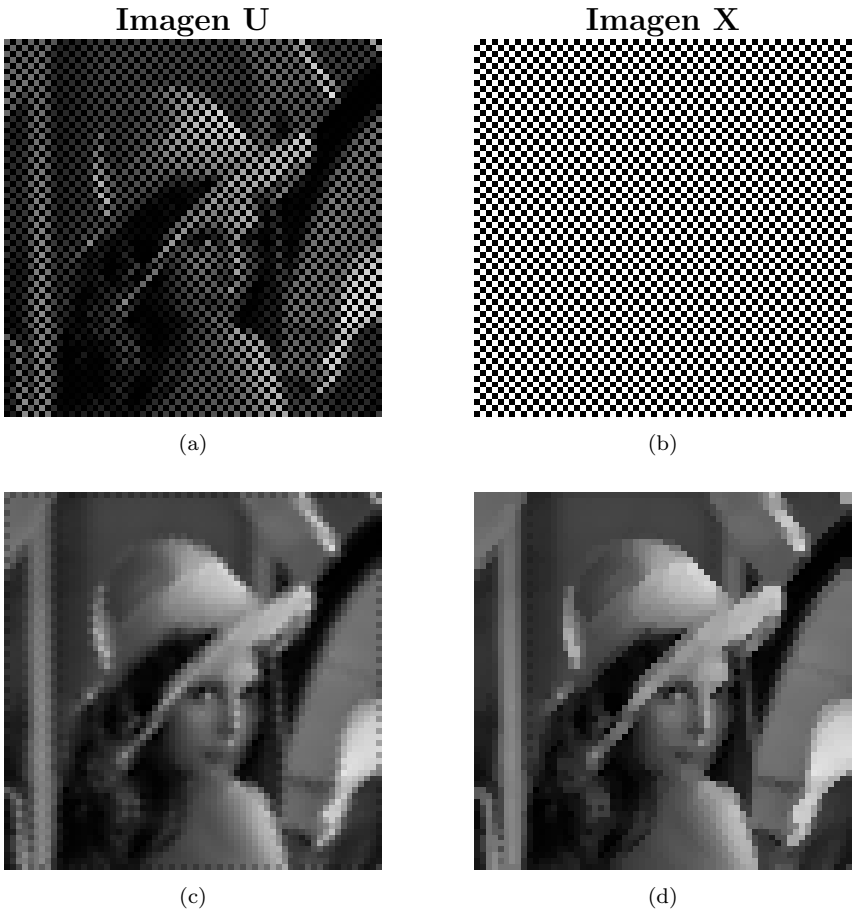


Figura 3.25: Ejemplo de procesamiento para la reconstrucción de píxeles del canal verde de una imagen en base a sus vecinos. (a) Canal verde que necesita ser reconstruido; (b) Patrón de reconstrucción, donde los ceros representan aquella ubicación en de los píxeles de U que deben ser reconstruidos; (c) Resultado implementando una interpolación lineal de sus 4 vecinos; (d) Resultado implementando la mediana de sus 4 vecinos.

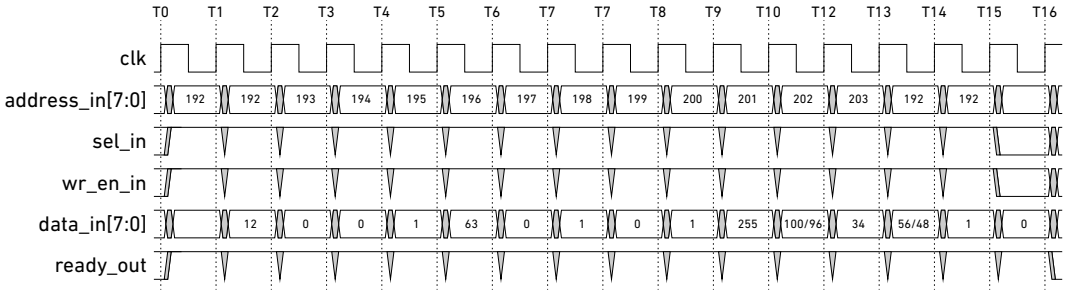


Figura 3.26: Temporizado de las señales IO para la configuración de los registros en *confUnit*.

3.3.3. Implementación y resultados experimentales

El diseño se realizó integramente en lenguaje VHDL, y luego se sintetizó siguiendo el flujo descrito en ApendiceN. Su fabricación fue en la tecnología CMOS 55nm LPX de GlobalFoundries, utilizando una librería de celdas estándar de bajo consu-

Tabla 3.15: Configuración de *confUnit* para implementar la función DeBayer/Mediana para la reconstrucción de U.

Parámetro	Valor	Descripción
sel_pri_sec	0	Se guarda el *bus_primario*
sel_cacheXU	0	El resultado se almacena en U
initial_address_rd	0	-
step_address_rd	1	-
final_address_rd	63	-
initial_address_wr	0	-
step_address_wr	1	-
initial_ramp_value	0	-
step_ramp_value	1	-
final_ramp_value	255	-
chop_image	0	Procesamiento no destructivo
sel_range	2/0	Para el DeBayer es necesario dividir por 4
sel_sec	X	Es indistinto ya que *bus_secundario* no se usa
en_mask	1	Procesamiento enmascarado por X
conf_U	1	Vecindario en cruz
conf_X	X	Es indistinto ya que no se utiliza para procesar
sel_border_U	2	Se copia el valor mas cercano para los bordes de U
sel_border_X	X	Es indistinto ya que no se utiliza para procesar
sel_FoG	2	Función OR para $\circ$
sel_Gu	6/5	Para DeBayer se selecciona Gu_0 ingresado previamente
sel_Fx	0	Función ZEROS para anular X y solo usarla como patrón

mo y de bajo voltaje, diseñadas en el laboratorio AndreouLab de la Univeridad de John Hopkins. En especial, la memoria de imágenes fue implementada utilizando un arreglo de 32 memorias RAM 6T de bajo consumo de 32 bits de ancho y 64 palabras de profundidad (Fig 3.27). El *core* del chip esta compuesto por $1,57$ millones de transistores, ocupando $0{,}76mm \times 1{,}13mm$ tal como se ilustra en Fig.3.28(a).

Para evaluar y testear el chip fabricado, se utilizó un placa personalizada montada a una de desarrollo OpalKelly XEM3010, con la cual, a través de la FPGA Spartan 3 que tiene integrada, se generan los vectores de excitación necesarios para testear la funcionalidad y el rendimiento del chip. Con una tensión de core de $V_{DD} = 1{,}2$V y una frecuencia de reloj igual a $F_{core} = 15$MHz, el chip consume $36{,}98\mu W$, donde $33{,}8\mu W$ corresponde a la potencia estática y $3{,}18\mu W$ a la dinámica. Cuando se escala la tensión a $V_{DD} = 0{,}5$V, la frecuencia máxima funcional pasa a $F_{core} = 1MHz$, dando un consumo total $7{,}38\mu W$, siendo $5{,}50\mu W$ y $1{,}88\mu W$ la potencia estática y dinámica respectivamente.

Tabla 3.16: Características del chip MORPHO8PWL en 55nm.

Tecnología	55nm de GlobalFoundries LPX
Tamaño del arreglo	Vector de 64 procesadores
Área de core	$0{,}76mm \times 1{,}13mm$ mm^2
Cantidad de transistores	$1,57$ millones
Precisión de entrada	8 bit
Precisión de parámetro	3 bit
Memoria de imágenes	2
Algoritmo	Linear a tramos
Máxima dimensión de función	5
Velocidad de trabajo	15 MHz
Cantidad de ciclos de reloj por procesamiento	$263 * 64 = 16832$
Operaciones por segundo	$444,061$ MOPs/s

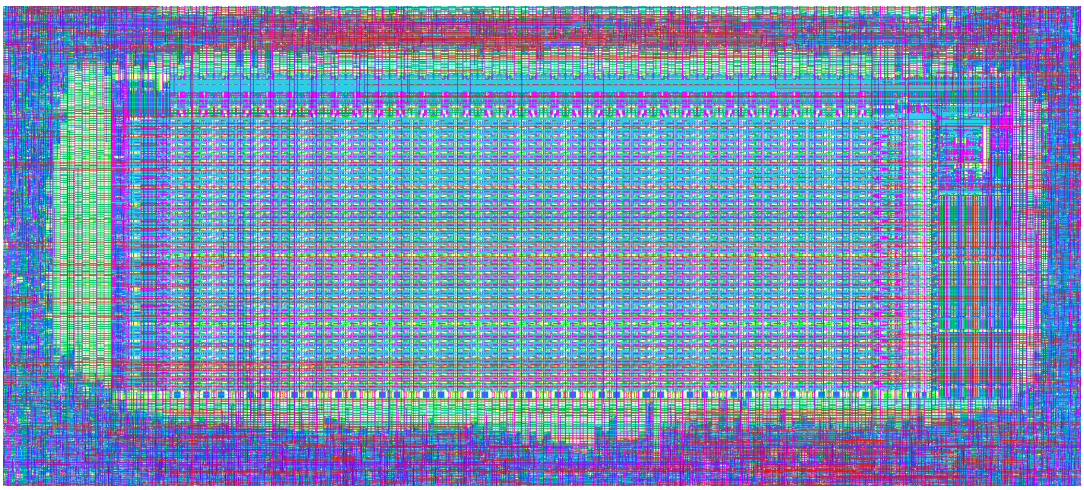
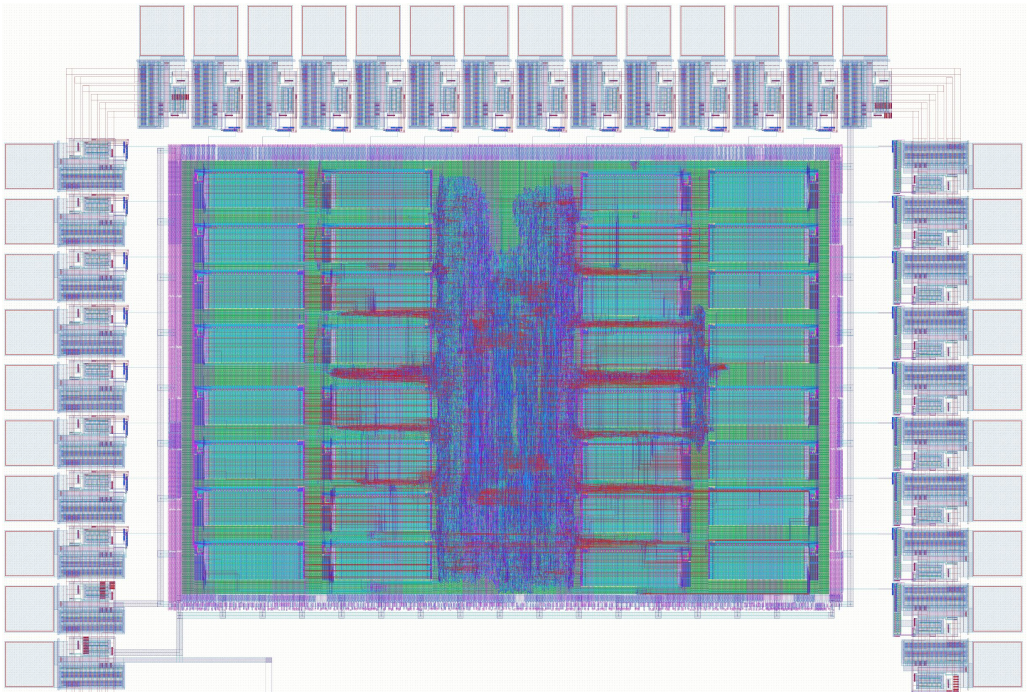
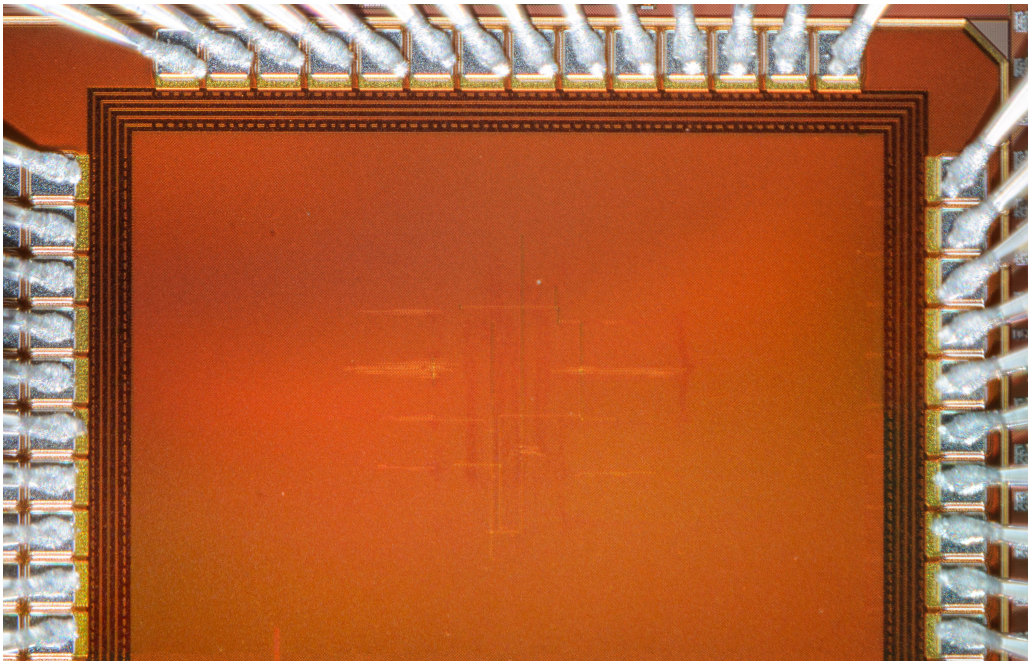


Figura 3.27: Máscara de la memoria RAM utilizada en arreglo para la cache de imágenes.

Para calcular su rendimiento y eficiencia primero se define una operación como la suma entre dos valores de 8 bits, lo que es equivalente a la actividad de $7,5$ bloques *"full-adder"* (FA). Por celda de procesamiento se realizan 6 comparaciones por ciclo de reloj, durante los 256 pasos de la rampa, lo que equivale a $3 \times 2 \times 256 = 1536$ sumas. En la ALU se realizan 12 operaciones de 3 bits ($2,5$FA), 6 selecciones de los parámetros de F y 6 de G, lo que añade 2,5 $\div 7,5 \times 12 = 4$ operaciones. Por último, se agregan 256 sumas de 8 bits del acumulador local (un sumador de una entrada de 12 bits y otra de 4 bits necesita $3FA + 9HA = 7{,}5FA$, equivalente a un sumador de 8bits). Por lo tanto el rendimiento del Sistema MORPHO8PWL es $(1536+12+256) \times 64 \times 64 \times 1 \times 10^6 \div ((256+4) \times 64) = 444,061$ MOPs/s, alcanzando 60.170 TOPs/W.



(a) Mascara del chip implementado en 55nm.



(b) Foto del chip fabricado MORPHO8PWL.

3.4. Conclusiones

En este capítulo se han presentado dos sistemas de procesamiento utilizando estructuras de tipo SCNN para cómputo binario y multibit (8 bits) fabricadas y testeadas con éxito en las tecnologías de 180nm de IBM y de 55nm de GlobalFoundries.

El Sistema multibit cuenta con la posibilidad de operar las entradas con precisión variable desde 1 bits hasta 8 bits y en diferentes rangos de valores a través de la configuración de la rampa de procesamiento. Esto permite no solo reducir la precisión de una imagen o su binarización, si no que también puede actuar como un divisor de la función. Por otro lado, se aumentó la precisión de los parámetros de función, lo que permite operar con funciones lineales para la implementación de filtros por convolución y operaciones morfológicas multi-nivel. Se agrega, además, la capacidad de enmascarar el procesamiento por píxel, de acuerdo a valores dentro de la imagen. Estas características sumada a la posibilidad de configurar la rampa resultan atractivas para la utilización de esta arquitectura dentro del mundo de las redes neuronales, dado que es posible inhibir celdas según su valor y su coordenada.

A diferencia de las versiones anteriores reportadas en la literatura, tales como el sistema SCDVP [12], fabricado en una tecnología de 90 nm, y el Sistema 3DNeuP [13], fabricado en una tecnología en 130nm, las arquitecturas realizadas en esta tesis no poseen elementos sensores de luz. En consecuencia, los diseños son puramente digitales y se han realizado de manera parametrizables, lo cual permite una flexibilidad amplia para sintetizar estructuras de distintos tamaños y en diferentes tecnologías. Esta flexibilidad se consigue a costa de una reducción en la densidad de transistores y en la eficiencia energética. A los efectos de automatizar el funcionamiento, se agregaron estructuras adicionales como máquinas de estados para el control de las distintas unidades, registros de configuración, y memoria de programas, entre otros.

En efecto, el Sistema MORPHO1PWL tiene una eficiencia ligeramente menor, $1,267$ TOPs/W contra $1,407$ TOPs/W en comparación con el Sistema SCDVP, cuando se tienen en cuenta operaciones de 5 vecinos y 1 bit de precisión (blanco y negro). Por otro lado, cuando se consideran operaciones en escalas de grises la eficiencia del Sistema MORPHO8PWL, configurado para 6 bits, es significativamen-

te mayor, 255,061 GOPs/W contra 64,969 GOPs/W del Sistema SCDVP (6 bits). Cuando se configura el Sistema MORPHO8PWL funciona al máximo de capacidad de precisión, 8 bits, su eficiencia pasa a 66,708 GOPs/W.

El MORPHO8PWL presenta una arquitectura similar al 3DNeuP, donde las celdas de procesamiento se encuentran separados de la memoria de imagen. Sin embargo, para optimizar los el accesos a los datos, se omitió el registros de acumulación de la memoria, mejorando así la eficiencia.

Para mejorar la eficiencia y velocidad de procesamiento es necesario aumentar en la cantidad de celdas dentro del vecindario y de esta manera evitar realizar cómputos temporales. Por otro lado, futuras implementaciones deben contemplar la capacidad de realizar *"clock-gating"* [76] en los registros de las celdas para disminuir la energía dinámica de los circuitos.

Capítulo 4

Procesadores morfológicos basados en funciones simétricas

4.1. Introducción

Las funciones lineales a tramos, definidas sobre un dominio simplicial de n dimensiones [8], requieren para su implementación de 2^n parámetros. En el caso de 5 variables de entrada, tal como los sistemas planteados en el capítulo anterior, se requieren de $2^5 = 32$ parámetros. Si bien las funciones de 9 variables se pueden representar mediante operaciones sucesivas sobre conjuntos de 5 variables, esto requiere mayor tiempo de procesamiento. Por ejemplo, para filtros morfológicos como la dilatación o la erosión en el MORPHO8PWL, solo basta con procesar dos veces la misma imagen utilizando las dos configuraciones de vecindades $\times$ y $+$, y luego procesar una vez los resultados en conjunto. Sin embargo, si se quiere calcular la mediana o la media selectiva de la región completa resulta imposible, y solo se puede llegar a aproximaciones. Por otro lado, si se quiere implementar un elemento estructurante de 9 entradas, es decir de 3×3 pixeles, utilizando el mismo algoritmo y arquitectura de procesador, se necesita que los multiplexores de parámetros por celda sean 16 veces más grande que en el caso de 5 entradas, lo cual resulta en una celda aproximadamente 16 veces mayor a la original, reduciendo así la densidad de píxeles a procesar en paralelo dado el mismo área.

En este capítulo se propone una alternativa para implementar funciones con ma-

yor cantidad de vecinos y complejidad reducida que consiste en la utilización de funciones simétricas PWL. De esta manera la cantidad de parámetros para una función de n entradas se reduce a $N\_param = n+1$, permitiendo el diseño de celdas de procesamiento pequeñas capaces de computar funciones tales como el máximo, el mínimo, la media, la mediana, entre otras, de más de 5 entradas. Además, se presentan las diferentes implementaciones en VLSI del algoritmo de PWL modificado para funciones simétricas, para imágenes de un bit o en escalas de grises. Previamente, se introducirá una breve descripción de las funciones simétricas multivariables, y las distintas modificaciones que se le realizan al algoritmo PWL existente.

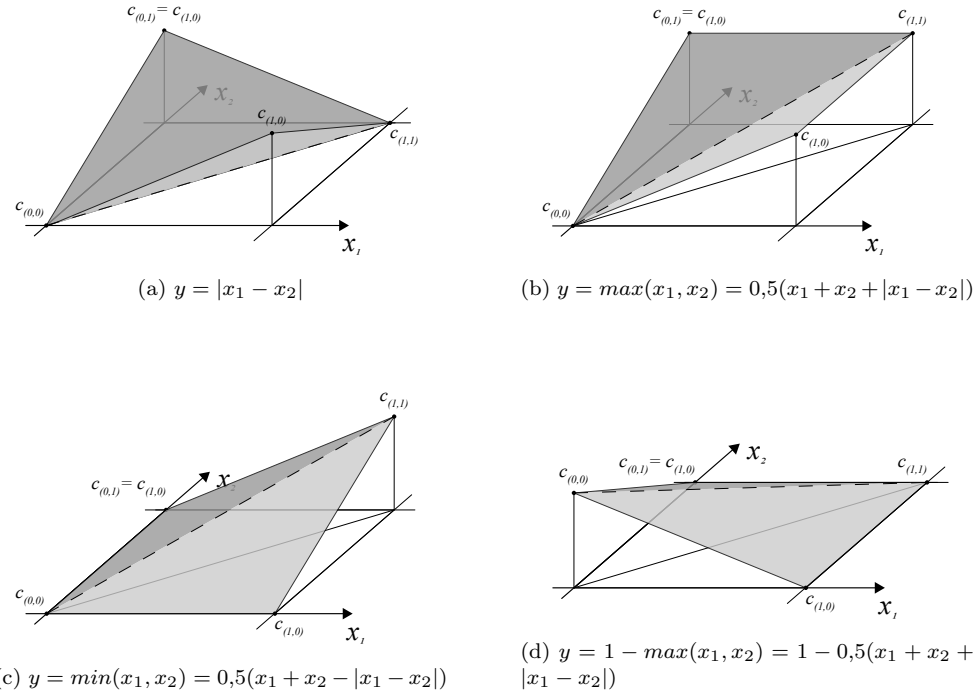
4.1.1. Funciones simétricas

Se define una función simétrica de n variables a una función cuyo valor es independiente del orden de sus variables [77]:

$$y = f(x_{i_1}, \cdots, x_{i_n}) = f(x_{j_1}, \cdots, x_{j_n}) \tag{4.1}$$

para dos conjuntos de indices $\{i_1, \cdots, i_n\}$, $\{j_1, \cdots, j_n\}$, tal que $i_k, j_k \in \{1, \cdots, n\}$, $i_k \neq i_l$, $j_k \neq j_l$ si $k \neq l$. Por ejemplo, las funciones $y = |x_1 - x_2|$, $y = max(x_1, x_2) = 0{,}5(x_1 + x_2 + |x_1 - x_2|)$, $y = min(x_1, x_2) = 0{,}5(x_1 + x_2 - |x_1 - x_2|)$, $y = 1 - max(x_1, x_2) = 1 - 0{,}5(x_1 + x_2 + |x_1 - x_2|)$, representadas en la Fig. 4.1a-d son simétricas. Cabe destacar que si el rango de las entradas es restringido a valores digitales, como por ejemplo, $\{0, 1\}$, las funciones anteriormente introducidas son: $XOR(x_1, x_2)$, $OR(x_1, x_2)$, $AND(x_1, x_2)$, y $NOR(x_1, x_2)$. Contrariamente, la función $y = x_1 + 2x_2$ es un ejemplo de una función que no es simétrica.

Dada una partición simplicial unitaria de dominio perteneciente a $\Re^2$, la cantidad de coeficientes necesarios para definir cualquier función lineal a tramos es $q = 2^n = 4$, denominados $c_{(0,0)}$, $c_{(1,0)}$, $c_{(0,1)}$, y $c_{(1,1)}$ según las coordenadas de sus vértices asociados. Si la función a implementar es simétrica, sus coeficientes $c_{(1,0)}$ y $c_{(0,1)}$ son iguales, reduciendo la cantidad de parámetros a $N_{param} = n + 1 = 3$. En el caso de una función en $\Re^3$ los coeficientes necesarios para definirla son $q = 2^n = 8$, $c_{(0,0,0)}$, $c_{(1,0,0)}$, $c_{(0,1,0)}$, $c_{(0,0,1)}$, $c_{(1,1,0)}$, $c_{(1,0,1)}$, $c_{(0,1,1)}$, $c_{(1,1,1)}$; sin embargo, $c_{(1,0,0)} = c_{(0,1,0)} = c_{(0,0,1)}$, y $c_{(1,1,0)} = c_{(1,0,1)} = c_{(0,1,1)}$ llevando a $N_{param} = n + 1 = 4$ la cantidad de parámetros a



(a) $y = |x_1 - x_2|$

(b) $y = max(x_1, x_2) = 0{,}5(x_1 + x_2 + |x_1 - x_2|)$

(c) $y = min(x_1, x_2) = 0{,}5(x_1 + x_2 - |x_1 - x_2|)$

(d) $y = 1 - max(x_1, x_2) = 1 - 0{,}5(x_1 + x_2 + |x_1 - x_2|)$

Figura 4.1: Funciones simétricas en el dominio simplicial.

guardar. Se desprende entonces que la función solo necesita ser definida en un solo símplice, y dado que existe uno, no es necesario realizar la búsqueda del mismo. Es conveniente entonces renombrar los coeficientes de acuerdo a la cantidad de variables cuyo valor sea distinto de cero. En $\Re^2$ los coeficientes pasan a ser $c_{(1,1)} \rightarrow c_2$, $c_{(1,0)} = c_{(0,1)} \rightarrow c_1$, y $c_{(0,0)} \rightarrow c_0$.

4.1.2. Implementación digital

En la Fig.4.2 se observa un diagrama en bloque de una opción para implementar de manera digital una función PWL simétrica para una partición por dimensión.

Todas la variables involucradas, las entradas x_i y los coeficientes c_j, donde $i \in \{1, \cdots, n\}$ y $j \in \{0, \cdots, n\}$, son almacenadas en memoria con precisión fija q y p respectivamente. El tiempo de procesamiento total N_{proc}, al tratarse de un sistema sincrónico, es igual a la cantidad de ciclos de reloj de un ciclo completo de rampa ascendente, también de q bits:

$$N_{proc} = \text{floor}\left(\frac{ramp_{end} - ramp_{ini}}{ramp_{step}}\right) \tag{4.2}$$

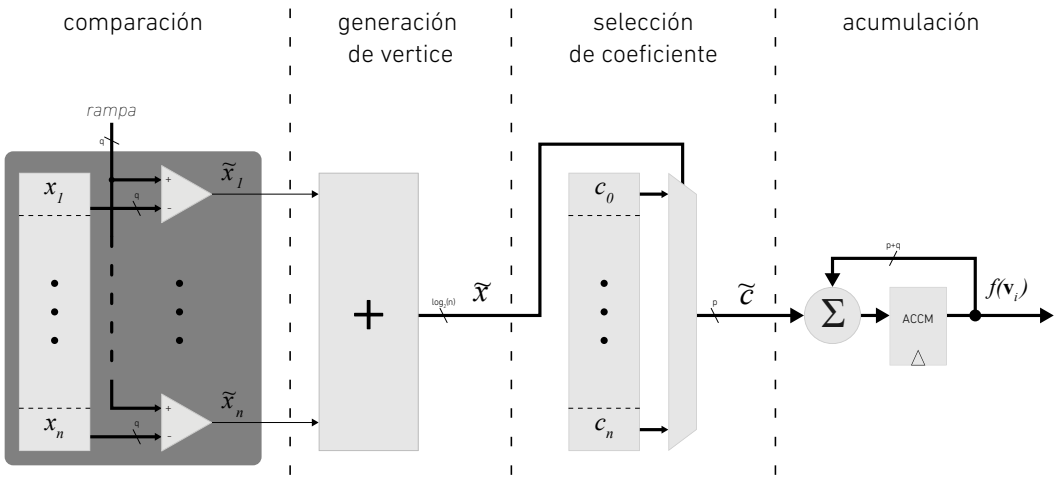


Figura 4.2: Esquemático de implementación digital para funciones simétricas a tramos.

donde $ramp_{ini}$ y $ramp_{end}$ es el valor de inicial y final de la rampa respectivamente, $ramp_{step}$ es el valor de incremento por ciclo de reloj, y la función $floor$ redondea su argumento al entero más cercano menor o igual al mismo. Cuando se inicia la rampa, se reinicia en cero el registro ACMM. Por cada paso de la rampa, su valor se compara con el de las entradas, formando entre todas las comparaciones una palabra $\tilde{x} = \sum_{i=0}^{n-1} \tilde{x}_n$ de $\log_2(n)$ bits que identifica el vértice a computar, y por lo tanto el coeficiente a usar. La acumulación de los $\tilde{c}$ por N_{proc} ciclos en el registro ACMM genera como resultado el nuevo valor de la función.

4.2. Procesador de un bit de vecindario extendido

En esta sección se describe el diseño del chip MORPHO1SYM, el cual utiliza una arquitectura similar al MORPHO1PWL, pero implementando el algoritmo modificado para el cálculo de funciones simétricas lineales a tramos. El arreglo bi-dimensional de CNN simpliciales está compuesto por 64×64 elementos de procesamiento (PE), cada uno interconectado localmente con sus ocho vecinos, capaz de operar funciones simétricas de un bit con dominio en $\Re^9$.

4.2.1. Arquitectura

En la Fig.4.3 se puede observar el diagrama simplificado de la arquitectura global del diseño. Al igual que el procesador anterior, el sistema esta compuesto por un arreglo de procesadores PWL *arrayPWL*, un controlador *controllerPWL* (que junto a su memoria de programa administra las señales que van a los PE) y una unidad de configuración que establece las funciones a implementar y el funcionamiento general del chip. Cada celda de procesamiento se conecta con sus ocho celdas vecinas más cercanas, implementando una función compuesta $z = f_x \circ g$, donde $y = f_x = f(x_0, ..., x_8)$ es una función simétrica PWL de nueve entradas, g es un valor local de 1 bit, y $\circ$ es una función lógica configurable (AND, OR, XOR o NOR). Debido a la naturaleza de la implementación, la cantidad de coeficientes a almacenar por función se reduce de 32 a 10, disminuyendo así el tiempo de procesamiento y la memoria para el almacenamiento de parámetros. Además se agregan los ya utilizados bloques *Decoder* y *Multiplexor*, lo cuales intervienen en la implementación del protocolo IO basado en AMBA AHB-Lite de ARM®.

A diferencia de su versión anterior, el procesador esta compuesto por 3 registros de estado denominados *regX*, *regU* y *regT* en Fig.4.4, los cuales guardan los valores de 1 bit X, U y T. Para el calculo de f_x, el valor local de X, junto con los 8 correspondientes de las celdas vecinas (Fig.4.5), se operan bit a bit, usando compuertas AND, con nueve señales globales de enmascaramiento $b_i \in \{0, 1\}, i = 0, ..., 8$ (*en_roi* en esquemático), y los nueve bits resultantes se suman produciendo el argumento de la función:

$$\arg(f) = b_0 x_0 + b_1 x_1 + \cdots + b_8 x_8 \tag{4.3}$$

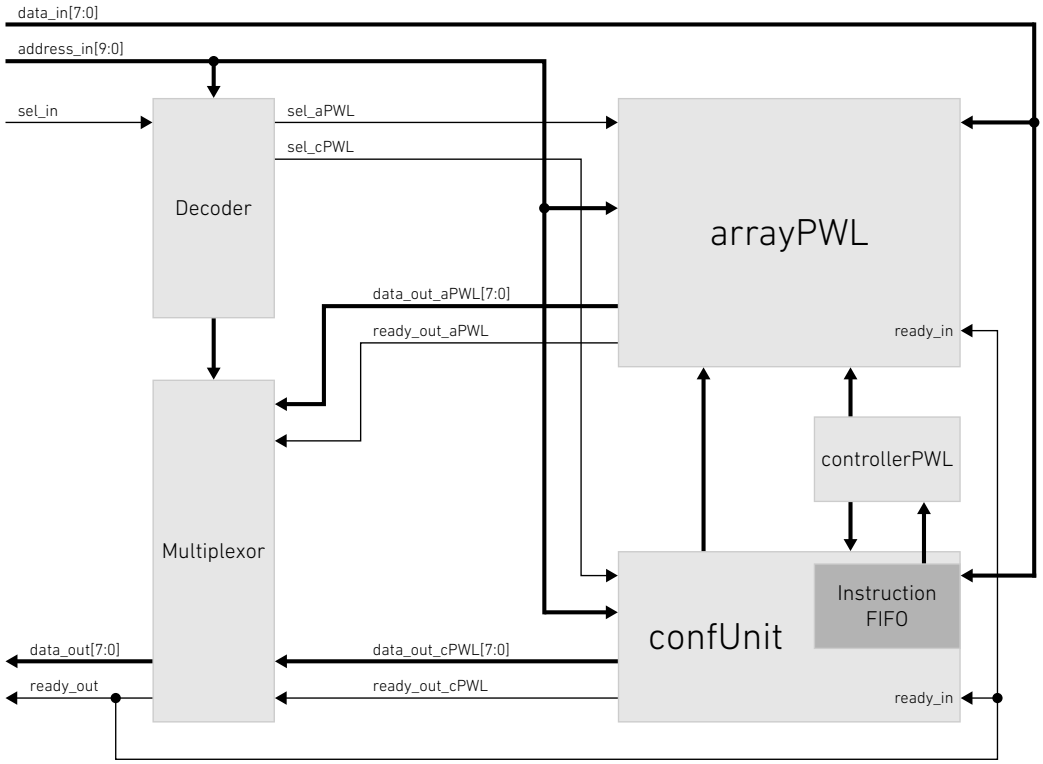


Figura 4.3: Esquemático general de la arquitectura.

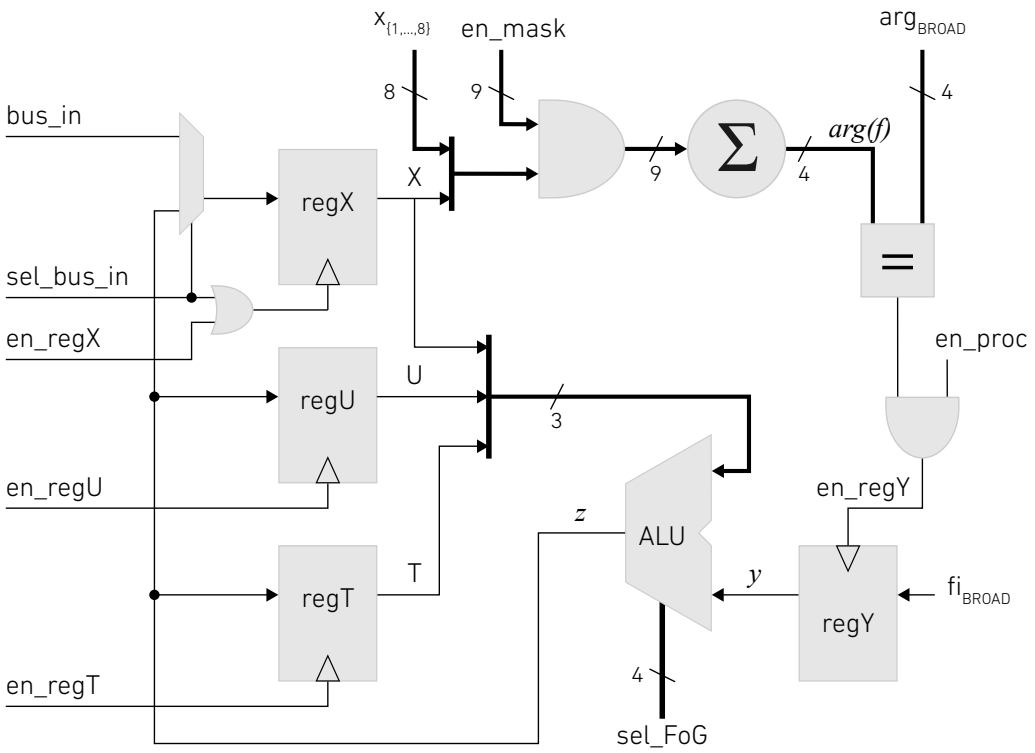


Figura 4.4: Esquema del elemento de procesamiento.

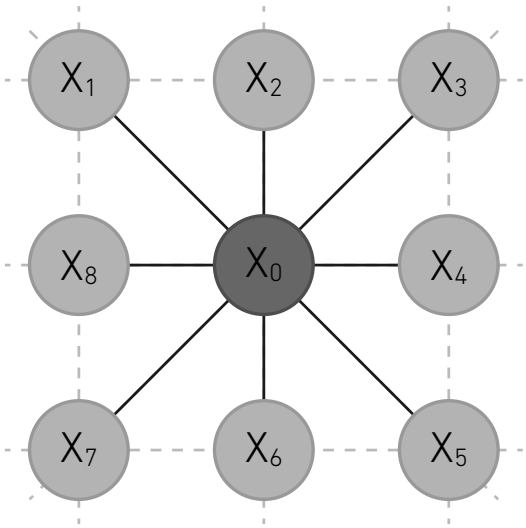


Figura 4.5: Conectividad local entre procesadores.

donde $0 \leq \arg(f) \leq 9$ es un valor de 4 bits. Durante diez ciclos de reloj, $arg(f)$ se compara con la señal global arg_{BROAD}, proveniente del controlador, y cuando ambas son iguales, habilitan el registro $regY$ para almacenar el valor de la función f_i correspondiente al valor arg_{BROAD} propagado. El valor y, junto con los otros tres, X, U y T, ingresan a una ALU, donde, con la señal *sel_FoG*, se selecciona la función lógica $\circ$, para generar el valor final z (Tabla 4.1). Por último, con las habilitaciones *en_regX* (Tabla 4.2), *en_regU* y *en_regT* se guarda el resultado z en el registro deseado. En la Fig.4.6 se ejemplifica el temporizado de las señales internas de la celda durante un periodo de procesamiento.

Los PE están organizados en un arreglo de dos dimensiones, localmente interconectados, denominado *arrayPWL*. En este diseño se mantiene la señal *sel_border* de dos bit que permite elegir como condiciones de borde del arreglo: '0', '1', o el valor igual a la de su vecino dentro del arreglo. Dado que la estructura general se mantiene, los detalles de la implementación se pueden observar el diseño MORPHO1PWL.

Para la configuración del procesamiento llevado a cabo en las celdas, se implementó la unidad denominada *confUnit* en el diagrama de la Fig.4.3. La unidad esta compuesta de una serie de registros, una memoria de funciones, y otra de tipo FIFO de programa, por las cuales el usuario configura las operaciones a realizar en los procesadores y el controlador de procesamiento *controllerPWL*. A continuación se detallan estos componentes, describiendo su funcionalidad en el sistema y su ubica-

Tabla 4.1: Operaciones de la ALU.

sel_FoG	Función
0	$z = \mathsf{AND}\,(y, X)$
1	$z = \mathsf{XOR}\,(y, X)$
2	$z = \mathsf{OR}\,(y, X)$
3	$z = \mathsf{NOR}\,(y, X)$
4	$z = \mathsf{AND}\,(y, U)$
5	$z = \mathsf{XOR}\,(y, U)$
6	$z = \mathsf{OR}\,(y, U)$
7	$z = \mathsf{NOR}\,(y, U)$
8	$z = \mathsf{AND}\,(y, T)$
9	$z = \mathsf{XOR}\,(y, T)$
10	$z = \mathsf{OR}\,(y, T)$
11	$z = \mathsf{NOR}\,(y, T)$
12	$z = y$
13	$z = X$
14	$z = U$
15	$z = T$

Tabla 4.2: Selección del próximo valor de *regX*.

Próximo *regX*	Condición
X	Si $en\_regX = 0$ o $sel\_bus\_in = 0$
z	Si $en\_regX = 1$ o $sel\_bus\_in = 0$
bus_in	Si $sel\_bus\_in = 1$

ción en memoria.

La primer memoria, ubicada entre las direcciones 512 y 537, permite almacenar 16 diferentes conjuntos de 10 parámetros de 1 bit que pueden ser utilizadas para operar en los procesadores. La selección de las diferentes funciones se realiza a través del valor del registro de 4 bits *sel_function*, el cual se describe más adelante. Para guardar las primeras 8 funciones, se debe escribir entre las direcciones 512 y 521, donde el primer parámetro de todas las funciones se encuentra en la dirección 512, el segundo en la 513, y así sucesivamente hasta llegar al décimo parámetro, ubicado en la 512. El primer bit del bus de datos de entrada corresponde a la función 0, y el último a la 7:

$$\mathsf{data\_in}[7:0] = \{f_x\_7, f_x\_6, f_x\_5, f_x\_4, f_x\_3, f_x\_2, f_x\_1, f_x\_0\} \tag{4.4}$$

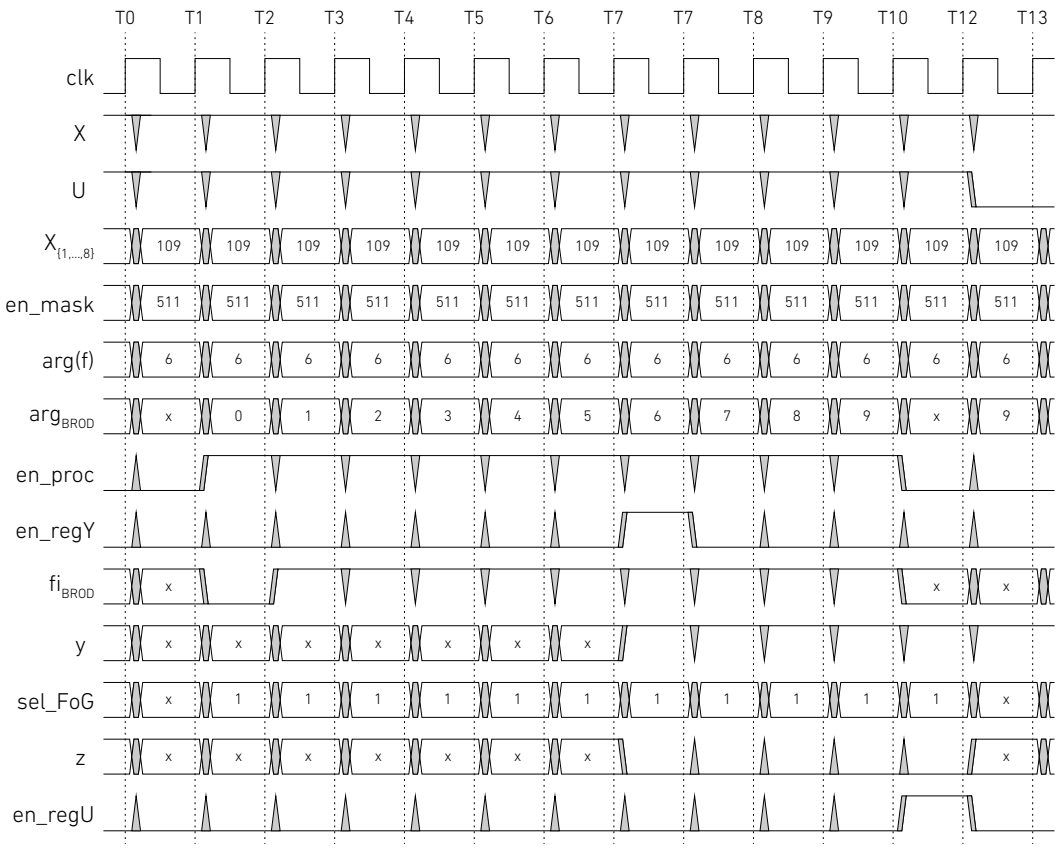


Figura 4.6: Temporizado de las señales del procesador durante un ciclo de procesamiento. La función a implementar es el cálculo del máximo de la vecindad (dilatación en morfología), y luego su resta con el valor X (función lógica XOR cuando se trata de valores de 1 bit). La operación resultante es util para el calculo de perímetros. Por último, el valor z es almacenado en *regU*. La cantidad de ciclos para el calculo de f_x es de un total de 10 ciclos para la distribución de los parámetros. Un ciclo de reloj más se suma, para el guardado del resultado, llevando a 11 la cantidad de ciclos total para una operación en el MORPHO1SYM.

Para el caso de las últimas 8 funciones, se debe escribir entre las direcciones de memoria 528 y 537, siguiendo la misma lógica anteriormente detallada. La función 8 se ingresa a través del bit menos significativo de *data_in*, y la última, en el bit más significativo:

$$\mathsf{data\_in}[7:0] = \{f_x\_15, f_x\_14, f_x\_13, f_x\_12, f_x\_11, f_x\_10, f_x\_9, f_x\_8\} \tag{4.5}$$

La segunda memoria, de tipo FIFO, provee al controlador *controllerPWL* de hasta 64 instrucciones de 10 bits de ancho. Para su carga, se debe acceder escribiendo entre las direcciones 544 a 546. Dado que el bus de entrada *data_in* es de 8 bits, se utilizan los dos bits menos significativos del bus de direcciones para formar la

palabra de instrucción. A diferencia del procesador MORPHO1PWL, se agrega la funcionalidad de poder reiniciar el puntero de lectura interno, para que una vez que haya sido completamente leída, los datos internos no se pierdan, y puedan ser utilizados nuevamente si así se desea. La FIFO implementa señales de bandera *full_flag* y *empty_flag* para indicar si se encuentra llena o vacía, respectivamente; y otra *empty_flag_rd* que indica con un 1 cuando la memoria ha sido completamente leída.

Los otros dos componentes que conforman la unidad de configuración son el registro de control y los registros de procesamiento. El registro de control es de 4 bits y se accede a través en la dirección 547. Su función es la de iniciar, pausar o detener el funcionamiento de la maquina de estado del controlador, así como la de dar reinicio el contenido a los punteros de la FIFO de instrucciones:

- **reset_fifo** : ubicado en el bit 0 del registro de control, si se encuentra en 1 reinicia completamente la FIFO de programa.
- **hold_proc** : ubicado en el bit 1 del registro de control, en 1 pausa el controlador de procesamiento.
- **start_proc** : ubicado en el bit 2 del registro de control, en 1 da comienzo a que el controlador de procesamiento empiece a leer la memoria de instrucciones.
- **stop_proc** : ubicado en el bit 3 del registro de control, en 1 detiene la lectura de la memoria de programa por parte del *controllerPWL*, y reinicia su maquina de estado.

El segundo grupo de registros, los de procesamiento, son configurados directamente por el controlado, con el objetivo de manejar las señales que se dirigen a los procesadores y otras que funcionan de señalización para el usuario:

- **is_processing**: registro de bandera que indica si el sistema se encuentra procesando ($is\_processing = 1$), o no. Su valor puede ser leído en el bit 3 de del bus de datos de salida, cuando se accede a la dirección 546.
- **sel_border**: registro de dos bits que configura las condiciones de borde a aquellos PE que se encuentran en la frontera del arreglo. Si es 0, los valores de borde

es 0, 1 si es 1, y de ser 2 o 3 se les asigna un valor igual a la de correspondiente de la celda más próxima.

- **sel_fx**: registro de cuatro bits que selecciona uno los dieciséis set de 10 parámetros almacenadas en la memoria de funciones.
- **sel_FoG**: registro de cuatro bits que configura la función FoG de la ALU dentro de las celdas (Tabla 4.1).
- **sel_regX**: registro de un bit que habilita los registros *regX* de los PE tomen guarden el valor z.
- **sel_regU**: registro de un bit que habilita los registros *regU* de los PE tomen guarden el valor z.
- **sel_regT**: registro de un bit que habilita los registros *regT* de los PE tomen guarden el valor z.
- **arg_broad**: registro de cuatro bits el cual que referencia el indice del parámetro de la función F que se esta propagando al arreglo. En el inicio de un procesamiento, toma el valor 0, y por cada ciclo de reloj, su valor incrementa en uno, hasta llegar a 10.
- **fi_broad**: registro de un bit que toma el valor del parámetro indexado por *arg_broad*, de la función F seleccionada por *sel_Fx*.
- **en_roi**: registro de un nueve bits que enmascara las X's de cada PE.

Vale resaltar que la señal IO de *ready_out* siempre se mantiene en alto, excepto cuando se intenta escribir en la memoria de programa y esta se encuentra llena. Por último, en la Tabla B.4, se detalla el las direcciones de acceso para la lectura de los registros anteriormente nombrados.

Controlador de procesamiento

En la Fig.4.7 se muestra el diagrama de flujo de la máquina de estados, implementada dentro del controlador *controllerPWL*, para el manejo de las señales que permiten el procesamiento dentro de las celdas. Cuando se pulsa la señal de reinicio

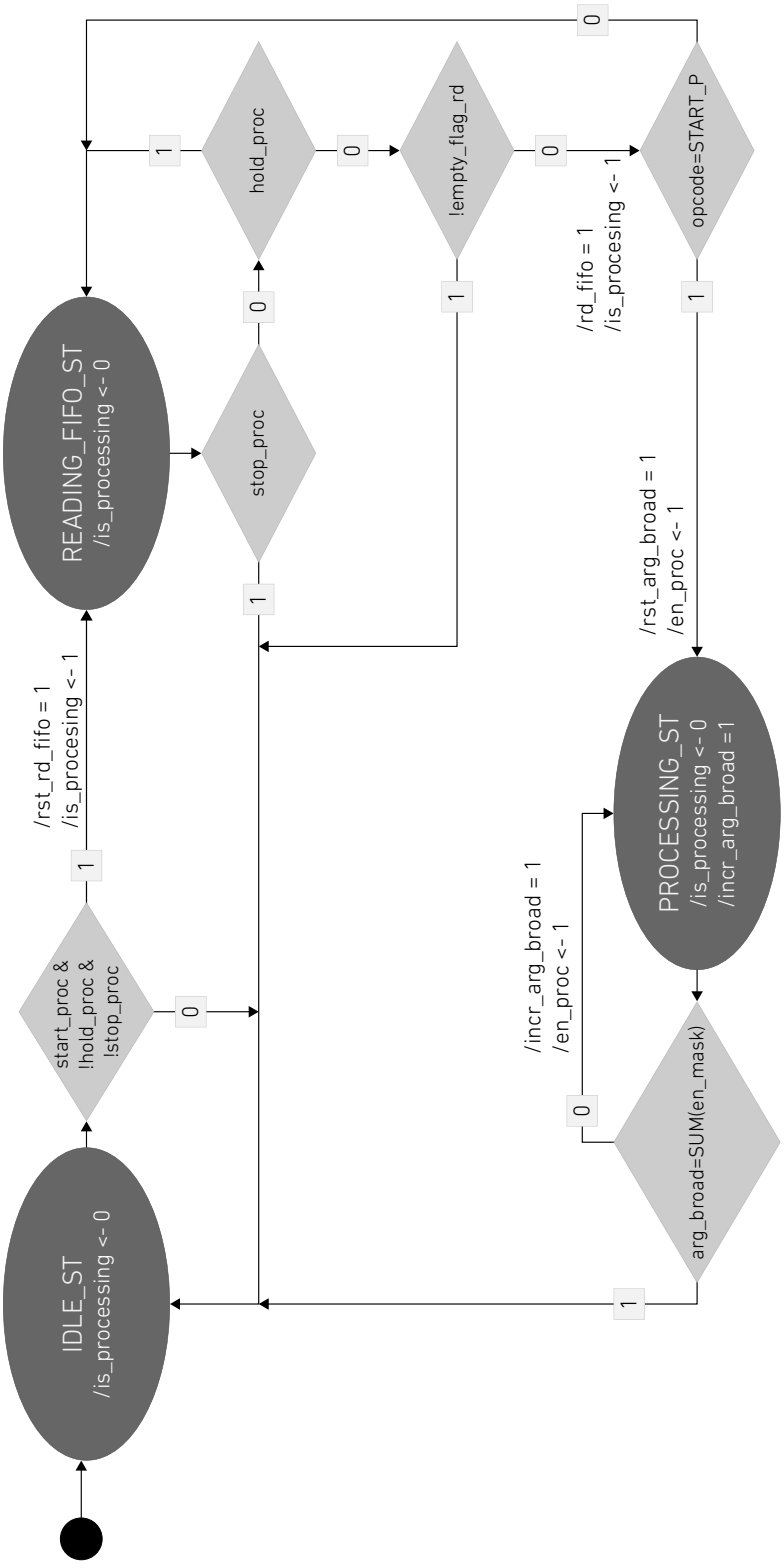


Figura 4.7: Diagrama de estados de la maquina implementada en *controllerPWL*. Las elipses representan los estados, los rombos evaluaciones condicionales, los cuadrados las distintas condiciones, y las flechas transiciones. En el controlador se implementa dos registros auxiliares, *last_read* y *last_proc*, como señales bandera de la última columna de datos de entrada y el último procesamiento realizado respectivamente.

general del sistema, la maquina se inicia en el estado IDLE_ST, y avanzará hacia los siguientes estados de acuerdo a los valores de los registros de control, y del contenido de la memoria de programa. La máquina de estados posee tres estados:

- **IDLE_ST** : primer estado, sensible a los registros de control *start_proc*, *hold_proc* y *stop_proc*. Cuando *start_proc* es 1 y los otros dos 0, se reinicia el puntero de lectura con *rst_rd_fifo* en alto, se lleva el registro *is_processing* a 1, y se pasa al estado READING_FIFO_ST.

- **READING_FIFO_ST** : si no se ha reiniciado la maquina de estados, y si no se la ha pausado, se evalúa si existen instrucciones a leer utilizando la señal de bandera *empty_flag_rd*. En caso positivo, se procede a leer la FIFO a través la señal *rd_fifo*, y se evalúan los dos bits más significativos de la instrucción leída. Las operaciones de programa o instrucciones se muestran en la Tabla 4.3. Si la instrucción es START_P, además de actualizar los registros para seleccionar la función y operación en las ALUs, se reinicia el registro *arg_broad* en 0, se lleva *en_proc* a 1 en el ciclo de reloj próximo iniciando un ciclo de procesamiento, y se pasa al estado PROCESSING_ST. En el caso de que no existan más instrucciones a leer, se retorna a IDLE_ST.

Tabla 4.3: Descripción de las instrucciones de programa que se leen durante el estado READING_FIFO_ST. En todos los casos se actualizan los registros que aparecen dentro de la instrucción. Por ejemplo, en CONF_1 se actualizan los tres bits mas significativos de *en_roi*, ademas de los registros que habilitan el almacenado dentro de las celdas de procesamiento.

Opcode	instrucción[9:0]									
	9	8	7	6	5	4	3	2	1	0
START_P	0	0	sel_fx				sel_FoG			
CONF_0	0	1	en_roi[5:0]						sel_border	
CONF_1	1	-			en_roi[8:6]			en_regT	en_regU	en_regX

- **PROCESSING_ST** : con el registro *is_processing* y la señal *en_proc* ambos en 1, se incrementa monotonamente, con pasos de 1, el registro *arg_broad*,hasta llegar al valor de la cantidad de vecinos habilitados por *en_roi*. Luego, da concluido el ciclo de procesamiento llevando *en_proc* a 0 y pasando al estado READING_FIFO_ST.

4.2.2. Funcionamiento de la arquitectura

A continuación se explica el funcionamiento del sistema ilustrando como se realizan las tareas básicas en la arquitectura. Las direcciones de acceso a los distintos componentes se listan en Tab.4.4

Tabla 4.4: Mapeo lógico de las direcciones para escritura y lectura.

Dirección	Sentido	Descripción
0-511	Escritura/ Lectura	Se accede a el registro *regX* de las celdas del arreglo.
512-521	Escritura/ Lectura	Memoria de parámetros de las primeras 8 funciones.
528-537	Escritura/ Lectura	Memoria de parámetros de las últimas 8 funciones.
544-546	Escritura	Memoria de instrucciones (Tab.3.5)
547	Escritura	Registro de control.
544-547	Lectura	Lectura de los registros de control y procesamiento (Tab.B.4)

Previo a realizar cualquier tarea, se debe pulsar la señal de reinicio del sistema *reset* para reiniciar el controlador *controllerPWL* al estado inicial IDLE_ST. Cuando se reinicia, también lo hacen los registros de configuración de procesamiento, permitiendo que los PE almacenen de manera estable los valores ingresados desde el exterior. A continuación se detalla la configuración de las tareas básicas para la escritura, lectura, y procesamiento de una imagen de entrada binaria.

La escritura y lectura de una imagen binaria de 64×64 pixeles se realiza siguiendo el mismo protocolo utilizado en el diseño MORPHO1PWL, con la excepción de que en este caso, el registro modificado en las celdas de procesamiento es *regX*.

Para la configuración del procesador, al igual que los casos anteriores, es necesario previamente cargar la memoria de funciones con las que van a ser utilizadas (Fig.4.8), y luego cargar las instrucciones, previo reinicio de la FIFO. Luego de que hayan sido cargadas, entonces queda darle inicio a la lectura y procesamiento de las mismas seteando el registro *start_proc* por un ciclo de reloj. A continuación, se describe un breve ejemplo de configuración.

En la Fig.4.9(a) se muestra una imagen binaria almacenada en X, a la cual se le detectó sus bordes, resultando en la imagen que aparece en la Fig4.9(b). El temporizado de las señales IO para configurar los registros de control y cargar las

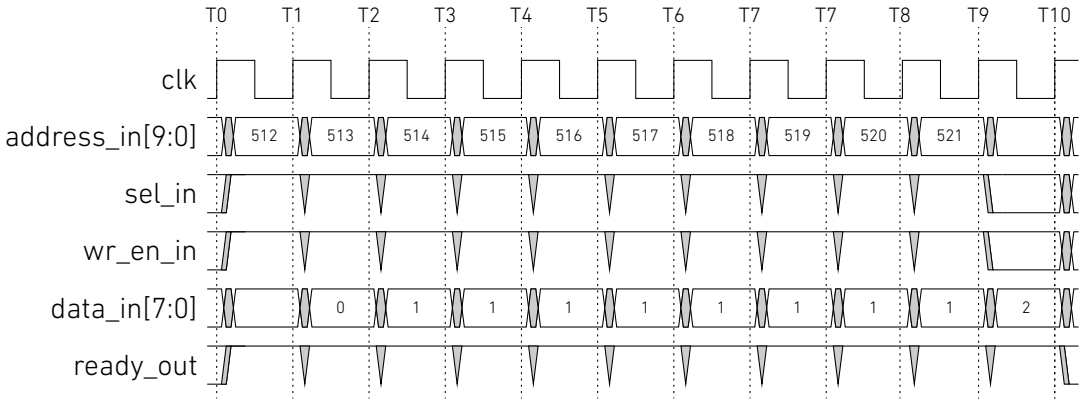


Figura 4.8: Temporizado de las señales IO para la carga de parámetros de función en MORPHO1SYM, para el ejemplo dado.

instrucciones son ilustradas en la Fig.4.10.

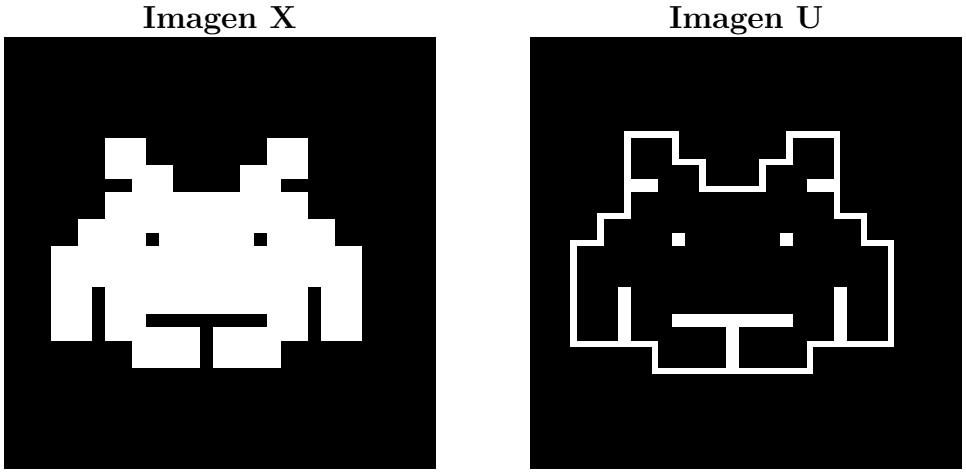


Figura 4.9: Ejemplo de procesamiento para la detección de bordes de una imagen binaria en un solo ciclo de procesamiento.

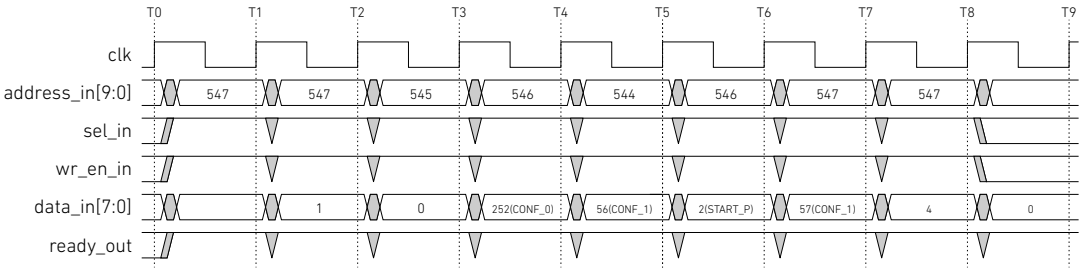


Figura 4.10: Ejemplo de temporizado de las señales IO configurando el módulo *confUnit* para la detección de bordes de X, y posterior guardado en U.

Luego de reiniciar la FIFO, se cargan luego 2 instrucciones para configurar *en_roi*, habilitando todas las entradas del vecindario, *en_regT*, *en_regU* y *en_regX* en 0, y

los bordes en cero con *sel_border* = 0. Después, se utiliza una instrucción START_P seleccionando, de la memoria de funciones (Fig.4.8), la de dilatación; y la función lógica XOR entre y y X, para la ALU. Por último, se habilita el registro *regU* para guardar el resultado, y se da comienzo al controlador para que lea la memoria de instrucciones. En la Tab.4.6 se listan los valores que se deben cargar para el ejemplo dado.

Tabla 4.5: Parámetros de las funciones utilizadas en el ejemplo para realizar la dilatación y detección de bordes.

Función	Parámetros (10x1 bits)
máximo	0 1 1 1 1 1 1 1 1 1

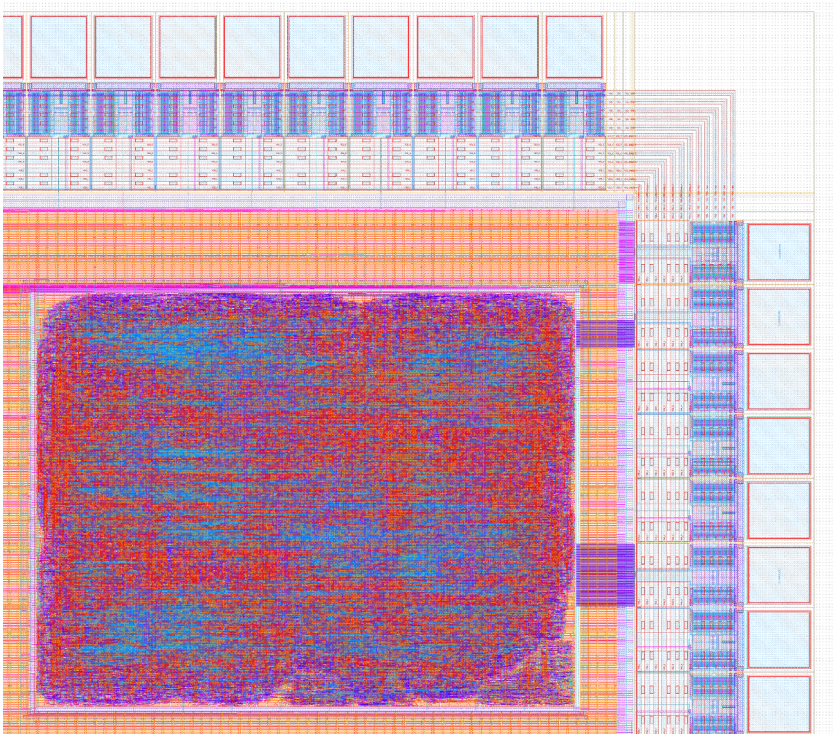
Tabla 4.6: Configuración de *confUnit* para implementar la la detección de bordes en X.

Parámetro	Valor	Descripción
en_roi	0b111111111	Se habilitan todos los valores del vecindario.
sel_border	0	Padding de ceros
sel_fx	0	Selección de la función 0 en la memoria de funciones.
sel_FoG	1	Función lógica $z = \mathsf{XOR}(y, X)$.

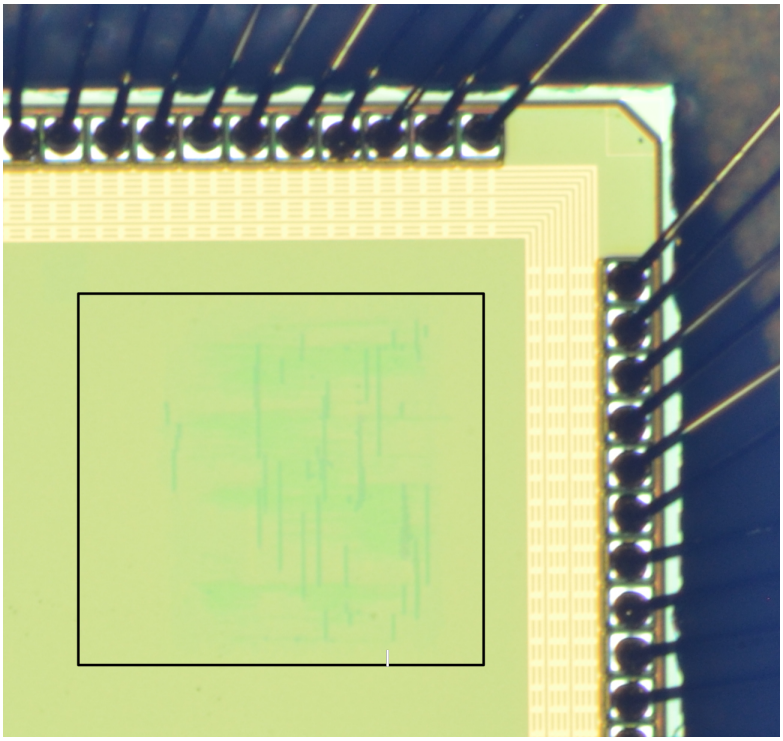
4.2.3. Resultados experimentales

La arquitectura fue íntegramente diseñada utilizando lenguaje HDL, e implementada en la tecnología CMOS 55nm LPX de GlobalFoundries. La fabricación se hizo conjuntamente con otros diseños, por lo que el anillo de PADs para las señales de entrada y salida fueron compartidas (exceptuando las de alimentación); y debido a la falta de área, en vez de implementar un arreglo de 64×64, se pudo fabricar uno de 48×48. Se utilizó una librería de celdas estándar digitales para bajo voltaje, permitiendo el escalado de la tensión de alimentación para correr el chip en un modo de bajo consumo. En la Fig.4.11(a) se observa el diseño enviado, el cual esta compuesto por 1,43 millones de transistores, ocupando un área de $0{,}85 \times 0{,}68 mm^2$.

El testeo se realizó utilizando una placa de desarrollo personalizada montada a una placa OpalKelly XEM6310, con la cual se generan los vectores de excitación necesarios para configurar y analizar el rendimiento del chip. Con una tensión de



(a) Mascara del chip implementado en 55nm.



(b) Foto del chip.

Tabla 4.7: Características del chip MORPHO1SYM en 55nm.

Tecnología	55nm de GlobalFoundries LPX
Tamaño del arreglo	48×48
Área de core	$0{,}85 \times 0{,}68$ mm^2
Cantidad de transistores	$1,43$M
Precisión de entrada	1bit
Precisión de parámetro	1bit
Memoria de imágenes	3
Algoritmo	Lineal a tramos simétricas
Máxima dimensión de función	9
Velocidad de trabajo	100 MHz
Cantidad de ciclos de reloj por procesamiento	11
Operaciones por segundo	$1,712$ TOPs/s

core de $V_{DD} = 0{,}6$V el chip funciona a una frecuencia de reloj igual a $F_{core} = 75$MHz, con un consumo de $3mW$.

Para el cálculo de su rendimiento y eficiencia se mantiene la definición de operación presentada en el Sistema MORPHO1PWL de una operación lógica de 1 bit. En cada celda se suman 9 bits al inicio de cada ciclo de cómputo (4 FA y 4 HA). Cada FA tiene 5 compuertas lógicas de 2 bits, y cada HA tiene 2. Esto representa un total de $20 + 8 = 28$. Luego se realizan una comparación de 4 bits durante 10 ciclos de reloj, agregando 8 operaciones por reloj; y por ultimo 1 operación de la ALU. Esto resulta en un total de $28 + 10 \times 8 + 1 = 109$ operaciones por celda en un ciclo de procesamiento; lo que equivale a $109 \times 48 \times 48 \times 75 \times 10^6 \div 11 = 1,712$ TOPs/s, loagrando una eficiencia de $570,763$ TOPs/W.

4.3. Procesador en escala de grises

El algoritmo de cómputo simplicial simétrico también puede aplicarse con mayor resolución, lo cual es de interés para el procesamiento de imágenes en escala de grises. Se realiza entonces un sistema que utiliza el procesamiento por columnas para entradas de 8 bits, en donde las celdas implementan la función:

$$y(k+1) = F\left(\mathbf{x}(k), \mathbf{c}\right) \circ G\left(\mathbf{u}(k), \mathbf{d}\right) \tag{4.6}$$

donde $F : \mathbb{R}^9 \longrightarrow \mathbb{R}^1$ y $G : \mathbb{R}^1 \longrightarrow \mathbb{R}^1$ son funciones simétricas simpliciales lineales a tramos, $\mathbf{u} \in \mathbb{R}^1$ y $\mathbf{x} \in \mathbb{R}^9$ son entradas de 8 bits que conforman el vector de estados correspondiente a la esfera de influencia (Fig.4.14), $\mathbf{c} \in \mathbb{R}^{10}$ y $\mathbf{d} \in \mathbb{R}^2$ son vectores de 10 y 2 parámetros respectivamente, ambos de 3 bits de resolución; $\circ$ es una función digital programable, y k es el índice de tiempo discreto. El diseño, denominado MORPHO8SYM, esta compuesto por un vector de procesadores simpliciales de tipo SIMD, los cuales, en paralelo procesan por columna una imagen alojada en una memoria local, aplicando funciones simétricas de 10 parámetros almacenadas en un conjunto de registros internos.

Arquitectura

En la Fig.4.12 se muestra el esquemático del sistema. El vector de 64 PE, procesa por columnas de 64 píxeles de alto, la imagen X alojada en una memoria cache RAM local de 32Kb. Al igual que en el diseño anterior, los resultados son también vectores columnas de igual tamaño, que se alojan en la memoria X, preservando su ubicación relativa en la imagen. Cada procesador tiene 3 registros de 8 bits, lo cuales almacenan por ciclo de procesamiento, el valor de los píxeles correspondientes a la columna de la izquierda, del centro, y de la derecha; que en conjunto con los correspondientes de las celdas de arriba y abajo, conforman el vector de estados de nueve elementos $\mathbf{x}$. Por otro lado, el valor U proviene de un registro interno de la celda, el cual almacena resultados parciales. Cada elemento de la vecindad, puede ser habilitado o no, a través de una señal de control, al igual que en MORPHO1SYM. Las funciones F y G pueden ser seleccionadas desde un banco de funciones local, capaz de guardar hasta 16 diferentes set de parámetros. Finalmente, una ALU de

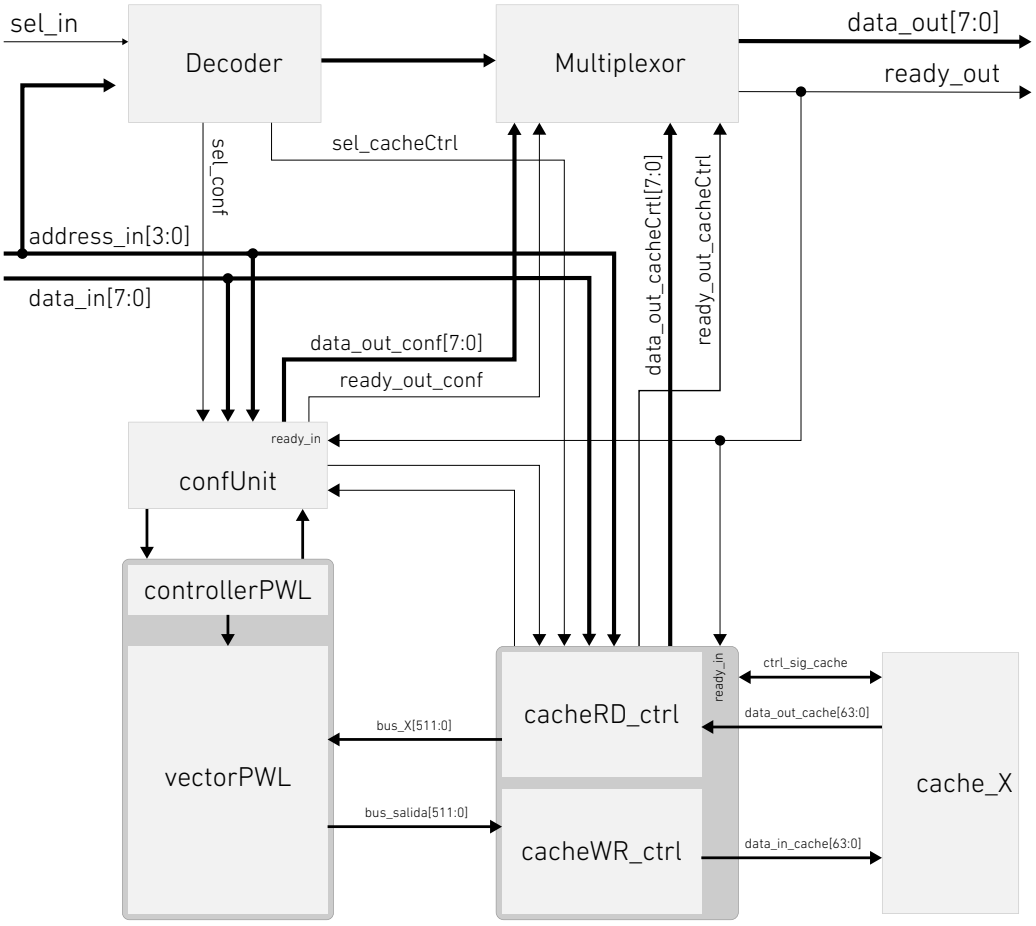


Figura 4.12: Esquemático básico del sistema.

ocho funciones es utilizada para dar como resultado $F \circ G$, la cual representa la salida del procesador, pero también puede ser almacenada localmente para tomar el valor de U. Al igual que en MORPHO8PWL, para el acceso a la memoria desde el exterior, o internamente, para procesar su contenido, dos unidades de control son implementadas: *cacheRD_ctrl* en la lectura, y *cacheWR_ctrl* para su escritura. El manejo de las señales de procesamiento que controlan en las celdas y la comunicación con las interfaces de la memoria, es llevada a cabo por un módulo de control denominado *controllerPWL*. Este ejecuta instrucciones provenientes de una memoria de tipo FIFO alojada en la unidad de configuración general *unitConf*. Por razones de implementación, el bus de dato de la cache no es del tamaño de una columna completa de 64 pixeles de 8 bits (512 bits), es necesario diseñar los controladores de memoria y de procesamiento teniendo en cuenta que se necesita acceder varias veces a la memoria para leer o escribir una columna entera. A continuación, se detallan

las distintas partes que conforman la arquitectura, empezando por el procesador.

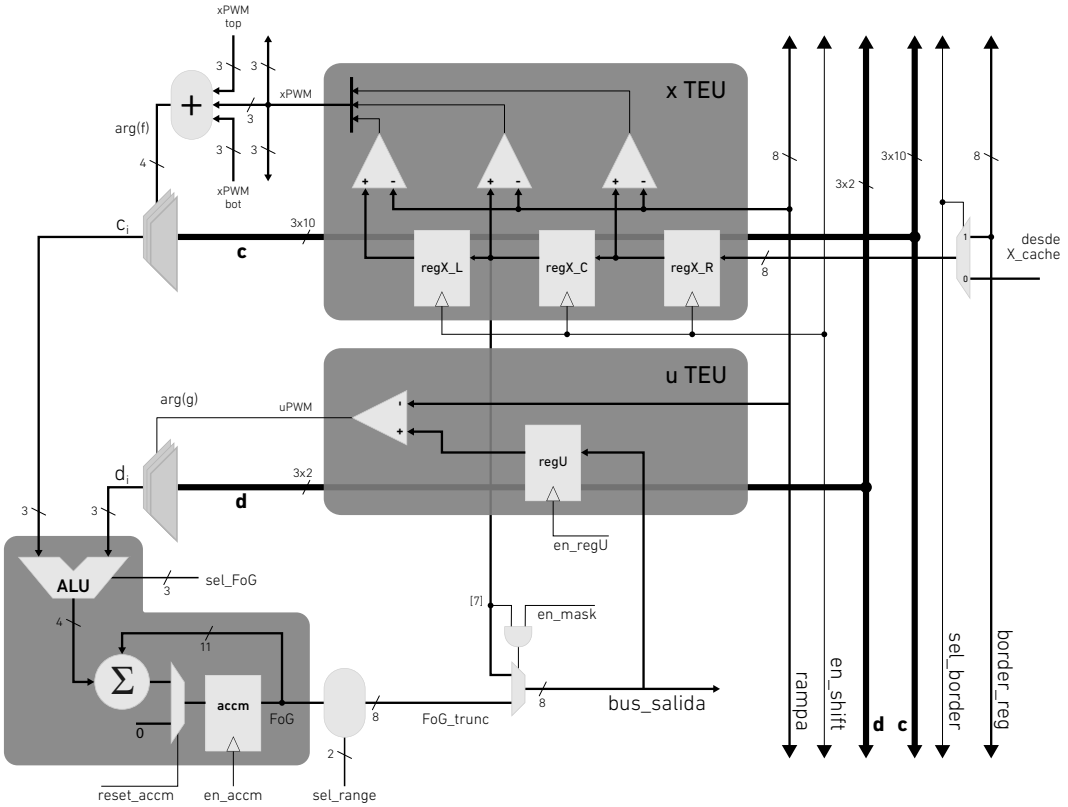


Figura 4.13: Esquema del elemento de procesamiento.

Para el cálculo de F, cada procesador posee tres registros de 8 bit, conectados en serie, denominados *regX_L*, *regX_C*, y *regX_R* en Fig.4.13, los cuales contienen el valor de los píxeles de tres columnas consecutivas provenientes de la memoria de imagen. Cada salida se compara ciclo de reloj con una rampa digital global, generado tres señales PWM de un bit, $\tilde{x}_L$, $\tilde{x}_C$ y $\tilde{x}_R$ respectivamente, que representan en tiempo los valores almacenados en los registros *regX*. Este set de tres señales xPWM, se suman junto que las 6 correspondientes a los dos procesadores vecinos, produciendo el argumento de la función:

$$\arg(f) = b_0\tilde{x}_0 + b_1\tilde{x}_1 + \cdots + b_8\tilde{x}_8 \tag{4.7}$$

donde $0 \leq \arg(f) \leq 9$ es un valor de 4 bits, y $en\_roi = \{b_0, b_1, ..., b_8\}$ es un vector de enmascaramiento, que permite la habilitación de las celdas intervinientes en la conformación del argumento, agregando la posibilidad de modificar la estructura de la vecindad. Por su parte, para el cálculo de G, el registro *regU* es cargado con

resultados previos realizados en el procesador, y su valor se compara con la misma rampa, generando el argumento $arg(g)$ de 1 bit.

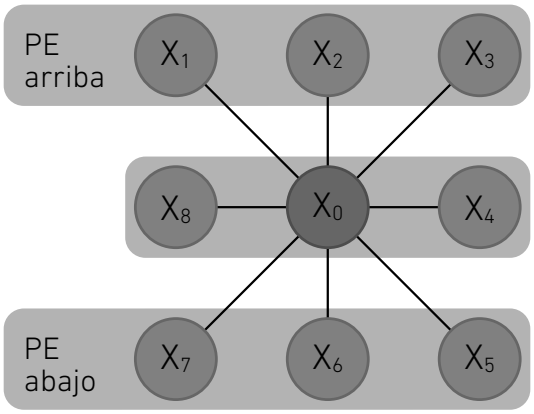


Figura 4.14: Configuración en equis × de $arg(f)$.

En un ejemplo ilustrado en Fig.4.15, se muestra como el vecindario de 9 elementos $\{x_0, x_1, ..., x_8\}$ (Fig.4.14) se compara con la rampa, generando el argumento de la función. Al inicio de un ciclo de procesamiento, $arg(f)$ es igual a 1001, y su valor va decreciendo de uno en uno a medida que la rampa incrementa y es superior a los x_i. Una vez que la rampa llega a su fin, $arg(f)$ se encuentra en el estado 0000, y la función F es calculada:

$$F\left(\mathbf{x}(k), \mathbf{c}\right) = \sum_{i=0}^{9} \mu_i \hat{c}_i \tag{4.8}$$

siendo $\{\mu_0, \mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6, \mu_7, \mu_8, \mu_9\} = \{\hat{x}_0, \hat{x}_1 - \hat{x}_0, \hat{x}_2 - \hat{x}_1, \hat{x}_3 - \hat{x}_2, \hat{x}_4 - \hat{x}_3, \hat{x}_5 - \hat{x}_4, \hat{x}_6 - \hat{x}_5, \hat{x}_7 - \hat{x}_6, \hat{x}_8 - \hat{x}_7, 1 - \hat{x}_8\}$, $\{\hat{x}_0, \hat{x}_1, \hat{x}_2, ..., \hat{x}_8\} = sort\{x_0, x_1, ..., x_8\}$ el vector de estado ordenado, y los $\hat{c}_i$ los parámetros direccionados por $arg(f)$. Los valores iniciales y finales de la rampa, así como el valor de su incremento, pueden también ser programados externamente sobre el controlador, habilitando funcionalidades al procesador.

La señal $arg(f)$, a través de un multiplexor, selecciona uno de los 10 parámetros de $\mathbf{c}$ de 3 bits correspondiente a la función elegida. La señal $arg(g)$, al ser de 1 bit solamente, selecciona uno de los dos parámetros de $\mathbf{d}$, también de 3 bits. Cada c_i seleccionado es operado con d_i en una ALU, generando una señal de 4 bits, denominada FoG_i, la cual, según el valor de la señal de selección *sel_FoG*, puede

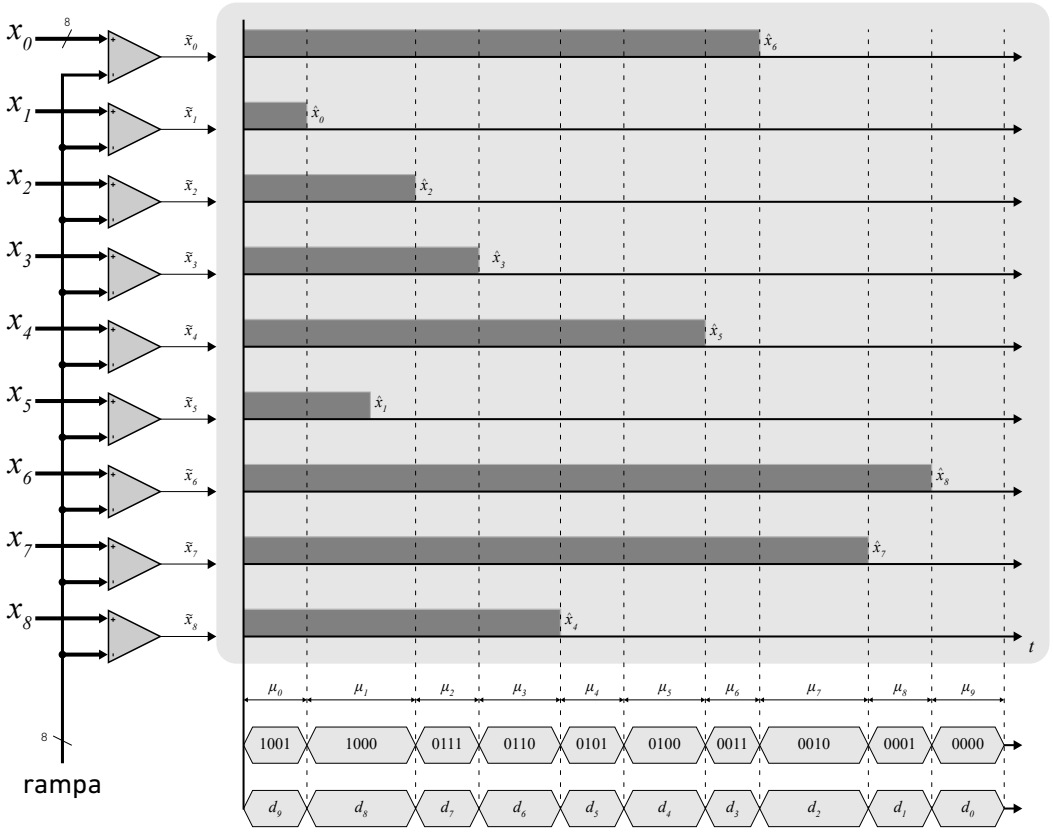


Figura 4.15: Actividad en un ciclo de procesamiento, mostrando el ordenamiento en tiempo del vector de estados y el direccionamiento de los parámetros. En este ejemplo

tomar los siguientes valores: $c_i + d_i$, $\mathsf{MAX}(c_i, d_i)$, $\mathsf{MIN}(c_i, d_i)$, $|c_i - d_i|$, o las funciones binarias $\mathsf{AND}(c_i, d_i)$, $\mathsf{OR}(c_i, d_i)$, $\mathsf{NOR}(c_i, d_i)$ y $\mathsf{XOR}(c_i, d_i)$ (Tabla 4.8).

Tabla 4.8: Selección de función en la ALU.

sel_FoG	Función
0	$\mathsf{AND}(c_i, d_i)$
1	$\mathsf{XOR}(c_i, d_i)$
2	$\mathsf{OR}(c_i, d_i)$
3	$\mathsf{NOR}(c_i, d_i)$
4	$\mathsf{MAX}(c_i, d_i)$
5	$\mathsf{MIN}(c_i, d_i)$
6	$c_i + d_i$
7	$\lvert c_i - d_i \rvert$

Posteriormente, FoG_i es acumulado en un registro de 11 bits *ACCM* que, una vez finalizada la rampa, almacena el resultado de la función compuesta *FoG*. Con la señal *sel_range* se procede al truncado de *FoG*, generando una señal de 8 bits

(*FoG_trunc*) que pueda ser leída y guardada (Tabla 4.9) en la memoria de imagen X, a través del *bus_salida*, y en el registro *regU* llevando *en_regU* a uno.

Tabla 4.9: Selección rango de *FoG* con *sel_range*.

sel_range	*FoG_trunc*
0	FoG[7:0]
1	FoG[8:1]
2	FoG[9:2]
3	FoG[10:3]

Por último, se mantiene la prestación de habilitar el procesamiento por píxel presentada en MORPHO8PWL. En este caso, cuando la señal de control *en_mask* se encuentra en alto, se evalúa si el bit más significativo de *regX_C* es 1 para que suceda el enmascaramiento del resultado. Como se puede observar, la utilización de esta función disminuye la precisión efectiva de los datos de entrada a 7 bits, además de ser necesaria una reconfiguración del valor final de la rampa en el módulo *confUnit*.

Las celdas de procesamiento se instancian en un vector denominado *vectorPWL* en la Fig.4.16. Cada PE se conecta localmente con los ubicados arriba y abajo a través de las señales PMW de los X para la conformación de la vecindad de cómputo. Las señales *rst_accm*, *en_accm*, $\boldsymbol{c}$, $\boldsymbol{d}$, *rampa*, *sel_FoG*, *en_roi*, *sel_range*, *en_mask* provenientes del modulo de control, agrupadas en*señales globales*, ingresan y se distribuyen globalmente a todos los procesadores para su control y configuración. El vector de datos de entrada *vector_bus_X* de $8 \times 64 = 512$ bits, proveniente de la memoria de imagen X, ingresa a cada procesador en forma de palabra de 8 bits según la posición del PE. Por otro lado, las palabras de *bus_salida* se sacan del vector para podes ser almacenado en la memoria de imagen. Dado que el primer y último procesador no tienen el vecindario completo, a estos ingresa una señal PWM repetida tres veces, producto de la comparación entre *rampa* y el valor del registro *border_value*, ubicado en la unidad *confUnit*.

Para la configuración del controlador *controllerPWL* que administra la señáles de La unidad de interfaz *confUnit* se accede desde el exterior para la configuración del controlador y los registros que intervienen en la etapa de procesamiento. Se

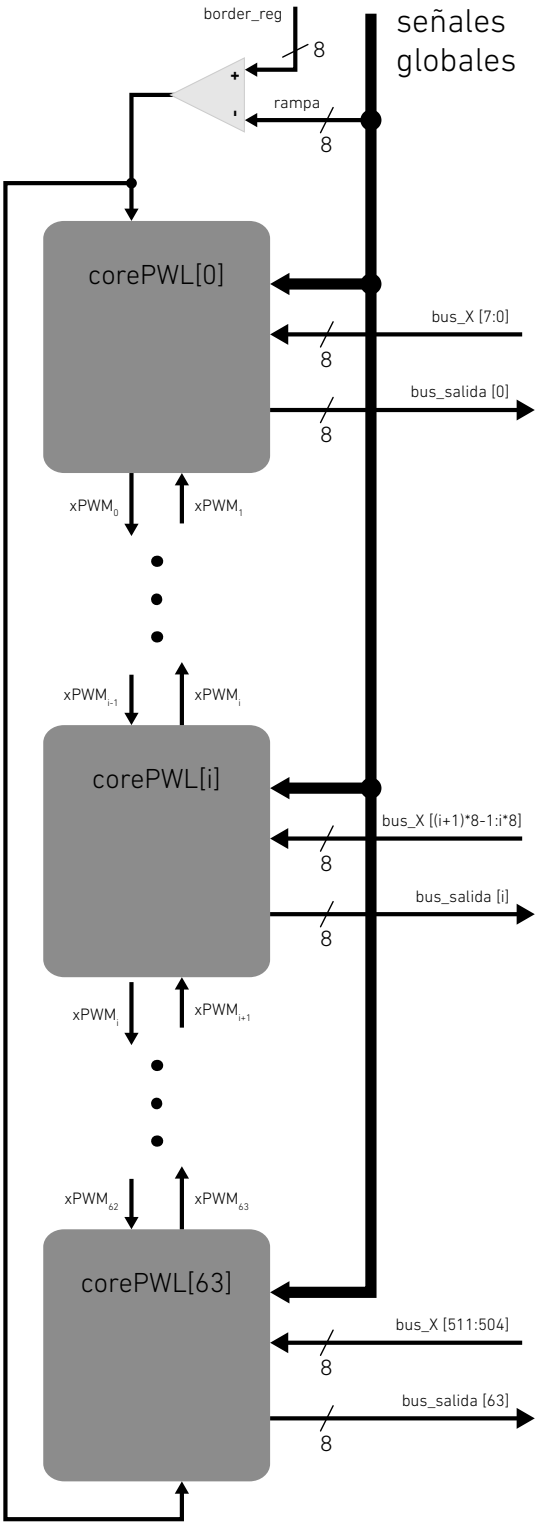


Figura 4.16: Diagrama del vector de procesadores *vectorPWL*.

encuentra conformado por una memoria de almacenamiento de funciones **c** y **d**, otra de selección de vecindario, una memoria de programa de tipo FIFO, y una serie de registros de control y configuración de procesamiento, los cuales son listado y

detallados a continuación.

Memoria de parámetros de función. Banco de registros que permiten almacenar hasta ocho sets de parámetros de 3 bits asociados a las funciones **c** y **d**, y ocho configuraciones de vecindario *en_roi*. Para su carga, se escribe en serie la dirección lógica 8, 21 parámetros totales: 10 de **c**, 2 de **d**, y 9 de *en_mask*. En el bit 0 del bus de entrada se carga el primer set de cada uno, en el bit 1 el segundo, y así sucesivamente hasta el bit más significativo que corresponde al octavo set :

$$data\_in[7:0] = \{set\_7, set\_6, set\_5, set\_4, set\_3, set\_2, set\_1, set\_0\} \tag{4.9}$$

La selección de **c** y **d** se realiza en conjunto a través de la señal de 4 bits *sel_f*, y *en_roi* con *sel_roi* de 3 bits; ambas provenientes de los registros de configuración. En especial, con *sel_f* se puede seleccionar funciones que se encuentran *hardcodeadas* (Tab.4.10).

Tabla 4.10: Selección de funciones de la memoria de parámetros utilizando la señal *sel_f*.

sel_f	**c** bits	**d** bits
0	$\{0, 0, set\_0\}$	
1	$\{0, 0, set\_1\}$	
2	$\{0, 0, set\_2\}$	
3	$\{0, 0, set\_3\}$	
4	$\{0, 0, set\_4\}$	
5	$\{0, 0, set\_5\}$	
6	$\{0, 0, set\_6\}$	
7	$\{0, 0, set\_7\}$	
8	$\{set\_2, set\_1, set\_0\}$	
9	$\{set\_5, set\_4, set\_3\}$	
10	$\{0, set\_7, set\_6\}$	
11	SUM_4	COPY
12	MAX	COPY
13	MIN	COPY
14	MEDIAN	COPY
15	MAX_INV	COPY

Memoria de instrucciones. Memoria de tipo FIFO de 8 bits de palabra y profundidad igual a 16, capaz de almacenar dos tipos de instrucciones (Tab.4.12). Para

cargarla se debe escribir en la dirección 0, previo reinicio a través de los registros de control. Su lectura esta a cargo del controlador *controllerPWL* cada vez que se inicia un ciclo de procesamiento, sin la perdida del contenido de la memoria, al igual que MORPHO1SYM.La FIFO implementa señales de bandera *full_flag* y *empty_flag* para indicar si se encuentra llena o vacía, respectivamente; y otra *empty_flag_rd* que indica con un 1 cuando la memoria ha sido completamente leída.

Registros de configuración y control de procesamiento. Esta compuesto por dos grupos de registros: el primero, integrado por *sel_f*, *sel_FoG*, *sel_range* y *sel_roi*, es configurado por el controlador a través de la lectura de la memoria de programa; el segundo es seteado desde el exterior (Tab.4.11), y funcionan para la configuración del procesamiento y el control de la maquina de estado que integra *controllerPWL*:

- **sel_f**: registro de 4 bits que selecciona el set de parámetros **c** y **d** para procesar (Tab.4.10). Es seteado cuando el controlador lee la instrucción START_P (Tab.4.12).
- **sel_FoG**: registro de 3 bits que selecciona el la función a implementar en la ALU de los procesadores (Tab.4.9). Es seteado cuando el controlador lee la instrucción START_P (Tab.4.12).
- **sel_range**: registro de 2 bits que que selecciona los 8 bits de *FoG* de los procesadores que se desean almacenar (Tabla 4.8). Es seteado cuando el controlador lee la instrucción CONFIG (Tab.4.12).
- **sel_roi**: registro de 3 bits que selecciona de uno set de parámetro de configuración de vecindario *en_roi*, de los 8 previamente almacenados (Tabla 4.8). Es seteado cuando el controlador lee la instrucción CONFIG (Tab.4.12).
- **start_ramp_value** : registro de 8 bits que almacena el valor inicial de la rampa de procesamiento. Se configura escribiendo en la dirección 3.
- **op_ramp_value** : registro de 8 bits que almacena el valor de incremento de la rampa de procesamiento. Se configura escribiendo en la dirección 4.

- **final_ramp_value** : registro de 8 bits que almacena el valor final de la rampa de procesamiento. Se configura escribiendo en la dirección 5.
- **border_value** : registro de 8 bits que almacena el para la condición de borde la imagen a procesar. Se configura escribiendo en la dirección 6.
- **reset_fifo** : registro de 1 bits que reinicia la memoria de instrucciones de tipo FIFO. Se configura escribiendo el bit 0 en la dirección 7.
- **reset_ctrl** : registro de 1 bits que reinicia el controlador de procesamiento. Se configura escribiendo el bit 2 en la dirección 7.
- **en_mask** : registro de 1 bits que habilita el procesamiento enmascarado dentro de los procesadores. Se configura escribiendo el bit 3 en la dirección 7.

Tabla 4.11: Ubicación en memoria de los registros de configuración recordando que el bus de datos es de 8 bits.

Dirección	bits del bus de dato							
	7	6	5	4	3	2	1	0
0	Memoria de instrucciones (Tab.4.12)							
1	-							
2	-							
3	start_ramp_value							
4	op_ramp_value							
5	final_ramp_value							
6	border_value							
7	-				en_mask	reset_ctrl	-	reset_fifo

Controlador de procesamiento PWL

En la Fig4.18 se ilustra el diagrama en bloques de la máquina de estados implementada dentro del controlador de procesamiento *controllerPWL*. Este se comunica con los controladores de memoria para recibir datos de entrada, y enviar los resultados a la cache; además de administrar las señales de procesamiento que se dirigen a

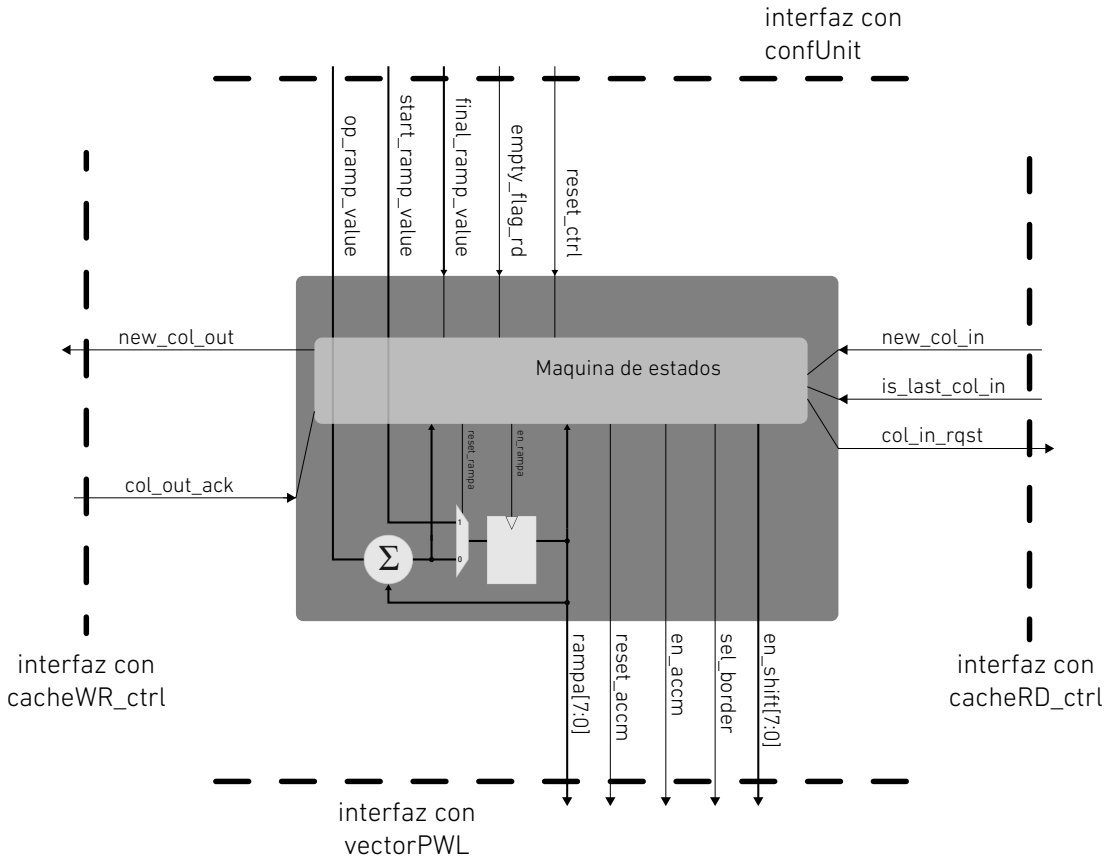


Figura 4.17: Esquemático simple del controlador PWL y sus conexiones con los controladores de la memoria cache, y la unidad de configuración.

las celdas (Fig.4.17). Como se enuncio en la introducción de la arquitectura, el controlador es capaz de cargar por sectores o bloques las distintas partes que componen un vector columna de entrada. Es por eso que, como se puede observar en Fig.4.17, la señal *en_shift_reg* es un vector de 8 lineas, los cuales ingresan cada una a un grupo de 8 PE, para activar por sectores la habilitaciones de los registros *regX*.Cuando la señal interna *rst_col_block* es uno, las lineas toman el valor 0, y luego se activan en orden creciente,de a uno, conforme *incr_col_block*. Cuando el bit 7 de *en_shift_reg* es uno, la señal interna de bandera *last_col_block* se activa, indicando que todos los sectores ya han guardados sus respectivos bloques de la columna de datos de entrada. La señal *en_all_block* activa todo el vector *en_shift_reg* para el caso de que se carguen los bordes de relleno de la imagen. Además, un acumulador de 8 bits se implementa para generar la señal *rampa*. Cuando la señal interna *rst_rampa* es 1, este se inicia con el valor alojado en *start_ramp_value*. Con *incr_ramp_value* su valor

se va incrementando a un paso dado por *op_ramp_value*, para finalmente llegar al valor *final_ramp_value*, generando que la señal de bandera interna *ramp_end* pase a un estado lógico alto. A continuación se listarán y detallaran los 7 estados intervinientes de la maquina. El reinicio de la misma se da al pulsar la señal *reset_ctrl*. Entonces el primer estado es asignado, y la maquina espera la existencia del primer vector de entrada que los controladores de memoria se inicialicen para dar comienzo a la lectura, procesamiento, y guardado de datos:

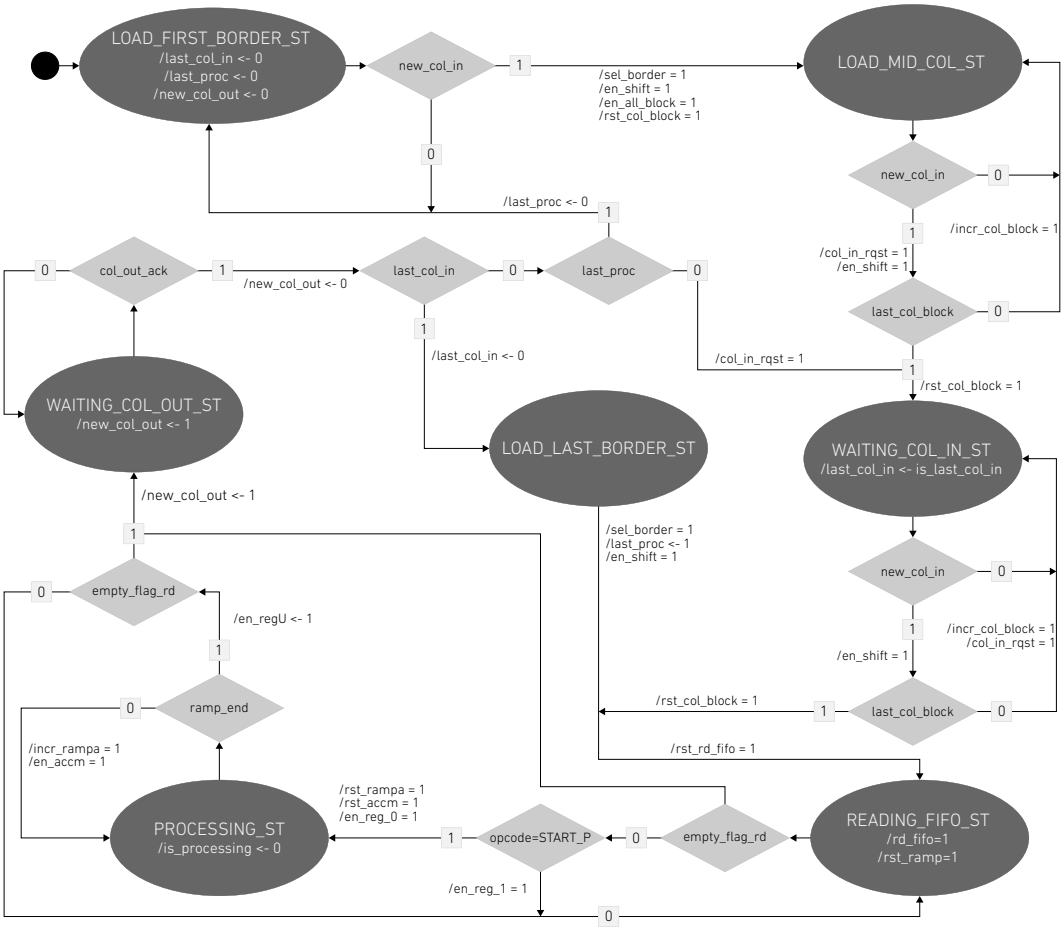


Figura 4.18: Diagrama de estados de la maquina implementada en *controllerPWL*. En el controlador se implementa dos registros auxiliares, *last_col* y *last_proc*, como señales bandera de la última columna de datos de entrada y el último procesamiento realizado respectivamente.

- **LOAD_FIRST_BORDER_ST** : se espera que el módulo *cacheRD_ctrl* se inicialice externamente y que indique la lectura del primer bloque de datos a través de la señal *new_col_in*. Sin embargo estos no son utilizados; se cargan los PE con el valor de borde almacenado en *border_value* como primer columna, y se avanza al estado LOAD_MID_COL_ST.

- **LOAD_MID_COL_ST** : se registran sucesivamente los distintos bloques de datos leídos bajo demanda (*col_in_rqst* = 1), que conforman la primer columna de imagen, en sus respectivos sectores de PE; y se avanza a WAITING_COL_IN_ST donde se espera por la tercer columna, para así empezar el primer ciclo de programa.

- **WAITING_COL_IN_ST** : con *col_in_rqst* en uno, el controlador pide nuevos datos a memoria, hasta completar una columna. Se reinicia el puntero de lectura de la memoria de instrucciones y se procede al estado READING_FIFO_ST.

- **READING_FIFO_ST** : se lee la FIFO de programa y se evalúan las instrucciones leídas. Cuando la señal de bandera *empty_flag_rd* es uno, indicando que no quedan más por leer, se pasa al estado WAITING_COL_OUT_ST. Si la instrucción es de tipo CONFIG, se pulsa *en_reg_1* para configurar los registros *sel_range* y *sel_roi*, y se sigue leyendo. En el caso de que la instrucción es de tipo START_P, se configuran los registros *sel_f* y *sel_FoG* con *en_reg_0*, se reinicia el contador de la *rampa*, y se procede a inicial un ciclo de procesamiento yendo al estado PROCESSING_ST. Las instrucciones de programa se listan en Tab.4.12.

Tabla 4.12: Descripción de las instrucciones de programa que se leen durante el estado READING_FIFO_ST. En todos los casos se actualizan los registros que aparecen dentro de la instrucción. Por ejemplo, en CONFIG se actualizan los tres bits mas significativos de *en_roi*, ademas de los registros que habilitan el almacenado dentro de las celdas de procesamiento.

Opcode	instrucción[7:0]							
	7	6	5	4	3	2	1	0
START_P	0	sel_FoG			sel_f			
CONFIG	1 - -			sel˙roi				sel_range

- **PROCESSING_ST** : estado en el cual se realiza el cálculo en las celdas PWL. A medida que se incrementa el valor de la rampa, se mantienen habitados los registros ACCM para la acumulación de los resultados parciales. Cuando la rampa llega a su valor final, el valor de *FoG* ya se encuentra almacenado en el acumulador, y su valor truncado se almacena en *regU*. Por último se evalúa si aún quedan instrucciones a leer en la memoria de programa. En caso positivo

se vuelve a READING_FIFO_ST, de lo contrario, se le avisa al controlador de escritura *cacheWR_ctrl* que hay un vector de resultados listos para ser guardados en la cache ($new\_col\_out = 1$) y se ingresa al estado WAITING_COL_OUT_ST.

- **WAITING_COL_OUT_ST** : se espera que *cacheWR_ctrl* termine de escribir los resultados en memoria, y se evalúan los registros internos de bandera *last_col_in* y *last_proc*. Si ya la última columna de entrada fue registrada en los procesadores ($last\_col\_in = 1$), se pone en alto *last_proc* anunciando que el próximo cilo de procesamiento es el último, y se ingresa al estado LOAD_LAST_BORDER_ST, donde se registrará en los PE el valor de borde configurado. En el caso contrario, se evalúa si el último procesamiento ya ocurrió ($last\_proc = 1$). En caso psitivo, se da por concluido el procesamiento de la imagen completa, y se retorna al estado inicial LOAD_FIRST_BORDER_ST. En caso negativo, se continúa normalmente solicitando un nuevo bloque de datos de entrada ($col\_in\_rqst = 1$), y se avanza a WAITING_COL_IN_ST.

- **LOAD_LAST_BORDER_ST** : estado en el cual se carga en paralelo el valor contenido en *border_value* en los procesadores. Se setea que *last_proc* en 1, ya que el próximo ciclo de programa es el último, se reinicia el puntero de lectura de la FIFO, y se avanza a READING_FIFO_ST para comenzar a leer las instrucciones nuevamente.

Memoria de imagen y sus controladores

La memoria cache, donde se guardan la imagen de entrada y de salida de 64×64 píxeles de 8 bits de precisión, es un memoria RAM de 32Kb de un solo puerto de 64 bits de ancho, y 9 bits de direcciones. Utiliza como protocolo de comunicación interno uno de dos fases, semejante el AMBA AHB-Lite, donde en el primer ciclo de reloj se presenta la dirección, y en el segundo el dato de entrada o de salida. Las imágenes se acceden por bloques de 8 píxeles; y una columna de imagen está compuesto por 8 bloques consecutivos. Si se quiere acceder a la primer columna, entonces se debe escribir/leer los 8 bloques de las direcciones que van de 0 a 7; la segunda columna corresponde a las direcciones desde 8 a 15, y así sucesivamente (Fig.4.19).

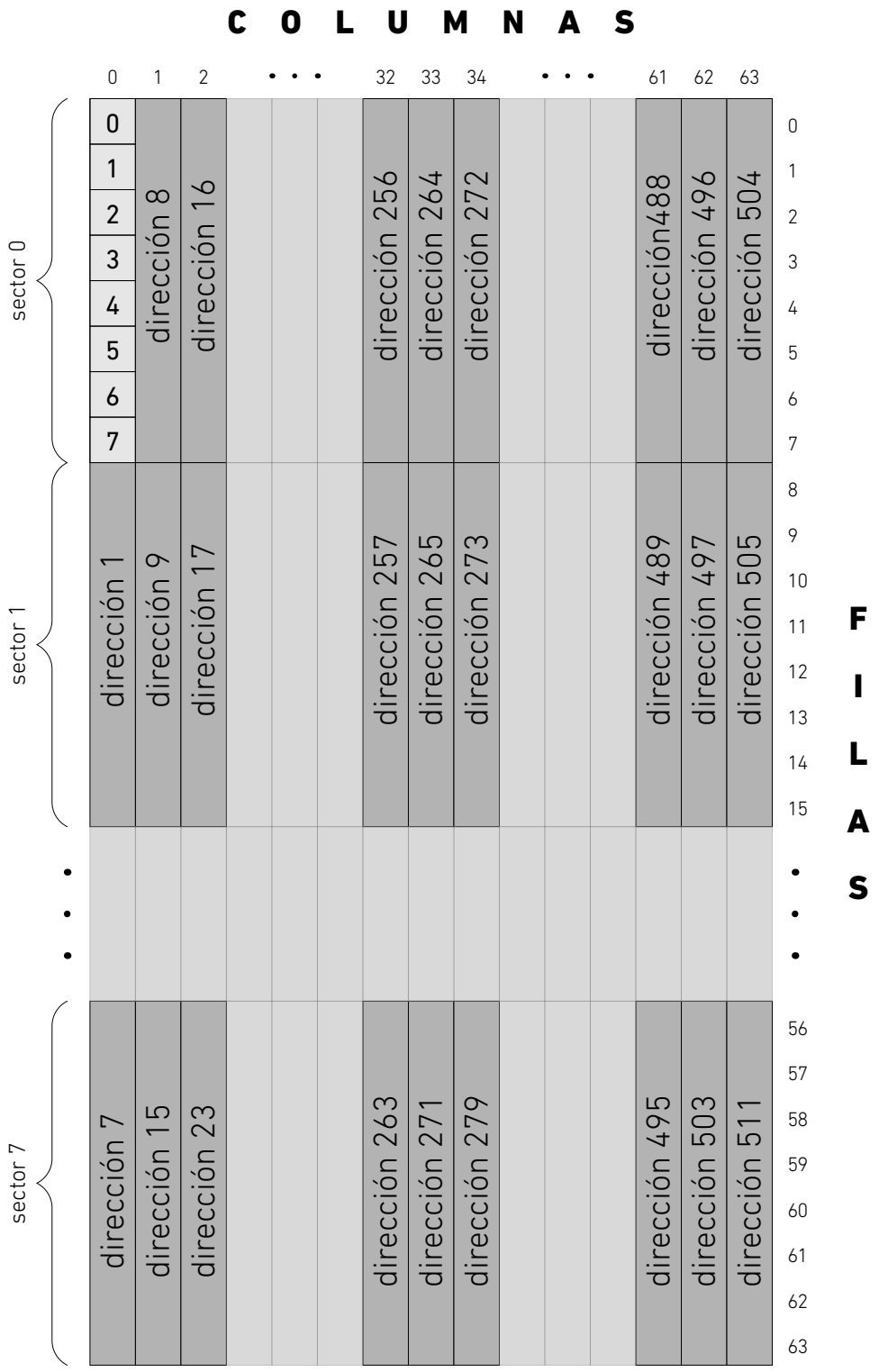


Figura 4.19: Mapeo en la memoria cache de las columnas que conforman una imagen.

Para arbitrar el acceso externo con el interno(durante el procesamiento), se implementan dos módulos controladores de memoria, uno de escritura denominado *cacheWR_ctrl*, y el segundo de lectura *cacheRD_ctrl* en Fig.4.20. Además, tres registros de un bit, configurables externamente accediendo a las dirección 14 (Tab.4.13),

manejan el inicio y suspensión del procesamiento (*start_reg* y *stop_reg*), y el reinicio de contadores de direcciones de 9 bits que cada controlador posee (*rst˙address˙reg*).

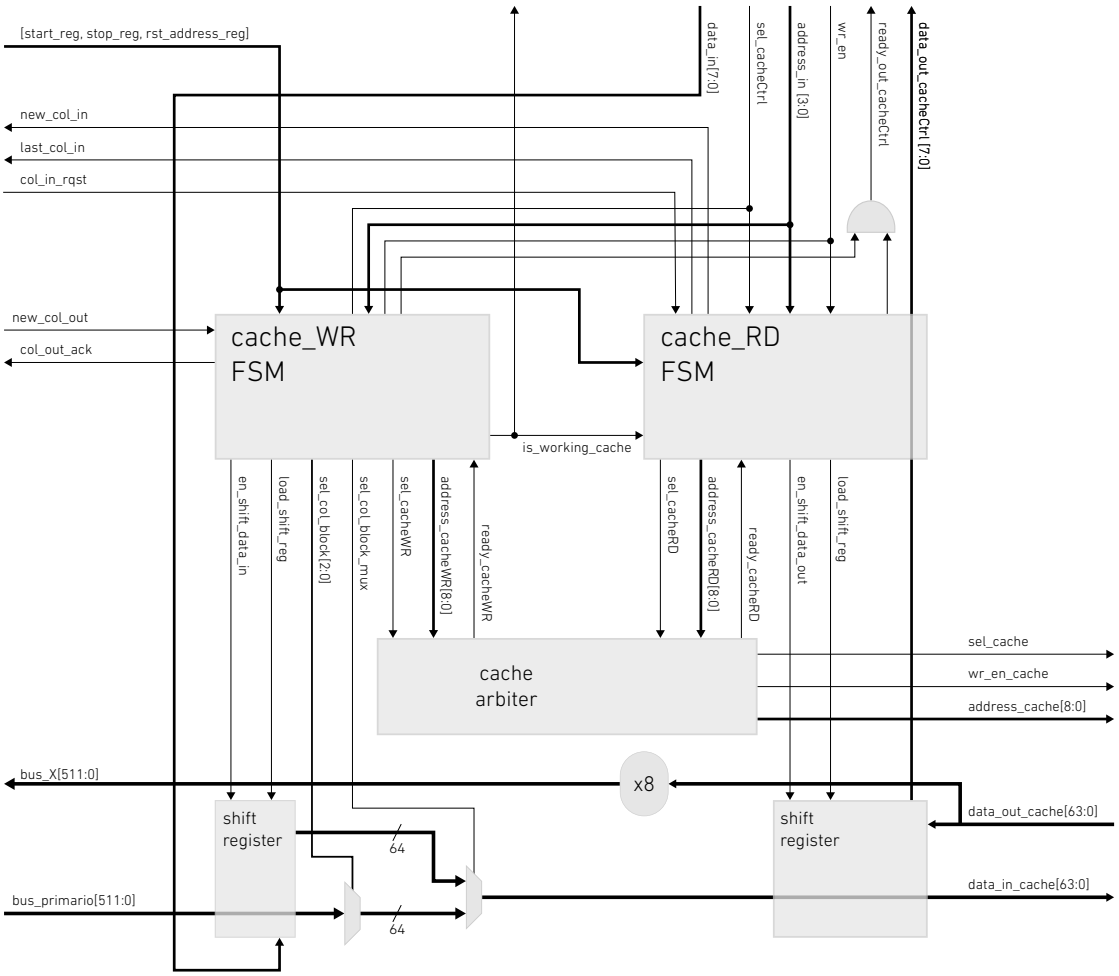


Figura 4.20: Esquemático del módulo del controlador de la memoria de imágenes.

Tabla 4.13: Palabra de control de los controladores de escritura y lectura de la memoria de imagen.

data_in[7:0]							
7	6	5	4	3	2	1	0
-			rst_address_reg	-	stop_reg	start_reg	

Para el acceso del usuario, los controladores de memoria utilizan, cada uno, un registro desplazamiento de 8 bits de ancho y 8 palabras de profundidad. La dirección de memoria de imagen está dada por el contador interno, el cual se incrementa de a uno cada vez que un proceso de escritura o lectura haya concluido, y se reinicia con

rst˙address˙reg. Para la escritura de una imagen completa, se reinicia el contador de direcciones, y se debe escribir de a bloques de 8 píxeles en serie a la dirección 12, y de manera automática el controlador *cacheWR_ctrl* carga en paralelo los distintos bloques en la memoria. Para lectura completa de una imagen, se repite el proceso de reiniciado del registro de direcciones del controlador, y se cuando se empieza a leer en la dirección 13, el controlador *cacheRD_ctrl* carga el primer bloque de 8 píxeles, y los desplaza hacia el exterior a través del bus de salida, mientras se mantenga la condición de lectura por parte del usuario a la dirección 13. Ejemplos de carga y lectura de una imagen se pueden observar en la Fig.4.23(a) y Fig.4.23(b) de la sección de Tareas.

Al igual que en MORPHO8PWL, un arbitro denominado *cache arbiter* es implementado para controlar el acceso la memoria de imagen por parte de los controladores; siempre dando prioridad al controlador de lectura. A diferencia del diseño homólogo anterior, los bloques de datos de entrada y salida a memoria durante el procesamiento no son *buffereados* por los registros de desplazamiento. A continuación se describen ambos controladores, su interacción entre ellos, con la memoria, el exterior del sistema y el *controllerPWL*.

Controlador de escritura de cache Para la implementación del controlador de escritura *cacheWR_ctrl*, se utiliza una maquina de 3 estados para la administración de las señales IO de escritura que se dirigen a la memoria de imagen, y un contador de 9 bits, que sirve de puntero para las direcciones de memoria. En la Fig.4.21 se muestra el diagrama de flujo de la maquina de estado.

Si se desea ingresar, desde el exterior, un bloque de 8 píxeles, el controlador habilita el ingreso en serie de los datos en el registro de desplazamiento, y una vez que este se encuentra cargado, el controlador ejecuta un proceso de escritura a la memoria, e incrementa el puntero. Por otro lado, si el sistema se encuentra procesando, cuando el controlador recibe la señal de que existe un nuevo vector de resultados, se inicia la escritura de 8 bloques en direcciones contiguas. En ambos casos, al inicio de una escritura o procesamiento de una imagen completa, el contador de direcciones se reinicia en 0, externamente o internamente. Los buses de datos provenientes del registro de desplazamiento y de la columna de procesadores, son multiplexados

utilizando *sel_col_block_mux* como selección. Además, la señal *sel_col_block*, producto de un contador de 3 bits, selecciona de cual de los 8 sectores de PE que contiene el *vectorPWL* se obtendrán los datos a guardar.

A continuación se listan y detallan los estados intervinientes en los distintos modos. Para acceder al primero es necesario llevar la señal de *stop_reg* por al menos un ciclo de reloj, a modo de reinicio.

- **IDLE_ST**: es el estado inicial del controlador. Si se da comienzo al procesamiento de la imagen X ($start\_reg = 1$), se reinicia el contador de direcciones y se asigna WAITING_COL_OUT_ST como próximo estado. Si se desea escribir externamente un bloque de datos ($phase\_sel\_cacheCtrl = 1$ y $phase\_wr\_en = 1$), se habilita el registro de desplazamiento, y se avanza a SHIFT_DATA_IN_ST.

- **WAITING_COL_OUT_ST**: estado en el que se espera, a través de *new_col_out*, la existencia de un nuevo vector de resultados que debe ser guardado en la RAM. De ser así, se almacenan consecutivamente en memoria los 8 bloques que confeccionan una columna, incrementando el contador de direcciones y de sector o bloque. Cuando la columna completa fue cargada, se la señal de respuesta *col_out_ack* es llevada a uno y se reinicia el puntero de sector. Por último, se evalúa en cada paso si se llegó a la dirección final igual a 511, y de ser así, se vuelve al estado inicial IDLE_ST.

- **SHIFT_DATA_IN_ST**: durante este estado se evalúa la condición de escritura externa para el registrado de los datos de entrada. Cuando La condición no es dada, se supone que el usurario ya ha cargado el bloque de píxeles completo, y se inicia el almacenado del mismo n memoria, regresando, por último al estado IDLE_ST.

Controlador de lectura de cache El módulo controlador de lectura *cacheRD_ctrl* se encuentra compuesto por una máquina de 5 estados; y al igual que el de escritura, se implementa un contador 9 bits que sirve de puntero de direcciones para la memoria, y un registro de desplazamiento de 8 bits de palabra y 8 palabras de profundidad. Para la lectura desde el usuario, el registro de desplazamiento se carga

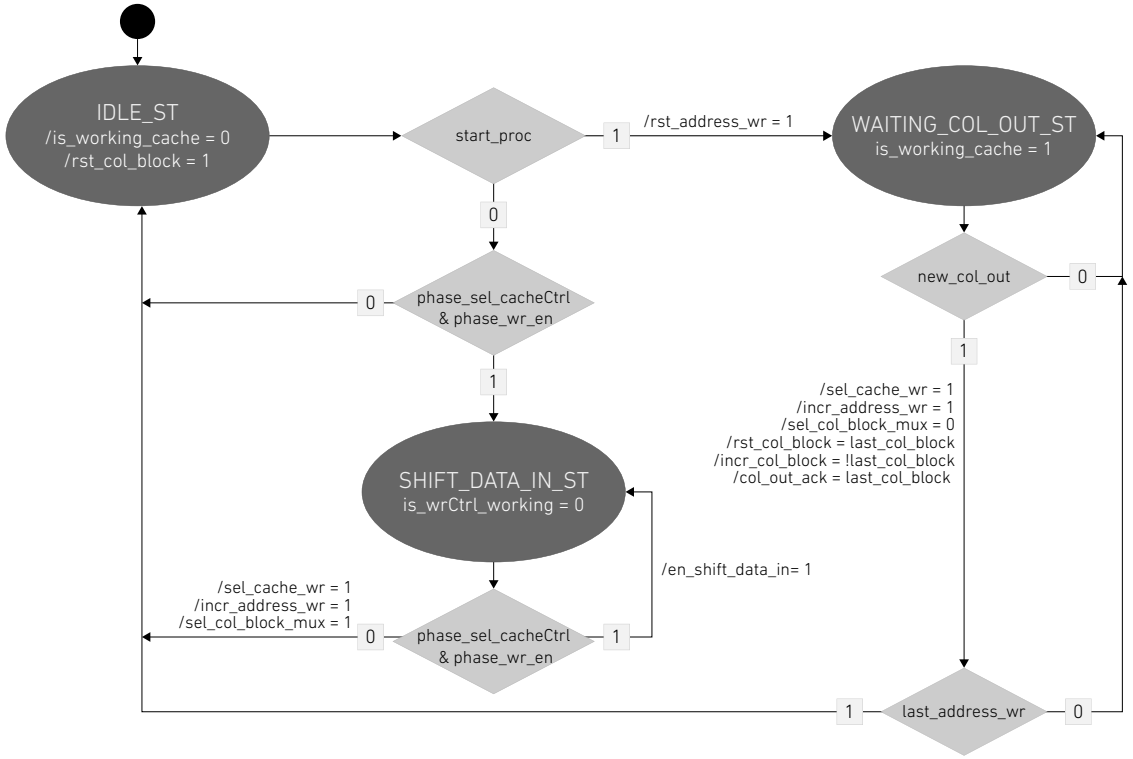


Figura 4.21: Diagrama de estados de la máquina implementada en *cacheWR_ctrl*.

en paralelo con el dato de memoria apuntado por el contador, y luego es leído en serie a traves del bus de datos de salida. En el caso de que el sistema se encuentre procesando, el controlador procede a realizar una lectura de la memoria bajo solicitud del *controllerPWL*. En ambos casos, al inicio de una escritura o procesamiento de una imagen completa, el contador de direcciones se reinicia en 0, externamente o internamente.

A continuación se listan y detallan los estados intervinientes en los distintos modos. Para acceder al primero es necesario llevar la señal de *stop_reg* por al menos un ciclo de reloj, a modo de reinicio.

- **IDLE_ST**: durante el estado inicial se evalúa que la señal *start_reg* que sea uno para reiniciar el puntero, y empezar a procesar la imagen en memoria avanzando al estado PROC_ADDR_PHASE_ST; o que desde el exterior se desee leer un bloque de columna, para iniciar la lectura de memoria a la dirección dada por el puntero, y asignar READ_MEM_OUT_ST como próximo estado.

- **PROC_ADDR_PHASE_ST**: se selecciona la memoria para su lectura, y si

se encuentra disponible, se incrementa el contador de direcciones, se le avisa al controlador de procesamiento que va a haber un dato nuevo disponible en el próximo ciclo ($new\_col\_in = 1$), y se avanza al estado PROC_DATA_PHASE_ST si no se ha leído en la última dirección; caso contrario, se lleva *lat_col_in* a uno, indicando que el último bloque va a ser leído, y se retorna al estado inicial IDLE_ST.

- **PROC_DATA_PHASE_ST**: a diferencia del PROC_ADDR_PHASE_ST, antes de acceder a memoria, se evalúa si el *controllerPWL* solicitó un nuevo dato ($col\_in\_rqst = 1$). De ser positivo, realiza un acceso a memoria del mismo modo en el que se procede en PROC_ADDR_PHASE_ST.

- **READ_MEM_OUT_ST**: se registra el dato de salida de la memoria en el registro de desplazamiento y se avanza a SHIFT_DATA_OUT_ST.

- **SHIFT_DATA_OUT_ST**: estado en el que se realiza la lectura en serie de los píxeles a través del bus de salida habilitando el registro de desplazamiento ($en\_shift\_data\_out = 1$), hasta que el usuario de-selecciona el módulo, y se vuelve al estado inicial.

4.3.1. Tareas y funcionamiento

A continuación se explica con ejemplos la realización de distintas tareas básicas en el procesador. La direcciones de memoria de los distintos módulos internos se listan en la Tab.4.14.

Tabla 4.14: Mapeo lógico de las direcciones para escritura y lectura.

Dirección	Sentido	Descripción
0-7	Escritura/ Lectura	Acceso a la unidad de configuración *confUnit* (Tab.4.11).
12	Escritura	Almacenamiento la imagen en memoria.
13	Lectura	Acceso a los bloques de la imagen alojada en memoria.
14	Escritura/ Lectura	Acceso los registros de control de los controladores de memoria (Tab.4.13).

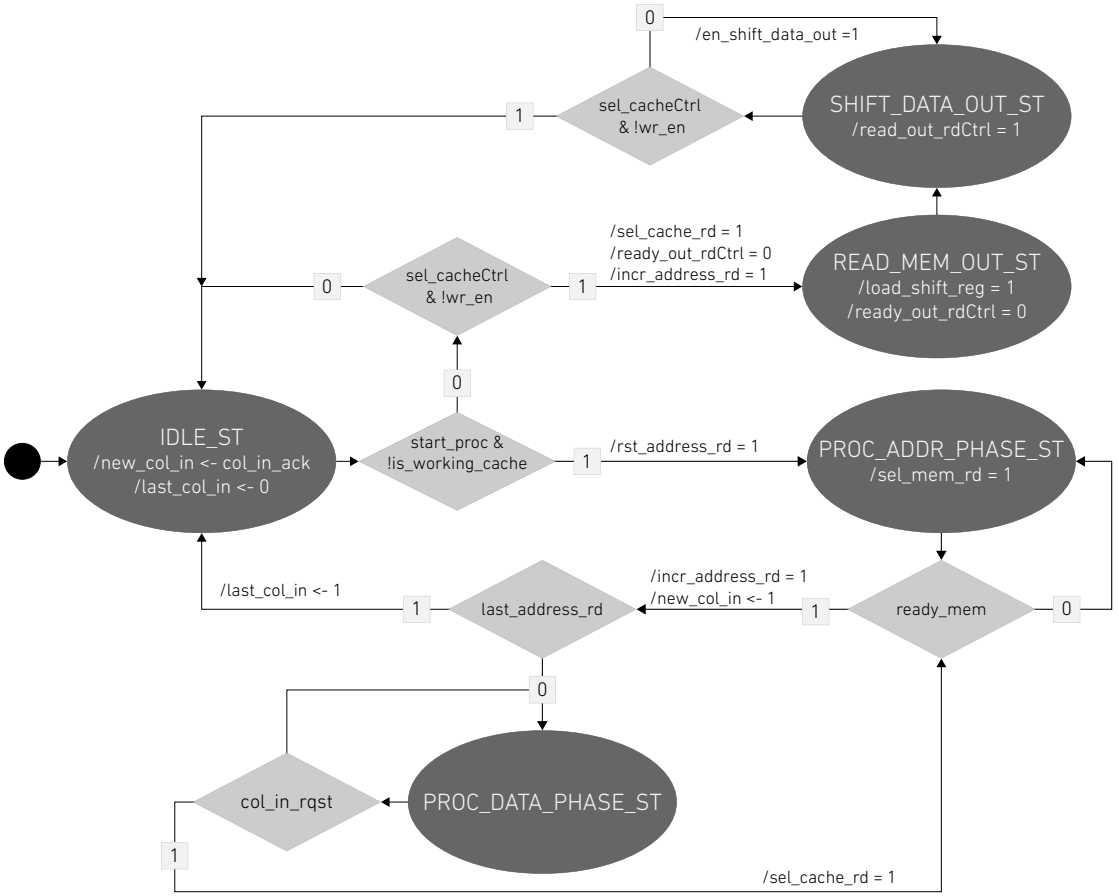


Figura 4.22: Diagrama de estados de la máquina implementada en *cacheRD_ctrl*.

Para dar inicio al sistema, es necesario pulsar la señal IO *reset*, para reiniciar las máquinas de estado de los controladores. A continuación se explica los procedimientos de escritura y lectura de las imágenes, y de configuración de los controladores para el inicio del procesamiento.

Dado que las direcciones de la memoria de imagen provienen de contadores con paso unitario, controlados por *cacheWR_ctrl* y *cacheRD_ctrl*, la escritura y lectura de una imagen debe en serie, en bloques de 8, de forma ordenada creciente, siguiendo el mapeo ilustrado en la Fig.4.19. Por lo tanto, previo a cualquier acceso completo de una imagen, es necesario reiniciar los punteros y controladores, escribiendo un 1 y 0 en los registros *rst_address_reg* y *stop_reg* en la dirección 14 (Tab.4.13). En la Fig.4.23(a) se muestra el temporizado de la señales IO para reiniciar el contador de direcciones y el ingreso del primer y segundo bloque de datos, que componen la columna 0 de X. Entre bloque y bloque, es necesario des-seleccionar el módulo para

reiniciar la maquina de estado del *cacheWR_ctrl*, y así empezar con la escritura del segundo bloque; hasta llegar al último. Para la lectura de la imagen, también se reinicia el puntero y el controlador, y se lee en serie de a 8 píxeles (Fig.4.23(b)).

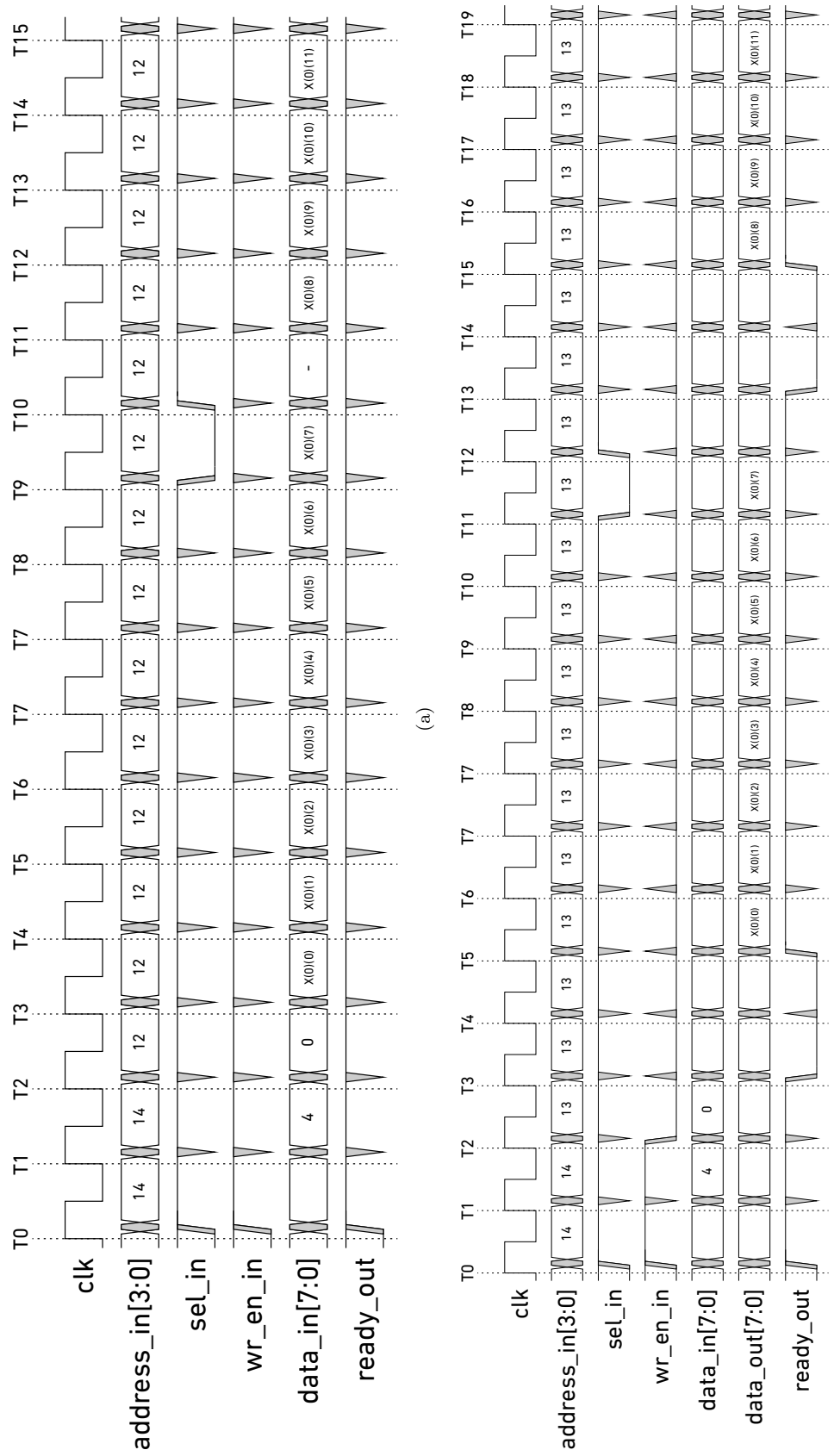


Figura 4.23: Temporizado de las señales del protocolo de comunicación del sistema para escribir y leer una imagen completa en la memoria cache.

A continuación se presenta la configuración de los parámetros y procesadores para la detección de bordes es una imagen (Fig.4.24(a)) que posee ruido "sal y pimienta" ("salt & pepper" en inglés) en un solo ciclo de lectura de imagen. Para ello es necesario, utilizar dos ciclos de rampa por vector de entrada, cuyos resultados parciales se ven ilustrados en Fig.4.24.

En el primer ciclo se hace una erosión sin buscar el mínimo, ya que en el vecindario donde haya ruido "pepper" se copiara el valor negro. Como el valor del registro auxiliar U es indefinido, se le debe aplicar la función cero. El resultado entonces es la "erosión parcial", la cual se almacena en *regU* (Fig.4.24(c)).

En el segundo ciclo se hace una dilatación sin buscar el máximo, ya que se podría copiar el ruido "salt". Dado que en U se encuentra el resultado del ciclo anterior, solo queda copiar su valor, y calcular la diferencia entre $F(x)$ y $G(u)$ para obtener la detección de bordes (Fig.4.24(f)).

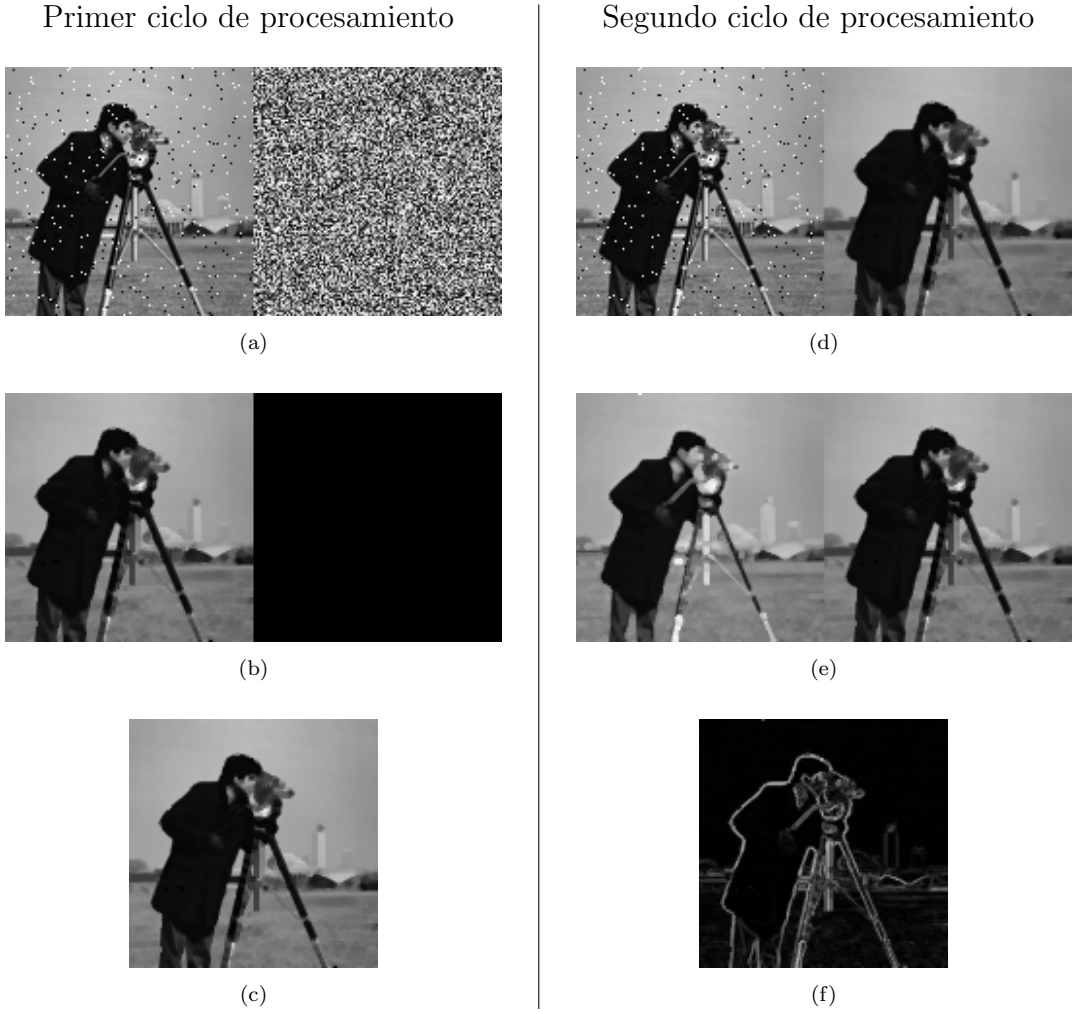


Figura 4.24: Ejemplo de eliminación de ruido "salt & pepper", y detección de bordes con un solo ciclo de lectura de memoria, y dos ciclos de procesamiento. (a) Izquierda: imagen de entrada X, Derecha: registro U (al ser el primer ciclo, su valor no es predecible y se representa con ruido); (b) Izquierda: resultado de la función de erosión en X ($F(x)$), Derecha: resultado de la función cero en U ($G(u)$); (c) Resultado de la función OR en FoG, del primer ciclo de procesamiento; (d) Izquierda: imagen de entrada X, Derecha: registro U, resultado del primer ciclo; (e) Izquierda: resultado de la función dilatación en X ($F(x)$), Derecha: resultado de la función de copia en U ($G(u)$); (f) Resultado de la función resta en FoG, del segundo ciclo de procesamiento.

En la Fig.4.25 se muestra el temporizado de las señales de interfaz para la escritura de la memoria de parámetros para las funciones y vecindarios. Los 10 primeros ciclos se ingresan en serie los de **c**, luego los dos de **d**, y luego los 9 de *en_roi*. Para implementar la "erosión parcial", MIN3, se usa la búsqueda del tercer mínimo para filtra hasta dos negros por vecindario; para la "dilatación parcial", MAX3, se realiza la búsqueda del tercer máximo (Tab.4.15).

Tabla 4.15: Parámetros de las funciones utilizadas en el ejemplo para realizar la erosión y la dilatación.

Función	Parámetros (10x3 bits)
MIN3	0 0 0 0 0 0 0 1 1 1
MAX3 parcial	0 0 0 1 1 1 1 1 1 1

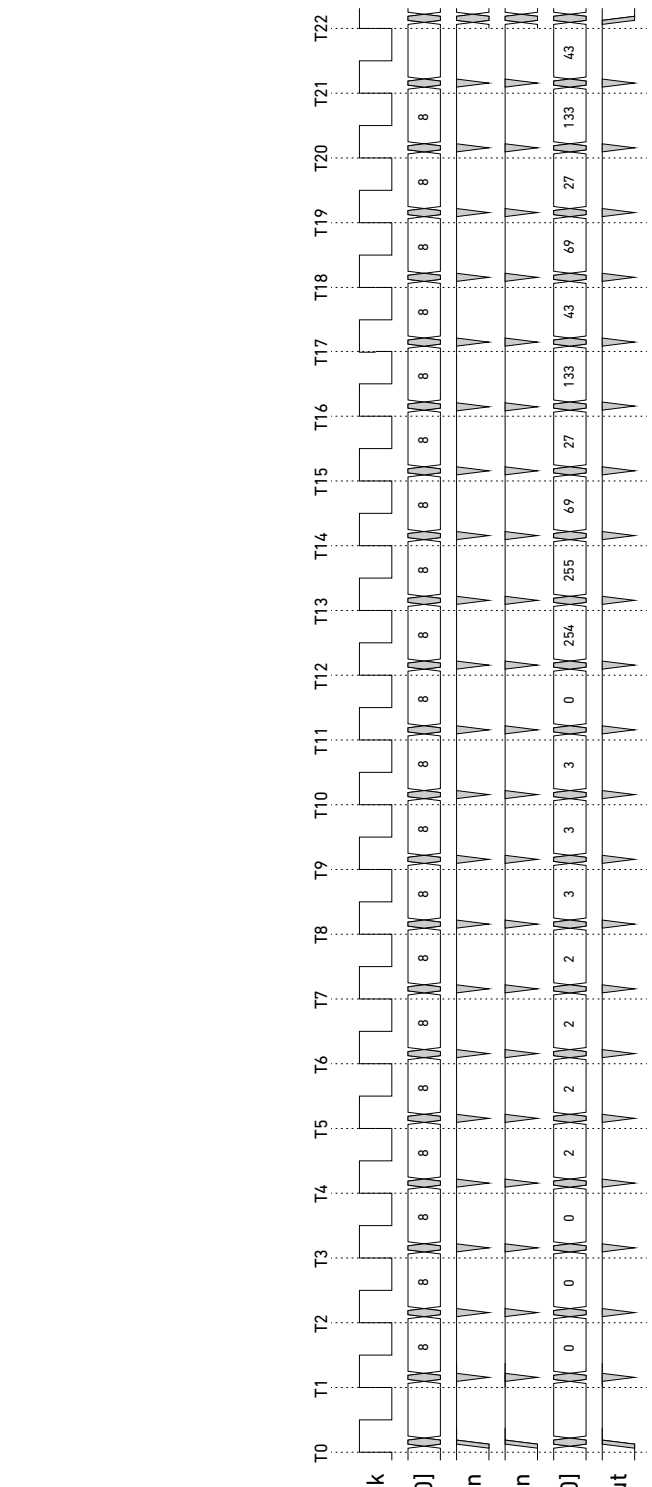


Figura 4.25: Temporizado de las señales del protocolo de comunicación del sistema para la escritura de la memoria de parámetros. Se carga la función MIN3 y MAX3 para **c**, CERO y COPIAR para **d**, la habilitación de todo el vecindario.

Una vez cargados los parámetros necesarios, resta escribir los registros y memoria de instrucciones de la unidad *confUnit*. Así como se realizó para el acceso de la memoria de imagen, se reinicia la FIFO y el controlador de procesamiento (primeros dos ciclos en Fig.4.26), se configura el inicio, paso y final de la rampa en 0, 1 y 255 respectivamente, y el valor de borde en 127. Como instrucción, se programa para todos los dos ciclos la habilitación de todos las celdas para el vecindario, y la selección de rango de los primeros 8 bits de la salida. Para el primer ciclo se elige la función MIN3 y CERO alojada en la ubicación 0, y la OR para la función compuesta. En el segundo ciclo MAX3 y COPIA alojada en la ubicación 1, y la diferencia ($|c_i - d_i|$) para la compuesta. En la Tab.4.16 se listan la configuración completa de los diferentes registros de procesamiento intervinientes. Por último, se da comienzo pulsando el registro de *start_reg* de los controladores de memoria.

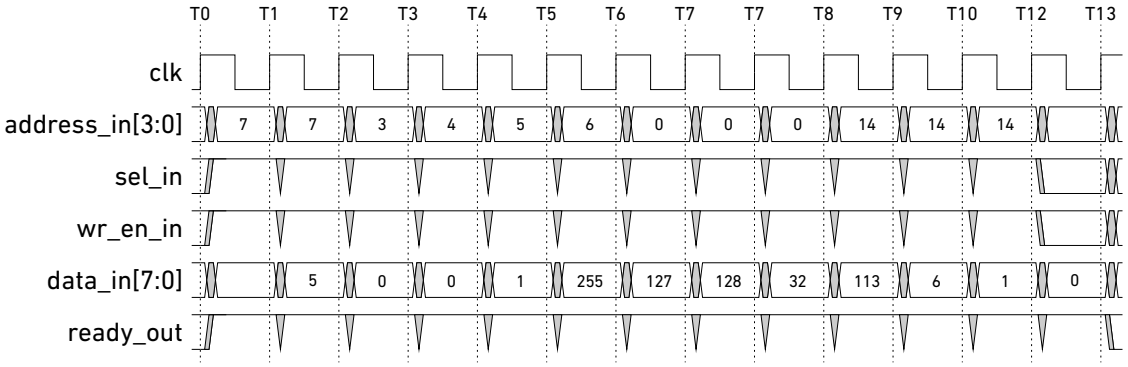


Figura 4.26: Temporizado de las señales IO para la escritura de los registros de configuración y la memoria FIFO de programa.

4.3.2. Implementación y resultados experimentales

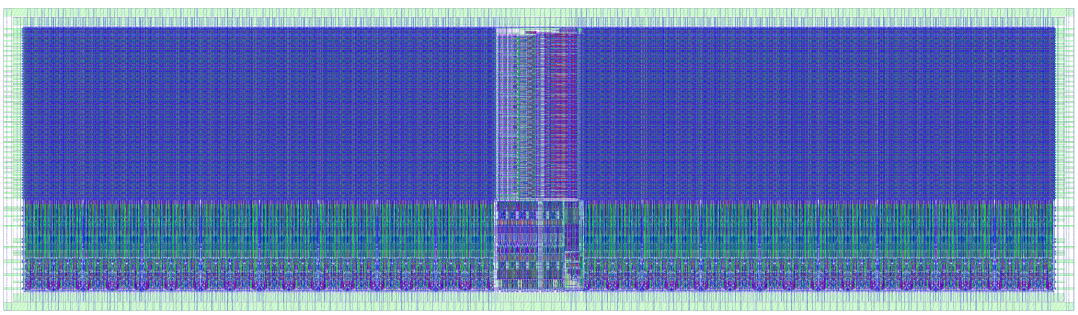


Figura 4.27: Máscara de la memoria RAM utilizada en arreglo para la cache de imágenes en MORPHO8SYM.

Tabla 4.16: Valores de configuración de los registros necesarios para el procesamiento ejemplo para MORPHO8SYM.

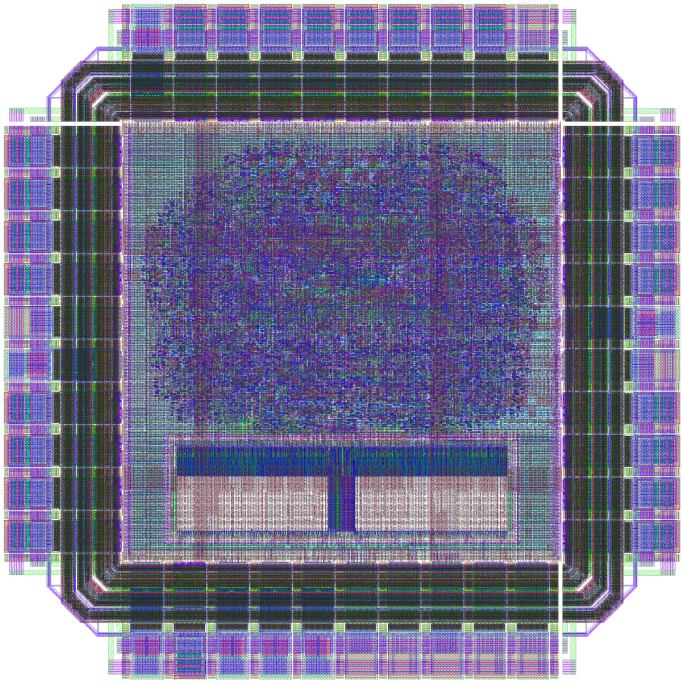
Registro	Valor
start_ramp_value	0
step_ramp_value	1
final_ramp_value	255
border_value	127
en_mask	0
sel_roi	0
sel_range	0
sel_FoG (1^{er})	2
sel_FoG (2^{do})	7
sel_f (1^{er})	0
sel_f (2^{do})	0

El diseño se realizo en lenguaje VHDL, y luego se sintetizo siguiendo el flujo descrito en ApendiceN. Su fabricación fue en la tecnología CMOS 130nm de GlobalFoundries, utilizando una librería de celdas estándar provistas por ARM©. En especial, la memoria de imágenes fue implementada utilizando un generador de memorias para la tecnología también de ARM©(Fig 4.27). El *core* del chip esta compuesto por 0,49 millones de transistores, ocupando $0{,}878mm \times 0{,}880mm$ mm^2 tal como se ilustra en Fig.4.28(a).

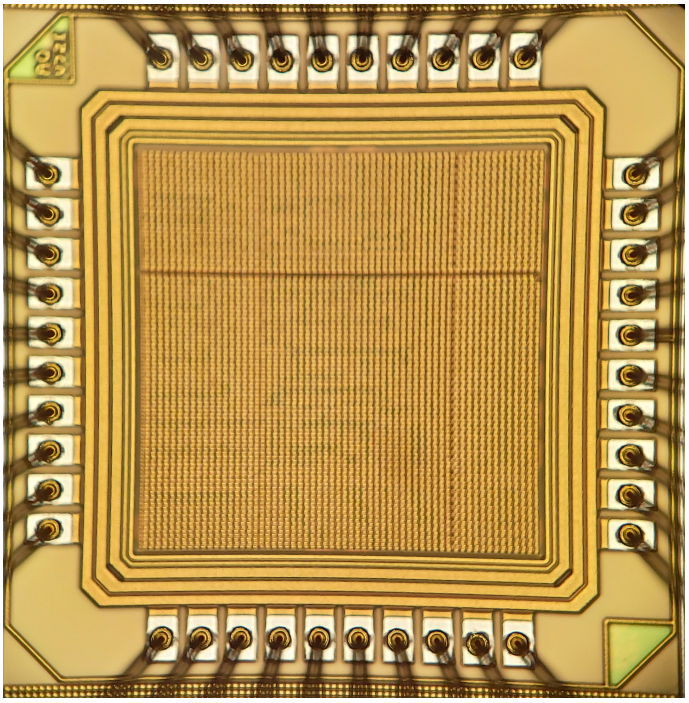
Tabla 4.17: Características del chip MORPHO8SYM en 130nm.

Tecnología	130nm de GlobalFoundries
Tamaño del arreglo	Vector de 64 procesadores
Área de chip	$1{,}4 \times 1{,}4$ mm^2
Área de core	$0{,}878mm \times 0{,}880mm$ mm^2
Cantidad de transistores	$0,49$M
Precisión de entrada	8 bit
Precisión de parámetro	3 bit
Memoria de imágenes	1
Algoritmo	Linear a tramos simétrica
Máxima dimensión de función	9
Velocidad de trabajo	100 MHz
Ciclos de reloj por procesamiento de imagen	$(256 + 8 + 8 + 2 + 2) \times 64 = 17644$
Operaciones por segundo	

Se utilizó un placa personalizada montada a una de desarrollo OpalKelly XEM6310,



(a) Mascara fabricación del en MORPHO8SYM.



(b) Foto del chip fabricado MORPHO8SYM.

con la cual, a través de la FPGA Spartan 6 que tiene integrada, se generan los vectores de excitación necesarios para testear la funcionalidad y el rendimiento del chip. Con una tensión de core de V_{DD} = 1,2V y una frecuencia de reloj igual a

$F_{core} = 100$MHz, el chip consume 1,04mW.

Para el cálculo de rendimiento y eficiencia se mantiene el criterio de definir una operación como una suma entre dos valores de 8 bits, utilizado en el Sistema MORPHO8PWL. Por celda se realizan 4 comparaciones de 8 bits por reloj, es decir $4 \times 256 = 1024$ operaciones por rampa. Para el calculo del argumento se suman 9 señales de 1 bit (4FA + 4HA=6FA) que se realizan 10 veces, lo que da $6 \div 7{,}5 \times 10 = 8$ operaciones. A esto se le debe agregar un total de $2,5 \div 7,5 \times 11 = 3{,}7$ operaciones que sucede en la ALU, al sumar 11 veces dos valores de 3 bits; y además las sumas que se da en el acumulador. De esta manera, el Sistema MORPHO8SYM tiene un rendimiento de $(1024 + 8 + 3{,}7 + 256) \times 64 \times 64 \times 100 \times 10^6 \div ((256 + 20) \times 64) = 29,952$ GOPs/s, alcanzando una eficiencia de $28,8$ TOPs/W.

4.4. Conclusiones

En este capítulo se ha introducido un nuevo algoritmo para la implementación de funciones lineales a tramos simétricas. Esto permite realizar operaciones morfológicas simples y complejas (dilatación, erosión, apertura, clausura, etc), funciones estadísticas de orden, y funciones lineales, utilizando menor cantidad de parámetros para igual número de variables. El algoritmo tradicional para cálculo de funciones lineales a tramos de n dimensiones requiere de 2^n parámetros, pero el nuevo algoritmo solo necesita $n+1$. Por ejemplo, para implementar un vecindario de 5×5 con el algoritmo original se requiere 33 554 432 de parámetros, mientras que con funciones simétricas solo se necesitan 26.

Utilizando este algoritmo, se desarrollaron nuevas estructuras SCNN basadas en las arquitecturas presentadas en el capítulo anterior, en donde la esfera de influencia se extiende a una región de 3×3 píxeles. Las celdas de procesamiento tienen un único motor de procesamiento, pero la capacidad de almacenar datos temporales junto a la disponibilidad de señales de habilitación de celdas de vecindario, permiten realizar funciones compuestas complejas .

El Sistema MORPHO1SYM, fabricado en la tecnología de 55nm de GlobalFoundries, procesa imágenes binarias de 48×48 píxeles con una eficiencia de $5,934$ TOPs/W, cuando se tiene en cuenta operaciones de 9 vecinos y 1 bit de precisión, superando ampliamente implementaciones anteriores con vecindarios más chicos.

En el caso del Sistema multibit MORPHO8SYM, se agregaron controladores de lectura y escritura más sofisticados que permiten ejecutar varios sistemas en serie, realizando un procesamiento de tipo *"pipeline"*. Este Sistema fue fabricado en 130nm de GlobalFoundries, y es capaz de procesar imágenes de 64×64 con una eficiencia de $24,615$ GOPs/W para una precisión de 8 bits, y $94,117$ GOPs/W para cómputos de 6 bits. La versión no simétrica MORPHO8PWL tiene un mejor rendimiento absoluto ($255,061$ GOPs/W). Esto se explica por diversos factores. En primer lugar, la nueva implementación solo implementa una función, versus dos (F y G) de la versión anterior, es decir, se realiza la mitad de operaciones, mientras que la energía consumida para la distribución de la rampa se mantiene constante. La diferencia en tecnologías también ayuda a que la brecha en consumo sea más grande. Mientras

que en 55nm se utilizó un conjunto de compuertas lógicas que pueden operar en 0.6V, en 130nm se utilizaron un kit de celdas estándar de ARM orientadas a mayor rendimiento en velocidad que son más rápidas pero no permiten bajar la tensión de alimentación. Sin embargo, no se debe olvidar que la celda nueva procesa un vecindario de 9 celdas, con capacidad de implementar múltiples funciones compuestas sin necesidad de realizar transferencias a memoria para el almacenamiento de datos temporales. Además, en el Sistema MORPHO8SYM se pudo implementar la técnica de habilitación de reloj por zonas, lo cual reduce la actividad de los árboles de reloj en circuitos que no se están utilizando mientras se realiza el procesamiento.

Para mejorar la eficiencia de las estructuras en futuras implementaciones, se debería realizar diseños personalizados de circuitos que poseen mayor actividad de movimiento de señales, tales como los comparadores pertenecientes en las celdas de procesamiento, los acumuladores, entre otros. Otras posibilidades de mejora son la utilización memorias para generar rampas no lineales, o la implementación de mayor cantidad de registros internos de almacenamiento temporal que puedan ser habilitados entre ciclos de rampa para realizar procesamiento espacial por fila o columna. Por otro lado, si se desea implementar las arquitecturas propuestas en sistemas con múltiples procesadores, se debe explorar la posibilidad de agregar de estructuras periféricas, como micro-procesadores dedicados o DMAs [78], que permitan expandir la abstracción del tipo de procesamiento de imágenes, y crear flujos de datos más complejos para, por ejemplo, poder extraer descriptores de alto nivel, o poder implementar redes neuronales más sofisticadas.

Capítulo 5

Integración de un System On Chip (SoC)

5.1. Introducción

El sistema Nano-Abacus SoC es un Sistema en chip (SoC), desarrollado en la Universidad de Johns Hopkins, con participación del grupo de Microelectrónica de la Universidad del Sur. El sistema fue diseñado para proveer una alta capacidad de cómputo de datos con alta eficiencia energética, utilizando múltiples procesadores no convencionales, orientado a sistemas móviles en aplicaciones de Internet de las cosas. El sistema se compone de chips multiprocesadores (CMP), una memoria principal 3D de acceso aleatorio de alto ancho de banda (DiRAM [79]) y una FPGA Zynq 7100. Cada CMP, la memoria y la FPGA constituyen circuitos integrados diferentes ("dies") que se integran sobre un *"interposer"* de Silicio especialmente diseñado. En este capítulo, inicialmente, se brinda detalles de la implementación del sistema y se describe la arquitectura de los CMP, resaltando su diseño modular, red de comunicación interna (NoC del ingles *Network On Chip*) y sus interfaces con el resto de los *dies*. Una vez introducido el sistema, se procede a describir el diseño de uno de los chips, denominado *Salamis Tablet* y tres procesadores principales, GF6MORPHO1, GF6MORPHO8 y GF6LINEAL, cuyas arquitecturas están basadas en la implementación con funciones lineales a tramos.

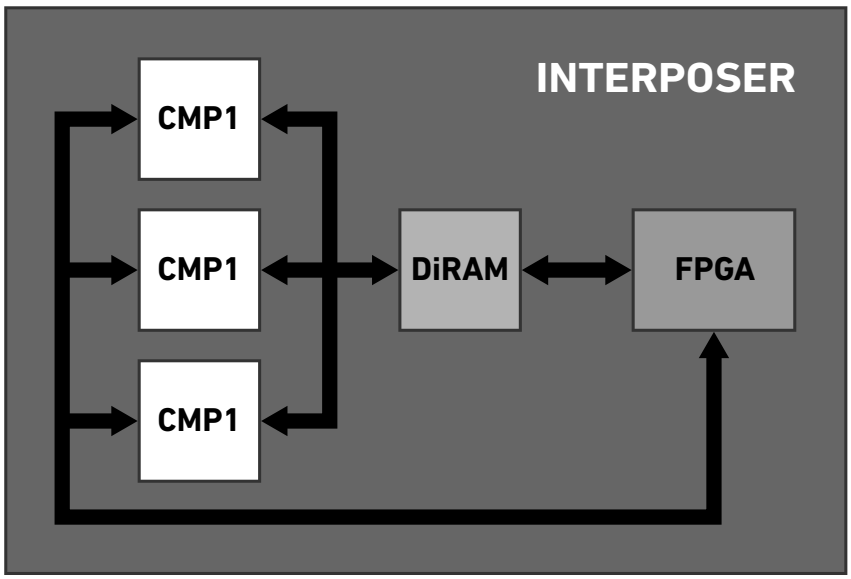


Figura 5.1: Visión de arriba del diseño 2.D Nano-Abacus SoC.

5.2. Arquitectura e implementación del sistema 2.D Nano-Abacus SoC

El 2.D Nano-Abacus SoC esta compuesto de 3 chips de multiprocesadores (CMP), cada uno fabricado en una área de $12,133\mu m$ por $17,166\mu m$ en la tecnología CMOS de $55nm$ de GlobalFoundries. Estos se interconectan a través de DiRAM y una FPGA, sobre el *"interposer"* de $50mm \times 64mm$ fabricado sobre un proceso de $1\mu m$. En la Fig.5.1 se ilustra el esquemático de los distintos componentes sobre el interposer.

Cada CMP está compuesto por un arreglo modular de una cierta cantidad de unidades de procesamiento (PU), especializadas en diferentes tipos de operaciones, una interfaz de memoria de alto ancho de banda para comunicarse con la memoria DiRAM, una interfaz de propósito general (GPIO) para comunicarse con la FPGA, y una NoC de dos niveles que permite comunicar los distintos PU con las distintas interfaces.

El primer nivel de la red (L1-NoC) es un *"token ring"* [80] que conecta las unidades de procesamiento con la DiRAM; el segundo nivel (L2-NoC) es una red mallada (o *"mesh network"*) que comunica los PU con la FPGA (Fig.5.2). Dado que la estructura interna del CMP es modular, todos los PUs tienen las mismas dimensiones y los mismos puertos de conexión con la NoC, por lo cual pueden ser intercambiados

o reemplazados durante el diseño e implementación de los CMPs. Además, todas las unidades de procesamiento tienen dominios de reloj y de alimentación separados para lograr un rendimiento óptimo de velocidad y eficiencia individualmente.

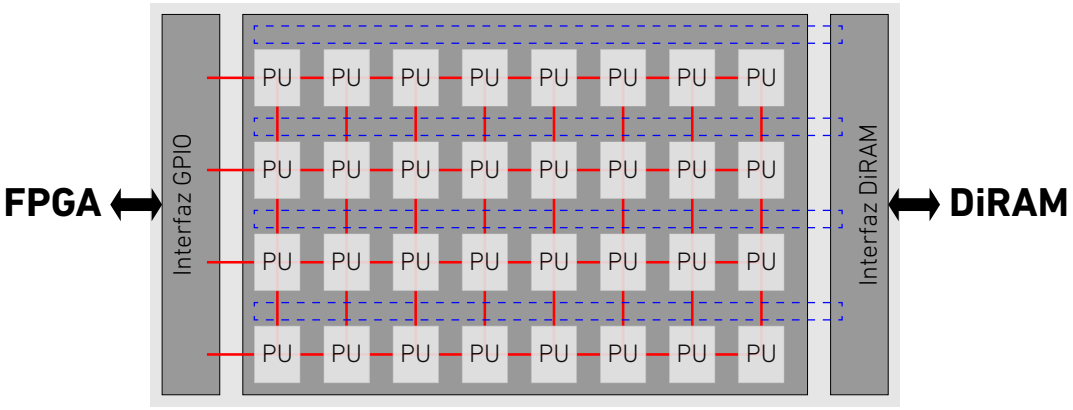


Figura 5.2: Diseño modular del CMP. Se muestra el arreglo de PU interocnectados por el "token ring" de la red L1 (en azul), y el "mesh" de la red L2 (en rojo).

La topología de la red de nivel L2 consiste en un red mallada de nodos organizados en una cuadrícula de dos dimensiones de M filas y N columnas, los cuales están compuestos por una unidad de ruteo de datos (RU) y la unidad de procesamiento PU. Cada nodo posee conexiones en las cuatro direcciones, norte, sur, este y oeste, las cuales utiliza para distribuir paquetes de datos hacia otros nodos y hacia puertos de entrada y salida, de forma sincrónica. La RU realiza un control de tráfico de paquetes de datos a través de una tabla de ruteo adaptiva que incorpora la capacidad de desviar un paquete para evitar el uso de *buffers* internos, y así reducir el tamaño de la estructura. La capacidad adaptiva de ruteo se complementa con un micro-sistema que detecta las conexiones "rotas" entre nodos. Un ejemplo de una red mallada de 2×2 nodos es ilustrada en la Fig.5.3. Dada la regularidad de la estructura, esta topología de red resulta muy atractiva para la implementación de sistema modulares escalables con unidades de procesamiento de diversa naturaleza no homogéneos.

Cada PU se conecta con la NoC a través de una unidad de control de interfaz o ICU (sigla del ingles *"Interface Controller Unit"*) embebida dentro de la unidad. El ICU es un procesador dedicado de 8 instrucciones (Tab.B.14) con un conjunto de registros internos especiales (Tab.B.15), que tiene por objetivo realizar transferen-

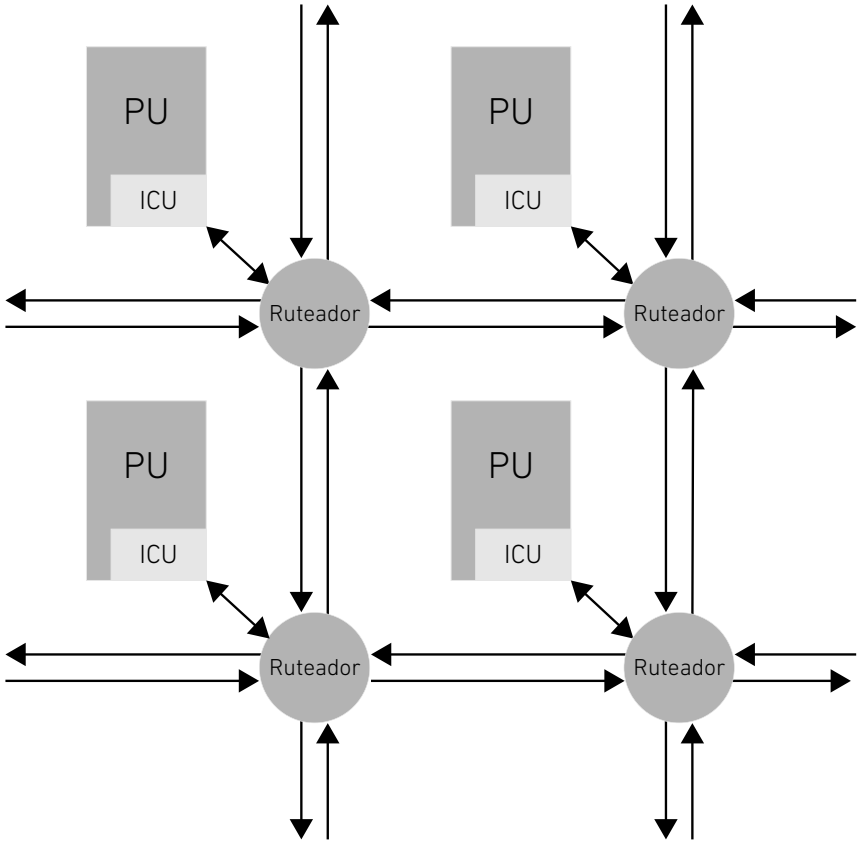


Figura 5.3: Una red de malla de 2×2 nodos.

cias de datos sofisticadas entre el PU y el resto de la red (L1 y L2). Esto permite que múltiples unidades de procesamiento trabajen en conjunto para realizar tareas complejas con gran cantidad de datos sin necesidad de usar la DiRAM como intermediario. Su programación se realiza a través de la red L2, por parte de una unidad de configuración (CU), la cual puede ser la FPGA externa o un CPU principal interno.

Del lado del PU, la comunicación se realiza mediante el protocolo AHB-Lite, detallado en el Apéndice A, en donde utilizan buses de datos de entrada y salida anchos, de 256 bits, para lograr altas tasas de transferencia de datos. Se cuenta además con dos señales especiales que pueden ser utilizadas para iniciar o detener un proceso interno del PU, o para comunicar, por ejemplo, que un cálculo ha finalizado. Del lado de NoC, se dispone de puertos para comunicación con L1 (Tab.B.16) y para comunicación con L2 (Tab.B.17).

Los datos transferidos entre la memoria principal y los PU a viajan a través de la red L1 sobre bus de datos de 256 bits de ancho, los cuales son escritos y leídos en la red utilizando un protocolo de comunicación asíncrona de 4 fases. Por otro lado,

las PU tienen la posibilidad de comunicarse entre ellas a través de la red L2, donde, al igual que en la red L1, se utiliza un bus de datos de 256 bits.

5.3. El chip multiprocesador *Salamis Tablet*

Dentro del marco de esta tesis se diseñó uno de los CMP que componen el sistema Nano-Abacus, denominado *Salamis Tablet* (Fig.5.4). El *Salamis Tablet* es un sistema heterogéneo de procesadores basados en la implementación de funciones lineales a tramos. En este CMP hay 2 PU basados en el MORPHO8SYM denominados GF6MORPHO8, 4 basados en MORPHO1SYM llamados GF6MORPHO1, y 50 unidades de procesamiento que realizan el producto escalar entre dos vectores de hasta 4096 componentes, denominados GF6LINEAL. Además se agregan un PU que contiene 8 KBytes de memoria local para guardar datos temporales, y una serie de unidades auxiliares tales como generadores de reloj, DSPs, y referencias bandgap, entre otros. Para el control del sistema, se implementan dos procesadores Cortex-M0 de ARM, cuya función es programar los distintos PU a través de la red L2.

A continuación se describen la arquitectura de los tres procesadores, resaltando las estructuras distintivas diseñadas especialmente para este CMP.

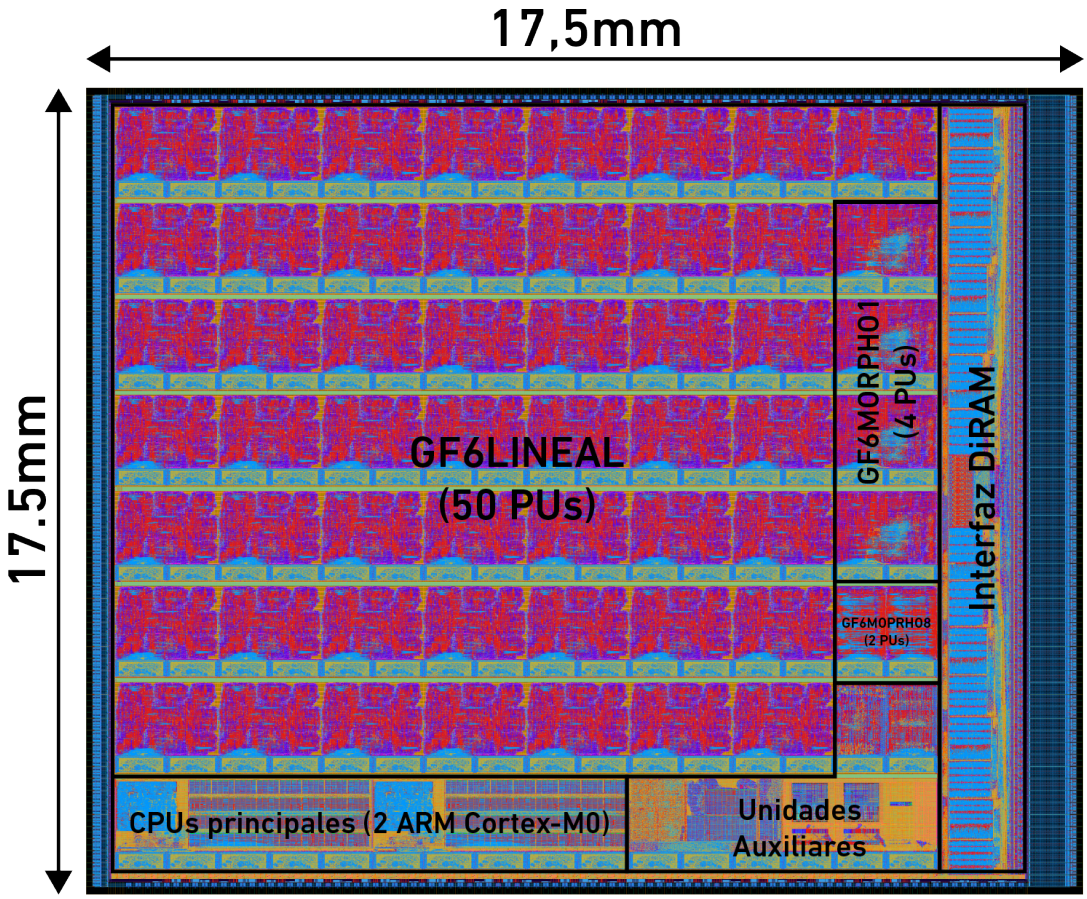


Figura 5.4: Máscara de fabricación del chip multiprocesador *Salamis Tablet*. Posee un total aproximado de 454 millones de transistores.

5.3.1. Sistema de procesamiento para imágenes binarias GF6MORPHO1

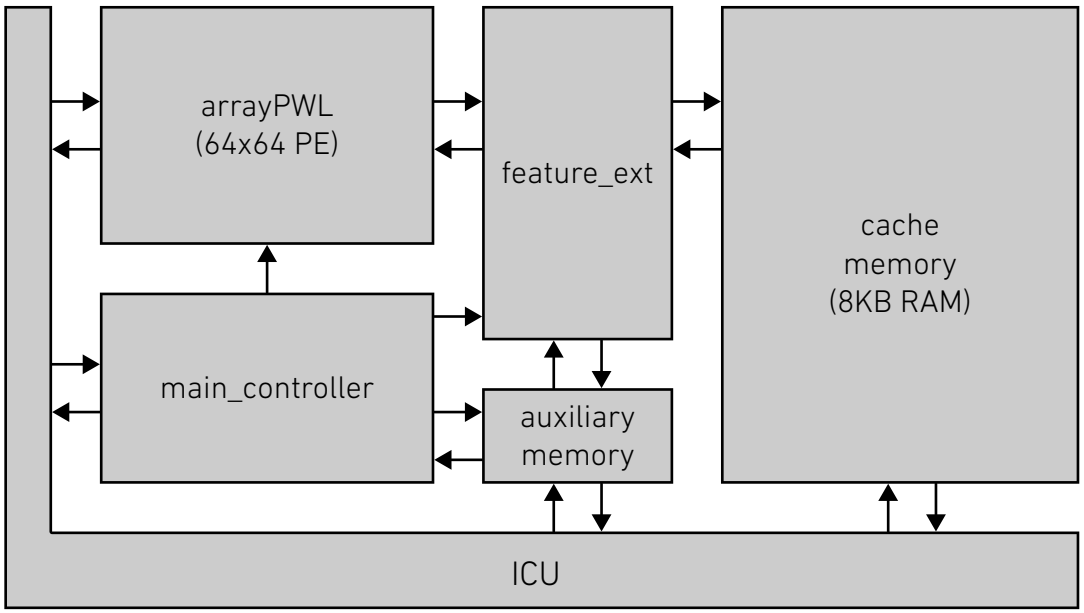


Figura 5.5: Esquemático general del sistema GF6MORPHO1.

En la Fig.5.5 se puede observar el diagrama en bloques de la unidad de procesamiento basado en el sistema MORPHO1SYM. La arquitectura se compone de un arreglo de elementos de procesamiento, una memoria RAM distribuida capaz de almacenar 16 imágenes de 1 bit de 64×64 píxeles y un procesador extractor de descriptores denominado *feature_ext*, el cual también sirve como DMA para la transferencia de datos entre el arreglo y la RAM. El PU es administrado por una unidad de control donde se implementa un CPU de propósito general con un conjunto reducido de instrucciones que opera con valores de 8 bits; una memoria auxiliar de 128 bytes permite el almacenamiento temporal de datos por parte del controlador, el módulo *feature_ext* y el ICU.

La estructura del elemento de procesamiento (PE) es similar al utilizado en el MORPHO1SYM, tal como ilustra Fig.5.6, pero en vez de enviar los parámetros **c** en 10 ciclos de reloj, los envía en un solo ciclo reduciendo el tiempo de procesamiento. Para ello, se implementa un multiplexor de 10 a 1 donde la entrada es el conjunto completo de parámetros, y la señal de selección es el argumento de la función $arg(f)$:

$$\arg(f) = b_0x_0 + b_1x_1 + \cdots + b_8x_8 \tag{5.1}$$

donde $b_i \in \{0,1\}, i = 0,...,8$ son los bits de enmascaramiento pertenecientes a la señal *en_roi* , $x_i \in \{0,1\}, i = 0,...,8$ los valores de las celdas pertenecientes al vecindario de influencia, y $0 \leq \arg(f) \leq 9$ es un valor de 4 bits.

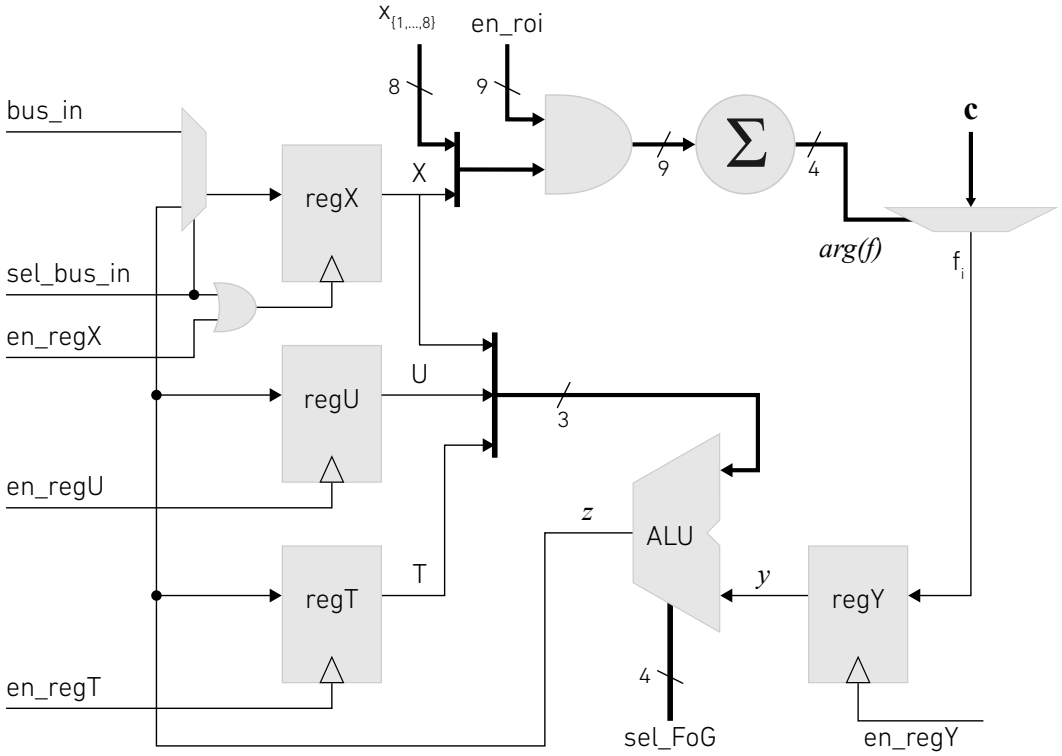


Figura 5.6: Arquitectura del elemento de procesamiento del GF6MORPHO1.

Los PE se agrupan en un arreglo bidimensional de 64×64 elementos interconectados localmente, denominado *arrayPWL* en Fig.5.5, capaz de almacenar hasta tres imágenes de 1 bit de precisión, dado que los elementos están compuestos por tres registros internos. Para su acceso, se implementa una interfaz multipuerto de 256 bits de ancho de datos, que arbitra la lectura y escritura de los registros internos *regX* por parte del módulo *feature_ext* y del ICU.

El control de la señales de procesamiento que se dirigen a los PE está a cargo de la unidad *main_controller*, en la cual se emplea una máquina de 4 estados de control del sistema (FSM), y un procesador de propósito general (μCPU) para la administración de los registros de procesamiento (Tab.B.5) y flujo de programa. El μCPU posee dos grupos de 16 registros internos de 8 bits (Tab.B.7), y es capaz de decodificar 16 instrucciones u *opcodes* (Tab.B.6) que modifican los registros de banderas *overflow*, *carry*, *zero* y *negativo*. El set de instrucciones se divide en ope-

raciones lógicas y aritméticas entre dos operandos, funciones de salto condicional, y de carga y lectura de la memoria auxiliar, entre otras. Además, se implementan dos conjuntos de registros para el almacenamiento de hasta 16 conjuntos de parámetros de funciones y otros 16 de configuración de vecindario.

Inicialmente, luego de una señal de reinicio general de sistema, la FSM comienza en el estado WAINTING_START, donde se espera un pulso de inicio desde el ICU, a través de la señal *start_in*, para reiniciar el procesador y los registros de procesamiento, y avanzar al segundo estado denominado CPU_WORKING. Aquí el μCPU lee y ejecuta instrucciones de 13 bits provenientes de una memoria local de programa *program_memory* de 256 palabras. Si el registro R01 es modificado, se habilita el procesamiento en *arrayPWL* además de configurar el vecindario y la función a implementar. Si se altera el MSB de R02, se seleccionan cuales que registros internos de los PE deben copiar el resultado alojado en los registros de salida *regY*. Si se quiere hacer uso del procesador extractor de descriptores, se debe escribir el registro interno R03. Además, el μCPU puede entrar en modo espera, donde la FSM pasa al estado cpu_stalled y retorna a la lectura de instrucciones cuando el ICU pulsa *start_in*. En el cuarto y último estado, denominado WAITING_ARRAY, el procesador aguarda que el módulo *feature_ext* termine la operación asignada a través de R03.

La unidad *feature_ext* fue diseñada con el fin de transferir datos internamente entre la memoria de imágenes y el arreglo, sin tener que acudir a la interfaz. Una máquina de estado local carga, en paralelo, el dato leído a un registro de desplazamiento de 256 bits, y, de acuerdo a lo que indique la palabra de configuración *feature_ext_setup*, realiza hasta tres operaciones sobre el mismo. La primera operación posible es contabilizar los valores iguales a 1 de todos los datos leídos y guardarlos en memoria. Además, se pueden almacenar sus respectivas coordenadas relativas a medida que se van acumulando; y por último, también se pueden escribir 1s en bits específicos de los datos, cuyas direcciones provienen de la memoria auxiliar. Si no se desea utilizar ninguna de las tres operaciones anteriores, el procesador se limita a funcionar como un DMA. La palabra de configuración, más tres registros de dirección fuente, destino, y la cantidad de datos, pueden ser configurados por el controlador interno o desde la NoC, a través de la memoria auxiliar.

En la Fig.5.7 se puede observar la máscara de fabricación con las distintas estruc-

turas principales resaltadas. La arquitectura fue íntegramente diseñada utilizando lenguaje HDL, e implementada en la tecnología CMOS 55nm LPX de GlobalFoundries. La unidad de procesamiento tiene el tamaño de dos slots, 1800μ de ancho, por 1281,6μm de alto con una cantidad total de 5,586 millones transistores.

Tabla 5.1: Especificación del diseño GF6MORPHO1

Tecnología	55nm de GlobalFoundries LPX
Área de core	2.30mm^2
Cantidad de PE	4096
Cantidad de transistores	5,586M
Precisión de entrada	1bits
Precisión de parámetro	1bits
Memoria de imágenes	19
Algoritmo	Lineal a tramos simétricas
Máxima dimensión de función	9
Frecuencia de reloj	300 MHz
Cantidad de ciclos de reloj por procesamiento	2
Operaciones por segundo	17,817TOPs/s

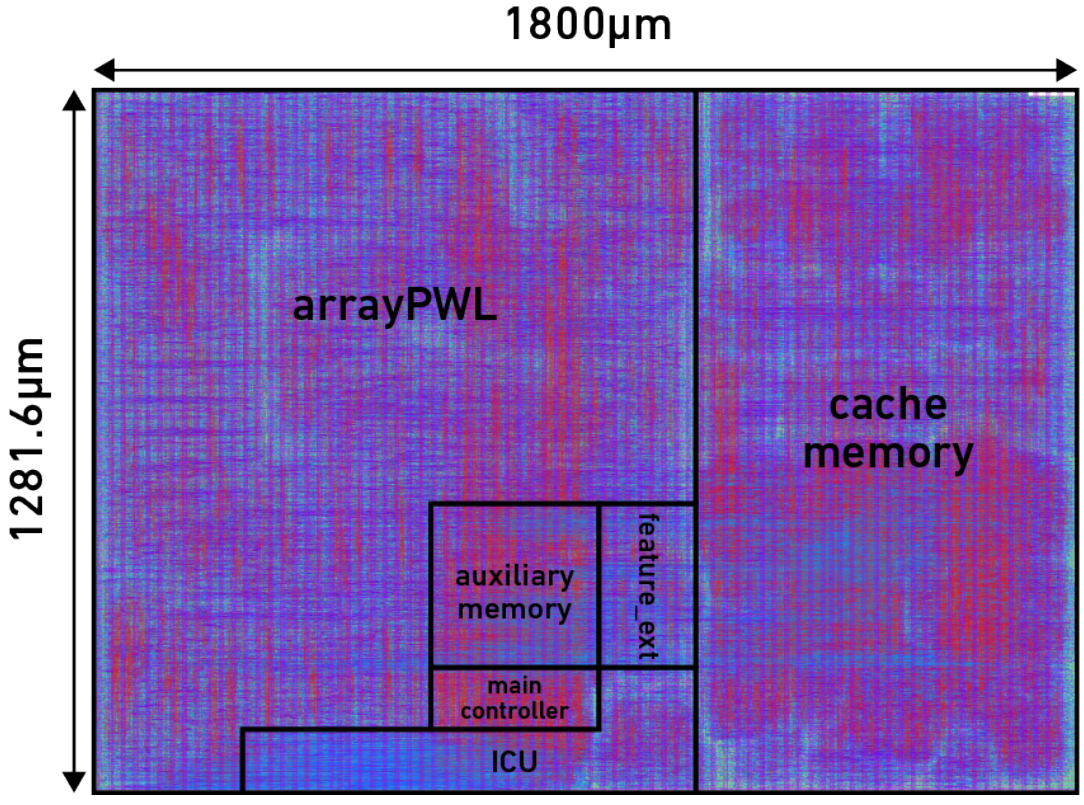


Figura 5.7: Arquitectura del elemento de procesamiento del GF6MORPHO1.

5.3.2. Sistema de procesamiento para imágenes en escala de grises GF6MORPHO8

Se diseñó un sistema para el procesamiento por columnas para datos de 8 bits basado en el sistema MORPHO8SYM, que implementa el algoritmo lineal a tramo para funciones simétricas $y = \Re^9 \rightarrow \Re^1$, con parámetros de 5 bits. La arquitectura, como se ilustra en Fig.5.8, se compone de dos partes: la primera es la interfaz de la NoC, el ICU, que comunica el núcleo o core del PU con el resto de la *network*; y la segunda es el core en si mismo, el cual contiene ocho procesadores *processorPWL*, los cuales pueden ser accedidos individualmente desde la NoC. Los procesadores están dispuestos lógicamente en serie, de manera tal que cada uno pueda leer los datos de salida del anterior (modo *pipeline*), para lo cual se requiere un protocolo de comunicación entre ellos. También pueden tomar datos de entrada escritos directamente desde la red.

Cada *processorPWL* está compuesto por un arreglo, *vectorPWL* en Fig.5.9, de 64 elementos de procesamiento SIMD, y dos vectores de registros de desplazamiento, *dinBuffer* y *doutBuffer*, de 64 bytes de largo, los cuales almacenan temporalmente los vectores de datos de entrada y salida, respectivamente. Para el control del vector de procesamiento se implementa la unidad *main_controller*, la cual se encarga de configurar los registros de funcionamiento del procesador, y de administrar el temporizado de las señales de comunicación entre los procesadores *processorPWL*. A continuación, se detallan los distintos módulos internos de la arquitectura.

El elemento de procesamiento, ilustrado en la Fig.5.10, utiliza la estructura de la celda implementada en MORPHO1SYM, con excepción del tamaño de los multiplexores de parámetros, que escalan de 3 a 5 bits, como también lo hace la ALU y el acumulador ACCM, que pasa de 12 a 14 bits. Se mantiene la opción de enmascaramiento del vector de estados y la posibilidad de almacenar local y temporalmente resultados parciales para realizar funciones compuestas. También se puede habilitar o deshabilitar el procesamiento en una celda de acuerdo al valor del MSB del registro central *regX˙C*.

Las celdas de procesamiento se instancian en un arreglo de 64 elementos, denominado *vectorPWL*. Cada PE se conecta localmente con los ubicados arriba y abajo a

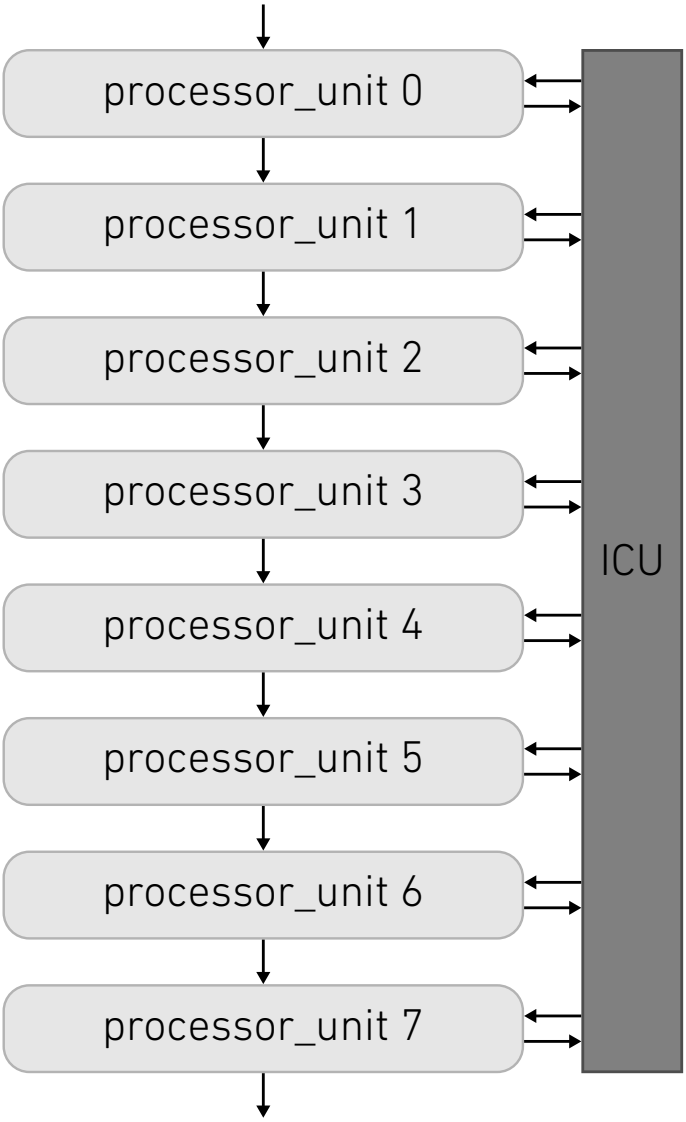


Figura 5.8: Esquemático general del sistema GF6MORPHO8.

través de las señales PMW de los X para la conformación de la vecindad de cómputo. Los datos de entrada provienen en conjunto desde el *dinBuffer*, y el *bus_salida* se conecta directamente a *doutBuffer*.

El control del vector de PEs está a cargo de la unidad *main_controller*, la cual esta compuesta por cuatro grupo de registros y una maquina de estado (FSM) con su correspondiente memoria de programa. El objetivo de la unidad es el de programar y controlar las celdas durante el procesamiento, y configurar el comportamiento de los *dinBuffer* y *doutBuffer*. El primer y segundo grupo de registros son configurados desde la NoC, y sirven para guardar ocho conjuntos de parámetros **c** y **d** de 5 bits por un lado, y 16 configuraciones de vecindad *en_roi* por el otro. El tercer grupo

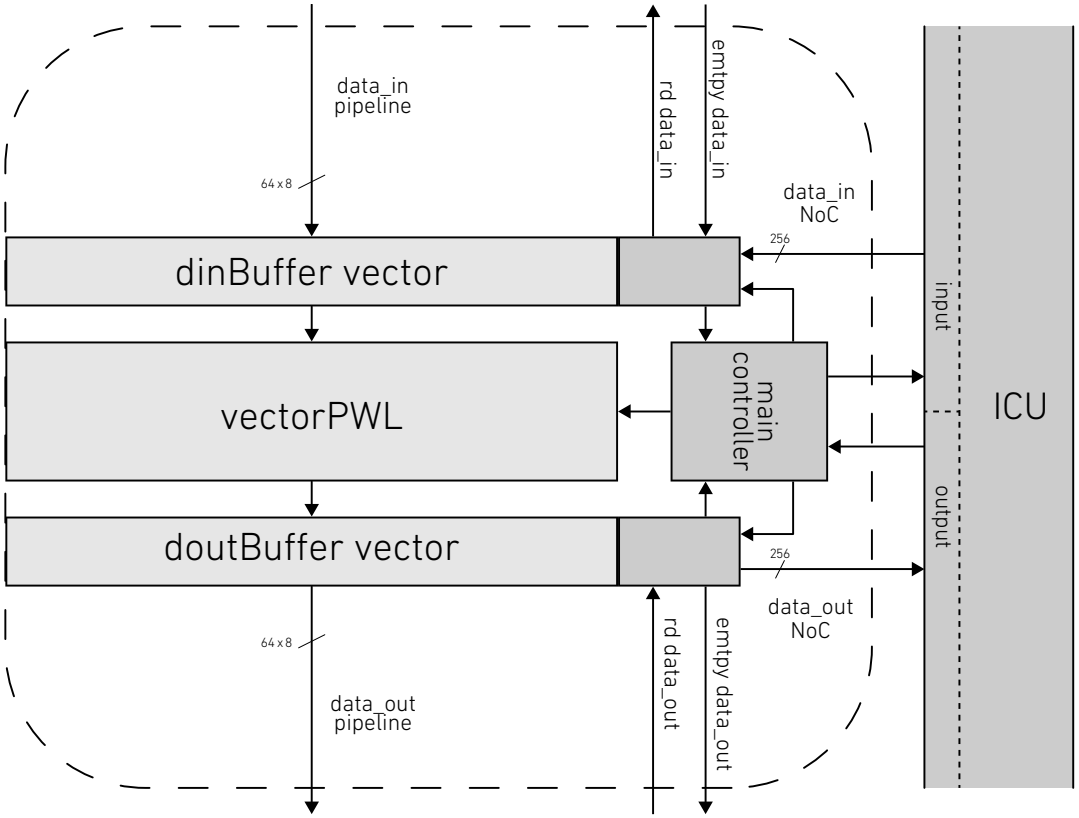


Figura 5.9: Arquitectura de una unidad de procesamiento *processor_unit* de GF6MORPHO8.

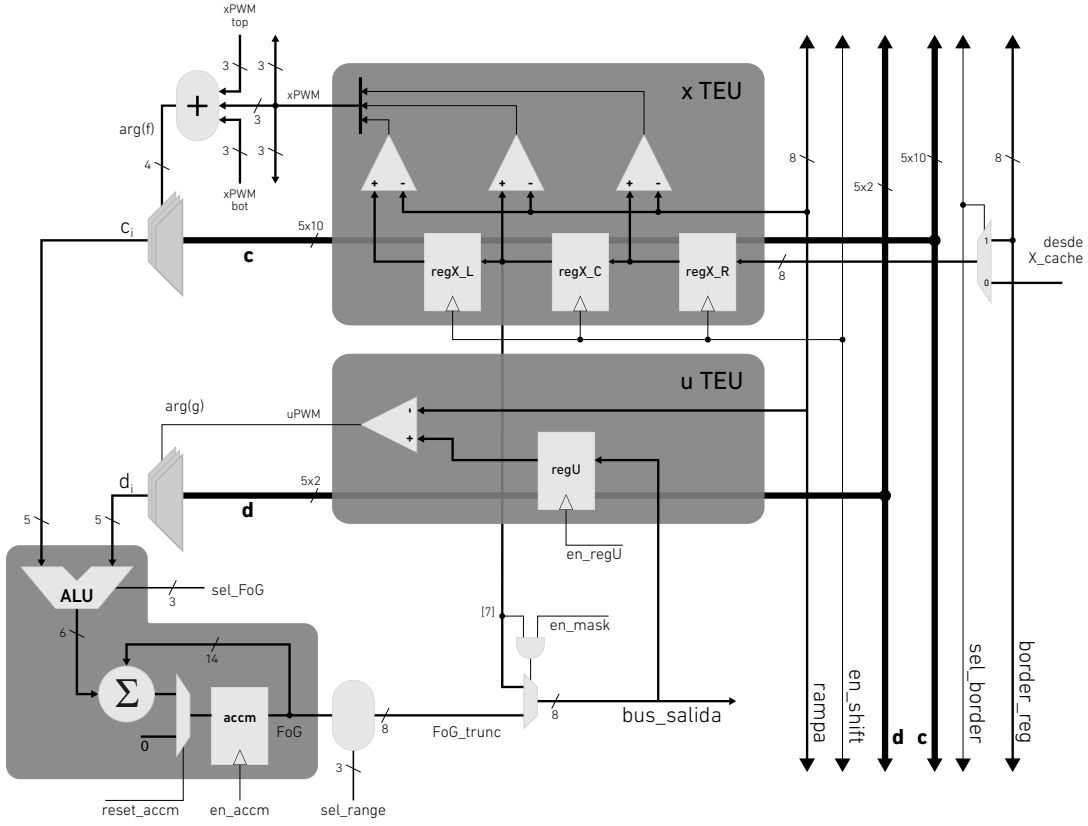


Figura 5.10: Estructura de ua celda de procesamiento de GF6MORPHO8.

de registros (Tab.B.9), también de configuración externa, establecen los valores de borde para la primer y última columna, la cantidad total de columnas que se van a procesar, el flujo de datos de *dinBuffer* y *doutBuffer*, el reinicio de la maquina de estado y el inicio del procesamiento para aceptar datos de entrada. El último grupo de registros, los de procesamiento, son configurados por la FSM a medida que está va decodificando las instrucciones leídas desde la memoria de programa (Tab.B.11). Estos configuran las funciones Fx y Gu a procesar, la función compuesta FoG de las ALUs de las celdas, el vector de enmascaramiento de la vecindad de su memoria, el valor de inicio, el paso y valor final de la rampa, y el truncado de la salida de los registros ACCM. En especial, el registro *end_reading_pm*, indica cuando la maquina estado debe dejar de leer la memoria, y avanzar en la lectura del nuevo vector de datos de entrada. A continuación se explican los 6 estados intervinientes en el procesamiento:

- IDLE_ST : es el primer estado luego del reinicio del sistema. La maquina de estados espera a que el registro *start_processing* sea para dar inicio a la lectura de vectores de entrada, y de eso modos, empezar a procesar. Entonces se un contador que registra la cantidad de vectores de entrada procesados, y se avanza al estado LOAD_FIRST_VECTOR_ST.

- LOAD_FIRST_VECTOR_ST : estado al que se ingresa una sola vez, en el cual se registran en los PEs el primer vector de datos de entrada. Dependiendo del valor del registro *padding_mode*, se utiliza el valor alojado en *padding_value* como primer dato, o se espera a que haya un nuevo dato disponible en *dinBuffer*. Entonces se incrementa el contador de datos si $padding\_mode = 0$, y la maquina de estados pasa a LOAD_MIDDLE_VECTOR_ST.

- LOAD_MIDDLE_VECTOR_ST : se espera que el registro de desplazamiento *dinBuffer* tenga un dato nuevo para ingresar en los registros centrales *regX_C* de los elementos de procesamiento, para luego avanzar al estado WAITING_INPUT_VECTOR_ST.

- WAITING_INPUT_VECTOR_ST : este es el estado principal para la lectura de datos de entrada y el envío de datos de salida. Cuando el *doutBuffer* se encuen-

tra listo para aceptar el nuevo vector de resultados proveniente del *vectorPWL*, y el *dinBuffer* posee un nuevo vector de entrada, se incrementa el contador de vectores, y se reinicia el puntero de dirección de la memoria de programa. Por defecto se avanza al estado READING_PROGRAM_MEMORY_ST, sin embargo, si el contador de vectores de entrada alcanzó el valor dado por *total_num_proc*, se evalúa nuevamente *padding_mode*, y si este último es cero, se retorna al estado IDLE_ST. En el caso de que *padding_mode* sea uno, se activa una señal bandera interna indicado que el próximo será el último ciclo de programa. Si dicha señal bandera ya se encontraba activada, significa que el último ciclo de programa ya se ha realizado, y se debe retornar al estado IDLE_ST una vez que *doutBuffer* haya registrado el último vector de resultados.

- READING_PROGRAM_MEMORY_ST : se lee la memoria de programa y se incrementa su dirección. Dependiendo de la instrucción leida, se actualizan los diferentes registros de procesamiento, o se reinicia el contador de *rampa* que va hacia los con PEs (Tab.B.10). De haberse leído la instrucción para comenzar a procesar, se pasa al último estado PROCESSING_ST. Si se leyó la última instrucción de programa, se retorna a WAITING_INPUT_VECTOR_ST.

- PROCESSING_ST : se habilitan los comparadores y el acumuluador ACCM de cada elemento de procesamiento. Se incrementa *rampa* que se distribuye a lo largo de todo el *vectorPWL* monotonamente a un paso dado por el registro *step_ramp_value*, hasta que llega al valor del registro *end_ramp_value*. Cuando esto sucedem se da por finalizado el ciclo de procesamiento, habiltando los registros *regU* para que copien los resultados internos, y dependiendo del valor del registro *end_reading_pm* se retorna a la lectura de instrucciones (READING_PROGRAM_MEMORY_ST), o se avanza a WAITING_INPUT_VECTOR_ST.

Por su parte, los módulos *dinBuffer* y *doutBuffer* también poseen simples máquinas de estados locales que administran el temporizado de las señales de comunicación con el controlador principal y con las demás unidades externas al propio *processor_unit* que quieran escribir o leer datos de la misma.

En la Fig5.11 se puede observar la máscara de fabricación de la implementación final, resaltando la ubicación de los diferentes módulos. El diseño ocupa un solo slot, con un área total de 1,15mm^2. En particular, la arquitectura fue preparada para poder ser instancia multiples veces de forma contigua, y así crear un pipeline más largo de procesadores vectores de procesamiento *procesor_unit*. Los detalles de la implementación son listados en la Tab.5.1.

Tabla 5.2: Especificaciones de implementación del diseño GF6MORPHO8

Tecnología	55nm de GlobalFoundries LPX
Área de core	1.15mm^2
Cantidad de PE	512
Cantidad de transistores	3,041M
Precisión de entrada	8bits
Precisión de parámetros	5bits
Memoria de imágenes	19
Algoritmo	Lineal a tramos simétricas
Máxima dimensión de función	9
Frecuencia de reloj	300MHz
Cantidad de ciclos de reloj por procesamiento	257
Operaciones por segundo	$825,304$ GOPs/s

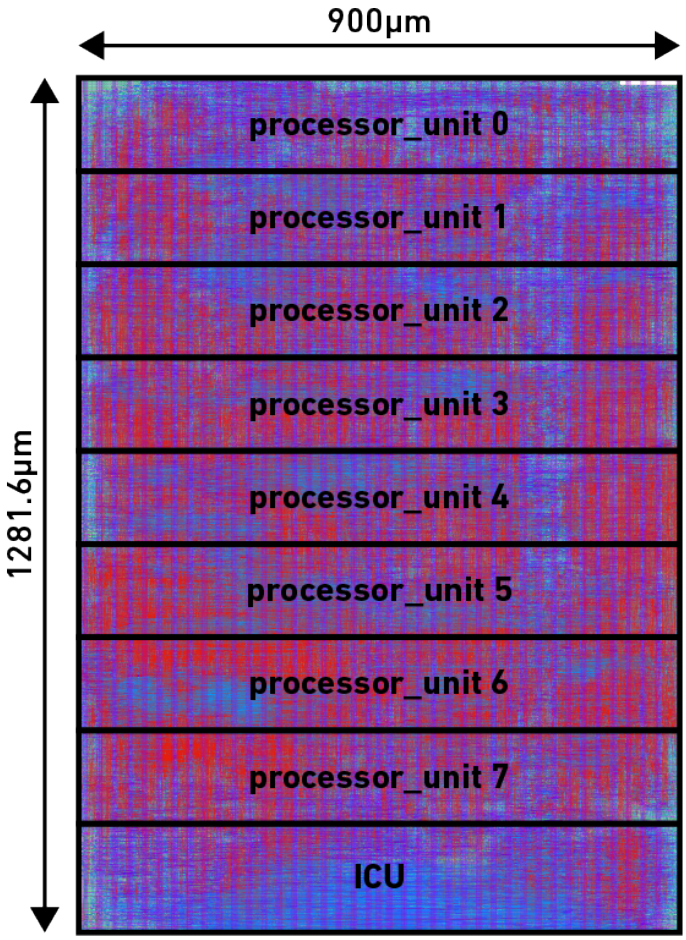


Figura 5.11: Mascara de fabricación del diseño GF6MORPHO8.

5.3.3. Procesador de multiplicador de Vector-Vector GF6LINEAL

La tercer arquitectura diseñada tiene como objetivo calcular el producto escalar entre dos vectores utilizando una modificación del algoritmo de cálculo de funciones lineales a tramos simplicial. El sistema permite realizar convoluciones discretas y circulares entre valores reales donde la cantidad entradas pueda ser variable, con la opción de poder correlacionar una entrada con un patrón específico. A continuación se explica las distintas modificaciones que se le hace al algoritmo original, se propone una estructura digital para su realización, y luego se detalla el diseño implementado utilizando como base la estructura propuesta.

Modificaciones del algoritmo PWL para funciones lineales

Para el cálculo de funciones lineales implementando el algoritmo PWL simplicial, los procedimientos explicados en 2.3.2 se mantienen: identificación del símplice y sus vértices, búsqueda de los correspondientes coeficientes, y por último la interpolación de los coeficientes a través de la combinación convexa dada por la Ec.2.3.2. Si se restringe el universo de funciones PWL al de funciones multivariables afines, se puede observar que para un hipercubo genérico, si bien el número de vértices se mantiene en 2^n, la cantidad de parámetros a guardar en memoria es igual a $n + 1$, como se explica a continuación.

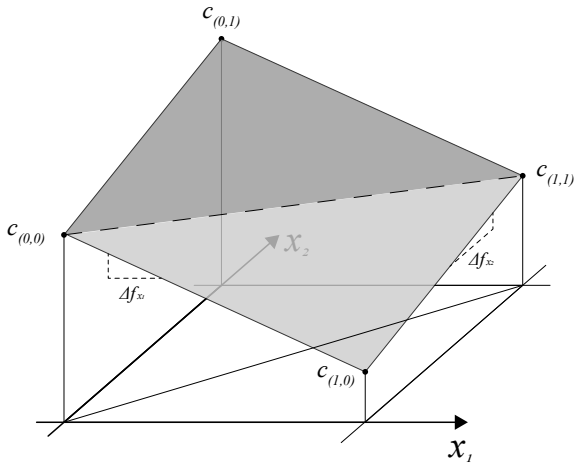


Figura 5.12: Ejemplo de función lineal en una partición simplicial en $\Re^2$.

Dada una partición simplicial unitaria de dominio perteneciente a $\Re^2$ (Fig.5.12), la cantidad de coeficientes necesarios para definir cualquier función PWL es $q =$

$2^n = 4$, denominados $c_{(0,0)}$, $c_{(1,0)}$, $c_{(0,1)}$, y $c_{(1,1)}$ según las coordenadas de sus vértices asociados. Si la función a implementar es un polinomio f de primer grado el valor de sus derivadas parciales en el dominio son constantes:

$$\Delta f_{x_i} = \frac{\partial f}{\partial x_i} \Delta x_i \tag{5.2}$$

Por lo tanto los parámetros $c_{(i_1,i_2)}$ pueden ser expresados a partir de un valor constante y de las derivadas parciales. Si bien cualquier coeficiente puede ser utilizado como valor constante de referencia, es conveniente utilizar el valor perteneciente al vértice $(1, 1)$, y calcular el resto a partir de este:

$$\left.\begin{array}{l} c_{(1,1)} = c_{(0,1)} + \Delta f_{x_1} \\ c_{(1,1)} = c_{(1,0)} + \Delta f_{x_2} \end{array}\right\} \Rightarrow \begin{array}{l} c_{(0,1)} = c_{(1,1)} - \Delta f_{x_1} \\ c_{(1,0)} = c_{(1,1)} - \Delta f_{x_2} \end{array} \tag{5.3}$$

Para hallar el coeficiente $(0, 0)$ se procede de la siguiente manera:

$$c_{(0,1)} = c_{(0,0)} + \Delta f_{x_2} \Rightarrow c_{(0,0)} = c_{(0,1)} - \Delta f_{x_2} \tag{5.4}$$

Si se reemplaza la expresión de $c_{(0,1)}$ hallada en (5.3), dentro de (5.4), se obtiene la expresión deseada :

$$c_{(0,0)} = c_{(1,1)} - \Delta f_{x_1} - \Delta f_{x_2} \tag{5.5}$$

En este ejemplo de dos dimensiones, solo es necesario guardar $c_{(1,1)}$, Δf_{x_1} y Δf_{x_2}. Extendiendo el concepto a un dominio en $\Re^n$, cualquier coeficiente puede ser calculado como :

$$c_{(i_1,i_2,\ldots,i_n)} = c_q + \sum_{j=1}^{n} (i_j - 1) \Delta f_{x_j} \tag{5.6}$$

donde i_j son los índices de las coordenadas de los vértices asociados a los coeficientes c, y c_q es el coeficiente asociado al vértice cuyas coordenadas $i_j = 1$, para todo j desde 1 hasta n; necesitando solo $n + 1$ valores para el cálculo de f.

Implementación digital En la Fig.5.13 se observa el diagrama en bloque de una opción de implementación para funciones lineales. La primer etapa de la estructura

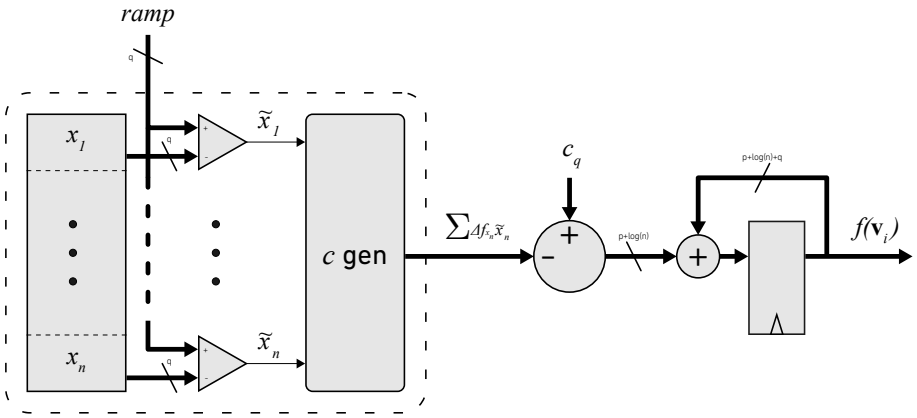


Figura 5.13: Esquemático de implementación digital.

propuesta es un arreglo de comparadores digitales, los cuales comparan las entradas x_i de q bits, con una rampa común monótonamente creciente. Si la rampa es menor al valor de entrada, el resultado $\tilde{x}_i$ es uno, y si es mayor o igual, es cero. Al inicio del procesamiento, la rampa inicia en $ramp_{ini}$, y por cada ciclo de reloj, se incrementa su valor en pasos $ramp_{step}$. El resultado de la comparación genera una palabra $\tilde{x} = \{\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_i, ..., \tilde{x}_n\}$ de n bits que identifica el vértice a computar a lo largo del tiempo. En el primer ciclo de reloj, el vector inicia en el valor $\tilde{x} = \{1, 1, ..., 1, ..., 1\}$, y durante el transcurso de la rampa, sus componentes eventualmente se igualan a cero. En el momento de finalización de la rampa, se obtiene $\tilde{x} = \{0, 0, ..., 0, ..., 0\}$.

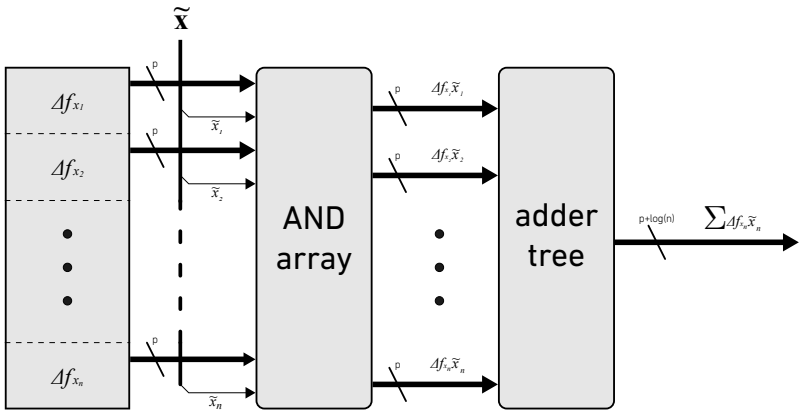


Figura 5.14: Módulo *c_gen* de la Fig.5.13 , donde se genera el segundo término del coeficiente de Ec(5.6).

La segunda etapa, "generador c", a la cual le ingresa el vector $\tilde{x}$, está compuesta por una memoria que almacena las derivadas parciales Δf_{x_i} de p bits, asociadas a las entradas x_i; un arreglo de compuertas AND, y un árbol de sumadores de n entradas

y una salida de $p + \log n$ bits de resolución (Fig.5.14). Cada señal $\tilde{x}_i$ habilita el valor de la derivada parcial correspondiente a la dimensión i-ésima, contribuyendo a la formación del segundo término de la Ec.(5.6) $\sum_{j=1}^{n} (i_j - 1)\Delta f_{x_j}$. Dicha salida ingresa a la tercer etapa, donde se suma con el valor c_q, de $p+\log n$ bits, para formar el coeficiente $c_{(i_1,i_2,...,i_n)}$ (Ec.(5.6))asociado al vértice $\tilde{x} = \{\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_n\}$. Finalmente, en la cuarta etapa de procesamiento, el registro ACCM, de $p+\log n+q$ bits, acumula los sucesivos coeficientes $c_{\tilde{x}}$, para generar como resultado el nuevo valor de la función.

El tiempo de procesamiento total N_{proc}, por tratarse de un sistema sincrónico, es igual a la cantidad de ciclos de reloj de un ciclo completo de la rampa de q bits:

$$N_{proc} = \text{floor}\left(\frac{ramp_{end} - ramp_{ini}}{ramp_{step}}\right) \tag{5.7}$$

donde $ramp_{ini}$ y $ramp_{end}$ es el valor de inicial y final de la rampa respectivamente, $ramp_{step}$ es el valor de incremento por ciclo de reloj, y la función $floor$ redondea su argumento al entero más cercano menor o igual al mismo.

Arquitectura de gf6linear

El PU denominado GF6LINEAR (Fig.5.15) está compuesto por un arreglo de 8 procesadores *coreLinear*, ocho unidades de detector de máximos y mínimos, y un controlador principal, *main_controller* para la configuración del sistema y administración de las señales de procesamiento. Cada *coreLinear* se basa en esquema propuesto en la Fig.5.13. La estructura está compuesta por una memoria distribuida capaz de almacenar 512 entradas de 8 bits, otra memoria para guardar los respectivos 512 parámetros signados de 6 bits cada uno, un árbol de sumadores, y una unidad de acumulación, donde el coeficiente c_q, proveniente de un registro de configuración del *main_controller*, se suma con el resultado del árbol de sumadores y se acumula en un registro local ACCM.

La memoria de entradas tiene la capacidad de desplazar sus valores en una dirección, alimentando el primer registro con el último valor de la memoria, o con un valor externo a la estructura proveniente de otro *coreLinear*, o de una memoria de tipo FIFO perteneciente al controlador principal. Por su parte, la salida del árbol de sumadores también puede combinarse con las respectivas salidas de otros procesa-

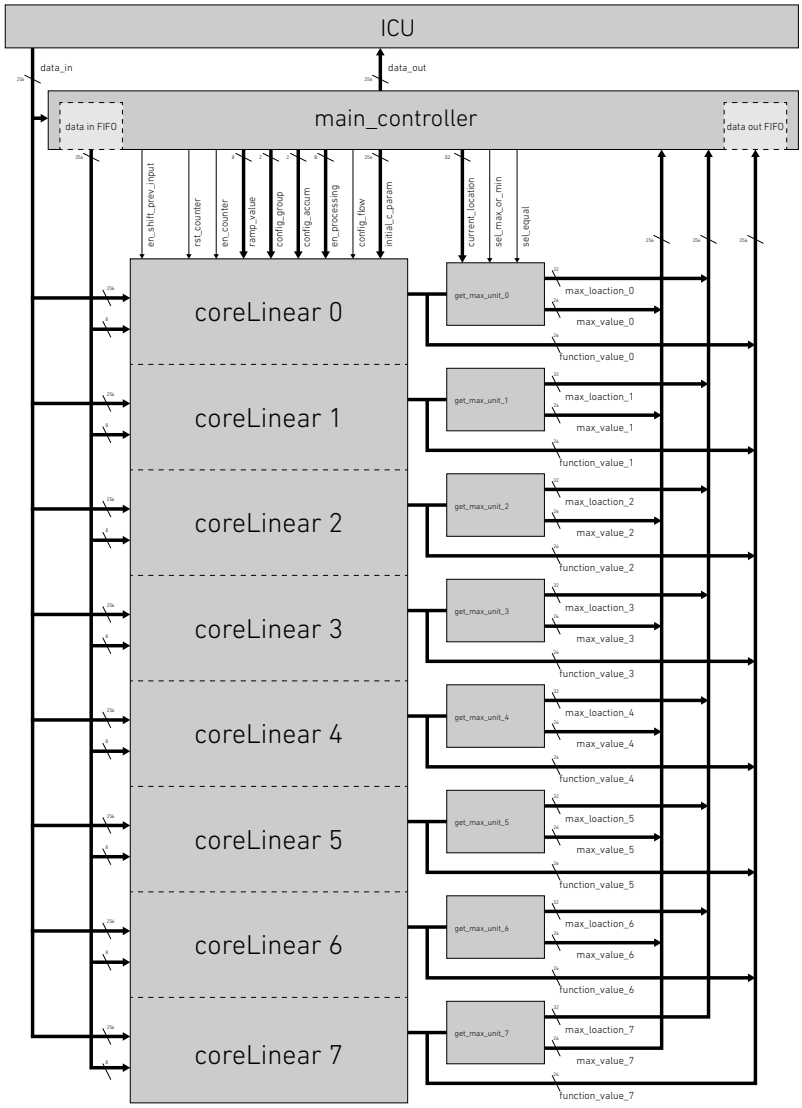


Figura 5.15: Arquitectura simplificada de la unidad de procesamiento gf6linear.

dores. Estas dos características permite que puedan agrupar varios *coreLinear* para poder procesar hasta una función de 4096 variables de entrada.

La salida de cada procesador ingresa a una unidad *get_max_unit* que se encarga de guardar el valor máximo o mínimo a lo largo de sucesivos ciclos de procesamiento (*max_value* en Fig.5.15), como también el número de ciclo de procesamiento al que pertenece dicho resultado (*max_location* en Fig.5.15). Por otro lado, las salidas se almacenan en una FIFO dentro del controlador, la cual tiene la capacidad de almacenar temporalmente hasta 64 resultados provenientes de los 8 procesadores, hasta que el ICU lea de la misma.

La configuración de las distintas unidades y la administración del temporizado

de las señales de procesamiento, se encuentra a cargo del módulo *main_controller* que está compuesto por un procesador personalizado que contiene un grupo 32 de registros internos de 8 bits, y un grupo reducido de 6 instrucciones especiales de 16 bits (Tab.B.12) diseñadas específicamente para la arquitectura. El procesador lee las instrucciones de una memoria de programa de 16 bits de ancho, con capacidad para almacenar hasta 128 palabras, la cual se carga desde el ICU. Las instrucciones fueron diseñadas para poder cargar los registros internos con valores inmediatos y así poder configurar las unidades y parámetros de procesamiento (Tab.B.13), dar inicio a un ciclo de rampa, leer la FIFO de datos de entrada para ingresarlas en la memoria de x, realizar un desplazamiento en los registros de las variables entrada, y poder saltar a otros lugares de la memoria de programa para realizar ciclos *for*. Por último, un registro de un bit denominado *start_proc* se encuentra mapeado en memoria, y debe ser pulsado desde el ICU para permitir que el procesador de control empiece a leer la memoria de programa.

En la Fig5.11 se puede observar la máscara de fabricación de la implementación final, resaltando la ubicación de los diferentes módulos. El diseño ocupa dos slots, con un área total de $2,30\text{mm}^2$ y $5,187$ millones de transistores. La arquitectura fue íntegramente diseñada utilizando lenguaje HDL, e implementada en la tecnología CMOS 55nm LPX de GlobalFoundries. La Tab.5.3 brinda los detalles de la implementación.

Tabla 5.3: Especificaciones de implementación del diseño GF6LINEAL

Tecnología	55nm de GlobalFoundries LPX
Área de core	2.30mm^2
Cantidad de transistores	5,187M
Cantidad de entradas	4096
Precisión de entrada	8bits
Precisión de parámetros	6bits signado
Algoritmo	PWL lineal
Frecuencia de reloj	300 MHz
Ciclos de reloj por procesamiento	256
Operaciones por segundo	$4,8$ GOPs/s

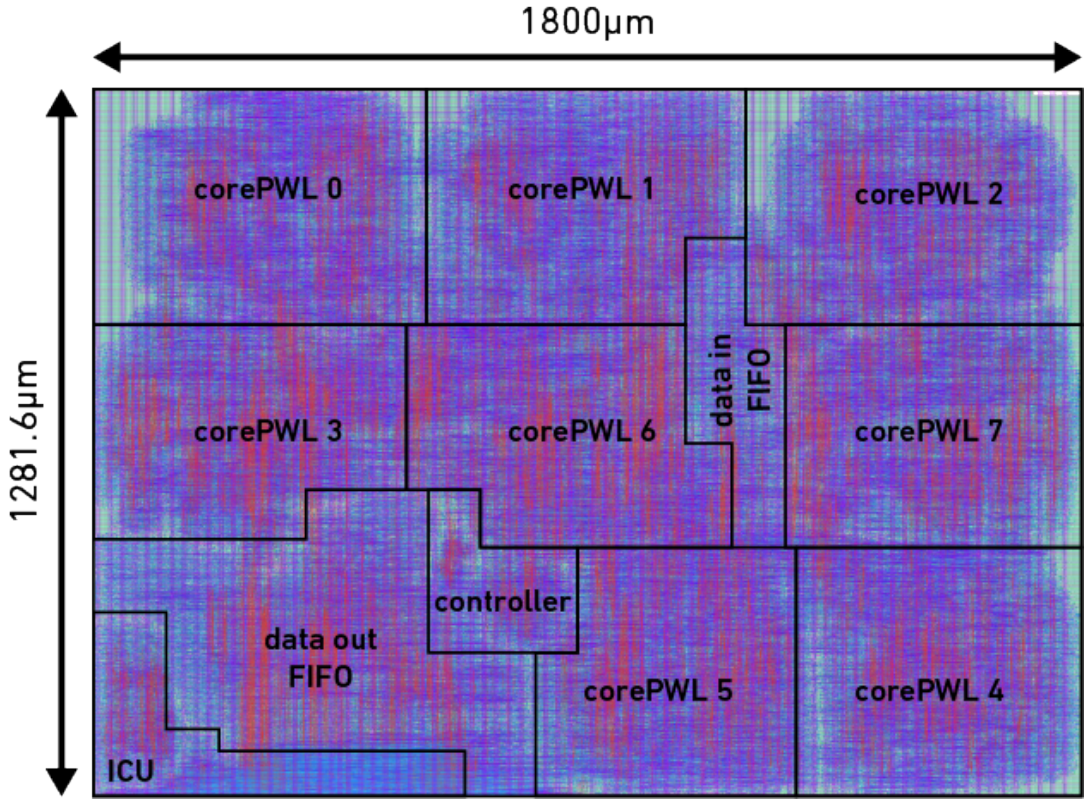


Figura 5.16: Mascara de fabricación del diseño GF6LINEAL.

5.4. Conclusiones

Como finalización de esta tesis, en este capítulo se presentó el diseño de procesadores digitales personalizados utilizando funciones lineales a tramos, en el marco del diseño de un sistema multi-chip 2.5D de gran rendimiento de cómputo y energéticamente eficiente. El Sistema 2.5D Nano-Abacus System-On-Chip, esta consituído por tres chips multi-procesadores (CMP) conectados a una memoria 3D DiRAM de TezzaronSemiconductor y a una FPGA Zynq 7100 de Xilinx, integrados verticalmente a través de una placa *"interposer"*.

Cada CMP está compuesto por un arreglo bidimensional de procesadores modulares, los cuales se comunican por medio de una red de comunicación interna, mejor conocida como Network-On-Chip o NoC, de dos niveles de jerarquía. El primer nivel, identificado como L1, es un "token-ring" que interconecta por fila los procesadores y la interfaz que comunica el chip con la memoria de DiRAM. El segundo nivel o L2, es una red mallada o *"mesh"* que interconecta los PU a través del arreglo y la interfaz de la FPGA, utilizando unidades de ruteo de paquetes inteligente.

Para poder realizar un procesamiento no-lineal eficiente, se desarrollaron dos sistemas basados en los diseños presentados en el Capítulo 4, los cuales implementan funciones simétricas de 9 dimensiones, para entradas binarias y multibit.

El Sistema GF6MORPHO1 está compuesto por un arreglo de elementos de procesamiento capaz de procesar una imagen binaria de 64×64 píxeles con un rendimiento de $614,40$ TOPs/s a una frecuencia de reloj de 300MHz. Para reducir los tiempos de procesamiento con respecto al Sistema MORPHO1SYM, se distribuyeron los parámetros de función en paralelo. Debido a esto, el tamaño de la celda de procesamiento aumentó, pero mejoró la eficiencia de cómputo dado que se redujo la energía utilizada en la distribución del argumento y su comparación por celda. Para expandir la capacidad de procesamiento del sistema, se agregó un microprocesador dedicado y un módulo de extracción de descriptores, que en conjunto, permiten la segmentación e identificación de objetos, el cálculo de sus áreas, perímetros, factores de forma, redondez, etc, y el cálculo de sus coordenadas basado en sus posiciones relativas en la imagen. Para disminuir la transferencia de datos entre el sistema y la NoC, se instanció un banco de memoria local distribuida que permite el almacenamiento de

16 imágenes binarias.

Para el procesamiento multibit, se diseñó el Sistema GF6MORPHO8 compuesto por 8 vectores de 64 celdas de procesamiento, similares al Sistema MORPHO8SYM, capaz de procesar entradas de hasta 8 bits de precisión, con un rendimiento de $590,769$ MOPs/s a una frecuencia de reloj de $300MHz$. El sistema se puede configurar para que cada vector de celdas pueda leer el resultado del vector previo y así trabajar en modo *"pipeline"*. Además, se aumentó la precisión de los parámetros a 5 bits con el fin mejorar la capacidad del procesador para su utilización en redes neuronales. Dado que la capacidad del bus de datos con la NoC es alto, se eliminó la memoria de almacenamiento local de imágenes para ahorrar área de silicio y mejorar la densidad del sistema.

Por último, se diseñó exclusivamente para el Nano-Abacus, el Sistema GF6LINEAL que realiza el producto escalar entre dos vectores de 4096 componentes, uno de entrada con precisión 8 bits y el otro de parámetros de 6 bits signado. Para ello, se desarrolló una variación del algoritmo de cálculo simplicial, en el cual se calculan los parámetros de la función en términos de las derivadas parciales de la función lineal a implementar. La rampa de comparación se mantiene, por lo que se puede realizar procesamiento multibit, y procesar entradas con precisión reducida en menor tiempo. Además, el sistema posee un microprocesador dedicado que controla el flujo de datos del vector de entrada y configura los distintos periféricos para poder realizar convoluciones lineales, circulares y calcular correlaciones. Con una frecuencia de operación de $300MHz$, el Sistema GF6LINEAL tiene un rendimiento de $4,800$ GOPs/s para operaciones del tipo MAC (multiplicación y acumulación) de 8 bits y de $76,8$ GOPs/s para una MAC de 4 bits.

Finalmente, los tres sistemas de procesamiento PWL fueron integrados es un chip CMP denominado SoC *Salamis Tablet*. El *Salamis Tablet* fue fabricado en un area de silicio de 14×17 mm^2 en la tecnología 55nm de GlobalFoundries de bajo consumo, y contiene un arreglo de unidades de procesamiento compuesto por 50 sistemas GF6LINEAL, 4 sistemas GF6MORPHO1, y 2 sistemas GF6MORPHO8; más dos procesadores ARM Cortex-M0 y unidades auxiliares. El chip es capaz de realizar 240 GMACs por segundo, $24,572$ TOPs/s de funciones simétricas binarias y $1,181$ GOPs/s de funciones simétricas de 8 bits. Si bien el *Salamis Tablet* no pudo

ser testeado debido un error fatal cometido durante el proceso final de ensamblado de máscaras, simulaciones y verificaciones formales post-layout han validado el funcionamiento y rendimiento del sistema completo.

Capítulo 6

Conclusiones

En esta tesis se han presentado distintas alternativas para el diseño de sistemas digitales basados en estructuras CNN simpliciales para procesamiento de imágenes binarias y multibit, fabricados en tecnologías CMOS de 180nm, 130nm y 55nm.

La primer diferencia con respecto a trabajos pasados fue la separación del procesamiento binario y el multibit, para optimizar los procesos de segmentación y cálculo de descriptores. De esta manera se pudieron realizar arreglos de celdas de un bit más densas, y vectores de procesamiento multibit relativamente pequeños y fácilmente escalables que pueden procesar imágenes a bajo-nivel sin comprometer la velocidad de operación. Además, se agregó la capacidad de ajustar la precisión de cómputo en los sistemas multibit a demanda, lo que permite ajustar la velocidad y la energía consumida en función de los recursos disponibles.

Otra contribución dentro de la tesis es el desarrollo de un nuevo algoritmo, basado en cierto tipo de simetría, que permite implementar vecindades con elementos estructurantes que abarcan más celdas de manera eficiente.

En todos los casos, las arquitecturas propuestas, al ser totalmente parametrizables y sintetizables, resultan independientes de la tecnología a utilizar, lo que permite portabilidad del diseño sin un impacto drástico en la energía por operación, como se puede observar a lo largo de los cuatro chips fabricados. Este tipo de estructuras resultan convenientes frente a aquellas que son totalmente diseñadas manualmente, que si bien son altamente eficientes en área y energía, requieren más tiempo de diseño y pierden generalidad. Implementaciones futuras deberían considerar una

combinación de diseño manual en aquellos circuitos que son intanciados múltiples veces, y diseño con librerías de celdas estándar para agregar plasticidad al sistema.

Si bien los sistemas paralelos requieren mayor área, permiten mantener la cantidad de operaciones por segundo con un reloj más lento. Esto no es beneficioso para tecnologías con grandes corrientes de pérdida, sin embargo, con la utilización de tecnologías SOI [81] o FinFET [82] estos problemas están mitigados. Además si se utilizan frecuencias más bajas y tecnologías con capacidades y resistencias más pequeñas, es posible disminuir la tensión de alimentación, y así, bajar la potencia en un factor cuadrático. Eso se puede observar en las arquitecturas que se fabricaron en distintas tecnologías y con distintos tipos de librerías de celdas estándar. Los sistemas fabricados en la tecnología 55nm LPX fueron testeados a una tensión igual a la mitad de la tensión nominal, lo que permitió un aumento de cuatro veces en la eficiencia del cómputo.

Por último, utilizando el paradigma de paralelismo de cómputo y de sistemas digitales flexibles especializados en procesamiento de imágenes en el plano focal, se diseñaron tres sistemas basados en las arquitecturas anteriormente propuestas utilizando nuevos algoritmos de cómputo basados en funciones lineales a tramos en el dominio simplicial. Los tres sistemas se integraron en un sistema 2.5D muti-chip multi-procesador, fabricado en una tecnología de 55nm, llevado a cabo conjuntamente con la Universidad de Johns Hopkins. Cabe destacar la fundamental importancia de la colaboración internacional para sostener el desarrollo de sistemas basados en tecnologías de última generación.

Apéndice A

Protocolo de comunicación IO

En todas las arquitecturas diseñadas se utilizó un protocolo de comunicación IO basado en AMBA 3 AHB-Lite de ARM®en modo sistema esclavo. Dado la naturaleza de del protocolo, fue necesario implementar un módulo de decodificador de direcciones denominado *Decoder*, que genera las señales de selección de las unidades internas de cada sistema. El segundo módulo es un multiplexor llamado *Multiplexor*, el cual selecciona el bus de salida del módulo interno seleccionado. A continuación se listan las señales utilizadas en cada arquitectura:

- **clk** : señal global de entrada. Todo el temporizado de las señales están relacionadas con el flanco ascendente de *clk*.
- **reset** : señal global de entrada. Cuando se encuentra en alto, reinicia el funcionamiento de todo el sistema, iniciando todos los controladores a sus estados iniciales, y llevando a cero los registros vinculados con el protocolo de comunicación IO.
- **sel_in** : señal de entra de selección global del sistema. Cuando está en alto, la transferencia es válida. Posee el mismo temporizado que las señales de fase de direcciones.
- **wr_en_in** : señal de entrada que indica la dirección de transferencia. Cuando es alto, señala que se esta realizando una operación de escritura, cuando es bajo, una de lectura. Posee el mismo temporizado que las señales de fase de direcciones.

- **address_in [Na-1:0]** : bus de direcciones de entrada de Na bits, que ingresa al *Decoder* para generar las señales de selección internas, y a los módulos internos de cada sistema. Su tamaño depende de cada arquitectura.
- **data_in [Nd-1:0]** : bus de entrada de Nd bits para ingresa los datos durante las operaciones de escritura. Su tamaño depende de cada arquitectura.
- **data_out [Nd-1:0]** : durante las operaciones de lectura, los buses *data_out_x* transfieren datos desde los módulos internos hasta el *Multiplexor*. El multiplexor, luego, transfiere el dato hacia el exterior a través de *data_out*, según la selección interna generada por el *Decoder* durante la fase de dirección.
- **ready_out** : señal de salida generada por la unidad interna seleccionada. Cuando está en alto la señal indica que la transferencia es exitosa. Cuando la señal es cero se debe esperar para que la transferencia pueda ser terminada.

Apéndice B

Apéndice de Tablas

Tabla B.1: Descripción de los registro de configuración y procesamiento del MORPHO1PWL.

Nombre	# Bits	Descripción
is_processing	1	Indica si el sistema se encuentra procesando (*is_processing*=1) o no. Su valor puede ser leído en el bit 0 del bus de salida de *controllerPWL*. Además, modifica inversamente el valor de la señal *ready_out* del módulo *arrayPWL*.
sel_border	2	Configura las condiciones de borde a aquellos PE que se encuentran en la frontera del arreglo. Si es 0, los valores de borde es 0, 1 si es 1, y de ser 2 o 3 se les asigna un valor igual a la de correspondiente de la celda más próxima.
sel_Gu	3	Selecciona una las ocho funciones de G almacenadas en la memoria de funciones.
sel_Fx	4	Selecciona una las ocho funciones de F almacenadas en la memoria de funciones.
sel_FoG	2	Configura la función FoG de la ALU dentro de las celdas (Tab.3.1).
conf_U	1	Configura la formación de la esfera de influencia o el argumento para la función G (Fig.3.3(c) y Fig.3.3(c)).
conf_X	1	Configura la formación de la esfera de influencia o el argumento para la función F (Fig.3.3(a) y Fig.3.3(b)).
sel_U	2	Selecciona el próximo valor que tomaran los registros U de los PE una vez terminado el procesamiento (Tab.3.2).
sel_X	2	Selecciona el próximo valor que tomaran los registros X de los PE una vez terminado el procesamiento (Tab.3.3).
arg_brod	5	Indice del parámetro de la función G y F que se esta propagando al arreglo. En el inicio de un procesamiento, toma el valor 0, y por cada ciclo de reloj, su valor incrementa en uno, hasta llegar a 31.
img_Gu	1	Valor del parámetro indexado por *arg_brod*, de la función G seleccionada por *sel_Gu*.
img_Fx	1	Valor del parámetro indexado por *arg_brod*, de la función F seleccionada por *sel_Fx*.

Tabla B.2: Descripción de los registro de configuración del MORPHO8PWL.

Nombre	# Bits	Descripción
start_reg	1	Si es uno da inicio al procesamiento a través de los controladores de memoria.
hold_reg	1	En uno pone en pausa los controladores de memoria, evitando que se lean nuevos datos de entrada, hasta que su valor sea 0 nuevamente.
stop_reg	1	Si se encuentra en uno, se paran los controladores de memoria y se reinician sus maquinas de estado internas, dando por terminado el procesamiento.
rst_ctrl	1	Reinicia la maquina de estados de control de los procesadores.
sel_pri_sec	1	Si es uno, el controlador de memoria selecciona el *bus_primario* como vector de datos a guardar en las memorias; caso contrario se almacena *bus_secundario*.
sel_cacheXU	1	Selecciona la memoria cache de destino del vector de resultados generador por el arreglo de procesadores. Si es 1 la memoria destino es *cache_X*, si es 0 *cache_U*.
initial_address_rd	8	Almacena el valor de la columna de inicio a procesar para X y U.
op_address_rd	8	Almacena el valor de paso para el indice de direcciones de lectura de columnas.
final_address_rd	8	Almacena el valor de la columna final que se va a procesar.
initial_address_wr	8	Almacena el valor del indice de la primer columna donde se va a escribir el primer vector procesado.
step_address_wr	8	Almacena el valor de paso para el calculo de las direcciones de escritura del vector de resultado.
start_ramp_value	8	Almacena el valor inicial de la rampa de procesamiento.
op_ramp_value	8	Almacena el valor de incremento de la rampa de procesamiento.
final_ramp_value	8	Almacena el valor final de la rampa de procesamiento.
chop_image	1	Si es 0 se realiza una rellenado de los bordes de las imágenes a procesar con un valor seleccionable con sel_border_U y sel_border_X. De ser 1 no se agregan valores de borde, siendo un procesamiento destructivo.
sel_range	2	Selecciona los 8 bits de *FoG* de los procesadores que se desean almacenar (Tab 3.8).
sel_sec	2	Selecciona el valor de la salida *bus_secundario* (Tab.3.10).
en_mask	1	Habilita el enmascaramiento del procesamiento.
conf_U	1	Configura la vecindad para el calculo de la función G_u.
conf_X	1	Configura la vecindad para el calculo de la función F_x.
sel_border_U	2	Selecciona el valor del rellenado de los bordes de la imagen U.
sel_border_X	2	Selecciona el valor del rellenado de los bordes de la imagen X.
sel_FoG	3	Selecciona la operación $\circ$ a realizar en las ALUs de los procesadores.
sel_Fx	3	Selecciona de la memoria de funciones, la función c a utilizar durante el computo.
sel_Gu	3	Selecciona de la memoria de funciones, la función d a utilizar durante el computo.

Tabla B.3: Ubicación en memoria de los registros de configuración del MORPHO8PWL.

Dirección	bits del bus de dato							
	7	6	5	4	3	2	1	0
192	-	sel_cacheXU	sel_pri_sec	-	rst_ctrl	stop_reg	hold_reg	start_reg
193	initial_address_rd							
195	step_address_rd							
196	final_address_rd							
197	initial_address_wr							
198	step_address_wr							
199	start_ramp_value							
200	step_ramp_value							
201	conf_X	conf_U	en_mask	sel_sec		sel_range		chop_image
202	-	sel_FoG		sel_border_X		sel_border_U		
203	-		sel_Gu			sel_Fx		

Tabla B.4: Descripción del bus de salida cuando se accede a *confUnit* para su lectura en el sistema MORPHO1SYM.

Dirección	data_out[7:0]							
	7	6	5	4	3	2	1	0
544	sel_fx				sel_FoG			
545	en_roi[5:0]						sel_border	
546	-	en_roi[8:6]			is_processing	en_regT	en_regU	en_regX
547	-	registros de control				empty_flag	empty_flag_rd	full_flag

Tabla B.5: Descripción de los registros que manejan las señales involucradas en el procesamiento dentro de las celdas y del *feature_ext* en el MORPHO1GF6.

Nombre	Descripción
sel_f	Registro de 4 bits que selecciona el conjunto de 10 parámetros c de la memoria de funciones, para se implementada en los PE.
sel_FoG	Registro de 4 bits que selecciona la operación, que se lleva a cabo en la ALU, entre y y cualquiera de los registros internos de los PE.
sel_roi	Registro de 4 bits que selecciona una de las configuraciónes de enmascaramiento alojadas en memorias, a implementar en las celdas.
sel_border	Registro de 2 bits que selecciona las condiciones de borde para las celdas ubicadas en la frontera del arreglo. Los valores que pueden totar son 0, 1 o el valor del registro *regX* del PE geográficamente más próximo a la frontera.
en_regX	Habilita el registro *regX* para que guerda el dato de salida de la ALU.
en_regU	Habilita el registro *regU* para que guerda el dato de salida de la ALU.
en_regT	Habilita el registro *regT* para que guerda el dato de salida de la ALU.
feature_ext_setup	Registro de 3 bits que configura el procesador extractor de descriptores (Tab.B.8).

Tabla B.6: Descripción simple de las instrucciones implementadas en el procesador de propósito general del MORPHO1GF6.

Instrucción	Descripción
IMOV	Guarda una valor inmediato de 8-bit en el registro local destino dado por Rd. No modifica los registros de bandera.
IBRANCH	Salto condicional de acuerdo a los valores de los registros de bandera. El próximo contador de programa (PC) es el actual más el valor inmediato signado de 6-bits. No modifica los registros de banderas.
ADD	Realiza la suma entre los registros indexados por Rs y Rd, y se guarda el resultado en Rd. Modifica los registros de bandera.
SUB	Realiza la resta entre los registros indexados por Rs y rd, y guarda el resultado en Rd. Modifica los registros de bandera.
OR	Realiza la operación lógica OR entre los registros indexados por Rs y Rd, y guarda el resultado en Rd. Modifica los registros de bandera, exceptuando el de *carry*.
AND	Realiza la operación lógica AND entre los registros indexados por Rs y Rd, y guarda el resultado en Rd. Modifica los registros de bandera, exceptuando el de *carry*.
XOR	Realiza la operación lógica XOR entre los registros indexados por Rs y Rd, y guarda el resultado en Rd. Modifica los registros de bandera, exceptuando el de *carry*.
CMP	Realiza la resta entre los registros indexados por Rs y rd, sin guardar el resultado. Modifica los registros de bandera.
CMPC	Realiza la resta con *carry* entre los registros indexados por Rs y Rd, sin guardar el resultado. Modifica los registros de bandera.
ADC	Realiza la suma con *carry* entre los registros indexados por Rs y Rd, y se guarda el resultado en Rd. Modifica los registros de bandera.
SBC	Realiza la resta con *carry* entre los registros indexados por Rs y Rd, y se guarda el resultado en Rd. Modifica los registros de bandera.
LSR_ROR	Rota en un bit el registro indexado. No modifica los registros de bandera.
RJUMP	Salta a la dirección PC+[Rd]. No modifica los registros de bandera.
LDR	Lee la memoria auxiliar en la dirección dada por el registro interno R14 y carga el valor leído en Rd. No modifica los registros de bandera.
STR	Guarda en la memoria auxiliar, en la dirección dada por el contenido de R15, el valor alojado en Rd. No modifica los registros de bandera.
REGS	Selecciona que set de 16 registros internos van a ser utilizados como registros fuente, y registro destino, independientemente. No modifica los registros de bandera.

Tabla B.7: Consideración especiales de los registros internos del μCPU del MORPHO1GF6.

Nombre de registro	Descripción
R00	Registro de valor cero
R01	Registro reservado para configurar las señales *sel_FoG* y *sel_Fx*. Cuando es modificado, el μCPU habilita el procesamiento en los PE.
R02	Registro reservado para *en_regX*, *en_regU*, *en_regT*, y *sel_roi*. Cuando su MSB es 1, habilita los registros internos de los PE a almacenar el resultado del último computo en las celdas.
R03	Registro reservado para configurar *sel_border* y *feature_ext_setup*. Si el tercer bit es alto, el procesador entra en modo de espera hasta que el *feature_ext* haya terminado. Si el MSB es uno, entonces el procesador espera a que el ICU pulse la señal de interrupción *start_in* para continuar la lectura de la memora de instrucciones.
R14	Registro que almacena de dirección para la instrucción LDR.
R15	Registra que almacena la dirección para la instrucción STR.
R30	Registro cuyos primero cuatros bits corresponden a los registros de bandera (Carry, Zero, Negative and Complement-two overflow).
R31	Registro reservado para el contador de programa (PC).

Tabla B.8: Descripción del registro de configuración del procesador extractor de descriptores *feature_ext* en MORPHO1GF6

Valor	Descripción
0-1	Nulo.
2	Cuenta el número de unos que hay en los datos leídos.
3	Cuenta el número de unos que hay en los datos leídos y calcula sus coordenadas.
4	Escribe unos en los lugares de memoria de imagen, direccionados por los valores de cooredenadas guardados en la memoria auxiliar.
5	El procesador funciona como DMA unicamente.
6	Pausa el procesador.
7	Para el procesador.

Tabla B.9: Registros de configuración de una unidad de procesamiento de MORPHO8GF6.

Nombre	# Bits	Función
padding_value register	8	Valor de condición de borde para la primer y última celda de procesamiento, y primer y último vector de dato de entrada.
padding_mode register	1	Habilita si se agregan un vectores de borde.
total_num_proc	32	Configura la cantidad de vectores de entradas máximos que se deben procesar.
mode_data_rd	1	Si es uno configura el *dinBuffer* para tomar valores del *processor_unit* previo. Si es cero acepta los datos que el ICU le escribe.
mode_data_wr	1	Si es uno configura el *doutBuffer* para que sea leído por la unidad de procesamiento que le sigue. Si es cero, los datos de salida deben ser leídos por el ICU.
start_processing	8	Inicia la maquina de estado del *main_controller*.
reset_fsm	8	Reinicia el controlador de procesamiento.

Tabla B.10: Breve descripción de las instrucciones de programa implementada en el MORPHO8GF6.

Instrucción	Descripción
CONF_RAMP_0	Sets the *start_ramp_value* register and keeps reading the program memory.
CONF_RAMP_1	Sets the *step_ramp_value* register and keeps reading the program memory.
CONF_RAMP_2	Sets the *end_ramp_value* register and keeps reading the program memory.
CONF_PE_0	Sets *sel_Gu* and *en_mask* registers and keeps reading the program memory.
CONF_PE_1	Sets *sel_mask_cell* and *sel_range* registers and keeps reading the program memory.
START_PROC	Sets *sel_Fx*, *sel_Gu* registers, and *end_reading_pm* registers; resets the ramp to the *start_ramp_value* value, sends a reset signals to the accumulator registers in the PEs, and starts processing data going to the PROCESSING_ST state.

Tabla B.11: Registros de configuración de procesamiento del diseño MORPHO8GF6. Su valor es alterado por la maquina de estado cada vez que una nueva instrucción es decodificada.

Nombre	# Bits	Función
sel_Fx	3	Selecciona los parámetros **c** que conforman la función F_x almacenada en la memoria de parámetros, a ser utilizada en los PEs.
sel_Gu	3	Selecciona los parámetros **d** que conforman la función G_x almacenada en la memoria de parámetros, a ser utilizada en los PEs.
sel_FoG	3	Selecciona la función compuesta FoG a operar en la ALU de cada PE.
sel_roi	4	Selecciona el vector de enmascaramiento de vecindad a utilizar durante el procesamiento.
sel_range	3	Selecciona como se trunca el resultado del último procesamiento almacenado en los registros ACCM en cada PE.
en_mask	1	Habilita la opción de enmascaramiento de computo en cada PE dependiendo del valor del MSB del registro central *regX_C*. Si es uno, la celda no procesa. Está opción reduce la precisión de los valores de entrada a 7 bits.
start_ramp_value	8	Valor inicial de la *rampa* de procesamiento.
step_ramp_value	8	Valor de incremento de la *rampa* de procesamiento.
end_ramp_value	8	Valor final de la *rampa* de procesamiento.
end_reading_pm	1	Cuando es uno da por concluído el ciclo de programa.

Tabla B.12: Descripción de las instrucciones de programa de GF6LINEAL.

Instrucción	Descripción
IMOV	Carga un valor inmediato de 8 bits en el registro indexado por Rd.
LOOP	Decrementa en uno el registro indexado por Rd, y si llega a cero, salta al PC-[Rs].
SHIFT	Desplaza los registros de memoria x_i.
LBUFFER	Lee la FIFO de datos de entrada y lo guarda en un registro de almacenamiento temporal interno, el cual funciona como fuente de nuevos datos de entrada a registros de memoria x_i.
START˙PROC	Empieza a procesar tomando el MSB de Rd como el valor final de la rampa $ramp_{end}$, el LSB de Rd como el valor inicial de la rampa $ramp_{ini}$, y Rs como el valor de paso de la rampa $ramp_{step}$.
CONFREG	Configura los registros de configuración de las unidades de procesamiento (Tab.B.13) cargando un valor inmediato al registro indexado por Rd.

Tabla B.13: Registros de configuración de las unidades de procesamiento de GF6LINEAL.

Nombre	# Bits	Descripción
en_processing	8	Cada bit habilita el funcionamiento de un *coreLinear*.
en_get_max_unit	8	Cada bit habilita el funcionamiento de un *get_max_unit*.
en_location_cnt	1	Habilita el contador de cancidad de ciclos de procesamiento que se utiliza entro de las unidades *get_max_unit*. El contador de ciclos incrementa cada vez que la memoria de entradas es desplazada.
en_data_out_FIFO	1	Habilita la FIFO de salida para la salida de resultados.
config_group	2	Configura el la cantidad de*coreLinear* que integran un gurpo de procesamiento. Couando es 0 se forma un solo grupo de procesamiento con los 8 procesadores trabajando en conjunto logrando 4096 entradas. Cuando es 1, existen dos grupos de procesamiento de 4 procesadores cada uno. Cuando es 2, cuatro grupos de procesamiento 2 dos procesadores. Finalmente, si es 3, los ocho *coreLinear* funcionan por separado.
config_flow	1	Configura el flujo de datos de entrada dentro de la memoria x_i. Si es 0, se carga el dato leido FIFO de entrada en el primer registros del arreglo (CONVOLUCIÓN LINEAL). Cuando es 1, el primer registro toma el valor del último del grupo (CONVOLUCIÓN CIRCULAR).
config_acumm	2	Configura la fuente del coeficiente c de las unidades de acumuladores dentro de los *coreLinear*. Para un procesamiento estandar, su valor debe ser igual al de *config_group*.
sel_max_or_min	1	Configura si las unidades *get_max_unit* deben guardar el resultado máximo o mínimo.
sel_equal	1	Configura si las unidades *get_max_unit* deben ser sensibles a valores iguales a los guardados.

Tabla B.14: Descripción de las instrucciones de programa del controlador de la interfaz ICU.

Instrucción	Descripción
LOAD	Carga un valor inmediato en uno de los registros internos
ADD	Suma o resta un valor inmediato a uno de los registros internos
LOOP	Salto condicional a una dirección dada por un registro interno, cuando otro registro seleccionable es igual 0
READ	Realiza un pedido de lectura a L1
WRITE	Escribe a la L1 o L2
SHIFT	Hace un desplazamiento dado por un valor inmediato, del registro de dato temporal interno, que luego puede ser escrito dentro del PU
WAIT	El ICU entra en un estado de espera. Puede esperar la finalización de un contador interno, recibir cierta cantidad de paquetes de la NoC, o espera una señal de respuesta del CPU principal
DONE	Indica el final del programa a ejecutar por el ICU

Tabla B.15: Registros internos del ICU.

Nombre	# Bits	Dirección	Descripción
l1_rd_addr	28	0,1	Dirección de lectura a L1
l1_wr_addr	28	2,3	Dirección de escritura a L1
l1_addr_inc	28	4,5	Valor de incremente para la dirección de L1
l2_addr	8	6	Dirección destino de L@ (fila y columna)
pu_wr_addr	16	7	Dirección de escritura hacia el PU
pu_rd_addr	16	8	Dirección de lectura hacia el PU
byte_mask	32	9,10	Bytes de enmascaramiento para escribir a memoria
rd_cnt	16	11	Contador de paquetes recibidos desde el L1 y L2
wr_sel	2	12	Selección de escritura del buffer interno
loop_cnts	16	16-31	Contadores de la instrucción LOOP

Tabla B.16: Puertos IO de la interfaz ICU con red L1.

Nombre	Sentido	# Bits	Descripción
L1 a PU			
N1_PU_data_i	IN	256	Dato de entrada desde la red L1
N1_PU_addr_i	IN	256	Dirección de L1
N1_PU_tag_addr i	IN	24	Dirección de escritura hacia el PU
N1_PU_req_i	IN	1	Señal de solicitud de transacción
N1_PU_ack_o	OUT	1	Señal de respuesta de transacción
PU a L1			
PU_N1_data_o	OUT	256	Dato de salida hacia la L1
PU_N1_op_o	OUT	2	Comando (lectura y escritura)
PU_N1_addr_o	OUT	40	Dirección L1
PU_N1_tag_addr_o	OUT	24	Dirección de escritura del PU
PU_N1_req_o	OUT	1	Señal de solicitud de transacción
PU_N1_ack_i	IN	1	Señal de respuesta de transacción

Tabla B.17: Puertos IO de la interfaz ICU con red L2.

Nombre	Sentido	# Bits	Descripción
L2 a PU			
N2_PU_is_data_i	IN	1	Indica si es un paquete de datos o de configuración
N2_PU_data_i	IN	256	Dato de entrada
N2_ver_addr_i	IN	256	Dirección fila de L2
N2_hor_addr_i	IN	256	Dirección columna de L2
N2_PU_reg_addr_i	IN	10	Dirección PU de entrada
N2_PU_reg_part_i	IN	6	Identificador PU de entrada
N2_PU_req_i	IN	1	Señal de solicitud de transacción
N2_PU_ack_o	OUT	1	Señal de respuesta de transacción
PU a L2			
PU_N2_is_data o	IN	1	Indica si es un paquete de datos o de configuración
PU_N2_data_o	OUT	256	Dato de salida
PU_N2_dest_ver_addr_o	OUT	8	Dirección fila de L2
PU_N2_dest_hor_addr_o	OUT	8	Dirección columna de L2
PU_N2_reg_addr_o	OUT	10	Dirección PU de salida
PU_N2_reg_part_o	OUT	6	Identificador PU de salida
PU_N2_req_o	OUT	1	Señal de solicitud de transacción
PU_N2_ack_i	IN	1	Señal de respuesta de transacción

Bibliografía

[1] Massimo Alioto. *Enabling the Internet of Things: From Integrated Circuits to Integrated Systems*. Springer Publishing Company, Incorporated, 1st edition, 2017.

[2] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to End Learning for Self-Driving Cars. *arXiv*, April 2016.

[3] Andreas G Andreou, Tomas Figliolia, Kayode Sanni, Thomas S Murray, Gaspar Tognetti, Daniel R Mendat, Jamal L Molin, Martin Villemur, Philippe O Pouliquen, Pedro Julian, Ralph Etienne-Cummings, and Isidoros Doxas. Bio-inspired system architecture for energy efficient, BIGDATA computing with application to wide area motion imagery. In *2016 IEEE 7th Latin American Symposium on Circuits & Systems (LASCAS)*, pages 1–6. IEEE, 2016.

[4] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, October 2010.

[5] L Chua and L Yang. Cellular neural networks: theory. *Circuits and Systems, IEEE Transactions on*, 35(10):1257–1272, October 1988.

[6] L Chua and L Yang. Cellular neural networks: applications. *Circuits and Systems, IEEE Transactions on*, 35(10):1273–1290, October 1988.

[7] L Chua and T Roska. The CNN paradigm. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, 40(3):147–156, March 1993.

[8] P Julian, A Desages, and O Agamennoni. High-level canonical piecewise linear representation using a simplicial partition. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 46(4):463–480, April 1999.

[9] R Dogaru, P Julian, L Chua, and M Glesner. The simplicial neural cell and its mixed-signal circuit implementation: an efficient neural-network architecture for intelligent signal processing in portable multimedia applications. *Neural Networks, IEEE Transactions on*, 13(4):995–1008, July 2002.

[10] Pablo S Mandolesi, Pedro Julian, and Andreas G Andreou. A scalable and programmable simplicial CNN digital pixel processor architecture. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 51(5):988–996, May 2004.

[11] M Di Federico and Mandolesi. Experimental results of simplicial cnn digital pixel processor. *Electronics Letters*, 44(1):27–29, 2008.

[12] M Di Federico, P Julian, and P S Mandolesi. SCDVP: A Simplicial CNN Digital Visual Processor. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(7):1962–1969, 2014.

[13] M Di Federico, P Julian, A G Andreou, and P S Mandolesi. Fully functional fine-grain vertically integrated 3D focal plane neuromorphic processor. In *SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S), 2014 IEEE*, pages 1–2. IEEE, 2014.

[14] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian,

H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, June 2017.

[15] M. Villemur, P. Julian, T. Figliola, and A. G. Andreou. Neuromorphic cellular neural network processor for intelligent internet-of-things. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2018.

[16] M. Villemur, P. Julian, and A. G. Andreou. Energy aware simplicial processor for embedded morphological visual processing in intelligent internet of things. *Electronics Letters*, 54(7):420–422, 2018.

[17] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, Jan 2002.

[18] C. Lee, C. Hung, C. Cheung, P. Yang, C. Kao, D. Chen, M. Shih, C. C. Chien, Y. Hsiao, L. Chen, M. Su, M. Alfano, J. Siegel, J. Din, and B. Black. An overview of the development of a gpu with integrated hbm on silicon interposer. In *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*, pages 1439–1444, May 2016.

[19] N. E. Jerger, A. Kannan, Z. Li, and G. H. Loh. Noc architectures for silicon interposer systems: Why pay for more wires when you can get them (from your interposer) for free? In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 458–470, Dec 2014.

[20] M Villemur, M Di Federico, and P Julian. A vlsi convolutional neural network architecture for vanishing point computation. In *2015 Argentine School of Micro-Nanoelectronics, Technology and Applications (EAMTA)*, pages 53–57, July 2015.

[21] M Villemur, M Di Federico, P Julian, A G Andreou, F Masson, and E Nebot. Design of a vanishing point algorithm for custom asic. In *2015 49th Annual*

Conference on Information Sciences and Systems (CISS), pages 1–5, March 2015.

[22] Liang Wang, Fangliang Chen, and Huiming Yin. Detecting and tracking vehicles in traffic by unmanned aerial vehicles. *Automation in Construction*, 05 2016.

[23] Dongping Tian. A review on image feature extraction and representation techniques. *International Journal of Multimedia and Ubiquitous Engineering*, 8:385–395, 01 2013.

[24] Stephen M. Pizer, E. Philip Amburn, John D. Austin, Robert Cromartie, Ari Geselowitz, Trey Greer, Bart Ter Haar Romeny, and John B. Zimmerman. Adaptive histogram equalization and its variations. *Comput. Vision Graph. Image Process.*, 39(3):355–368, September 1987.

[25] Matt Hall. Smooth operator: Smoothing seismic interpretations and attributes. *The Leading Edge*, 26(1):16–20, 2007.

[26] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Prentice Hall, Upper Saddle River, N.J., 2008.

[27] Moncef Gabbouj and J Astola. Nonlinear order statistic filter design: Methodologies and challenges. *European Signal Processing Conference*, 2015, 03 2015.

[28] Juan Díz de León Santiago, Arturo Gamino, Julio Salgado, Valentín Trujillo, and Alicia Ortiz. Operadores k-estadistícos para morfología matemática de conjuntos. *Revista Facultad de Ingeniería Universidad de Antioquia*, pages 216 – 227, 06 2009.

[29] M. S. Darus, S. N. Sulaiman, I. S. Isa, Z. Hussain, N. M. Tahir, and N. A. M. Isa. Modified hybrid median filter for removal of low density random-valued impulse noise in images. In *2016 6th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*, pages 528–533, Nov 2016.

[30] Jean Serra. *Image Analysis and Mathematical Morphology*. Academic Press, Inc., Orlando, FL, USA, 1983.

[31] J.L.D. de León Santiago and C.Y. Márquez. *Introducción a la morfología matemática de conjuntos*. Ciencia de la computación. Instituto Politécnico Nacional, 2003.

[32] H. Hadwiger. Minkowskische addition und subtraktion beliebiger punktmengen und die theoreme von erhard schmidt. *Mathematische Zeitschrift*, 53(3):210–218, Jun 1950.

[33] Christian Lantuéjoul. *Skeletonization in Quantitative Metallography*, pages 107–135. 01 1980.

[34] Thresholding using the isodata clustering algorithm. *IEEE Transactions on Systems, Man, and Cybernetics*, 10(11):771–774, Nov 1980.

[35] G W Zack, W E Rogers, and SA Latt. Automatic measurement of sister chromatid exchange frequency. *The journal of histochemistry and cytochemistry : official journal of the Histochemistry Society*, 25:741–53, 08 1977.

[36] Lucas J van Vliet, Ian T Young, and Guus L Beckers. A nonlinear laplace operator as edge detector in noisy images. *Computer Vision, Graphics, and Image Processing*, 45(2):167 – 195, 1989.

[37] J. Lee, R. Haralick, and L. Shapiro. Morphologic edge detection. *IEEE Journal on Robotics and Automation*, 3(2):142–156, April 1987.

[38] D. Marr, E. Hildreth, and Sydney Brenner. Theory of edge detection. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 207(1167), 1980.

[39] P. W. Verbeek and L. J. van Vliet. On the location error of curved edges in low-pass filtered 2-d and 3-d images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):726–733, July 1994.

[40] F. Meyer and S. Beucher. Morphological segmentation. *Journal of Visual Communication and Image Representation*, 1(1):21 – 46, 1990.

[41] Fernand Meyer. The morphological approach to segmentation: the wa-tershed transformation. 1993.

[42] Jose Hilera and Victor Martinez Hernando. Redes neuronales artificiales : fundamentos, modelos y aplicaciones. 04 2019.

[43] Andreas Zell. Simulation neuronaler netze. 08 1994.

[44] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, Nov 1997.

[45] S. Lawrence, C. L. Giles, , and A. D. Back. Face recognition: a convolutional neural-network approach. *IEEE Transactions on Neural Networks*, 8(1):98–113, Jan 1997.

[46] L. O. Chua and L. Yang. Cellular neural networks: theory. *IEEE Transactions on Circuits and Systems*, 35(10):1257–1272, Oct 1988.

[47] L. O. Chua and L. Yang. Cellular neural networks: applications. *IEEE Transactions on Circuits and Systems*, 35(10):1273–1290, Oct 1988.

[48] Leon O. Chua. Cnn: A vision of complexity. *International Journal of Bifurcation and Chaos*, 07(10):2219–2425, 1997.

[49] P. Julian, R. Dogaru, and L. O. Chua. A piecewise-linear simplicial coupling cell for cnn gray-level image processing. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 49(7):904–913, July 2002.

[50] P. Julian, A. Desages, and B. D'Amico. Orthonormal high-level canonical pwl functions with applications to model reduction. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47(5):702–712, May 2000.

[51] P. Julian, A. Desages, and O. Agamennoni. High-level canonical piecewise linear representation using a simplicial partition. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 46(4):463–480, April 1999.

[52] M. Chien and E. Kuh. Solving nonlinear resistive networks using piecewise-linear analysis and simplicial subdivision. *IEEE Transactions on Circuits and Systems*, 24(6):305–317, June 1977.

[53] R. Dogaru, P. Julian, L. O. Chua, and M. Glesner. The simplicial neural cell and its mixed-signal circuit implementation: an efficient neural-network architecture for intelligent signal processing in portable multimedia applications. *IEEE Transactions on Neural Networks*, 13(4):995–1008, July 2002.

[54] P. S. Mandolesi, P. Julian, and A. G. Andreou. A scalable and programmable simplicial cnn digital pixel processor architecture. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 51(5):988–996, May 2004.

[55] S. Lee, M. Kim, K. Kim, J. Kim, and H. Yoo. 24-gops 4.5-mm^{2}digital cellular neural network for rapid visual attention in an object-recognition soc. *IEEE Transactions on Neural Networks*, 22(1):64–73, Jan 2011.

[56] Robert Desimone and John Duncan. Neural mechanisms of selective visual attention. *Annual Review of Neuroscience*, 18(1):193–222, 1995. PMID: 7605061.

[57] W. Zhang, Q. Fu, and N. Wu. A programmable vision chip based on multiple levels of parallel processors. *IEEE Journal of Solid-State Circuits*, 46(9):2132–2147, Sep. 2011.

[58] S. J. Carey, A. Lopich, D. R. W. Barr, B. Wang, and P. Dudek. A 100,000 fps vision sensor with embedded 535gops/w 256x256 simd processor array. In *2013 Symposium on VLSI Circuits*, pages C182–C183, June 2013.

[59] M. Di Federico, P. Julian, and P. S. Mandolesi. Scdvp: A simplicial cnn digital visual processor. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(7):1962–1969, July 2014.

[60] M. D. Federico, P. S. Mandolesi, P. Julian, and A. G. Andreou. Experimental results of simplicial cnn digital pixel processor. *Electronics Letters*, 44(1):27–29, January 2008.

[61] D. G. Bobrow. *"Determining optical flow": a retrospective*. MITP, 1994.

[62] M. Di Federico, P. Julian, A. G. Andreou, and P. S. Mandolesi. Fully functional fine-grain vertically integrated 3d focal plane neuromorphic processor.

In *2014 SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pages 1–2, Oct 2014.

[63] J. A. Schmitz, M. K. Gharzai, S. Balkir, M. W. Hoffman, D. J. White, and N. Schemm. A 1000 frames/s vision chip using scalable pixel-neighborhood-level parallel processing. *IEEE Journal of Solid-State Circuits*, 52(2):556–568, Feb 2017.

[64] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. Yodann: An ultra-low power convolutional neural network accelerator based on binary weights. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 236–241, July 2016.

[65] A. A. Bahou, G. Karunaratne, R. Andri, L. Cavigelli, and L. Benini. Xnorbin: A 95 top/s/w hardware accelerator for binary convolutional neural networks. In *2018 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, pages 1–3, April 2018.

[66] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.

[67] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, Jan 2017.

[68] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[69] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, Oct 2015.

[70] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659 – 1671, 1997.

[71] A. S. Cassidy, P. Merolla, J. V. Arthur, S. K. Esser, B. Jackson, R. Alvarez-Icaza, P. Datta, J. Sawada, T. M. Wong, V. Feldman, A. Amir, D. B. Rubin, F. Akopyan, E. McQuinn, W. P. Risk, and D. S. Modha. Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, Aug 2013.

[72] Adam Coates and Andrew Y. Ng. The importance of encoding versus training with sparse coding and vector quantization. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 921–928, USA, 2011. Omnipress.

[73] R. Kasturi, D. B. Goldgof, R. Ekambaram, G. Pratt, E. Krotkov, D. D. Hackett, Y. Ran, Q. Zheng, R. Sharma, M. Anderson, M. Peot, M. Aguilar, D. Khosla, Y. Chen, K. Kim, L. Elazary, R. C. Voorhies, D. F. Parks, and L. Itti. Performance evaluation of neuromorphic-vision object recognition algorithms. In *2014 22nd International Conference on Pattern Recognition*, pages 2401–2406, Aug 2014.

[74] M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, January 2018.

[75] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659 – 1671, 1997.

[76] N. Anand, G. Joseph, and S. S. Oommen. Performance analysis and implementation of clock gating techniques for low power applications. In *2014 International Conference on Science Engineering and Management Research (ICSEMR)*, pages 1–4, Nov 2014.

[77] F N David, M G Kendall, and D E Barton. *Symmetric Function and Allied Tables*. Cambridge University Press, 1968.

[78] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, May 2006.

[79] E. R. Hsieh, C. H. Chuang, and S. S. Chung. An innovative 1t1r dipole dynamic random access memory (diram) featuring high speed, ultra-low power, and low voltage operation. In *2016 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*, pages 1–2, April 2016.

[80] W. Bux, F. Closs, K. Kuemmerle, H. Keller, and H. Mueller. Architecture and design of a reliable token-ring network. *IEEE Journal on Selected Areas in Communications*, 1(5):756–765, November 1983.

[81] M. Yamaoka, K. Osada, R. Tsuchiya, M. Horiuchi, S. Kimura, and T. Kawahara. Low power sram menu for soc application using yin-yang-feedback memory cell technology. In *2004 Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No.04CH37525)*, pages 288–291, June 2004.

[82] T. Skotnicki, C. Fenouillet-Beranger, C. Gallon, F. Boeuf, S. Monfray, F. Payet, A. Pouydebasque, M. Szczap, A. Farcy, F. Arnaud, S. Clerc, M. Sellier, A. Cathignol, J. Schoellkopf, E. Perea, R. Ferrant, and H. Mingam. Innovative materials, devices, and cmos technologies for low-power mobile multimedia. *IEEE Transactions on Electron Devices*, 55(1):96–130, Jan 2008.