UNS UNIVERSIDAD NACIONAL DEL SUR

2024
Bahía Blanca, Argentina

# Tesis de Doctorado en Ingeniería Eléctrica

## Arquitecturas hardware de procesamiento para redes neuronales por eventos

Diego Gigena Ivanovich

**Director:** Dr. Pedro Julián / **Co-director:** Dr. Carlos De Marziani

# Prefacio

Esta tesis se presenta como parte de los requisitos para acceder al grado académico de Doctor en Ingeniería Eléctrica, 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 en investigaciones llevadas a cabo en el ámbito del Departamento de Ingeniería Eléctrica y de Computadoras (DIEC) de la Universidad Nacional del Sur, durante el período comprendido entre el 1 de Abril de 2018 al 1 de Julio de 2024, bajo la dirección del Dr. Pedro Julián, IIIE-CONICET, Departamento de Ingeniería Eléctrica y de Computadoras, y el Dr. Carlos De Marziani, Departamento de Ingeniería Electrónica de la Universidad Nacional de la Patagonia San Juan Bosco (UNPSJB).

Ing. Diego Gigena Ivanovich

UNIVERSIDAD NACIONAL DEL SUR

Subsecretaría de Posgrado

La presente tesis ha sido aprobada el 06/03/2025,

SAFE Martin Dario
Subsecretario de Posgrado
27.06.2025
15:47:32

mereciendo la calificación de Sobresaliente (10).

# Agradecimientos

En primer lugar quiero agradecer a mi familia, que me ha acompañado y brindado su apoyo incondicional en cada etapa de mi vida desde que tengo uso de razón. A mi compañera de vida, quien estuvo a mi lado en las incontables horas de arduo trabajo que hicieron posible la realización de esta tesis. A mi director, por abrirme las puertas al mundo y guiarme en estos primeros pasos de la vida profesional. Y a mis amigos y colegas, quienes, de una manera u otra, me acompañaron en este camino.

# Índice

# Lista de figuras

[1] *Application Specific Integrated Circuit*

# Lista de tablas

# Acrónimos

**CAM** *Content Addressable Memory* (Memoria Direccionable por Contenido) . . . . . 5

**FPS** *Frames Per Second* (Fotogramas Por Segundo) . . . . . XVII

**RAM** *Random Access Memory* (Memoria de Acceso Aleatorio) . . . . . 26

**AMBA** *Advanced Microcontroller Bus Architecture* (Arquitectura avanzada de Controladores de Bus) . . . . . 71

**AHB** *Advanced High-performance Bus* (Bus avanzado de alto rendimiento) . . . . . 71

**DVS** *Dynamic Vision Sensor* . . . . . 11

**ATIS** *Asynchronous Time Based Image Sensor* (Sensor de Imagen Asíncrono Basado en Tiempo) . . . . . 11

**DAVIS** *Dynamic and Active Pixel Vision Sensor* (Sensor de Visión Dinámico y de Píxel Activo) . . . . . 12

**AER** *Address Event Representation* . . . . . 4

**AVR** *Address Value Representation* . . . . . 23

**ADC** *Analog to Digital Converter* . . . . . 13

**NUC** *Non-Uniform Correction* . . . . . 13

**SNN** *Spiking Neural Networks* . . . . . 10

**CPU** *Central Processing Unit* . . . . . 10

**GPU** *Graphic Processing Unit* . . . . . 10

**CNN** *Convolucional Neural Network* . . . . . 13

**DNN** *Deep Neural Network* . . . . . 5

# Resumen

En los últimos años, el avance exponencial en la capacidad de procesamiento computacional, combinado con el desarrollo de la inteligencia artificial, en particular de las redes neuronales profundas, ha impulsado avances notables en áreas diversas como la medicina, el Internet de las cosas (IoT) y los vehículos inteligentes, alcanzando resultados sobresalientes gracias al aprendizaje profundo. Los sistemas de visión artificial capturan y procesan secuencias de fotogramas a una tasa fija de cuadros (FPS[2]), generalmente a través de operaciones de convolución que extraen características para generar un resultado. Sin embargo, este procesamiento es altamente demandante en términos energéticos dado que requiere el procesamiento completo de una imagen, lo cual se agrava a medida que se requieren tasas de cuadros más elevadas. Además, las cámaras convencionales presentan limitaciones como el desenfoque por movimiento cuando intentan capturar escenas rápidas.

En contraposición, los sensores de visión por eventos, debido a su principio de funcionamiento, registran únicamente los cambios en la dinámica de la escena, lo que permite alcanzar altas resoluciones temporales con un menor ancho de banda. Para aprovechar estas ventajas, es fundamental contar con plataformas de hardware especializadas que puedan procesar de manera eficiente.

En esta tesis se proponen técnicas para la adquisición y el procesamiento de datos por eventos mediante redes neuronales convolucionales profundas y arquitecturas para implementaciones energéticamente eficientes en circuitos integrados CMOS. Para ello se utilizan memorias direccionables por contenido (CAM, del inglés *Content Addressable Memories*), dado que presentan ventajas para la adquisición, actualización y almacenamiento de eventos y matrices dispersas en

[2] *Frames Per Second* (Fotogramas Por Segundo)

tiempo real.

Una contribución central de la presente tesis doctoral es la implementación de un Sistema en chip (SoC) constituido por dos memorias CAM capaces de almacenar 1024 coordenadas de una imagen de a lo sumo 512 x 512 pixeles. Además, el SoC integra un procesador RISC V dedicado para el manejo del sistema, un acelerador para el cómputo de características con su decodificador de coordenadas, 80 KiB de memoria RAM, dos bloques dedicados para la adquisición por eventos y ocho QSPI para tráfico de datos. Este sistema, que permite implementar en forma secuencial varias capas convolucionales, fue fabricado en una tecnología CMOS de 65 nm, produciendo un SoC de 9 $mm^2$, el cual fue probado exitosamente y verificado en tiempo real con datos tanto artificiales como provenientes de una cámara por eventos comercial.

Las mediciones de potencia y eficiencia realizadas, demostraron una eficiencia energética que supera en al menos un 16 % a los consumos obtenidos mediante un esquema de cómputo de matriz densa tradicional, particularmente para casos de matrices altamente dispersas con niveles de activación menores al 1 %.

# Abstract

In recent years, the exponential growth in computational processing power, combined with the development of artificial intelligence—particularly deep neural networks—has driven remarkable advances in diverse fields such as medicine, the Internet of Things (IoT), and intelligent vehicles, achieving outstanding results thanks to deep learning. Artificial vision systems capture and process frame sequences at a fixed frame rate (FPS), generally via convolution operations that extract features to produce an output. However, this processing is highly energy-intensive, since it requires the full processing of each image, an issue that becomes more severe as higher frame rates are demanded. In addition, conventional cameras suffer from motion blur when attempting to capture fast scenes.

In contrast, event-based vision sensors, due to their operating principle, record only changes in the scene's dynamics, allowing them to achieve high temporal resolution with lower bandwidth. To exploit these advantages, it is essential to have specialized hardware platforms that can process events efficiently.

This thesis proposes techniques for the acquisition and processing of event-based data using deep convolutional neural networks, as well as architectures for energy-efficient implementation in CMOS integrated circuits. To this end, content-addressable memories (CAMs) are employed, since they offer advantages for the real-time acquisition, updating, and storage of events and sparse matrices.

A central contribution of this doctoral thesis is the implementation of a system-on-chip (SoC) comprising two CAMs capable of storing 1024 coordinates from an image up to 512 × 512 pixels. The SoC also integrates a dedicated RISC-V processor for system management, a feature-compute accelerator with its coordinate decoder, 80 KiB of RAM, two dedicated event-acquisition blocks, and eight QSPI interfaces for data traffic. This system—able to sequentially imple-

ment multiple convolutional layers—was fabricated in a 65 nm CMOS process, yielding a 9 $mm^2$ SoC, which was successfully tested and verified in real time using both synthetic data and data from a commercial event-based camera.

Power and efficiency measurements demonstrated an energy efficiency at least 16 % higher than that of a traditional dense-matrix computation scheme, particularly in cases of highly sparse matrices with activation levels below 1 %.

# Capítulo 1

# Introducción

Los sistemas de visión convencionales producen imágenes estáticas a una tasa de cuadros fija. Típicamente consisten en un arreglo de píxeles fotosensibles que producen una señal correspondiente al valor absoluto de la iluminación en cada punto de una imagen, obteniendo imágenes ricas en colores y textura, seguido de un sistema hardware de cómputo de alta capacidad que permite extraer información sobre la misma para llevar a cabo una tarea.

La evolución de las redes neuronales, impulsada por el crecimiento exponencial en el desarrollo de sistemas hardware de cómputo, ha hecho posible la creación de sistemas avanzados de visión artificial. Sin embargo, procesar imágenes estáticas por medio de redes neuronales tiene un elevado costo computacional y energético, que se incrementa exponencialmente a medida que se necesita aumentar la tasa de cuadros por segundo para extraer información relacionada a la dinámica de la escena. El cerebro, por el contrario, lleva a cabo todas las funciones neuronales y cognitivas con un consumo promedio de solo 20 Watts. Esto ha motivado a la comunidad científica a buscar alternativas a los sensores convencionales, inspirándose en los mecanismos biológicos que utiliza el cerebro, por ejemplo para procesar señales visuales. La retina contiene 5 capas de neuronas, a través de las cuales la información fluye tanto verticalmente (entre una capa y otra), como horizontalmente (entre neuronas vecinas dentro de la misma capa). La primer capa está constituida por células fotorreceptoras denominadas conos y bastones. Los bastones son altamente sensibles y desempeñan un papel fundamental en situaciones con poca luz, mientras que los conos son menos sen-

sibles, y son los responsables de la visión a color. Estos fotorreceptores son los responsables de convertir la luz incidente en señales eléctricas y transmitirlas a las células de la siguiente capa, denominadas células horizontales. Estas células se conectan con los fotorreceptores y sus células vecinas, formando una membrana de potencial, cuyo valor en cada célula está determinado por el promedio pesado de los potenciales de sus células vecinas. La tercer capa de células, denominadas células bipolares, se conectan tanto con los fotorreceptores como con las células horizontales, y producen una señal proporcional a la diferencia de ambas, por lo que debe producirse movimiento, o un cambio de contraste en la intensidad lumínica de la escena para que éstas células propaguen la información. De darse el caso, la información producida por las células bipolares pasa a través de de la capa de células amacrinas hasta las células ganglionares, y de ahí hacia el nervio óptico en forma de impulsos eléctricos denominados *spikes*.

Esta modalidad de comunicación inspiró el desarrollo de la retina de silicio (*Silicon Retina*) por Misha Mahowald y Carver Mead en 1991 [7], un sensor basado en la retina humana que imita el comportamiento de las células de la retina del ojo humano antes descrito, y con este hito, mostró una nueva forma de realizar cómputo, con eficiencias comparables a las observadas en la biología, dando origen a la ingeniería neuromórfica. A partir de allí, los esfuerzos hacia la comprensión de los mecanismos de procesamiento y aprendizaje del cerebro y su implementación en circuitos electrónicos y sensores neuromórficos continuaron evolucionando hacia soluciones más prácticas y eficientes [8, 9, 10, 2] que incorporen no solamente los circuitos analógicos de sensado, sino también interfaces digitales con protocolos de comunicación como el de representación de eventos por coordenadas *Address Event Representation* (AER[1]) que permiten acceder a todo el poder de cómputo de los procesadores modernos. Actualmente, empresas como iniVation [11] y Prophesee [12] desempeñan un papel fundamental comercializando esta tecnología, lo que lleva a que grupos de investigación de todo el mundo hayan focalizado su atención hacia el desarrollo de algoritmos para procesar y extraer información de estos sensores, que desempeñan un rol fundamental en campos como la robótica [13, 14], vehículos inteligentes [15, 16, 17] y visión

[1] *Address Event Representation*

artificial [18].

El procesamiento de redes neuronales profundas (DNN[2]), especialmente las redes convolucionales aplicadas a sistemas de visión, demanda un alto consumo energético debido a la necesidad de procesar cada píxel de la imagen y propagar el resultado a través de múltiples capas de la red. En contraste, los sensores de visión por eventos registran y transmiten únicamente los cambios que ocurren en una escena, lo que da lugar a datos altamente dispersos y reduce significativamente la cantidad de información a procesar. Esta tesis propone una arquitectura de hardware optimizada para el procesamiento de redes neuronales convolucionales, cuyas entradas suceden por eventos, utilizando memorias direccionables por contenido (CAM[3]). Este enfoque permite almacenar y procesar los eventos de manera más eficiente, lo que resulta en un ahorro considerable de energía al evitar el procesamiento innecesario de píxeles sin cambios.

La tesis se encuentra organizada de la siguiente forma: El Capítulo 2 realiza una revisión de los sensores de visión por eventos existentes, e introduce conceptos generales de procesamiento de eventos, sus mecanismos de representación y tipos de algoritmos que están siendo utilizados actualmente para procesar dichos eventos. Se desarrollan conceptos básicos de redes neuronales convolucionales y redes neuronales dispersas, con énfasis en el procesamiento de eventos, y se realiza una revisión de las arquitecturas de hardware para procesamiento de redes neuronales por eventos más relevantes. A continuación, en el Capítulo 3 se define el mecanismo de cómputo de los algoritmos de redes convolucionales dispersas utilizando memorias CAM, y se proponen estrategias para ejecutar de manera secuencial las distintas capas. En el Capítulo 4, se presenta la distribución de estos algoritmos en bloques de cómputo de hardware específicos. Se incluye además un análisis detallado en términos de tiempo de procesamiento y dimensionamiento, y se realiza un análisis de energía, donde se analizan los cuellos de botella y se establecen las relaciones entre cantidad de recursos y desempeño. En el Capítulo 5, se describe el proceso de diseño, fabricación y verificación de un sistema en chip (SoC[4]) para adquisición de eventos y procesamiento de redes convoluciona-

[2] *Deep Neural Network*
[3] *Content Addressable Memory* (Memoria Direccionable por Contenido)
[4] *System on Chip*

les dispersas empleando memorias CAM en tecnología de 65 nm de la empresa TSMC. En el Capítulo 6, se realiza un análisis global de los resultados obtenidos, y se plantean potenciales líneas de trabajo futuro en esta temática.

Durante el desarrollo de este trabajo se han realizado diversas contribuciones. Entre ellas destacan el análisis de ruido del sensor por eventos FRIS, publicado en [19], la implementación de un dataset de sombras de manos basado en eventos mediante un algoritmo de emulación, y el entrenamiento de una red neuronal convolucional para tareas de clasificación, publicado en [20]. A partir de estos avances, se realizó una prueba de concepto en tiempo real mediante la integración de una cámara de eventos. Asimismo, en colaboración con investigadores de varias universidades europeas, se publicó un trabajo en el que se presenta un dataset de eventos para el reconocimiento del alfabeto Braille utilizando un sensor táctil [21].

Además, se diseñó e implementó el entorno de verificación del SoC DigineuronV1, publicado en [22], y se contribuyó al diseño de la arquitectura del SoC DigineuronV2, así como al desarrollo de su entorno de verificación, publicado en [23]. Posteriormente, se creó un mecanismo de cómputo de redes neuronales basadas en eventos utilizando memorias CAM, lo que derivó en la presentación de una solicitud de patente [24]. Se implementó un sistema en chip (SoC) de 9 $mm^2$ en tecnología CMOS de 65 nm que incorpora una arquitectura para adquisición de eventos y su procesamiento mediante redes neuronales convolucionales dispersas basada en un procesador RISC-V. Esta última contribución ha resultado en la redacción de un nueva publicación, actualmente en proceso de revisión al momento de redacción de esta tesis.

# Capítulo 2

# Conceptos Preliminares

En este capítulo, se presentan los conceptos preliminares necesarios para comprender el paradigma de procesamiento de eventos en sistemas neuromórficos. Se abordan los principios clave para extraer información significativa a partir de eventos capturados por sensores especializados, que difieren fundamentalmente de los sensores tradicionales en términos de captura y representación de datos. Además, se examinan diferentes mecanismos de procesamiento y transformación de estos eventos en representaciones que faciliten la aplicación de algoritmos de visión computacional y redes neuronales.

Este análisis incluye una discusión sobre las ventajas y limitaciones de los sensores de imágenes por eventos, como el Dynamic Vision Sensor (DVS), y cómo se ha adaptado el procesamiento a redes neuronales convolucionales (CNNs) mediante representaciones temporales y espaciales eficientes. También se exploran enfoques avanzados, como las redes convolucionales dispersas y su implementación en hardware dedicado, lo cual es clave para lograr una baja latencia y consumo energético en aplicaciones prácticas.

A lo largo del capítulo, se destacan las innovaciones en la integración de tecnologías y algoritmos para el procesamiento eficiente de datos dispersos y el reconocimiento de patrones, sentando las bases para el desarrollo de soluciones neuromórficas en tiempo real.

## 2.1. Procesamiento por Eventos

Uno de los aspectos clave a tener en cuenta en el paradigma de procesamiento por eventos, es cómo extraer información significativa de los datos obtenidos mediante los sensores, dado que los eventos individuales solo proporcionan información puntual sobre un único pixel. Existen distintos tipos de algoritmos para procesar eventos, algunos operan evento por evento, minimizando la latencia introducida por el procesamiento, mientras que otros operan en grupos o paquetes de eventos, los cuales introducen cierta latencia, pero permiten que estos sean transformados en otro tipo de representación alternativa, que permita extraer de manera más sencilla la información contenida, y así posibilitar el uso de algoritmos de visión computacional tradicionales como lo son las redes neuronales convolucionales. Aún sin considerar la latencia, los métodos basados en grupos (es decir, ventanas temporales) de eventos pueden seguir proporcionando una actualización del estado con la llegada de cada evento si la ventana se desliza por cada evento individual. Un solo evento no proporciona suficiente información para la estimación, por lo que se necesita información adicional en forma de eventos pasados o conocimiento extra.

Los eventos individuales son utilizados por mecanismos de procesamiento evento por evento, tales como filtros probabilísticos y *Spiking Neural Networks* (SNN). Este tipo de redes neuromórficas están inspiradas en la biología, y contienen en su estructura, información construida en base a eventos pasados o a información adicional provista por mecanismos de entrenamiento, que se fusiona de manera asincrónica con los eventos entrantes para producir un resultado. Este tipo de redes neuronales son extremadamente eficientes en término de cómputo y consumo energético, pero muy complejas de entrenar, y aún no están a la altura de los resultados que pueden obtenerse con redes neuronales convencionales.

Los eventos son procesados y transformados en representaciones alternativas que facilitan la extracción de información significativa ("características") para resolver una tarea determinada. Existen distintos tipos de representaciones dependiendo de la naturaleza del procesamiento. Varias de ellas surgen de la necesidad de agregar la poca información contenida por eventos individuales en ausencia de conocimiento adicional. Algunas representaciones son simples transformaciones

de datos hechas a mano, mientras que otras son más elaboradas. A continuación se describen algunos de los mecanismos de representación más utilizados en la literatura.

### Imagen o histograma en dos dimensiones (2D)

Los eventos dentro de un mismo vecindario espacio-temporal pueden convertirse en una representación matricial dispersa, contando eventos o acumulando la polaridad píxel a píxel, que puede ser utilizada por algoritmos de visión computacional tradicionales. Por otro lado, la desventaja que estos métodos tienen, es que cuantizan los eventos en tiempo, resultando en una pérdida de la información temporal que éstos contienen, y las imágenes resultantes son altamente sensibles a la cantidad de eventos utilizados. Sin embargo, es uno de los mecanismos de representación más utilizado en la literatura ya que permiten convertir un flujo de eventos en una representación matricial que contiene información espacial de los contornos de una escena, los cuales representan la mayor cantidad de información en una imagen, y son estructuras de datos compatibles con algoritmos de visión computacional tradicionales.

### Superficie temporal

Este mecanismo de representación es una estructura matricial donde cada píxel almacena un solo valor de tiempo, como puede ser la marca de tiempo del último evento en ese píxel [25]. De esta manera, los eventos son convertidos en una imagen cuya "intensidad" es una función de la historia del movimiento de la escena, donde los valores más elevados corresponden con movimiento más reciente. Esta representación pone de manifiesto de manera explícita la riqueza de la información temporal contenida en los eventos, y puede actualizarse de manera asincrónica. Comparada con otros mecanismos de representación matricial, la superficie temporal permite comprimir de manera significativa la información, ya que sólo conserva una marca de tiempo por píxel, y por lo tanto su efectividad disminuye significativamente en escenas que contienen gran cantidad de información en cuanto a textura, ya que los píxeles reciben eventos de manera más frecuente.

### Grilla de vóxeles

Es un histograma espacio-temporal de eventos en forma de matriz tridimensional, donde cada vóxel representa un píxel particular en un determinado intervalo de tiempo. Esta representación preserva mejor la información temporal de los eventos con respecto al histograma 2D, ya que evita colapsarlos en una matriz bidimensional. Ambos esquemas cuantizan las marcas de tiempo de los eventos, pero ésta provee una mayor resolución temporal.

### Conjunto de puntos en tres dimensiones (3D)

Los eventos en un vecindario espacio-temporal son tratados como puntos en un espacio 3D, donde $\epsilon = (x_k, y_k, t_k) \in \Re^3$, por lo que la dimensión temporal se convierte en una dimensión geométrica. Es una representación dispersa, por lo que es utilizada en mecanismos de procesamiento basados en puntos como el ajuste de planos [26] o redes neuronales de tipo PointNet [27].

Los sistemas de procesamiento de eventos se componen de varias etapas: pre-procesamiento (adaptación de entrada), procesamiento central (extracción y análisis de características) y post-procesamiento (generación de salida). Los métodos empleados para procesar eventos dependen tanto de la elección del mecanismo de representación como de la plataforma de hardware disponible, y estos tres factores están interrelacionados. Por ejemplo, es común utilizar representaciones densas y diseñar algoritmos optimizados para ejecutarse en procesadores estándar, como CPU[1] o GPU[2]. Paralelamente, procesar eventos individualmente mediante SNN[3] implementadas en hardware neuromórfico resulta natural en la búsqueda de soluciones más eficientes y de baja latencia. No obstante, también existen enfoques intermedios, como las redes convolucionales por eventos, que aprovechan la dispersión espacial de las representaciones matriciales sin comprometer significativamente la latencia o el consumo energético.

[1] *Central Processing Unit*
[2] *Graphic Processing Unit*
[3] *Spiking Neural Networks*

## 2.2. Sensores de imágenes por eventos

A diferencia de los sensores tradicionales que producen una imagen a una tasa de velocidad fija, los sensores basados en eventos capturan de manera asincrónica los cambios de intensidad lumínica por píxel, y generan un flujo de eventos asincrónico que codifica tiempo, ubicación espacial y signo de estos cambios. Este principio de operación en consecuencia le aporta a este tipo de sensores características muy interesantes como una alta resolución temporal (en el orden de los $\mu$s) [8], un alto rango dinámico (alrededor de los 140 dB, mientras que para los sensores tradicionales el rango dinámico típicamente ronda los 60 dB), muy bajo consumo de energía, y un gran ancho de banda por pixel (en el orden de los kHz), lo que resulta en una reducción del desenfoque por movimiento. Todo esto los hace especialmente atractivos para aplicaciones que requieran baja latencia, alta velocidad y alto rango dinámico, tales como sistemas para evitar obstáculos en drones [13], detección de peatones en sistemas de visión para vehículos autónomos [28], o aplicaciones de muy bajo consumo, como detección y seguimiento de gestos en sistemas de realidad aumentada [18].

La cámara de eventos DVS[4] [8] (del inglés *Dynamic Vision Sensor*) tuvo su origen en un diseño de retina de silicio basado en *frames*, donde el fotorreceptor de tiempo continuo estaba acoplado capacitivamente a un circuito de lectura que se reiniciaba cada vez que se muestreaba el píxel [29]. Aunque muchas aplicaciones se pueden resolver solo procesando los eventos DVS (es decir, los cambios de intensidad lumínica), se hizo evidente que algunas también requieren algún tipo de salida estática (es decir, intensidad lumínica "absoluta"). Para abordar esta limitación, ha habido varios desarrollos de cámaras que emiten de forma simultánea información dinámica y estática [10].

El sensor ATIS[5] [9] tiene píxeles que contienen un subpíxel DVS llamado detección de cambios, que activa otro subpíxel para leer la intensidad absoluta. El disparador reinicia un condensador a un nivel de tensión alto. La carga acumulada se va descargando de este condensador mediante otro fotodiodo. Cuanto más brillante sea la luz, más rápido se descarga el condensador. La lectura de la

[4] *Dynamic Vision Sensor*
[5] *Asynchronous Time Based Image Sensor* (Sensor de Imagen Asíncrono Basado en Tiempo)

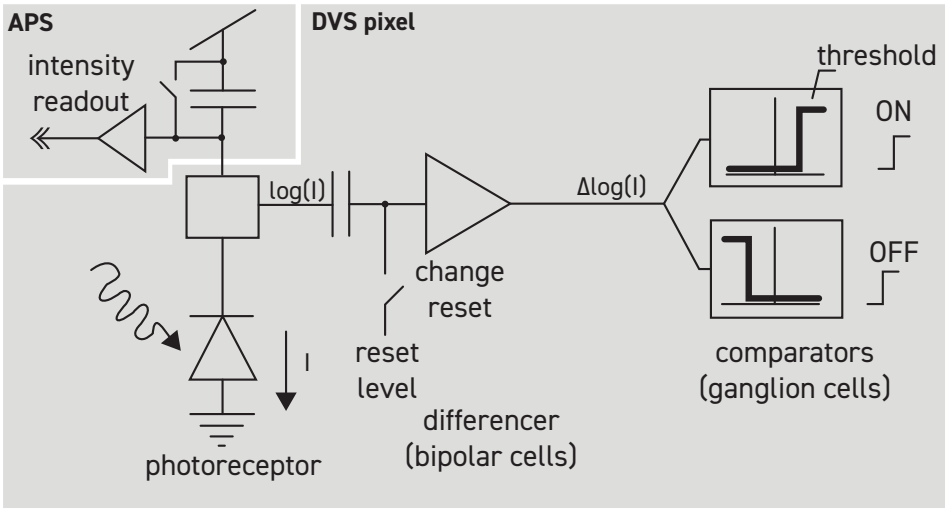



Figura 2.1: Esquemático simplificado de un pixel DAVIS [1].

intensidad en ATIS transmite dos eventos adicionales que codifican el tiempo entre el cruce de dos voltajes umbral, como en [30]. De este modo, solo los píxeles que cambian proporcionan sus nuevos valores de intensidad. Cuanto más brillante sea la iluminación, más corto será el tiempo entre estos dos eventos. El ATIS logra un amplio rango dinámico estático ($> 120$ dB). Sin embargo, tiene la desventaja de que los píxeles son al menos el doble de grandes que los píxeles de la cámara DVS. Además, en escenas oscuras, el tiempo entre los dos eventos de intensidad puede ser largo y la lectura de intensidad puede ser interrumpida por nuevos eventos.

El sensor DAVIS[6] [10] combina en el mismo píxel un sensor de píxel activo convencional (APS) con un sensor DVS compartiendo el mismo fotodiodo (Fig. 2.1), lo que le da una ventaja que sobre el ATIS ya que posee un tamaño de píxel mucho más pequeño, y el circuito de lectura solamente añade un 5 % al área del píxel DVS. Sin embargo, la lectura APS tiene un rango dinámico limitado (55 dB), y al igual que una cámara estándar, es redundante si los píxeles no cambian. Dado que tanto los diseños de píxeles de la ATIS y la DAVIS incorporan circuitería de píxeles de tipo DVS, a menudo se utiliza este término para referirse a la salida de eventos de polaridad binaria, o al circuito, independientemente de si proviene de una cámara DVS, ATIS o DAVIS.

Otro sensor desarrollado en los últimos años bajo el paradigma de eventos pe-

[6] *Dynamic and Active Pixel Vision Sensor* (Sensor de Visión Dinámico y de Píxel Activo)

ro con una tecnología de píxel distinta, es el *Flexible Readout Integration Sensor* (FRIS) [2]. Este sensor implementa un esquema de muestreo basado en eventos que permite solamente la salida de los píxeles que se encuentran dentro de un rango programable de tasas de flujo de fotones. En este sensor, cada píxel posee un conversor analógico digital (ADC[7]) sigma-delta de sobremuestreo, lo que permite la cuantización de señales de hasta 26bits y permite la corrección de errores no-uniforme (NUC[8]) de ganancia y offset antes de la lectura de los datos directamente en el sensor gracias a registros en cada píxel que almacenan estos parámetros (Fig. 2.2).

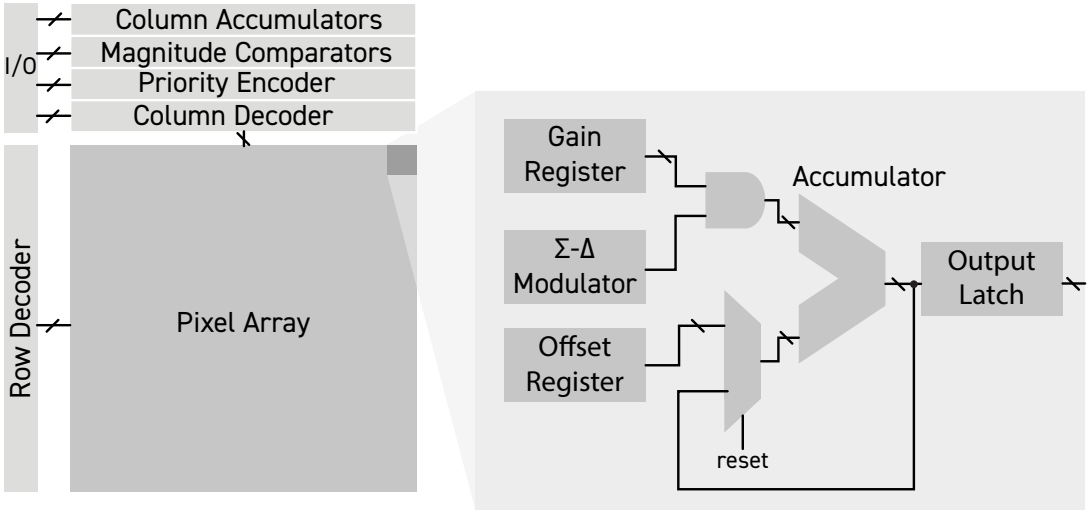



Figura 2.2: Diagrama en bloques de la arquitectura del sensor FRIS [2].

## 2.3. Redes convolucionales por eventos

Las redes neuronales convolucionales (CNN[9]) constituyen uno de los tipos de redes neuronales profundas (DNN), y son ampliamente utilizadas en procesamiento de imagen y video. Fueron introducidas en su forma moderna por Yann LeCun a finales de los años 80 donde publicó la primer red neuronal convolucional entrenada con el algoritmo de *backpropagation* (del inglés propagación hacia atrás), denominada "LeNet" que era capaz de reconocer códigos postales escritos a mano [31]. Este modelo de red neuronal pasó por diversas iteraciones de modi-

[7] *Analog to Digital Converter*
[8] *Non-Uniform Correction*
[9] *Convolucional Neural Network*

ficaciones y mejoras hasta que en 1998 el modelo LeNet-5 (Fig. 2.3) alcanzó una tasa de error menor al 0,95 % en el dataset de dígitos escritos a mano MNIST [3].

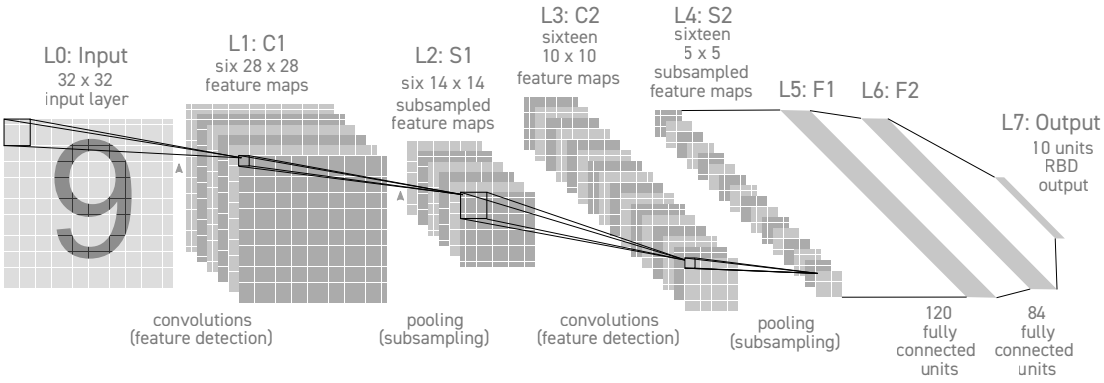



Figura 2.3: Arquitectura LeNet-5 [3].

La Fig. 2.3 muestra la arquitectura básica del modelo LeNet-5. La capa de entrada $L_0$ cumple la función de retina, recibiendo imágenes de caracteres escritos a mano (Dataset MNIST) los cuales se encuentran centrados y normalizados en tamaño. La siguiente capa $L_1$ está compuesta de varios **canales** o **mapas de características**, cuya función es extraer características simples de la imagen tales como bordes, esquinas, etc. En la práctica, cada mapa de características es una matriz cuadrada de pesos idénticos. Los pesos dentro de un mapa de características deben ser idénticos para que puedan detectar la misma característica local en la imagen de entrada. Los pesos entre los mapas de características son diferentes para que puedan detectar diferentes características locales. Cada unidad en un mapa de características tiene un **campo receptivo**. Este está conformado por una sub-región $n \times n$ o vecindad de la imagen de entrada, que puede ser percibida por una unidad en el mapa de características.

El proceso de multiplicar y acumular las características de un campo receptivo deslizante por un set de pesos, también llamado **kernel** o **filtro** dado que su función es la de extraer o resaltar ciertas características de una imagen, es lo que se conoce como **convolución** (técnicamente, es equivalente a la operación de correlación cruzada cuando el kernel es simétrico) (Fig. 2.4). El paso del deslizamiento es lo que se conoce como **stride**. Los kernels o filtros son aprendidos durante el entrenamiento de la red neuronal convolucional.

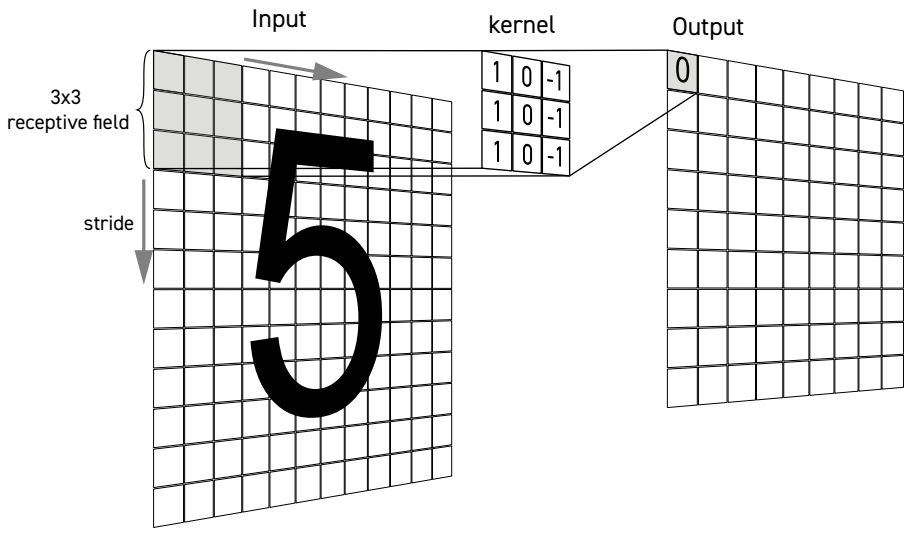



Figura 2.4: Operación de convolución estándar.

Una vez finalizada la operación de convolución, se han extraído las características de la imagen de entrada. No obstante, el mero hecho de que un conjunto de características esté presente no es suficiente para identificar a qué imagen corresponde. Es crucial conocer también su disposición relativa. Por ejemplo, si se detecta una "línea curva horizontal" en la parte inferior central, una "línea curva vertical" en el lateral derecho, una "línea recta vertical" en la esquina superior izquierda y una "línea recta horizontal" en el centro superior, se puede inferir que el número en cuestión es un "5". Esto se vuelve especialmente importante considerando que en imágenes del mundo real, como los números escritos a mano, existe una gran variabilidad en sus formas. Ninguna persona escribe los números exactamente igual. Por ello, se necesita que la red sea menos dependiente de la posición absoluta de las características y más sensible a su disposición relativa: aunque los números 5 escritos a mano varíen, la curva suele estar en la parte inferior y la línea recta en la superior. Este concepto se denomina invariancia a la translación local [32], y una forma de lograrlo es reduciendo la resolución espacial de la imagen, algo que se consigue mediante el **subsampling o pooling** (Fig. 2.5).

Existen diversas formas de submuestrear una imagen. En LeNet-5, esta operación realiza un promedio local sobre una sección del mapa de características, lo que reduce efectivamente la resolución del mapa en su conjunto y disminuye la sensibilidad de la red a desplazamientos y distorsiones en la imagen de entrada.

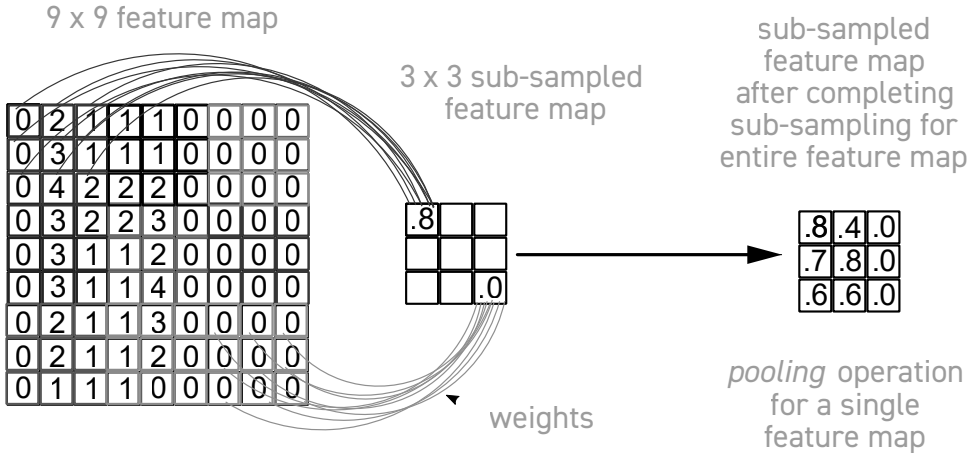



Figura 2.5: Operación de submuestreo o pooling.

Existe también otro tipo de operación de pooling muy utilizada, que en lugar de computar el promedio, toma el valor máximo. La capa de pooling tiene tantos mapas de características de salida, como mapas de características a la entrada. La mecánica de cómputo es muy similar a la capa de convolución, pero en este caso, el valor de **stride** es tal que no hay superposición entre los saltos de campo receptivo. Cada característica de entrada contribuye a una única característica de salida.

Una vez realizada toda la secuencia de capas de convolución y pooling, la red implementa una capa **fully connected** tradicional, como las que se implementan en el perceptron multicapa. La primera, F1 según la notación de la Fig. 2.3, tiene la función de "aplanar" (en inglés se conoce como *flatten*) las características provenientes de la capa de pooling C2. La siguiente capa oculta "comprime" la salida aún más, en un vector de 84 elementos. Finalmente, la capa de salida implementa una función de base radial euclideana con 10 neuronas para realizar la clasificación de números entre 0 y 9.

Este tipo de redes convolucionales como la recién descrita LeNet-5, fueron originalmente diseñadas para trabajar con imágenes densas, es decir, matrices completas de datos donde cada píxel contiene información relevante para la tarea de procesamiento. En este tipo de datos, la convolución es aplicada a todas las regiones de la imagen, ya que se espera que cada parte de la imagen contenga alguna característica significativa que la red pueda aprender. Sin embargo, en datos de naturaleza dispersa, como los encontrados en representaciones de eventos,

la mayoría de los valores pueden ser ceros, lo que genera ineficiencias cuando se aplican convoluciones estándar.

Por este motivo, en 2014, Graham introdujo el concepto de redes convolucionales espacialmente dispersas SCNN[10] [33], donde se propone un mecanismo de representación para las capas convolucionales basado en el uso de dos matrices por capa. La primera, llamada **matriz de características**, es una lista de vectores fila: uno que representa el estado de *ground*, que, en lugar de ser cero, puede estar determinado por el valor del **bias** (como suele ocurrir en redes neuronales), y un vector fila adicional para cada **sitio activo**, el cual contiene las características de dicho sitio. La segunda matriz, llamada **matriz de punteros**, cuyo tamaño coincide con el tamaño espacial de la capa convolucional, almacena el índice de la fila correspondiente en la matriz de características.

Bajo este enfoque, Graham implementa una red neuronal convolucional dispersa para el reconocimiento de caracteres chinos escritos a mano. Esto se debe a que, cuando un carácter se renderiza en alta resolución en una matriz bidimensional, adopta una forma dispersa, lo que permite aprovechar este esquema de representación de manera eficiente.

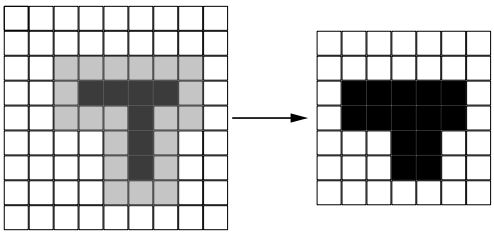


Figura 2.6: Convolución dispersa con un kernel de $2 \times 2$, cuya imagen de entrada posee 7 sitios activos.

Más tarde, en [4] se profundiza y formaliza matemáticamente este concepto, extendiéndose también a redes neuronales convolucionales 3D, y construye varios modelos para reconocimiento de objetos. La Fig. 2.6 ilustra el concepto de convolución dispersa, con un ejemplo de kernel de $2 \times 2$. El filtro convolucional necesita calcularse para cada paso de convolución que contenga al menos 1 sitio activo, ilustrado mediante la región sombreada. La matriz resultado (a la derecha) genera 14 sitios activos, como resultado del efecto de dilatación que se analiza en

[10] *Sparse Convolucional Neural Network*

dicho trabajo. En este caso, se ponen a prueba por primera vez estas redes en secuencias de video para reconocimiento de acciones humanas, donde a partir de videos capturados mediante cámaras convencionales (Fig. 2.7), se recrean videos cuyos cuadros se obtienen calculando la diferencia entre dos cuadros consecutivos del video, obteniendo así una representación altamente dispersa como la que se encuentra en datasets de eventos.


Figura 2.7: Imagen de los dos datasets de video utilizados en [4] y la diferencia entre dos cuadros consecutivos. La imagen (a) corresponde con el dataset [5], mientras que la imagen (b) corresponde con el dataset [6].

Como punto de partida en el desarrollo de esta tesis, se propone una técnica de emulación de eventos basada en el sensor FRIS[11] [2], la cual fue publicada en [20]. Este enfoque toma la diferencia entre cuadros consecutivos, similar a la metodología utilizada por Graham, pero con la particularidad de que en este trabajo los píxeles se "activan" según dos umbrales denominados *umbral incremental* y *umbral decremental*. El objetivo de este método es abordar el problema del reconocimiento de gestos de manos, orientado a una aplicación portátil de detección basada en sombras utilizando redes neuronales convolucionales. Este enfoque permite la adaptación a diferentes entornos de prueba, con condiciones de iluminación variables y fondos cambiantes.

Como prueba de concepto, se grabó un dataset de videos de sombras de manos (Fig. 2.8), que incluye tres gestos: paloma, conejo y perro, utilizando diversos fondos para evaluar el rendimiento de esta técnica en comparación con otros mecanismos, como el algoritmo de detección de bordes de Canny [34].

Se implementó un modelo de red neuronal convolucional diseñado para este propósito (Tabla 2.1),el cual se entrenó en tres versiones distintas: utilizando imágenes en escala de grises, datos basados en eventos, y la versión con detección

[11] *Flexible Readout Integration Sensor*

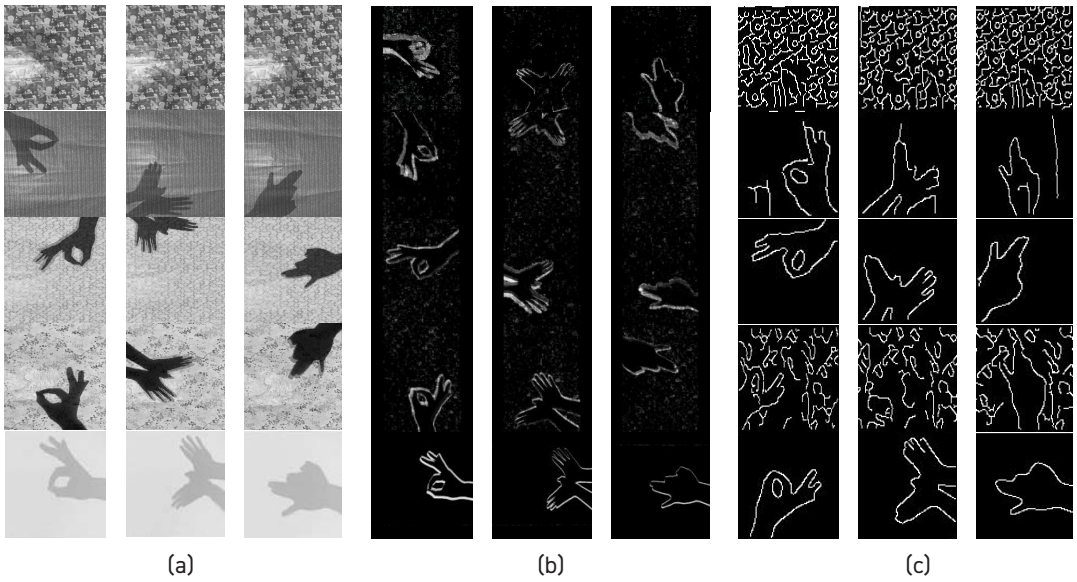



Figura 2.8: Dataset de sombras de manos donde las columnas representan los distintos gestos, y las filas los distintos fondos, (a) versión escala de grises, (b) versión por eventos, (c) versión algoritmo de Canny.

de bordes mediante el algoritmo de Canny. Para evaluar el desempeño, las redes fueron entrenadas excluyendo uno de los cuatro tipos de fondos, con el fin de validar los resultados con datos inéditos para la red. Este proceso se repitió para cada uno de los fondos, y se analizaron las estadísticas de desempeño (Fig. 2.9).

Se observó que el impacto de la interferencia del fondo y los cambios en condiciones de iluminación se reducen drásticamente en la red entrenada con datos del emulador FRIS. En el caso de la red entrenada con imágenes en escala de grises, se observó un desempeño poco adecuado en los casos donde las condiciones de fondo disminuyen el contraste de las sombras. Finalmente, la versión con detección de bordes de Canny mostró un buen comportamiento en fondos lisos, pero su rendimiento disminuyó drásticamente en fondos con objetos y patrones.

Esta red antes descrita, es compatible en su versión de eventos, con las técnicas implementadas por Graham. Sin embargo, sufre de la pérdida de dispersión por los efectos de dilatación de los sitios activos en las capas convolucionales (Fig. 2.6). En [35], Graham propone una variante del algoritmo de convolución para datos dispersos que soluciona el problema de dilatación. Sin embargo, éste algoritmo ya no es equivalente a la convolución tradicional. La convolución dispersa en subvariedades *Submanifold Sparse Convolution* (SSC[12]), consiste en una

[12] *Submanifold Sparse Convolutions*

Tabla 2.1: Modelo de red neuronal convolucional propuesto.

| **Capa** | **Tipo** | **Tamaño de entrada** | **Kernels** |
|---|---|---|---|
| 1 | Convolución | $128 \times 128 \times 1$ | $3 \times 3 \times 1$ |
| 2 | MaxPool | $126 \times 126 \times 16$ | $2 \times 2$ |
| 3 | Convolución | $63 \times 63 \times 16$ | $5 \times 5 \times 16$ |
| 4 | MaxPool | $30 \times 30 \times 16$ | $2 \times 2$ |
| 5 | Fully Connected | $15 \times 15 \times 16$ | 128 |
| 6 | Fully Connected | 128 | 64 |
| 7 | Fully Connected | 64 | 32 |
| 8 | Fully Connected | 32 | 3 |

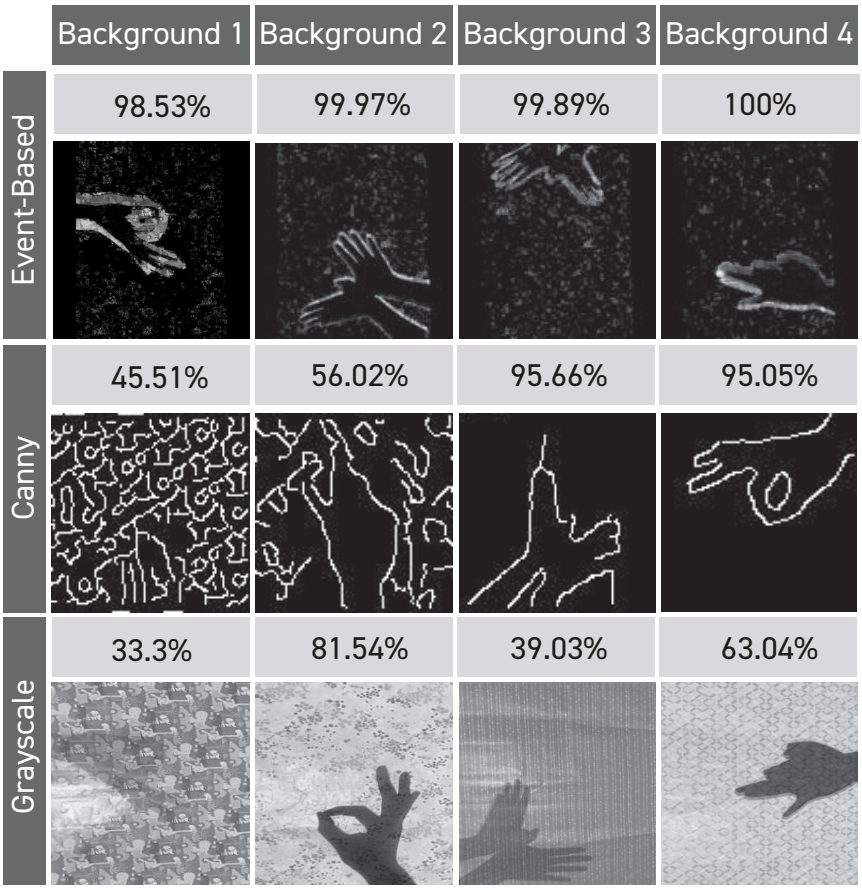



Figura 2.9: Resultados obtenidos en testeo con fondo nunca antes visto.

convolución dispersa como la descrita anteriormente, con la diferencia de que un sitio activo a la salida de la capa existe si y solo si existe también a la entrada (Fig. 2.10). La operación se efectúa de la siguiente manera. Primero, se agrega **padding** de ceros a la entrada, de manera que la salida tenga las mismas dimensiones que la entrada original. Luego, se restringe la salida, de manera que un sitio a la salida puede estar activo, si y solo si el sitio central en su campo receptivo está activo.

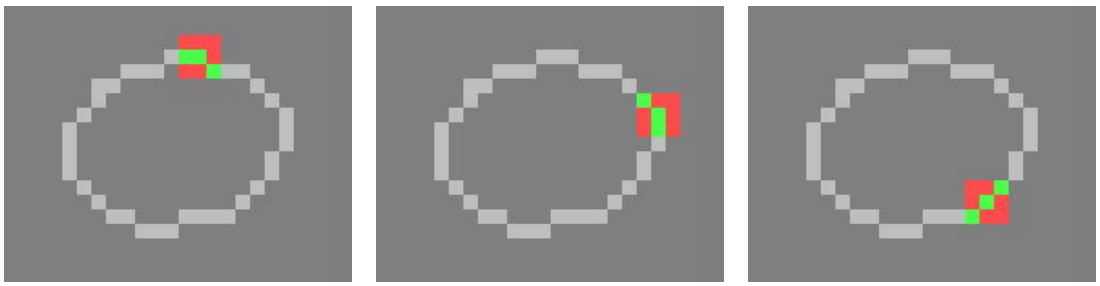


Figura 2.10: SSC cuyo campo receptivo se encuentra centrado en tres ubicaciones espaciales distintas. Las ubicaciones activas se muestran en verde, mientras que las rojas son ignoradas.

Este algoritmo resuelve el problema de la dilatación espacial, sin embargo se pierde la resolución temporal que aportan los sistemas de visión por eventos. Por tanto, en [36] Messikommer propone una modificación del algoritmo original SSC de Graham, que permite actualizar la convolución evento a evento, siendo completamente equivalente (si se colapsaran los eventos en el eje temporal) al algoritmo original.

## 2.4. Hardware dedicado para procesamiento de redes por eventos

Desde la aparición de las redes neuronales profundas (DNN), el volumen y la complejidad de las operaciones matemáticas necesarias para su entrenamiento y ejecución han crecido considerablemente. Uno de los desafíos que imponen las CNNs modernas es la elevada complejidad computacional y el gran número de parámetros involucrados, lo que implica que la inferencia con CNNs demande una considerable cantidad de energía y sea relativamente lenta cuando se ejecuta en plataformas de cómputo de propósito general. Esto se debe a la necesidad de

grandes cantidades de elementos de procesamiento en paralelo, que respondan a las demandas de latencia en aplicaciones de tiempo real, y en consecuencia se necesitan jerarquías de memoria sofisticadas para mover y abastecer a los elementos de procesamiento con las grandes cantidades de datos y parámetros que el cómputo de redes convolucionales conlleva, las cuales incrementan el costo energético. Por ello, resulta esencial desarrollar arquitecturas de hardware especializadas que optimicen el procesamiento de estos algoritmos, permitiendo su implementación eficiente en aplicaciones del mundo real, especialmente en dispositivos inteligentes de bajo consumo energético.

Las redes convolucionales dispersas reducen considerablemente los requerimientos de cómputo y memoria presentes en las redes convolucionales tradicionales, sin embargo presentan desafíos desde el punto de vista de hardware tales como la implementación de esquemas de representación de matrices dispersas que ocupen la menor cantidad de memoria posible, y complejos mecanismos de cómputo para evitar el cómputo con características nulas y ahorrar energía. En [37], se implementa un mecanismo de representación basado en una máscara binaria que reduce significativamente la cantidad de memoria necesaria para procesar redes convolucionales dispersas respecto a otros aceleradores convencionales, sin aumentar demasiado la complejidad de la lógica de cómputo.

Existen otro tipo de aceleradores eficientes desde el punto de vista energético, que trabajan bajo el paradigma de *Compute in Memory* (CIM[13]) tales como [38, 39, 40], que utilizan memorias CAM para el procesamiento de redes neuronales convolucionales, prescindiendo de multiplicadores y sumadores. Esto se logra utilizando las memorias CAM para almacenar las tablas de verdad de un sumador de $n$ bits, y así reemplazar las operaciones de multiplicación y acumulación, por múltiples operaciones de búsqueda. Sin embargo, las memorias CAM pueden ser aprovechadas de otras formas para el cómputo de redes convolucionales dispersas. En [41], se utilizan como recurso para almacenar vectores dispersos de manera eficiente en operaciones de tipo vector-vector, y vector-matriz, explotando la cualidad de búsqueda en un solo ciclo de reloj que este tipo de memorias aporta, sin embargo no se llega a expandir este concepto a redes neuronales.

[13] *Compute in Memory*

Algunas de las arquitecturas de hardware más eficientes para procesamiento de redes convolucionales, son arquitecturas neuromórficas para procesamiento de redes por eventos, con un esquema de memoria autocontenida. NeuronFlow [42], es una arquitectura de hardware híbrida que combina el direccionamiento por coordenadas similar al protocolo AER presente en los sensores y procesadores neuromórficos, pero en lugar de *spikes*, trabaja con valores numéricos de 16 bits, como en las arquitecturas de tipo *dataflow* tradicionales, denominando este esquema *Address Value Representation* (AVR[14]).

Grandes empresas de semiconductores como IBM e Intel han desarrollado sistemas en chip neuromórficos basados en eventos. IBM desarrolló TrueNorth [43], un SoC digital de 1M de neuronas *spiking* programables dispuestas en una NoC[15] de $64 \times 64$ núcleos, donde cada núcleo consta de 256 neuronas. La conectividad entre neuronas sigue un esquema por bloques: cada neurona puede conectarse a una línea de entrada de cualquier núcleo en el sistema y, desde allí, a cualquier neurona de ese núcleo a través de sus sinapsis locales. Soporta redes neuronales convolucionales bajo ciertas condiciones a través de una técnica de mapeo específica [44], que ha permitido el desarrollo de algunas aplicaciones como reconocimiento de gestos [45]. Por su parte, Intel desarrolló un SoC neuromórfico denominado Loihi [46], el cual se encuentra en su segunda versión. Esta arquitectura implementa núcleos de procesamiento asincrónicos capaces de procesar SNN con altos niveles de eficiencia, y soporte de entrenamiento en chip, lo que sirvió como catalizador de posteriores avances en el área de desarrollo de redes neuronales SNN. En [47], se propone una arquitectura de hardware basada en una NoC, que implementa una técnica de compresión de sinapsis que permite reducir drásticamente los requerimientos de memoria, y es capaz de ejecutar tanto redes convolucionales dispersas tradicionales, como SNN mediante modelos de neuronas de tipo integración y disparo con fuga *leaky-integrate and fire* (LIF[16]).

[14] *Address Value Representation*
[15] *Network on Chip*
[16] *Leaky Integrate-and-Fire*

## 2.5. Conclusiones

En este capítulo se han abordado los conceptos fundamentales necesarios para comprender el paradigma de procesamiento de eventos, las técnicas de representación y los avances en hardware dedicados a redes neuronales convolucionales por eventos. Se destacaron los principales mecanismos de procesamiento y las diversas representaciones espaciales y temporales que permiten extraer información útil de los eventos capturados por estos sensores neuromórficos. Asimismo, se discutieron las características y limitaciones de los sensores de imágenes por eventos, y se describieron técnicas y mecanismos mediante los cuales se pueden optimizar las redes neuronales convolucionales para extraer información significativa de estos sensores, sin realizar cómputos innecesarios. Finalmente, se abordaron los desafíos y soluciones en la implementación de hardware eficiente, subrayando la importancia de desarrollar tecnologías optimizadas para el procesamiento de eventos en tiempo real, con bajo consumo energético y alta eficiencia computacional.

# Capítulo 3

# Bloques de cómputo utilizando memorias CAM

Existen diversas arquitecturas de hardware capaces de trabajar con matrices dispersas, cada una con sus propias particularidades. Sin embargo, estas arquitecturas procesan normalmente matrices dispersas estáticas que requieren complejos mecanismos de codificación y no resultan prácticas en situaciones donde deben actualizarse dinámicamente. Dado que los eventos vienen codificados por su coordenada espacial, resulta natural trabajar con matrices dispersas codificadas en formato de lista de coordenadas y sus características asociadas. Este mecanismo, si bien resulta ideal, presenta un desafío en términos de tiempo y energía. Cada vez que llega un nuevo evento, debe buscarse en la matriz dispersa almacenada en memoria, para determinar si ya existe una entrada en la misma coordenada espacial. Este proceso de búsqueda es costoso, ya que debe leerse todo el arreglo de memoria para determinar si un evento ya existe o no. En este sentido, las memorias direccionables por contenido (CAM por sus siglas en inglés), son arreglos de memoria especiales que cuentan con un puerto adicional de búsqueda que permite determinar si un dato ya existe en la memoria en un solo ciclo de reloj, lo que resulta ideal para esta aplicación.

Este capítulo se centra en la representación de matrices dispersas utilizando memorias CAM, y en cómo estas pueden ser utilizadas para implementar algoritmos de cómputo eficientes para procesar flujos de eventos. En la Sección 3.1 se presentan los algoritmos de cómputo para procesar flujos de eventos utilizan-

do memorias CAM, y en la Sección 3.2 se discuten distintas arquitecturas de sistemas que implementan estos algoritmos.

## 3.1. Algoritmos de cómputo

Un flujo de eventos dentro de un intervalo de tiempo específico puede ser representado como un conjunto $\Omega = \{\epsilon_n\}_{n=1}^{N} \in \mathbb{R}^{N\times 4}$, donde cada evento $\epsilon_n = (x_n, y_n, t_n, p_n)$ consiste en coordenadas espaciales $(x_n, y_n)$, una marca de tiempo $t_n$ (que aumenta monótonamente con respecto al índice $n$) y una polaridad $p_n$ que denota el signo del cambio de intensidad relativo con respecto al evento más reciente en la misma ubicación espacial.

Para manejar la dispersión inherente en los datos basados en eventos, se utiliza un conjunto $\mathbf{S}$ de $N$ elementos únicos, donde $\mathbf{S} = \{s_n\}_{n=1}^{N}$. Cada elemento $s_n = (xs_n, ys_n, as_n)$ comprende una coordenada 2D única $(xs_n, ys_n) \subset \mathbb{Z}_+^2$ y un valor de intensidad no nulo $as_n \in \mathbb{R}^{d_i}$, donde $d_i$ representa la dimensión del vector de características del conjunto de valores de intensidad de entrada. Estos elementos se definen como:

$$\mathbf{S} = \{(xs_n, ys_n, as_n) \in \mathbb{Z}_+ \times \mathbb{Z}_+ \times \mathbb{R}^{d_i} \mid 0 \leq xs_n < H_i, 0 \leq ys_n < W_i\} \quad (3.1)$$

donde $H_i$ y $W_i$ denotan la altura y el ancho de la matriz dispersa de entrada, respectivamente. Las coordenadas se almacenan en una memoria CAM de entrada, mientras que las intensidades no nulas asociadas $as_n$ se almacenan en una memoria RAM[1] asociada.

De manera similar, la salida de la capa de la red neuronal se representa como un conjunto $\mathbf{Q}$ de $M$ elementos únicos, donde $\mathbf{Q} = \{q_m\}_{m=1}^{M}$. Cada elemento $q_m = (xq_m, yq_m, bq_m)$ tiene una intensidad no nula $bq_m \in \mathbb{R}^{d_o}$, donde $d_o$ denota la dimensión del vector de características del conjunto de valores de intensidad de salida, e incluye una coordenada 2D única $(xq_m, yq_m) \subset \mathbb{Z}_+^2$. Estos elementos se definen como:

[1] *Random Access Memory* (Memoria de Acceso Aleatorio)

$$\mathbf{Q} = \{(xq_m, yq_m, bq_m) \in \mathbb{Z}_+ \times \mathbb{Z}_+ \times \mathbb{R}^{d_o} \mid 0 \leq xq_m < H_o, 0 \leq yq_m < W_o\} \quad (3.2)$$

donde $H_o$ y $W_o$ denotan la altura y el ancho de la matriz dispersa de salida, respectivamente. Estas coordenadas de salida y sus valores de intensidad asociados se almacenan en una CAM de salida y su RAM asociada.

Para integrar los eventos $\Omega$ generados por un sensor basado en eventos, se emplea una memoria CAM junto con su RAM asociada para almacenar las coordenadas de los píxeles $(xs_n, ys_n)$ y su contenido $as_n$ en un conjunto $\mathbf{S}$. Al llegar un nuevo evento, su coordenada se compara con las coordenadas existentes $(xs_n, ys_n) \in \mathbf{S}$ ya almacenadas en la CAM. Si la coordenada coincide con alguna ya registrada, se actualiza su valor mediante la función $as_n|t = f(\epsilon, as_n|t-1)$, que utiliza información del evento como la polaridad $p_n$ y la marca de tiempo $t_n$, conforme a los mecanismos de representación matricial descritos en la Sección 2.1. En caso de que la coordenada no se encuentre en la CAM, esta se inicializa.

Tras un período de tiempo predeterminado, el conjunto $\mathbf{S}$ se congela y se habilita un nuevo conjunto CAM-RAM para continuar integrando datos. Este proceso es gestionado por el *Sistema de Adquisición de Eventos* (Fig. 3.1), que recibe los eventos, evalúa la función de actualización $f$ y la aplica al evento de entrada $\epsilon$, utilizando su coordenada $(xs_n, ys_n)$ y el valor anterior $as_n|_{t-1}$.

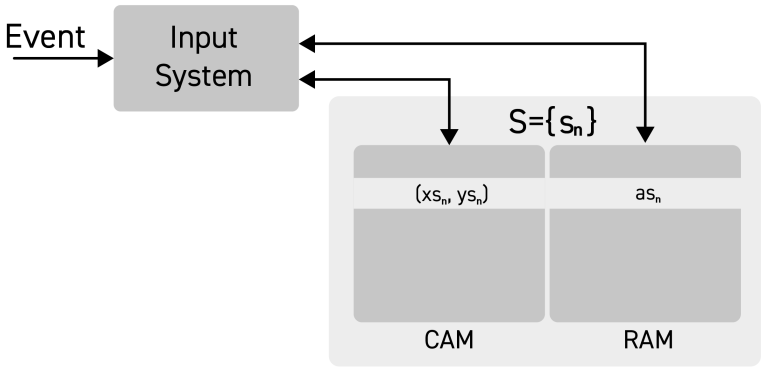



Figura 3.1: Integración de eventos con un conjunto CAM-RAM.

Una vez finalizado el proceso de adquisición, los eventos se encuentran completamente integrados en el conjunto de entrada $\mathbf{S}$, y dado un conjunto de salida $\mathbf{Q} = \varnothing$ inicialmente vacío, y un conjunto de $d_o$ *kernels* de tamaño $k^2 \times d_i$, el

cálculo de una capa convolucional se realiza de la siguiente manera. Se lee la primera entrada $(xs_0, ys_0)$ de $S$, y el conjunto $\Phi \subset Q$ de todos los elementos $q_m$ cuyas coordenadas de salida se ven afectadas por los elementos $(xs_0, ys_0)$ (Fig. 3.2) se define como:

$$\Phi = \{(\hat{xq}_m, \hat{yq}_m) \in \mathbb{Z}_+^2 \mid (\hat{xq}_m, \hat{yq}_m) = \frac{(xs_0 + p, ys_0 + p) - (i, j)}{\sigma}, \\ 0 \leq i < k,\ 0 \leq j < k\} \quad (3.3)$$

donde $p \in \mathbb{Z}_+$ es el *padding*, $\sigma \in \mathbb{Z}_{>0}$ es el *stride* (paso) de la capa convolucional, y $k \subset \mathbb{Z}_{>0}$ representa la dimensión del *kernel*. Esta ecuación solo está definida para valores de $(i, j) \subset \mathbb{Z}_+^2$ tales que el cociente en el miembro derecho de la Ec. 3.3 garantice que $(\hat{xq}_m, \hat{yq}_m) \in \mathbb{Z}_+^2$.

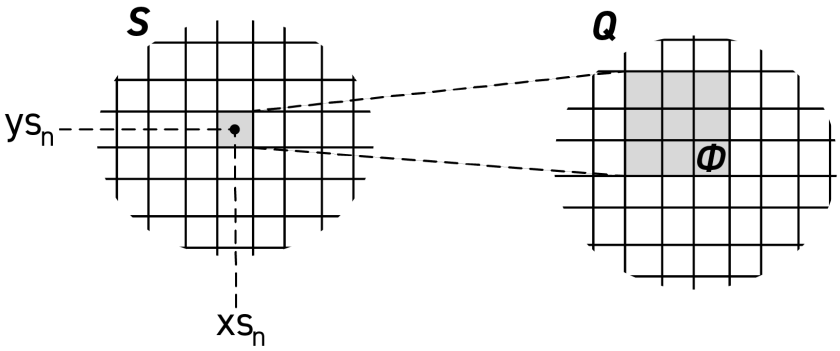



Figura 3.2: Coordenada de entrada $(xs_n, ys_n) \in S$ y el subconjunto de coordenadas de salida $\Phi \subset Q$ que se verán afectadas.

A continuación, se selecciona el primer elemento del conjunto $\Phi$ y se busca en $\mathbf{Q}$. Si está presente, indica un procesamiento previo, y se procede al siguiente elemento. De lo contrario, se calcula el conjunto de coordenadas de entrada $\Upsilon \subset S$ que contribuyen a esta coordenada de salida $(xq_m, yq_m)$ (Fig. 3.3):

$$\Upsilon = \{(\hat{xs}_n, \hat{ys}_n) \in \mathbb{Z}_+^2 \mid (\hat{xs}_n, \hat{ys}_n) = (xq_m, yq_m) \times \sigma + (i - p, j - p), \\ 0 \leq i < k,\ 0 \leq j < k\} \quad (3.4)$$

donde $k \subset \mathbb{Z}_{>0}$ representa la dimensión del *kernel*. Estos elementos $(\hat{xs}_n, \hat{ys}_n) \in \Upsilon$

se buscan en $\mathbf{S}$, y sus características asociadas respectivas $as_n$ se operan con los valores de núcleo $K(i,j)$, produciendo las $d_o$ características de salida en el elemento $(xq_m, yq_m) \in \Phi$:

$$bq_m = \sum_{(xs_n, ys_n) \in \Upsilon} as_n \times K(i,j). \tag{3.5}$$

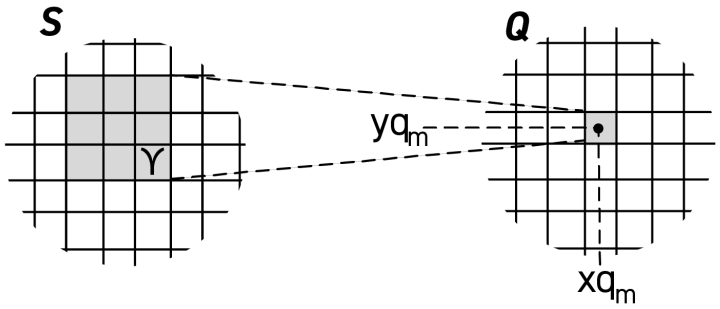



Figura 3.3: Coordenada de salida $(xq_m, yq_m) \in Q$ y el subconjunto de coordenadas de entrada $\Upsilon \subset S$ que contribuyen a ella.

Una vez que $\Phi$ ha sido procesado completamente, se selecciona el siguiente elemento en $\mathbf{S}$, y el proceso se repite hasta que se hayan calculado todos los elementos en $\mathbf{Q}$. Este doble procesamiento de coordenadas (conjuntos $\Phi$ y $\Upsilon$) se realiza para evitar el almacenamiento de valores parciales $bq_m$, lo cual en hardware implica un costo significativo en términos de la memoria RAM necesaria para almacenar las características. El algoritmo 1 más abajo detallado, presenta en formato de pseudocódigo el mecanismo de procesamiento descrito anteriormente.

El mecanismo de operación puede ampliarse para realizar operaciones de *pooling* de la siguiente forma: se asume que el conjunto de entrada $\mathbf{S} = \{s_n\}_{n=1}^{N}$ contiene las salidas de convolución obtenidas mediante el Algoritmo 1, almacenadas en una estructura de entrada tipo CAM-RAM. Las capas de *pooling* generalmente operan con una ventana deslizante de tamaño $2 \times 2$ y un paso de 2. El cálculo de las coordenadas de salida $(xq_m, yq_m) \in \mathbf{Q}$ representa un caso particular del procedimiento anterior, y es equivalente al caso de una convolución donde el *kernel* es de $2 \times 2$ y no hay solapamiento de los pasos de convolución, es decir, *stride* de 2. Dado que la salida tiene la misma cantidad de canales que la entrada, $d_o = d_i$, se procede a través de los pasos hasta la Ecuación 3.5, donde las intensidades de salida $b_m$ para el caso de *MaxPool* se calculan como el valor máximo entre las

**Algoritmo 1:** Convolución por eventos

**Entrada:**

- Conjunto de coordenadas de entrada $(xs_n, ys_n) \in \mathbf{S}$ almacenadas en CAM
- Conjunto de intensidades de entrada $as_n \in \mathbf{S}$ almacenadas en RAM
- Padding $p$
- Stride $\sigma$
- Dimensiones $H_i$ y $W_i$ de la matriz rala de entada
- Pesos $K \in \mathbb{R}^{d_0 \times k^2 \times d_i}$ almacenados en RAM

**Salida:**

- Coordenadas de salida $(xq_m, yq_m) \in \mathbf{Q}$ almacenadas en CAM
- *features* (intensidades) de salida $bq_m \in \mathbf{Q}$ almacenadas en RAM

Inicializar $\mathbf{Q} = \varnothing$
**foreach** $(xs_n, ys_n) \in \mathbf{S}$ **do**
  Inicializar $\Phi = \varnothing$
  **for** $i = 0$ *to* $k - 1$ **do**
    **for** $j = 0$ *to* $k - 1$ **do**
      $(\hat{xq}_m, \hat{yq}_m) = \frac{(xs_n, ys_n) - (i+p, j+p)}{\sigma}$
      **if** $(\hat{xq}_m, \hat{yq}_m) \in \mathbb{Z}_+^2$ **then**
        $\Phi \leftarrow (\hat{xq}_m, \hat{yq}_m)$
      **else**
        Descartar elemento
  **foreach** $(xq_m, yq_m) \in \Phi$ **do**
    **if** $(xq_m, yq_m) \notin \mathbf{Q}$ **then**
      $\mathbf{Q} \leftarrow (xq_m, yq_m)$
      Inicializar $\Upsilon = \varnothing$
      Inicializar $bq_m = 0$
      **for** $i = 0$ *to* $k - 1$ **do**
        **for** $j = 0$ *to* $k - 1$ **do**
          $(\hat{xs}_n, \hat{ys}_n) = (xq_m, yq_m) \times \sigma + (i - p, j - p)$
          **if** $\hat{xs}_n \in [0, H_i)$ ***and*** $\hat{ys}_n \in [0, W_i)$ **then**
            $\Upsilon \leftarrow (\hat{xs}_n, \hat{ys}_n)$
            Leer $as_n$ de RAM
            $bq_m = bq_m + as_n \times K(i, j)$
      Guardar $bq_m$ en RAM

intensidades de entrada $a_n \in \Upsilon$:

$$b_m = \text{máx}\{a_n \in \Upsilon\}. \tag{3.6}$$

En el caso de *AveragePool*, el cálculo de las intensidades de salida $b_m$ se consigue mediante el cómputo de una convolución con un *kernel* cuyos valores de pesos son iguales a $1/k^2$.

Después de calcular la secuencia de capas de convolución y pooling correspondientes a la etapa de filtrado de una red CNN, es necesario procesar las llamadas capas *Fully Connected* en redes de clasificación. Aunque estas capas pueden presentar cierto grado de dispersión, su densidad es demasiado elevada para ser procesada con el mecanismo de cómputo descrito. Sin embargo, la matriz dispersa de salida de la última capa de filtrado debe convertirse a una representación densa para permitir el procesamiento en otras arquitecturas. Para ello, se introduce un mecanismo de cómputo que toma como entrada la matriz dispersa de salida de la última capa de filtrado y realiza el cálculo de la primera capa *Fully Connected*, almacenando su salida en formato denso en la memoria RAM (algoritmo 2). La función principal de este mecanismo es re-mapear las coordenadas bidimensionales $(xs_n, ys_n) \in \mathbf{S}$ a coordenadas unidimensionales $idx$. Este mapeo se realiza por columnas mediante una función $f : \mathbb{Z}_+^2 \to \mathbb{Z}_+$, definida sobre el conjunto $\mathbf{S}$ de la siguiente manera:

$$f(xs_n, ys_n) = ys_n \times H_i + xs_n \tag{3.7}$$

donde $H_i$ representa la altura de la matriz dispersa de entrada. Además, se considera una matriz de pesos $K \in \mathbb{R}^{T \times d_i \times d_o}$, donde $T = H_i \times W_i$ representa el número máximo de elementos en $\mathbf{S}$ en ausencia de dispersión, $d_i$ es el número de canales del mapa de características de entrada y $d_o$ es el número de neuronas de salida.

**Algoritmo 2:** Fully Conected Dispersa a Densa

**Entrada:**

- Conjunto de coordenadas de entrada $(xs_n, ys_n) \in \mathbf{S}$ almacenadas en CAM.
- Conjunto de características de entrada $as_n \in \mathbf{S}$ almacenadas en RAM.
- Dimensiones $H_i$ y $W_i$ de las entradas.
- Pesos $K \in \mathbb{R}^{T \times d_i \times d_o}$ almacenados en RAM.

**Salida:**

- Vector de características de salida densa **ofm** de $d_o$ elementos.

```
1 Inicializar ofm = 0
2 for xq_m ∈ [0, d_o) do
3     foreach (xs_n, ys_n, as_n) ∈ S do
4         idx ← ys_n × H_i + xs_n
5         ofm[xq_m] = ofm[xq_m] + as_n × K(idx, xq_m)
```

## 3.2. Opciones de arquitectura

En esta sección se presentan y discuten diversas opciones de arquitecturas para procesamiento de redes CNN mediante los algoritmos descritos en la Sección 3.1. Se exploran implementaciones con distintos grados de complejidad que permiten optimizar el uso del área de silicio y minimizar la pérdida de datos durante el procesamiento. Estas implementaciones constan esencialmente de un bloque de adquisición y procesamiento, que realiza todas las operaciones de cómputo descritas en los algoritmos de la Sección 3.1, y gestiona una serie de bloques CAM-RAM que almacenan las matrices dispersas de todas las capas de filtrado de la red CNN, y se interconectan entre sí a través de una arquitectura tipo bus.

La implementación más intuitiva consiste en un sistema como el de la Fig. 3.4, que consta de $N + 1$ conjuntos CAM-RAM, donde $N$ es el número de capas en la red neuronal que necesita ser procesada. En esta configuración, los eventos entrantes se integran utilizando el primer conjunto. Después de un período de integración especificado $T$, se detiene la recolección de eventos, y los datos integrados en el primer conjunto se utilizan como entrada de la primera capa de la red neuronal. El segundo conjunto se utiliza para almacenar la salida de esta ca-

pa. Este proceso continúa de manera que cada conjunto CAM-RAM subsiguiente sirve como la salida para la siguiente capa, culminando en el conjunto $N+1$, que contiene la salida final que produce el sistema.

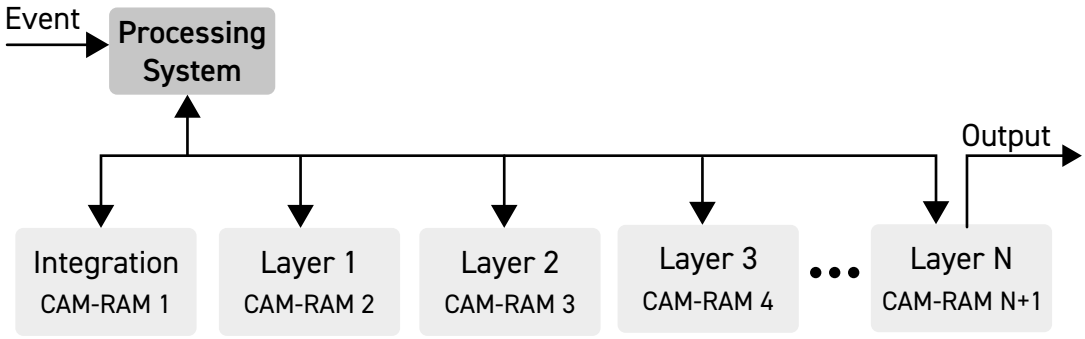



Figura 3.4: Sistema básico que consta de N+1 conjuntos CAM-RAM, donde cada capa se almacena en un conjunto diferente.

En esta implementación, el número de conjuntos CAM-RAM necesarios para procesar la red neuronal depende directamente del número de capas, con todas las capas internas de la red almacenadas en el sistema. Este esquema presenta la ventaja de que la red puede actualizarse en tiempo real de manera incremental a medida que llegan nuevos eventos, aprovechando así la resolución temporal completa que éstos ofrecen, y logrando el máximo desempeño en términos de velocidad de procesamiento.

No obstante, presenta la limitación de vincular el sistema a una arquitectura de red con un número máximo de capas, lo que, en el caso de redes con menos capas, resulta en un uso subóptimo del sistema. Además, el costo de las memorias CAM en términos de área, sumado al hecho de que las redes neuronales modernas suelen constar de numerosas capas, implicaría una cantidad significativa de área para implementar el número requerido de conjuntos CAM-RAM, haciendo que esta implementación sea poco práctica. Sin embargo, resulta una alternativa interesante si se busca el máximo desempeño y se dispone de una red CNN pequeña y altamente optimizada.

La alternativa que se propone en esta tesis es un sistema que consta de dos conjuntos CAM-RAM, ambos inicialmente vacíos. Se utiliza el primer conjunto para acumular eventos entrantes durante un período de tiempo $T_I$, como se muestra en la Fig. 3.5a. Al finalizar este período, se detiene la acumulación y se utiliza

este conjunto integrado como la entrada para la primera capa de la red neuronal (NN), almacenando la salida calculada en el segundo conjunto CAM-RAM. Una vez que esta operación concluye, se procede a calcular la segunda capa. Para este cálculo, los roles de los conjuntos se invierten: el conjunto de salida de la operación anterior ahora sirve como entrada. Este proceso alternante continúa hasta que los cálculos de la NN se completan en un tiempo total de procesamiento denominado $T_P$ (Fig. 3.5b). En este punto, se exportan los resultados finales del sistema y se reinicia el ciclo de adquisición de datos (Fig. 3.5c).

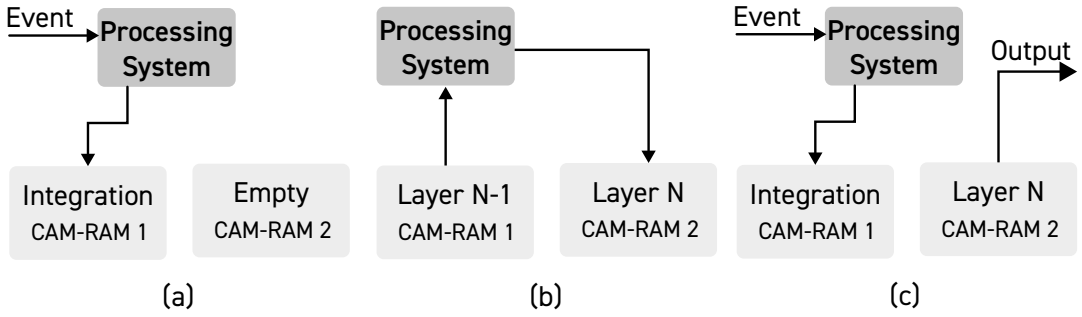



Figura 3.5: Sistema mínimo de 2 conjuntos CAM-RAM: (a) fase de integración inicial.; (b) proceso cíclico de cálculo de la red neuronal; (c) salida del sistema mientras el segundo conjunto integra el siguiente conjunto de eventos.

Este enfoque aborda el problema de los requisitos espaciales, y es independiente del número de capas de la red CNN; sin embargo, se pierde la resolución temporal de los eventos al procesar de manera discreta cada vez que se cumple el período de integración $T_I$, introduce una latencia igual al tiempo de procesamiento total $T_P$, y es incapaz de aceptar nuevos eventos mientras se procesa la red neuronal, lo que resulta en una ventana de pérdida de información durante el tiempo que dure el procesamiento de la red, y se reinicie el ciclo de adquisición. Para resolver esto, se propone la introducción de un tercer conjunto CAM-RAM (Fig. 3.6).

En esta configuración, se utiliza el primer conjunto para acumular eventos entrantes durante un período de tiempo $T_I$. Después de este período, se entregan los datos a un sistema de procesamiento que utilizará este conjunto de entrada junto con otro conjunto, para calcular toda la red neuronal ciclando entre roles de entrada y salida en una ventana de tiempo $T_P$, mientras que el $3^{\text{er}}$ conjunto recién incorporado se utilizará para iniciar el nuevo ciclo de integración en paralelo.

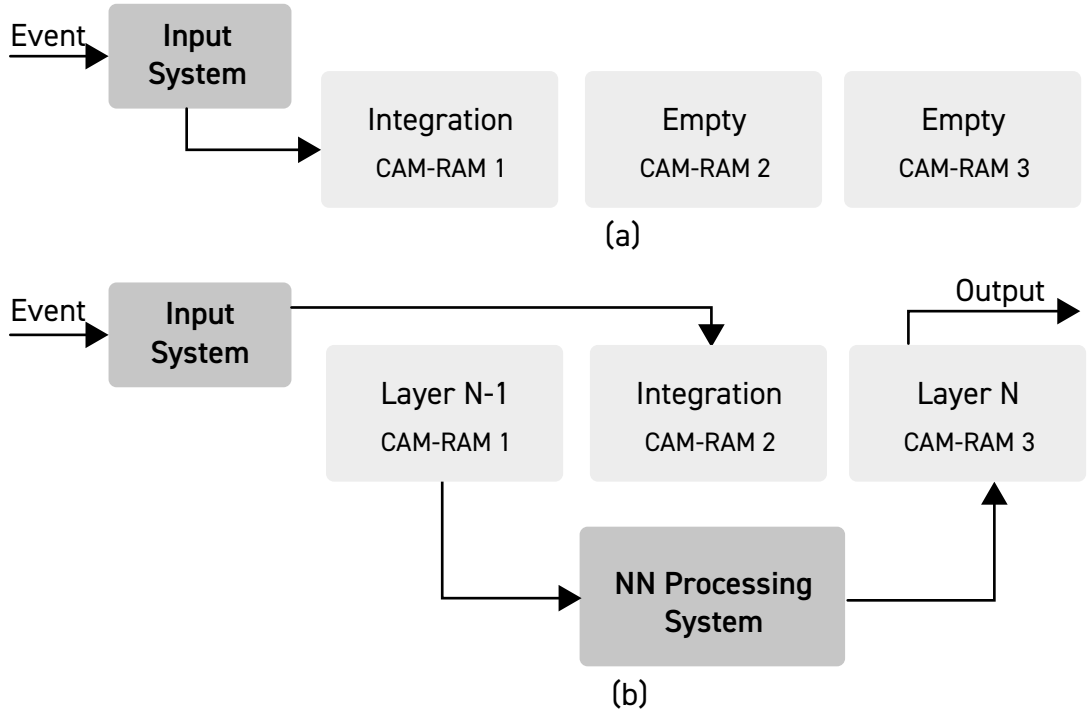



Figura 3.6: Sistema de 3 conjuntos CAM-RAM.

Este enfoque funciona bien siempre que el tiempo de procesamiento de la red neuronal sea menor o igual al tiempo de integración de eventos, de modo que toda la red pueda ser calculada mientras el conjunto de entrada sigue integrando. El punto de operación óptimo del sistema ocurre cuando el tiempo de procesamiento $T_P$ es igual al tiempo de integración $T_I$ (Fig. 3.7a). En estas circunstancias el sistema se encuentra procesando constantemente. Si el tiempo de procesamiento es menor, el sistema entra en un estado subóptimo donde los dos conjuntos CAM-RAM utilizados para procesar se encuentran desocupados hasta que el sistema termine el proceso de integración (Fig. 3.7b). Sin embargo, cuando el tiempo de integración $T_I$ es menor que el tiempo de procesamiento de la red neuronal $T_P$, ocurre pérdida de datos durante un tiempo $T_P - T_I$, ya que no hay conjuntos CAM-RAM disponibles. Esta condición puede suceder en escenarios con altas tasas de datos basados en eventos.

Este concepto sugiere intuitivamente que la relación entre los tiempos de integración y procesamiento puede ajustarse mediante la adición de más conjuntos CAM-RAM, permitiendo que el sistema siempre disponga de un conjunto libre para iniciar un nuevo ciclo de integración. Por ejemplo, al agregar dos conjuntos

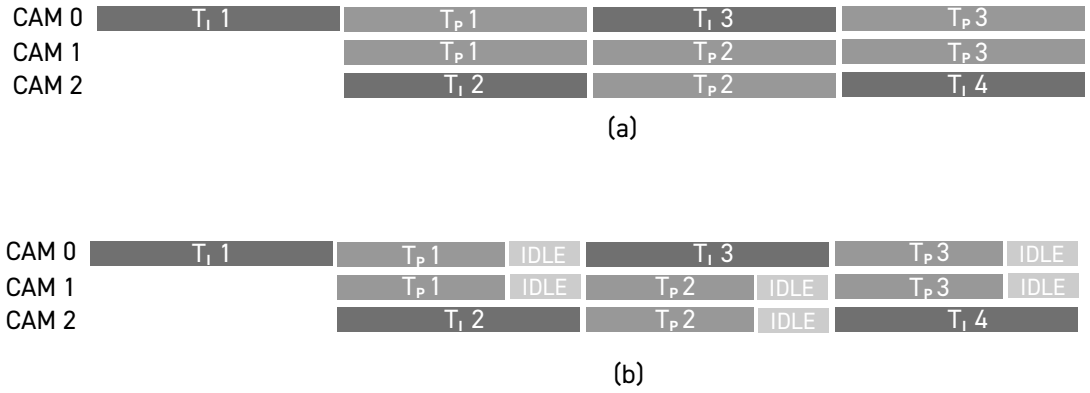



Figura 3.7: Pipeline de operación de un sistema de 3 conjuntos CAM-RAM. (a) Caso óptimo donde $T_I = T_P$; (b) Caso subóptimo donde $T_I > T_P$.

CAM-RAM adicionales al sistema de la Fig. 3.6, el punto de operación óptimo se alcanza cuando el tiempo de integración es la mitad del tiempo de procesamiento (Fig. 3.8), lo cual reduce el rango de pérdida de información para tiempos de integración inferiores a $T_P/2$.

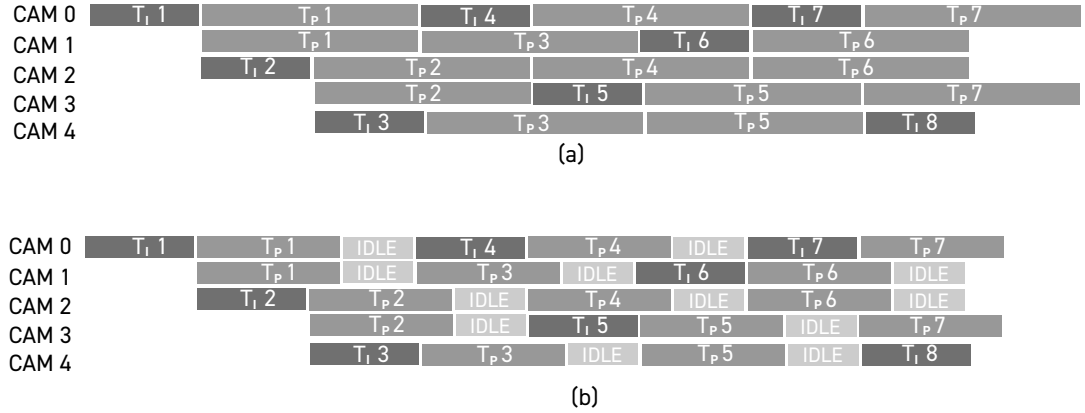



Figura 3.8: Pipeline de operación de un sistema de 5 conjuntos CAM-RAM. (a) Caso óptimo donde $T_I = T_P/2$; (b) Caso subóptimo donde $T_I > T_P/2$.

Esto puede ampliarse para un número arbitrario de conjuntos CAM-RAM impares, donde la relación entre el tiempo de integración y el tiempo de procesamiento se rige por la siguiente expresión matemática:

$$\frac{T_P}{T_I} = \frac{(N-1)}{2} \tag{3.8}$$

donde $T_P/T_I$ representa la relación entre el tiempo de procesamiento y el tiempo de integración, y $N$ es el número de conjuntos CAM-RAM totales que dispone el sistema.

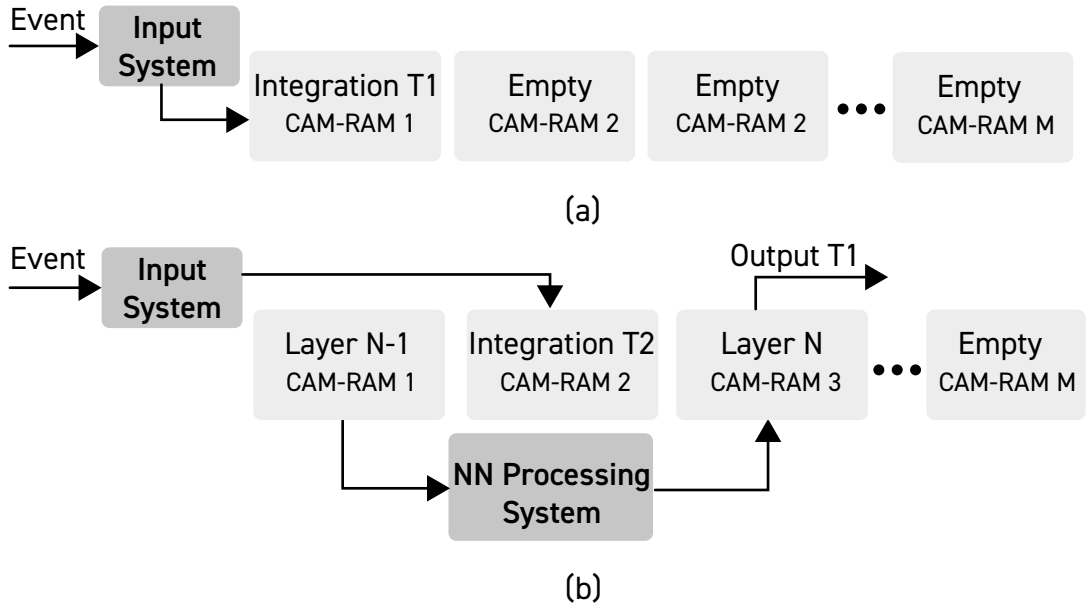



Figura 3.9: Sistema que consta de $M$ conjuntos CAM-RAM.

Para este caso, se propone una configuración del sistema con un número arbitrario $M$ de conjuntos CAM-RAM. En este sistema, un conjunto se dedica a integrar eventos entrantes, mientras que otros dos se asignan para calcular la red neuronal. Una vez que se completa la integración de un conjunto, otro conjunto disponible comienza a integrar el siguiente lote de eventos, con el conjunto previamente integrado en cola para ser procesado tan pronto como el motor de cálculo de la red neuronal esté libre. Esta configuración garantiza una integración y procesamiento de datos continuos, independientemente de la demanda operativa de la red neuronal. Aquí, a diferencia de la configuración representada en la Fig. 3.4, el número de conjuntos CAM-RAM en la Fig. 3.9 no se correlaciona con el número de capas de la red neuronal, sino que es inversamente proporcional al tiempo mínimo de integración factible $T_{IM}$ que evita la pérdida de datos para un tiempo de procesamiento de la red neuronal dado. La Fig. 3.10 ilustra la relación entre el tiempo de integración y el tiempo de procesamiento y la cantidad de conjuntos CAM-RAM mínimos necesarios para evitar pérdida de información. El número de conjuntos CAM-RAM crece exponencialmente a medida que se busca reducir la relación entre tiempo de integración y procesamiento. Esto demuestra que esta metodología resulta eficiente siempre y cuando la cantidad de conjuntos CAM-RAM se mantenga en un número que resulte en una implementación razonable en término de área. Si se necesita incrementar el desempeño, se

puede optar por incrementar la frecuencia de reloj del sistema, lo que reduciría el tiempo de integración mínimo en términos absolutos.

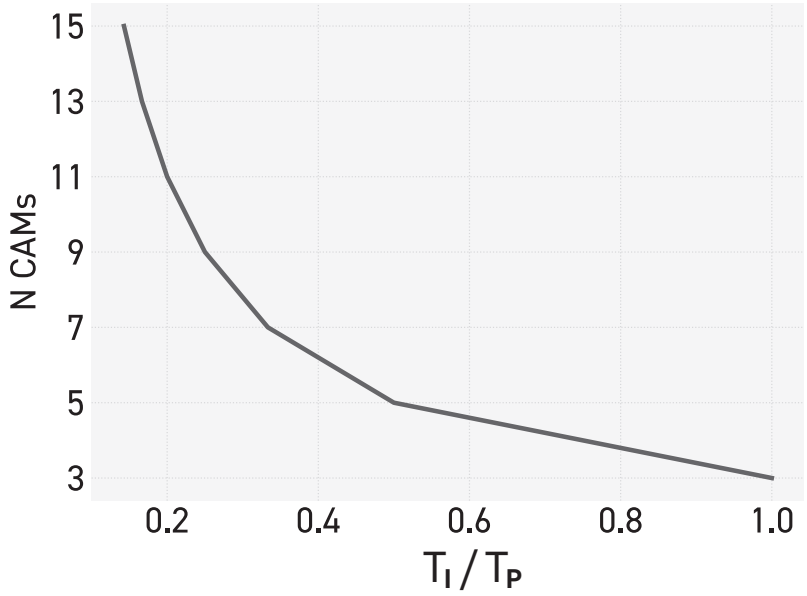



Figura 3.10: Cantidad de conjuntos CAM-RAM necesarios en función de la relación entre el tiempo de integración y el tiempo de procesamiento.

## 3.3. Conclusiones

En este Capítulo se ha abordado la representación y el procesamiento de datos dispersos basados en eventos dentro de un marco de redes neuronales, utilizando memorias CAM. Se introdujeron algoritmos para el procesamiento de capas de convolución y *pooling*, que constituyen la etapa de filtrado y extracción de características en una red neuronal convolucional (CNN). Asimismo, se presentó un algoritmo para procesar la primera capa *Fully Connected*, y mapear la representación matricial dispersa, almacenada en un conjunto CAM-RAM, a una representación densa que puede ser procesada por arquitecturas de procesamiento convencionales.

Se evaluaron distintas implementaciones de arquitecturas en términos de flexibilidad, área y desempeño. Se presentaron dos modelos de arquitectura: en primer lugar, una arquitectura que permite el procesamiento y la actualización de la red neuronal evento por evento, aprovechando toda la resolución temporal

de los mismos. Sin embargo, el número de conjuntos CAM-RAM necesarios depende de la cantidad de capas en la red, lo que limita su flexibilidad en favor del desempeño. En segundo lugar, se introdujo una arquitectura en la que el número de conjuntos CAM-RAM es independiente de la cantidad de capas, otorgando mayor flexibilidad al sistema, pero a costa de una menor resolución temporal y una latencia igual al tiempo de procesamiento total de la red neuronal. Además, se derivó una relación matemática entre el número de conjuntos CAM-RAM disponibles en el sistema y la relación entre los tiempos de integración y procesamiento, proporcionando una herramienta esencial para elegir la implementación más adecuada según los requisitos del sistema y la red.

# Capítulo 4

# Arquitectura del Sistema

En el capítulo anterior, se desarrollaron algoritmos para el procesamiento de redes neuronales convolucionales dispersas utilizando memorias CAM y se exploraron distintas opciones de arquitectura en función del desempeño y la flexibilidad. En este capítulo se presenta y analiza una arquitectura flexible basada en $N$ CAMs, como la propuesta en el Capítulo 3, que permite implementar un número arbitrario de capas y lograr un buen desempeño con una cantidad razonable de conjuntos CAM-RAM.

Este capítulo se organiza de la siguiente manera: en la Sección 4.1, los algoritmos se clasifican según las operaciones que requieren y se describe la microarquitectura de los bloques de hardware necesarios para su ejecución. En primer lugar, un bloque de adquisición recibe los eventos y los almacena temporalmente, tras lo cual un controlador de adquisición actualiza la matriz dispersa de entrada en uno de los conjuntos CAM–RAM. A continuación, el módulo de cálculo de la red convolucional integra tres componentes clave: un decodificador dual de coordenadas que gestiona todas las operaciones relativas a las coordenadas de entrada, salida y pesos; un arreglo de unidades MAC (multiplicación y acumulación) encargado de procesar las características; y un bloque de escalado y redondeo que ajusta el resultado final e implementa las funciones de activación. De este modo, cada etapa del flujo de datos —desde la adquisición de eventos hasta la obtención de la salida convolucional— queda claramente delimitada en componentes hardware especializados.

En la Sección 4.2, se desarrolla un modelo funcional de la interacción entre

los distintos bloques, considerando la microarquitectura diseñada. Esto permite modelar los tiempos necesarios en ciclos de reloj para el procesamiento de las capas de la red, identificando cuellos de botella y las relaciones de compromiso entre la complejidad de implementación y el desempeño.

## 4.1. Bloques Funcionales

La Fig. 4.1 muestra un diagrama en bloques de la arquitectura del sistema, que incluye los cinco bloques funcionales necesarios para el procesamiento de una red convolucional por eventos: sistema de adquisición, decodificador de coordenadas, arreglo de elementos de procesamiento, memorias CAM y sus RAM asociadas.

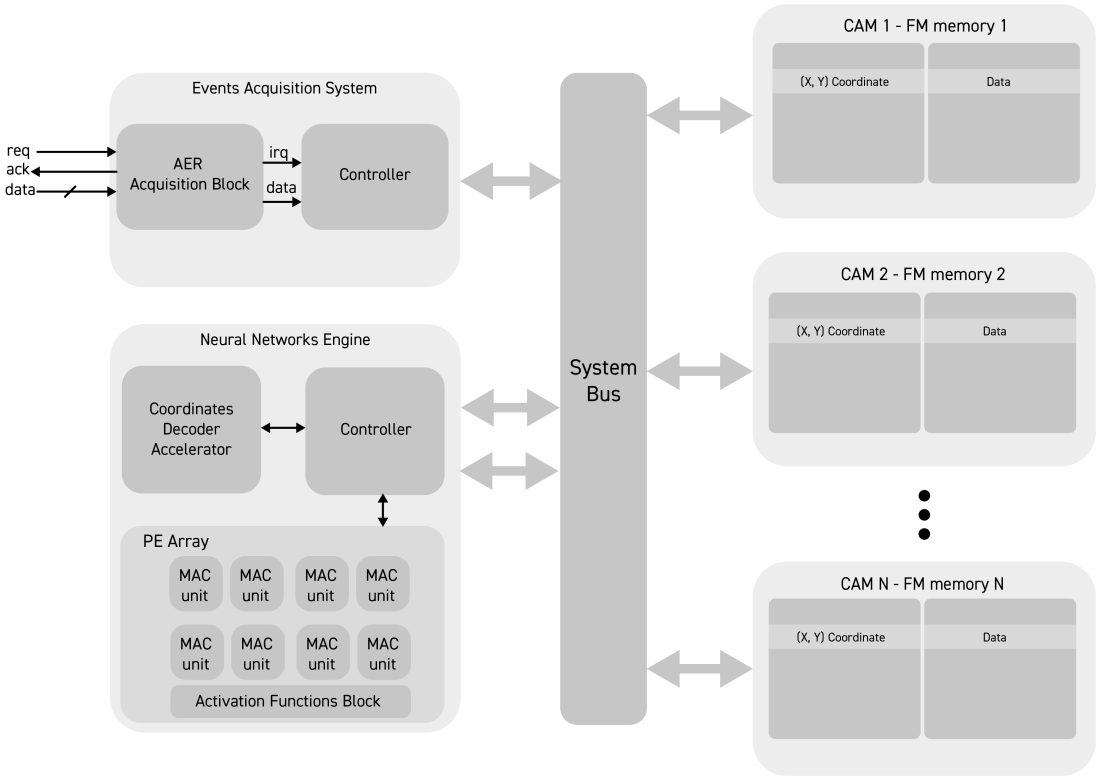



Figura 4.1: Diagrama en bloques de una arquitectura con $N$ CAMs, sistema de adquisición de eventos y acelerador de redes neuronales por eventos.

Los controladores independientes tanto en el sistema de adquisición como en el acelerador de redes neuronales permite realizar el cómputo en tiempo real. Mientras el sistema de adquisición recibe y procesa eventos, el acelerador opera en paralelo con datos de una ventana de tiempo anterior, optimizando así el rendimiento del sistema.

A continuación se describe la microarquitectura de cada uno de los bloques del sistema, mencionados anteriormente.

### 4.1.1. Bloque CAM-RAM

El bloque CAM-RAM, como su nombre lo indica, está compuesto por dos partes fundamentales: un bloque de memoria CAM utilizado para almacenar coordenadas de manera eficiente y una memoria RAM de alta densidad destinada a almacenar las características asociadas. Aunque se consideran parte del mismo bloque debido a que su combinación permite almacenar matrices dispersas, cada una de estas memorias posee su propio puerto y se accede de manera completamente independiente a través del bus principal del sistema.

En la Fig. 4.2 se muestra un diagrama en bloques del módulo CAM, que consta del arreglo de memoria propiamente dicho [48], complementado con lógica adicional para facilitar las operaciones necesarias en el procesamiento de coordenadas. Estas operaciones incluyen la escritura de nuevas coordenadas y la búsqueda y lectura de coordenadas previamente almacenadas. Dado que las coordenadas almacenadas son siempre únicas y las operaciones se realizan de manera ordenada, tanto la escritura como la lectura se efectúan de forma secuencial. Esta configuración permite un manejo eficiente y preciso de las coordenadas, optimizando el rendimiento del sistema en tareas específicas de búsqueda y actualización de datos.

La lógica de control de la memoria CAM gestiona tanto las operaciones de escritura y lectura secuenciales como las operaciones de búsqueda. Su diseño incluye un mecanismo de detección de datos válidos para evitar hits con información inválida, así como un mecanismo de punteros de escritura y lectura, similares a los que se encuentran en diseños de memorias FIFO sincrónicas.

El arreglo de memoria incluye un puerto de lectura/escritura de acceso aleatorio y un puerto independiente para operaciones de búsqueda, lo que permite comparar una palabra de N bits con todo el arreglo en una latencia constante, independientemente de su posición. Las coordenadas almacenadas son únicas y se inicializan de manera secuencial con la llegada de nuevos eventos, como se describe en los algoritmos del Capítulo 3, por lo que resulta conveniente la incor-

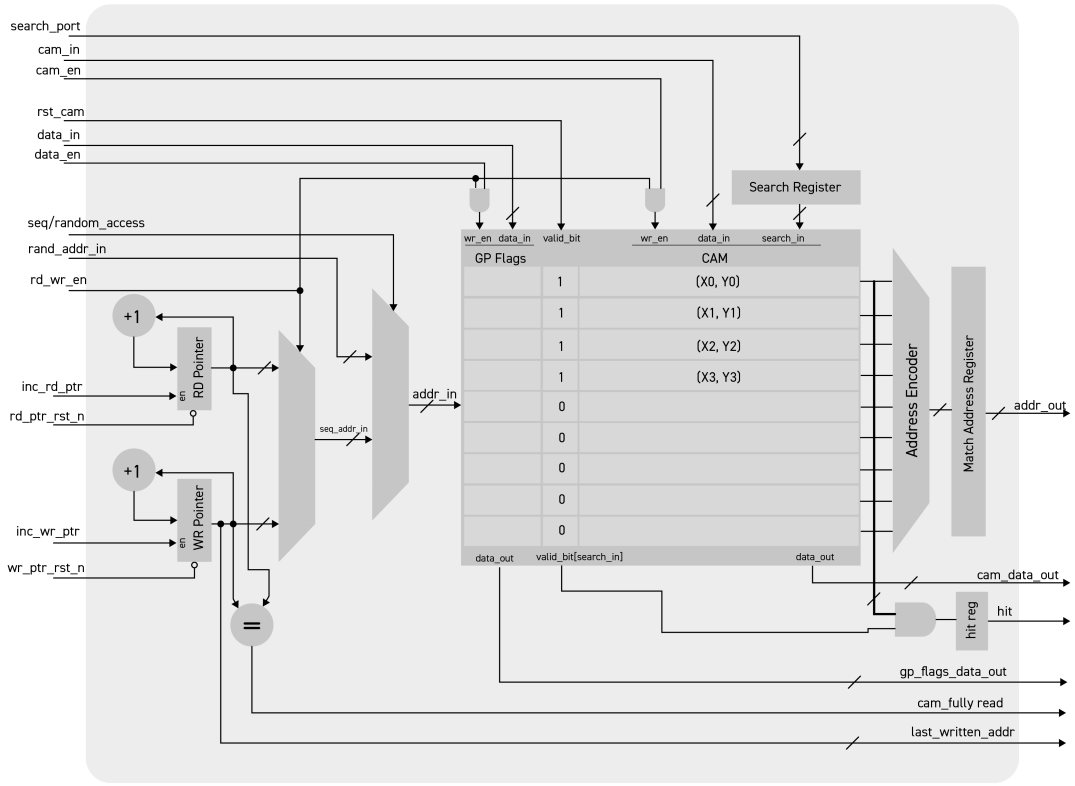



Figura 4.2: Diagrama en bloques del módulo CAM.

poración de un puntero de escritura que se auto-incrementa para evitar sobrescribir información. Cuando se realiza una escritura, la lógica de control configura el bit de dato válido para garantizar que la coordenada quede disponible para operaciones de búsqueda.

En operaciones de convolución o MaxPool, es esencial procesar todas las coordenadas de entrada. Por ello, al igual que en la escritura, se incorpora un puntero de lectura que se auto-incrementa con cada lectura de coordenada. Dado que la cantidad de coordenadas almacenadas puede variar, se incluye un mecanismo de detección de CAM leída, que compara los punteros de manera similar al mecanismo de detección de estado lleno y vacío en las memorias FIFO. Una vez procesadas todas las coordenadas, una señal de reset reinicia a cero todos los bits de dato válido y los punteros de escritura y lectura, dejando la memoria lista para ser reutilizada por otra capa de la red. Estas señales de CAM leída, vacía, y llena que se generan a partir de la comparación de los punteros, determinan estados clave del arreglo de memoria, y por ende, en conjunto con señales de habilitación, son utilizadas como señales de interrupción al controlador principal

del sistema.

Este diseño además incorpora por cada registro CAM, un registro adjunto sin la circuitería de comparación, para almacenar banderas de propósito general, (*General Purpose Flags*) "GP Flags" en la Fig. 4.2, que pueden o no ser escritas y leídas en conjunto con las coordenadas, dependiendo de una señal de habilitación. Esta funcionalidad, si bien no tiene un propósito específico, podría ser utilizada en implementación de algoritmos de integración de eventos que requieran de etiquetado de grupos de coordenadas que satisfagan ciertas condiciones, o incluso podría en ciertos casos donde se requiera poca precisión, reemplazar a la memoria RAM de alta densidad.

Tabla 4.1: Operaciones CAM y ciclos de reloj de cada operación.

| **Operación** | **T[clk]** | **Descripción** |
|---|---|---|
| sequential_read | 2 | Lectura secuencial. |
| sequential_write | 2 | Escritura secuencial. |
| search | 3 | Operación de búsqueda. Consta de una operación de escritura seguida de una operación de lectura para leer el resultado de la misma. |

Por último, se añade un multiplexor al puerto de direccionamiento del arreglo CAM, que permite a demás de la escritura/lectura secuencial, acceso de lectura/escritura aleatorios, en caso de que alguna implementación de algoritmo lo requiera. Todas las señales de entrada y salida de este módulo se mapean directamente a registros de manera ordenada a partir de una dirección base en el mapa de memoria del sistema, de acuerdo a la Tabla A.7, detallada en el Anexo A.

### 4.1.2. Bloque de Adquisición de Eventos

El bloque de adquisición se encarga de recibir eventos y almacenarlos temporalmente. La Fig. 4.3 muestra la representación en diagrama de bloques de dicho módulo. El puerto de entrada de eventos (ubicado del lado izquierdo en la Fig. 4.3) se diseña en concordancia con la implementación del protocolo AER por la empresa IniVation en sus cámaras DAVIS346 AER [49]. En dichas cáma-

ras, la lectura de eventos ocurre por filas. Para aprovechar el ancho de banda, la transmisión siempre comienza por la coordenada **y** (dirección de la fila), seguida de todas las coordenadas **x** que han sido disparadas. Por esta razón, el módulo receptor, luego de que la señal de *request* se sincroniza con el reloj del sistema mediante la lógica de sincronización, detecta el flanco descendente de la señal de *request* y emite un pulso a la máquina de estados mediante la señal *new_data_req*. Si la señal de habilitación de eventos *events_en* se encuentra activa y la FIFO no se encuentra llena, se activa la máquina de estados *FSM Controller* mediante la señal *start_fsm* que controla la adquisición.

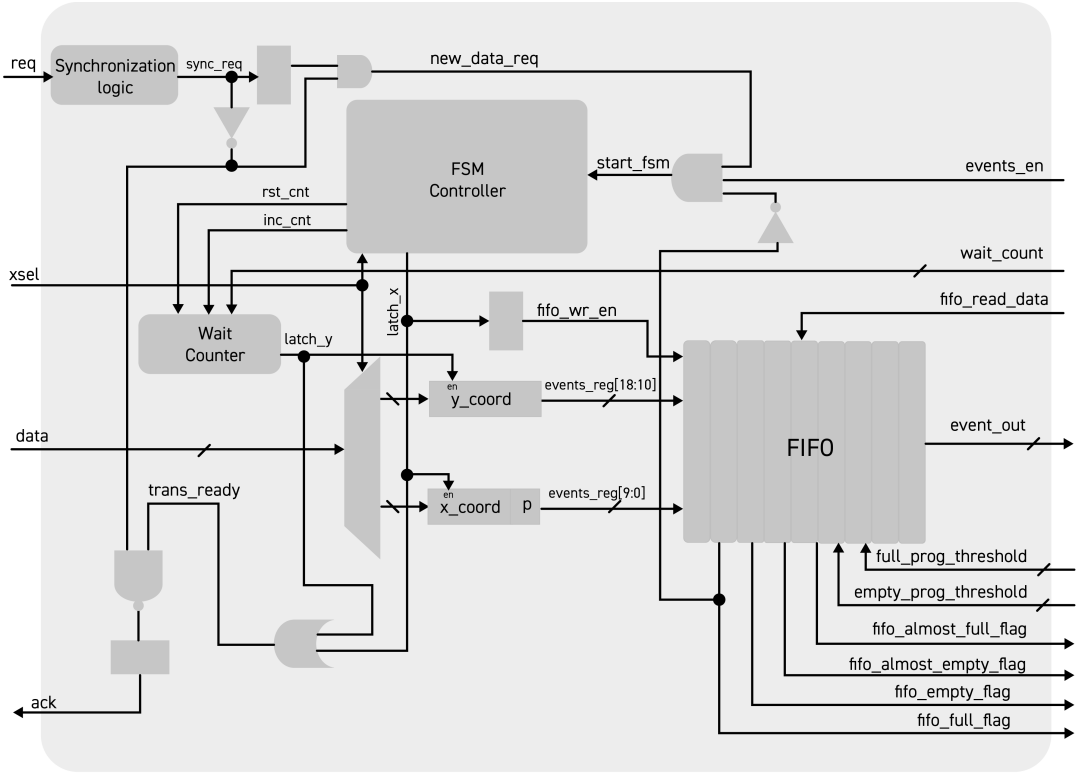



Figura 4.3: Diagrama en bloques del módulo de adquisición de eventos. Del lado izquierdo se encuentran las señales correspondientes al protocolo AER, mientras que del lado derecho se encuentran las señales de control y lectura de eventos.

En la Fig. 4.4 se ilustra el diagrama de flujo de la operación de la máquina de estados *FSM Controller*. Esta se encuentra inicialmente en el estado IDLE_ST manteniendo el contador de espera *Wait Counter* en estado de reset. Cuando se habilita la operación mediante la activación de la señal *start_fsm*, el valor de la señal *xsel* determina de qué tipo de coordenada se trata. Si se trata de una coordenada **Y**, transiciona al estado LATCH_YCOORD_ST, donde se habilita

el contador de espera hasta que la cuenta alcanza el valor estipulado por la señal *wait_count*. Al llegar a este valor, se habilita la señal *latch_ycoord* que permite guardar la coordenada en el registro temporal *y_coord*, dispara la lógica que acciona la señal *ack* inidicando al transmisor que los datos han sido guardados, y retorna al estado IDLE_ST. Luego de cada transacción donde se recibe una coordenada **Y**, se espera recibir una segunda transacción con una coordenada **X**. En este caso, la máquina de estados transiciona al estado LATCH_XCOORD_ST donde dicha coordenada en conjunto con la polaridad del evento son registrados inmediatamente [1] mediante la señal *latch_xcoord* que habilita la escritura del evento en FIFO, y por otro lado, al igual que en el caso anterior acciona la lógica de *ack* para dar por finalizada la transacción.

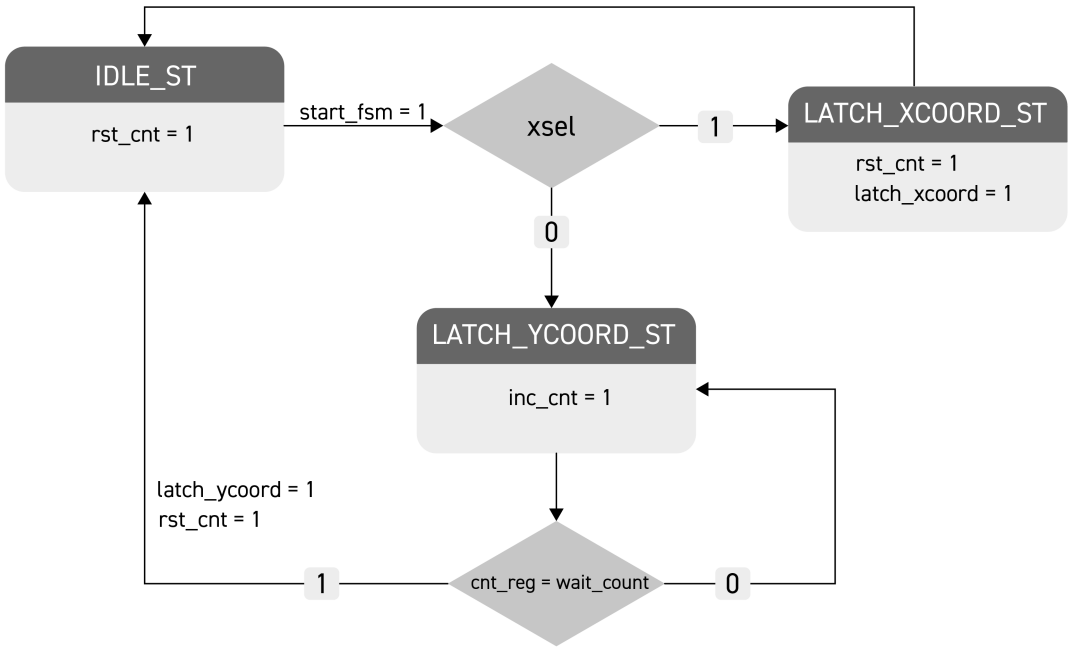



Figura 4.4: Máquina de estados del controlador del bloque de adquisición (Módulo *FSM Controller* en el diagrama en bloques de la Fig. 4.3).

Las señales de control y lectura de eventos de la FIFO se mapean a registros de memoria de acuerdo con la Tabla A.4. La FIFO consta de punteros de umbral programables que permiten mediante señales de habilitación, enviar interrupciones al controlador del sistema cuando ésta se encuentra medio llena, para que los

[1] Debido a limitaciones de implementación del protocolo AER por la empresa IniVation [49], cuando se realizan transacciones de coordenadas **Y** se debe esperar un mínimo de $50ns$ para garantizar la estabilidad del dato. Sin embargo, cuando se transmiten coordenadas **X**, el dato puede considerarse válido instantáneamente.

eventos sean leídos y procesados y así evitar cuellos de botella en la adquisición.

### 4.1.3. Decodificador de Coordenadas

El decodificador de coordenadas es un elemento clave en el procesamiento de operaciones de convolución y MaxPool sobre matrices dispersas. Su función principal es calcular las coordenadas involucradas en estas operaciones, optimizando así el manejo de datos dispersos. Este módulo opera a partir de una matriz dispersa de entrada, almacenada en un bloque CAM-RAM (denotado como el conjunto **S**, según la notación introducida en el Capítulo 3), y utiliza parámetros específicos de la capa como el paso (*stride*), el tamaño del *kernel*, y el *padding*. Con base en estos parámetros, dada una coodenada de la matriz dispersa de entrada, el decodificador determina todas las coordenadas de la matriz dispersa de salida cuyas características se ven afectadas por la características de dicha coordenada de entrada (conjunto $\Phi$) y, además, el conjunto de todas las coordenadas de la matriz dispersa de entrada cuya combinación lineal de características genera cada una de las características de salida (conjunto $\Upsilon$). La Fig. 4.5 presenta un diagrama en bloques de los componentes principales que intervienen en este proceso. Dada una coordenada de la matriz dispersa de entrada y un conjunto específico

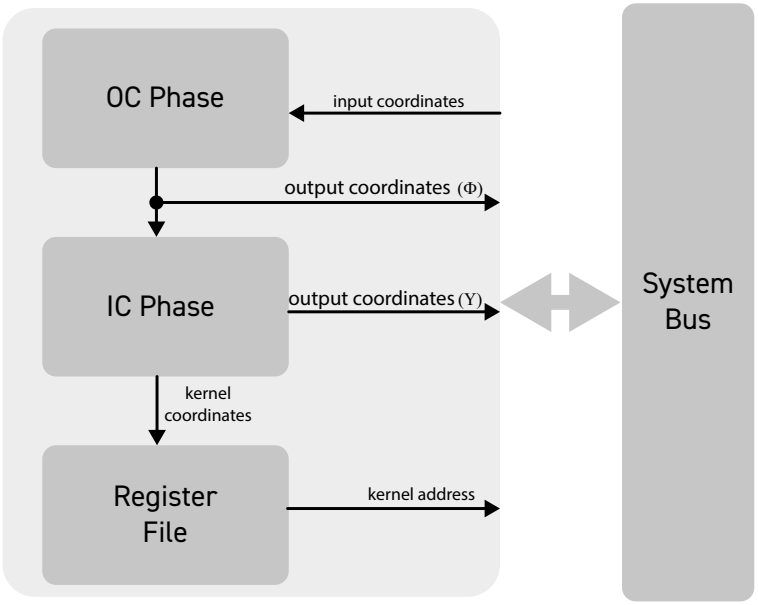



Figura 4.5: Diagrama de bloques del módulo Decodificador de Coordenadas.

de parámetros de capa, el bloque *OC Phase* calcula el conjunto de coordenadas

de salida $\Phi$ y las almacena temporalmente. La salida de este bloque es accesible a través del bus del sistema, permitiendo su lectura por el controlador, y además está conectada a la entrada del bloque *IC Phase*. Este segundo bloque, a partir de una coordenada del conjunto $\Phi$, determina las coordenadas del conjunto $\Upsilon$ de la matriz dispersa de entrada, junto con las coordenadas del *kernel*, y las almacena temporalmente en dos *buffers*. Ambos *buffers* son accesibles mediante el bus del sistema; sin embargo, el *buffer* que contiene las coordenadas del *kernel* está conectado a un banco de registros que asocia a cada coordenada la dirección física de memoria donde se almacenan los pesos correspondientes al *kernel*. Este proceso se ilustra en la Fig. 4.6.

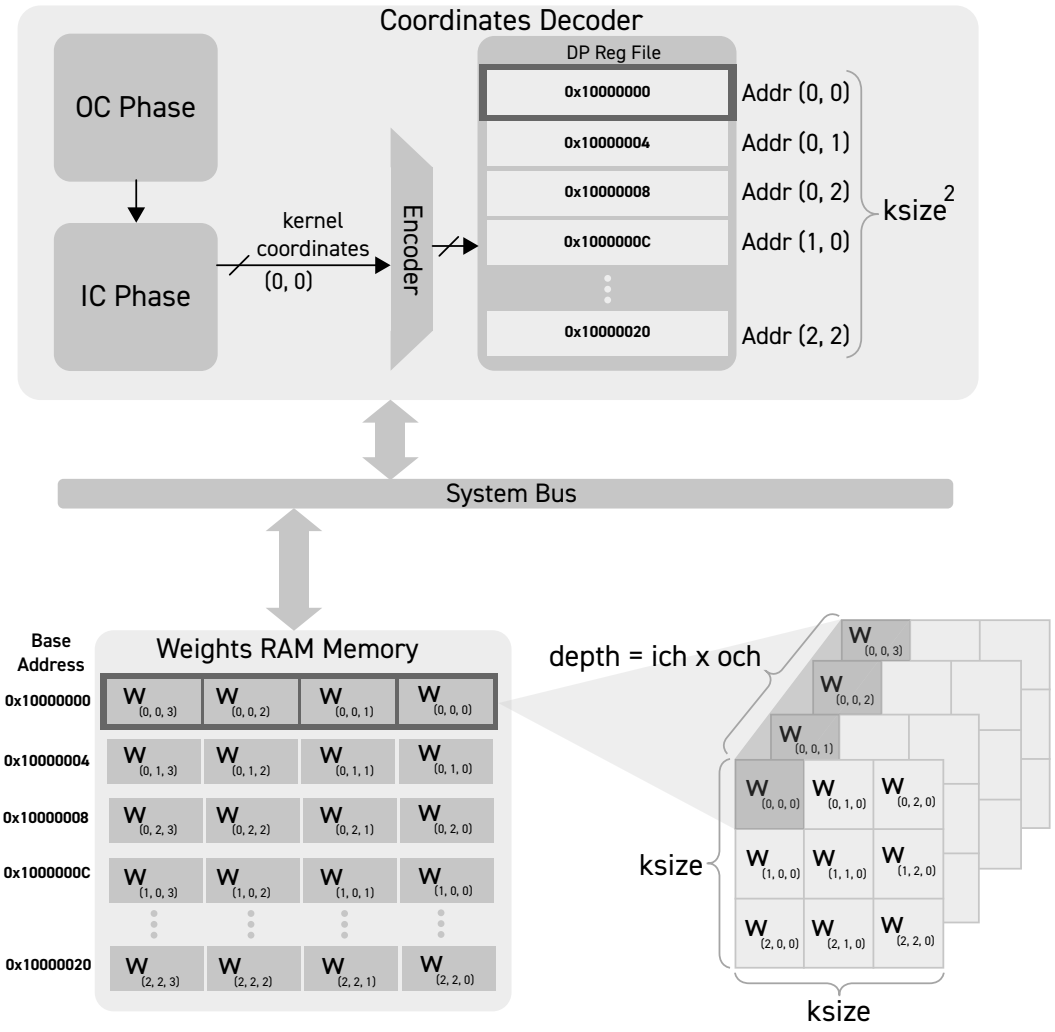



Figura 4.6: Diagrama de bloques simplificado del Decodificador de coordenadas y su interconexión a través del bus del sistema con la memoria RAM que almacena los pesos de una capa convolucional a procesar.

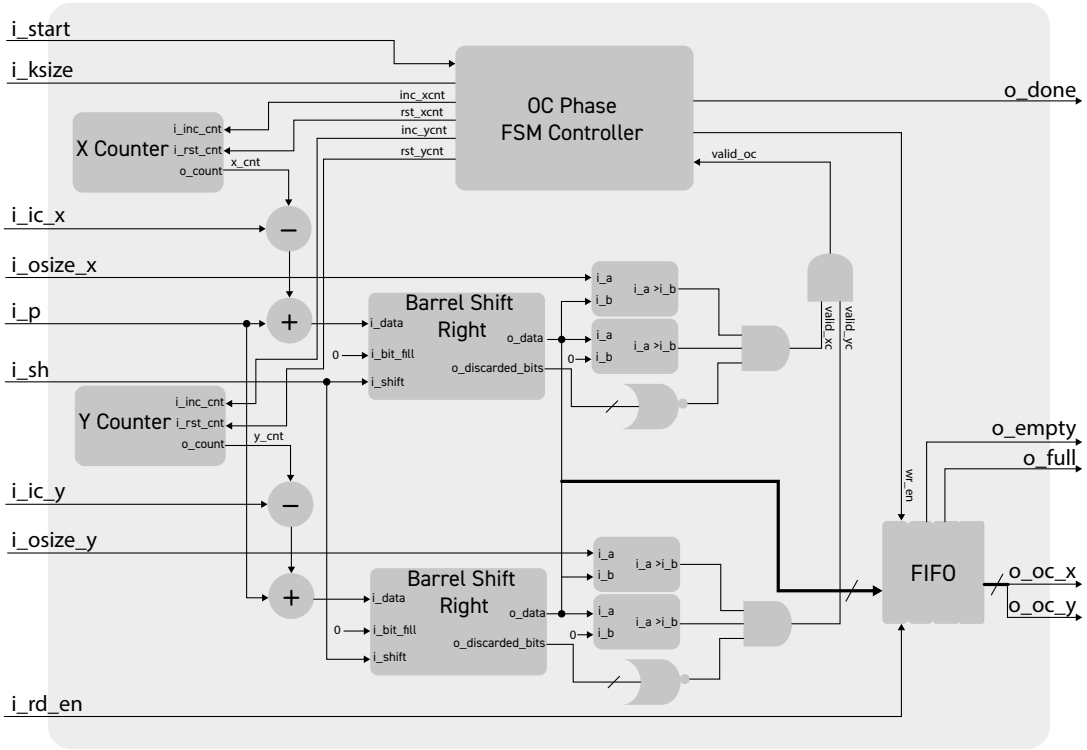



Figura 4.7: Diagrama de bloques del módulo Decodificador de Coordenadas de Salida (*OC Phase*).

### OC Phase

El módulo *OC Phase* recibe una coordenada $(i\_ic\_x, i\_ic\_y)$ de la matriz dispersa de entrada. Utilizando las señales *i_sh* (determina el paso), *i_ksize* (tamaño de kernel), *i_p* (*padding*), y $(i\_osize\_x, i\_osize\_x)$ (dimensiones de la matriz dispersa de salida), este módulo calcula las coordenadas correspondientes en la matriz dispersa de salida cuyas características son afectadas por la característica de entrada ubicada en $(i\_ic\_x, i\_ic\_y)$ (conjunto $\Phi$). La microarquitectura del bloque de hardware que lleva a cabo este proceso se detalla en la Fig. 4.7. El cálculo se basa en la Ec. 3.3, a la cual se le han impuesto restricciones para simplificar el diseño del hardware. En su versión revisada (Ec. 4.1), el *stride* (denotado como $\sigma$ según la notación del Capítulo 3), se restringe a valores que son potencias de 2, lo que reduce la complejidad de implementación de la operación de división mediante la inclusión de dos *barrel shifters* a derecha, que operan las coordenadas **x** e **y** respectivamente. Además, se ha reformulado la notación para alinearla con la terminología utilizada en la Fig. 4.7, de la siguiente manera:

$$(o\_oc\_x, o\_oc\_y) = \frac{(i\_ic\_x + i\_p, i\_ic\_y + i\_p) - (x\_cnt, y\_cnt)}{2^{i\_sh}} \tag{4.1}$$

donde $i\_p$ es el *padding*, $2^{i\_sh}$ es el *stride* de la capa a procesar e $i\_sh$ es un número entero que representa la cantidad de lugares a desplazar por el *barrel shifter*; $(i\_ic\_x, i\_ic\_y)$ es la coordenada de la matriz dispersa de entrada a procesar, $(x\_cnt, y\_cnt)$ representa todas las coordenadas de la matriz de pesos en el rango $i\_ksize$, y $(o\_oc\_x, o\_oc\_y)$ representa todas las coordenadas de las características de la matriz dispersa de salida que serán afectadas por la influencia de $(i\_ic\_x, i\_ic\_y)$ al iterar sobre dos contadores que generan los valores de $(x\_cnt, y\_cnt)$. Dado que las coordenadas son valores enteros positivos en el rango de las dimensiones de la matriz, esta ecuación está definida únicamente cuando se cumplen las siguientes condiciones:

1. $(i\_ic\_x, i\_ic\_y) - (x\_cnt, y\_cnt) + i\_p$ es múltiplo de $2^{i\_sh}$.

2. $(0, 0) \leq (o\_oc\_x, o\_oc\_y) \leq (i\_osize\_x, i\_osize\_y)$.

La primera condición puede garantizarse verificando los bits descartados como resultado de la división; si todos ellos son cero, el cociente es entero. La segunda condición se verifica mediante la incorporación de dos conjuntos de comparadores: el primero verifica que la coordenada resultante sea mayor o igual a cero, mientras que el segundo compara contra las señales *(i_osize_x, i_osize_y)* de manera que las coordenadas de salida estén dentro del rango de la matriz. Esta secuencia de acciones necesarias para producir las coordenadas del conjunto $\Phi$ están coordinadas mediante un controlador dedicado denominado *OC Phase FSM Controller*, cuya operación se ilustra mediante el diagrama de flujo de la Fig. 4.8. Los contadores *X Counter* e *Y Counter* iteran sobre todos los valores posibles de $(x\_cnt, y\_cnt)$. Si se produce una coordenada $(o\_oc\_x, o\_oc\_y)$ válida, se habilita la señal *valid_oc* y el controlador *OC Phase FSM Controller* habilita la escritura de dicha coordenada en la FIFO para ser almacenada. Una vez que ambos contadores alcanzan el valor $i\_ksize - 1$, el controlador habilita la señal *o_done*, indicando al controlador del sistema que la FIFO está lista para ser leída.

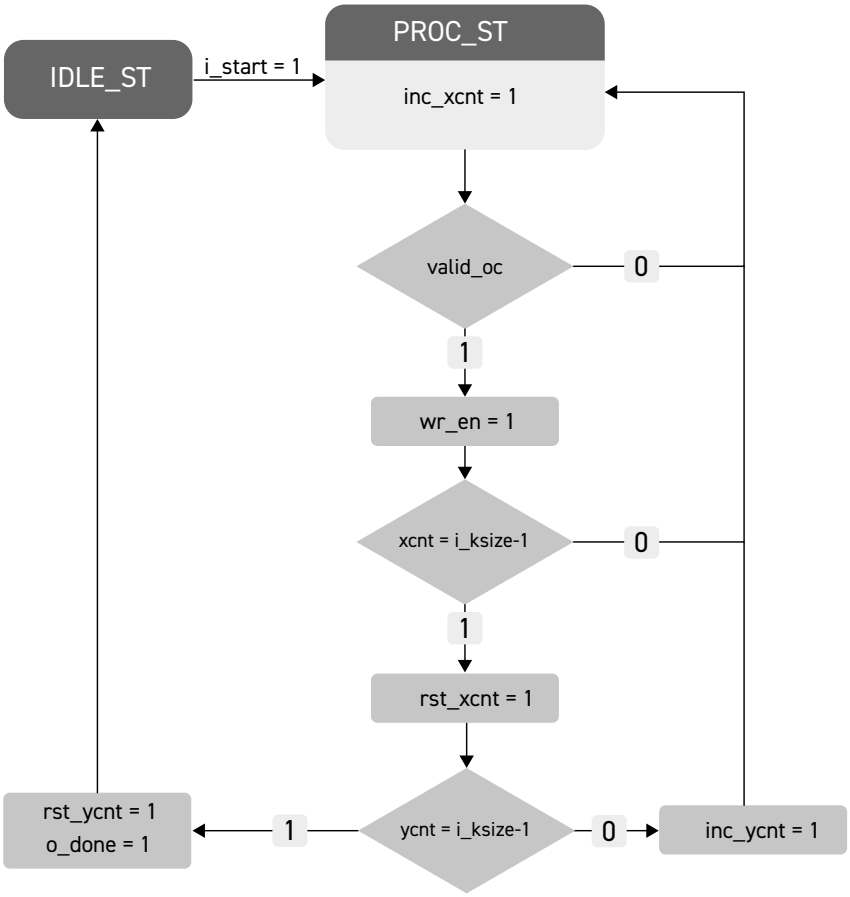



Figura 4.8: Máquina de estados del módulo Decodificador de Coordenadas de Salida (*OC Phase*).

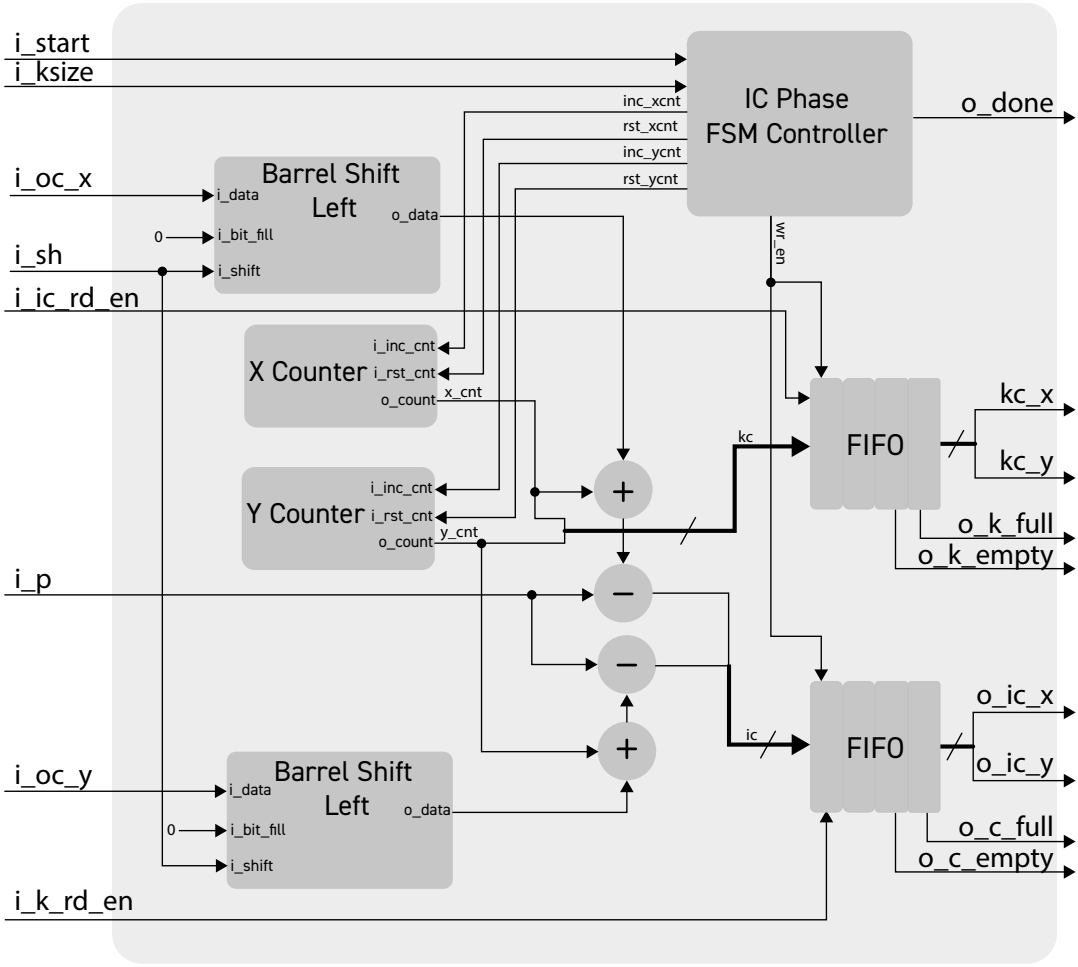



Figura 4.9: Diagrama de bloques del módulo Decodificador de Coordenadas de Entrada (*IC Phase*).

### IC Phase

El bloque *IC Phase*, a partir de la coordenada $(i_oc\_x, i\_oc\_y)$ del conjunto $\Phi$ y a las señales de *i_sh*, *i_p* e *i_ksize*, calcula los pares de coordenadas $(o\_ic\_x, o\_ic\_y)$ y $(kc\_x, kc\_y)$ correspondientes a las coordenadas de la matriz dispersa de entrada pertenecientes al conjunto $\Upsilon$ y al conjunto de coordenadas de la matriz de pesos cuyas características influyen en el valor de la característica de salida correspondiente a la coordenada $(i\_oc\_x, i\_oc\_y)$, conforme a la Ec. 3.4, que se adapta y reescribe en función a la terminología empleada en el diseño de la microarquitectura de hardware representada en la Fig. 4.9, de la siguiente manera:

$$(o\_ic\_x, o\_ic\_y) = (i\_oc\_x, i\_oc\_y) \times (2^{i\_sh}) + (x\_cnt - i\_p, y\_cnt - i\_p). \quad (4.2)$$

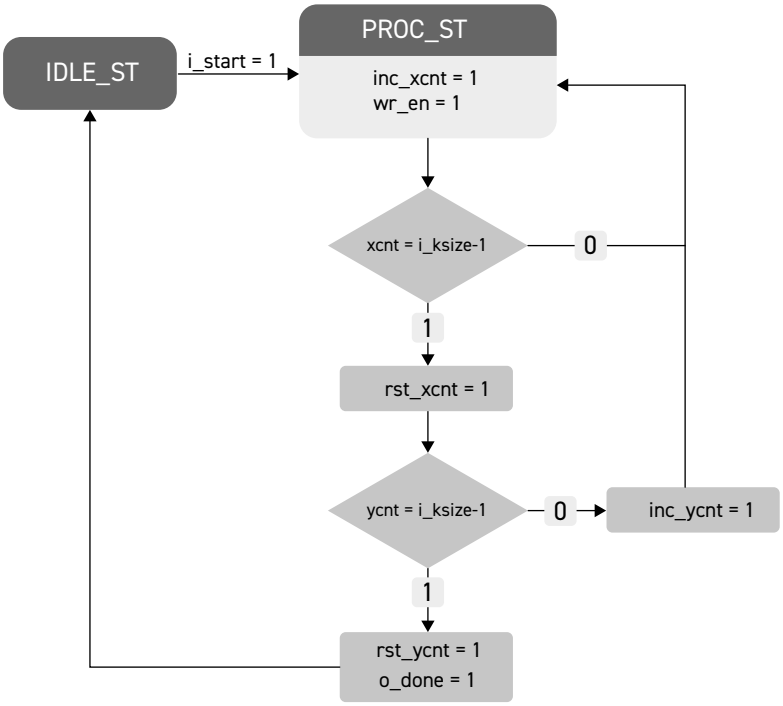



Figura 4.10: Máquina de estados del módulo Decodificador de Coordenadas de Entrada (*IC Phase*).

Al igual que ocurre con el módulo *OC Phase*, en este caso el parámetro de paso ($\sigma$ según la notación del Capítulo 3) se restringe a valores potencia de 2 para simplificar el hardware, que en el caso de la Ec. 4.2 se encuentra multiplicando, por lo que se incluyen dos *barrel shifters* a izquierda.

El mecanismo de operación de este bloque (Fig. 4.10) funciona de la siguiente manera: La máquina de estados se encuentra inicialmente en el estado IDLE_ST, con los contadores *X Counter* e *Y Counter* en estado de reset. Cuando recibe el flanco ascendente de la señal *i_start*, transiciona al estado de procesamiento PROC_ST. En este estado, se calcula el resultado de la primera coordenada utilizando los *barrel shifters* y los circuitos de suma y resta, almacenándose la coordenada resultante y el valor del contador en dos FIFOs. Se guardan tanto la coordenada de la matriz dispersa de entrada ($o\_ic\_x, o\_ic\_y$) como la coordenada de la matriz de pesos ($x\_cnt, y\_cnt$), necesarias para localizar en la memoria RAM la característica de la matriz dispersa de entrada y el peso correspondiente.

Es importante notar que tanto el módulo *OC Phase* como el módulo *IC Phase* deben iterar sobre todo el rango de coordenadas del kernel $(0,0) \leq (x\_cnt, y\_cnt) \leq (i\_ksize - 1, i\_ksize - 1)$. Por tal motivo, el tiempo de pro-

cesamiento de los conjuntos $\Phi$ y $\Upsilon$ es constante y depende del cuadrado del valor $i\_ksize$ (Ec. 4.3). Esto mismo ocurre dado que el decodificador de coordenadas no posee información espacial acerca de las coordenadas vecinas almacenadas en la memoria CAM, y en consecuencia no se puede predecir de antemano qué combinaciones de coordenadas producirán resultados válidos

$$T_{proc}[clks] = i\_ksize^2. \tag{4.3}$$

Este diseño asegura que todas las posibles combinaciones de coordenadas se consideren, garantizando así la precisión y completitud en el cálculo de las características de la matriz dispersa de salida, sin importar la distribución espacial de las coordenadas de la matriz dispersa de entrada.

### 4.1.4. Arreglo de Elementos de Procesamiento

El arreglo de elementos de procesamiento es el bloque encargado de realizar todas las operaciones de multiplicación y acumulación entre las características de la matriz dispersa de entrada y las características provenientes de la matriz de pesos, necesarias para procesar las características de la matriz dispersa de salida (Fig. 4.11). Este bloque consta de un arreglo de $N$ unidades de Multiplicación y Acumulación de 8 bits (MAC, por sus siglas en inglés), cuyas entradas se alimentan desde un banco de memorias de tipo FIFO. Cada unidad MAC incluye un multiplicador de 8 bits cuya salida se conecta a un sumador. Este sumador combina el producto obtenido con la suma parcial de operaciones previas, almacenada en un registro acumulador. Este registro puede inicializarse en un valor específico gracias a un multiplexor ubicado en su entrada, que permite seleccionar entre una fuente de datos externa o la salida del sumador, facilitando así la incorporación de un *bias* en la capa. Adicionalmente, cada unidad MAC dispone de una señal de habilitación controlada manualmente, la cual debe ser activada de forma independiente por el controlador general del sistema. Esta funcionalidad permite optimizar el consumo energético, ya que únicamente se activan las unidades MAC necesarias para el procesamiento, dejando las restantes inactivas cuando no son requeridas.

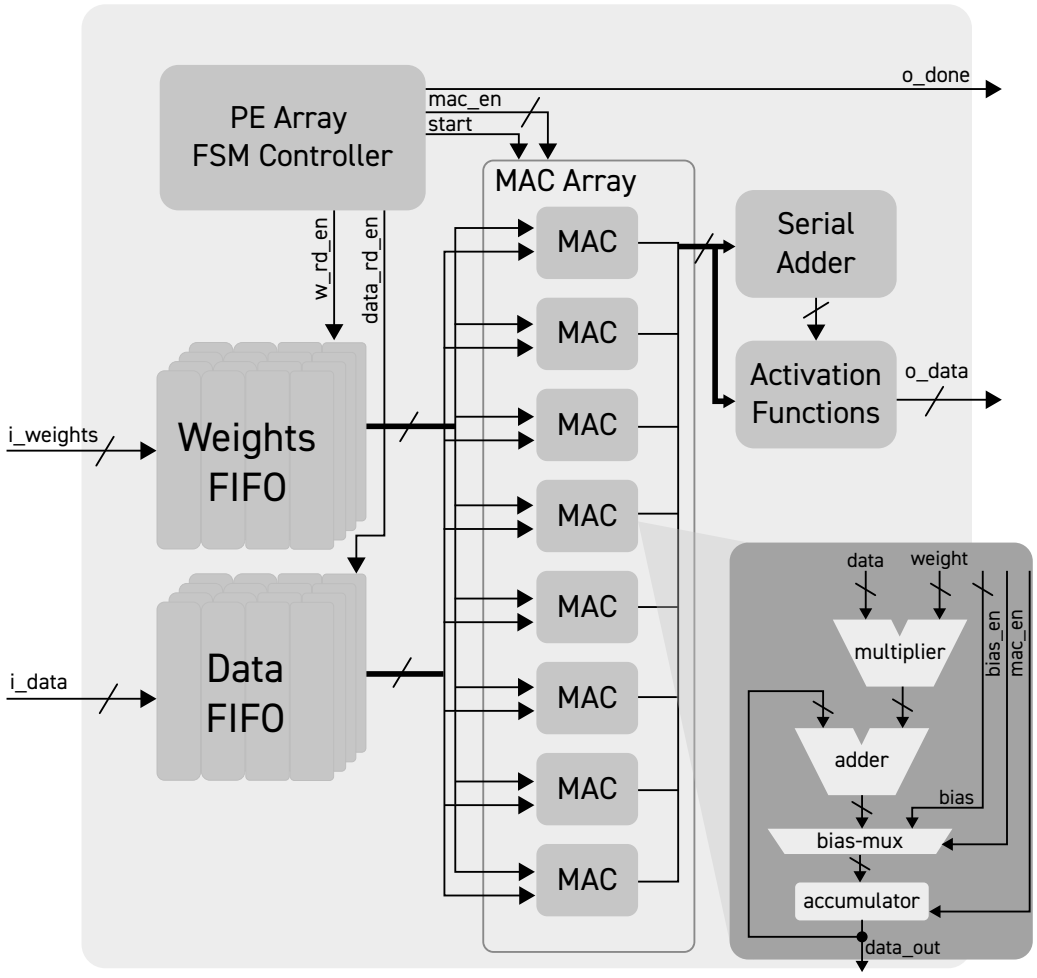



Figura 4.11: Diagrama en bloques del arreglo de elementos de procesamiento (*PE Array*).

Este módulo posee dos modos de operación. Las salidas de las unidades MAC se dirigen, por un lado, a un arreglo de N bloques que realizan operaciones de truncado, redondeo y funciones de activación, permitiendo que cada bloque opere de manera independiente sobre diferentes características de la matriz dispersa de salida (modo independiente), y por otro lado, a un sumador serie inspirado en [50] que acumula todas las salidas individuales en una misma característica de salida cuando la cantidad de canales en la matriz dispersa de entrada es muy numerosa, reduciendo así los tiempos de procesamiento (modo cooperativo). Esta última función resulta de particular utilidad en casos donde la cantidad de productos a acumular es múltiplo de la cantidad de unidades MAC disponibles. Si se dispone de N MACs, el sumador tardará N ciclos en combinar las salidas, y por lo tanto, si por ejemplo se necesitan procesar $N \times N$ características de entrada, éstas se pueden llevar a cabo utilizando las N unidades MAC disponibles durante N ciclos. Una vez transcurrido ese tiempo, los N resultados parciales son acumulados en otros N ciclos por el sumador serie, mientras las MAC procesan otro grupo de características, teniendo los resultados de $N \times N$ multiplicaciones y acumulaciones listos solamente cada $2 \times N$ ciclos. El ejemplo anterior ilustra el caso donde todas las unidades MAC disponibles están siendo usadas, lo que no siempre ocurre. Para estos casos, el sumador serie cuenta con una entrada de configuración que permite establecer el número de MACs a acumular. Estas están numeradas desde la 0 a la $N-1$, por lo que comenzando siempre desde la 0, se puede programar cuántas se desean utilizar. La cantidad de ciclos de reloj de procesamiento necesarios para esta configuración de unidades MAC trabajando en conjunto con el sumador serie puede modelarse matemáticamente a partir de este concepto de acuerdo con la si guiente ecuación:

$$T[clk] = \left\lceil \frac{n\_coords \times i\_ch}{N} + N \right\rceil \times o\_ch \tag{4.4}$$

donde el operador $\lceil\ \rceil$ denota la operación techo, garantizando una cantidad entera de ciclos de reloj $T[clk]$. Los términos $i\_ch$ y $o\_ch$ son parámetros de capa, y representan la cantidad de canales o características que van a estar asociadas a cada coordenada de entrada y salida respectivamente. Por otro lado, $n\_coords$ representa la cantidad de coordenadas a procesar para un determinado paso de

convolución. Si se estuviesen procesando matrices densas, $n\_coords$ tendría un valor constante e igual a $ksize^2$, pero como se están procesando matrices dispersas, cada paso de convolución va a procesar una cantidad de coordenadas distintas, que va a estar determinado por la distribución espacial de las coordenadas de la matriz dispersa de entrada asociadas a características no nulas, estando este número comprendido entre $1 \leq n\_coords \leq ksize^2$. Si en lugar de utilizar el sumador serie se opta por procesar cada característica de salida $o\_ch$ de manera independiente utilizando distintas unidades MAC, la Ec. 4.4 se simplifica de la siguiente manera:

$$T[clk] = n\_coords \times i\_ch \times \left\lceil \frac{o\_ch}{N} \right\rceil \tag{4.5}$$

donde en este caso los ciclos de reloj van a estar determinados principalmente por la cantidad de características a procesar para el paso de convolución en cuestión siempre y cuando hayan suficientes unidades MAC disponibles para procesar todas las características de salida $o\_ch$ en simultáneo. De no ser este el caso, se deberá reutilizar las unidades MAC tantas veces como indique el término $\lceil o\_ch/N \rceil$. De esto surge la idea intuitiva de que por un lado, si se tienen muchos canales de entrada y pocos canales de salida, resulta conveniente la utilización del sumador serie (modo cooperativo), dado que el proceso de cómputo se realiza en serie con respecto a los canales de salida, mientras que el procesamiento de características asociadas a una misma característica de salida se realiza de forma paralela. En este caso la máxima eficiencia se obtiene cuando la cantidad de características a procesar, es decir, el producto entre la cantidad de canales de entrada (características asociadas a una sola coordenada de entrada) y la cantidad de coordenadas por paso de convolución es múltiplo de la cantidad de unidades MAC disponibles. Esto puede verse claramente en la Fig. 4.12a, donde los "saltos" se producen para tales valores, denotando para este ejemplo con 8 unidades MAC y 6 canales de salida ($o\_ch = 6$), que tanto si se procesan 1 u 8 características, la cantidad de ciclos de reloj necesarios es la misma, y lo que baja o sube es la eficiencia. Si tomamos el rango de $i\_ch$ entre 1 y 8, se ve que en el caso donde los canales de salida $o\_ch$ trabajan en paralelo (modo independiente), la cantidad de ciclos escala de manera lineal, ya que es directamente proporcional al producto $n\_coords \times i\_ch$, lo que resulta más conveniente para $i\_ch < 8$. Si consideramos

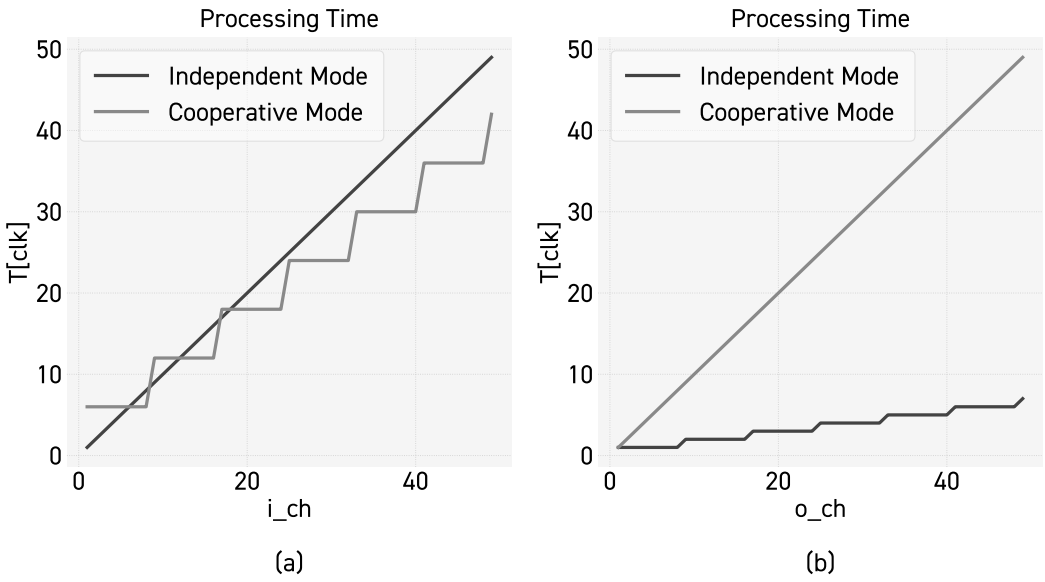



Figura 4.12: Comparación de tiempos de cómputo para un arreglo de 8 unidades MAC, asumiendo el procesamiento de las características de entrada de 1 elemento de la matriz dispersa de entrada ($n\_coords = 1$) para los casos donde se tiene (a) 6 canales de salida, es decir $o\_ch = 6$, y (b) 1 canal de entrada, es decir $i\_ch = 1$.

el caso de la Fig. 4.12b, donde la cantidad de canales de entrada se mantiene constante, y se evalúa el desempeño respecto a la cantidad de canales de salida, resulta evidente la conveniencia de la paralelización de canales de salida la cual va a ser aprovechada al máximo, cuando la cantidad de canales de salida sean múltiplos de la cantidad de unidades MAC disponibles.

Las memorias FIFO destinadas a almacenar elementos de la matriz de pesos cuentan con umbrales programables de FIFO llena y vacía capaces de disparar la máquina de estados que controla el proceso, habilitando el procesamiento de forma automática al cruzar dicho umbral. Además de dichos umbrales, la máquina de estados posee señales de habilitación que permiten, una vez que todos los datos presentes en las memorias FIFO hayan sido procesados, habilitar automáticamente las unidades de truncado y redondeo, y emitir una señal de interrupción al controlador del sistema indicando que los resultados están listos para ser almacenados. Además de la salida de interrupción al final de la cadena de procesamiento, la máquina de estados puede emitir interrupciones intermedias cuando las FIFO se encuentran vacías, es decir, inmediatamente después de haber terminado de procesar, y antes de ejecutar truncado y redondeo, y puede

emitir una interrupción al terminar la operación del sumador serie. La máquina de estados también puede deshabilitarse por completo, y mediante señales de habilitación manuales, se puede disparar cada etapa de procesamiento ciclo a ciclo en forma manual.

Las memorias FIFO destinadas a almacenar características de la matriz dispersa de entrada tienen dos funciones adicionales: replicado y re-lectura. La función de replicado permite escribir una serie de características de entrada distintas de manera secuencial en cada FIFO, mientras que en el puerto de lectura, que está conectado a los elementos de procesamiento, cada característica de 8 bits se replica en cada una de las entradas de características de los elementos de procesamiento gracias a un contador que multiplexa la salidas de las FIFOs habilitadas. Esta funcionalidad es crucial cuando los elementos de procesamiento operan de manera independiente con pesos provenientes de distintos filtros.

La función de re-lectura resulta de importancia cuando se desea procesar más cantidad de filtros que de unidades MAC disponibles en el arreglo de hardware. En este caso, se cargan las características de la matriz dispersa de entrada, se carga el primer set de filtros, y se realiza el procesamiento. Una vez finalizado, se guarda el resultado en memoria RAM, y se carga el segundo set de filtros. En lugar de volver a cargar el mismo set de entradas para procesar el segundo set de salidas, simplemente habilitando el bit de re-lectura, se resetean los punteros de lectura de las FIFOs de datos, permitiendo ahorrar tiempo de carga de datos redundantes, resultando en una mejora tanto en latencia como en energía.

## 4.2. Modelado de la operación del sistema

En la sección anterior se describió la microarquitectura de cada bloque funcional y su operación individual. En esta sección se presenta una arquitectura que integra dichos módulos, con el objetivo de caracterizar los algoritmos desarrollados en el Capítulo 3 en términos de área, consumo energético y desempeño. Existen diversas métricas para medir el grado de dispersión de una matriz [51]; dada la naturaleza de los algoritmos a implementar, se optó por utilizar la si-

guiente:

$$\ell = 1 - \frac{\#\{(i,j), a_{(i,j)} \neq 0\}}{H \times W}. \tag{4.6}$$

En esta ecuación, el numerador del segundo término representa la cantidad de coordenadas con valores no nulos, mientras que el denominador corresponde al total de coordenadas de la matriz. De esta manera, el valor de $\ell$ crece proporcionalmente al grado de dispersión, oscilando entre 0 y 1.

Entre los algoritmos presentados en el Capítulo 3, se selecciona el de convolución como caso de estudio, ya que presenta la mayor diversidad de operaciones y, con modificaciones mínimas, el modelo de sistema puede adaptarse a los demás algoritmos expuestos en dicho capítulo. Se parte de una matriz dispersa de entrada que consta de 1 canal de características y N elementos, es decir, N coordenadas que disponen de una característica asociada de un elemento, cuyas coordenadas se encuentran almacenadas en una memoria CAM, llámese CAM0, y sus características se almacenan en una memoria RAM asociada, llámese RAM0. En el Decodificador de Coordenadas se escriben los parámetros de la capa a procesar *stride*, *padding*, y tamaño de filtro *ksize*. El procesamiento comienza con una lectura secuencial del primer registro CAM0, del cual se obtiene una coordenada de la matriz dispersa de entrada $(x_i, y_i)$. Esta coordenada se ingresa al bloque Decodificador de Coordenadas, más específicamente al módulo *OC Phase*. Este módulo calcula en base a los parámetros de capa todas las coordenadas de la matriz dispersa de salida cuyas características se verán influenciadas por la característica de la matriz dispersa de entrada correspondiente a la coordenada $(x_i, y_i)$. Al cabo de $ksize^2$ ciclos de reloj, se lee la primer coordenada $(x_o, y_o)$ y se realiza la búsqueda de la misma en la CAM de salida, llámese CAM1. Si no se encuentra, se realiza una escritura secuencial de la misma, y se registra la dirección en la cual dicha coordenada fue guardada. A continuación se calculan todas las coordenadas de entrada que influyen sobre la coordenada de salida $(x_o, y_o)$ (lo que tradicionalmente se conoce como un paso de convolución) haciendo uso del módulo *IC Phase* del Decodificador de Coordenadas y nuevamente, al cabo de otros $ksize^2$ ciclos de reloj, se lee la primer coordenada. Se busca en la CAM0, y si se encuentra, se lee la dirección en la cual se almacena en memoria RAM, la característica asociada a dicha coordenada, y se la carga en

la FIFO de características del Arreglo de Elementos de Procesamiento. Se lee del banco de registros de doble puerto del bloque Decodificador de Coordenadas la dirección donde se encuentran almacenados los pesos necesarios, se los lee de la memoria RAM destinada a almacenar la matriz de pesos, y se los carga en la FIFO de parámetros del Arreglo de Elementos de Procesamiento. Se lee la siguiente coordenada del módulo *IC Phase*, y se repite el proceso antes descrito. Una vez que se leyeron y procesaron todas las coordenadas del módulo *IC Phase*, y se encuentran todas las características de entrada necesarias y sus respectivos pesos almacenados en el Arreglo de Elementos de Procesamiento, se habilita el procesamiento de dicho módulo. El tiempo de procesamiento va a depender directamente de la cantidad de canales de la matriz dispersa de entrada $i\_ch$, de la cantidad de canales de la matriz dispersa de salida que se intenta calcular $o\_ch$, de la cantidad de coordenadas de la matriz dispersa de entrada encontradas en la CAM0 $n\_coords$, y se modela como se vio en la sección anterior, de acuerdo a las Ec. 4.4 y 4.5. Una vez terminado el procesamiento y obtenido las características de salida correspondientes a la coordenada $(x_o, y_o)$, se guarda en la memoria RAM1 en la dirección asociada a $(x_o, y_o)$, y se procede a leer la siguiente coordenada de salida de la FIFO del módulo *OC Phase* del bloque Decodificador de Coordenadas. Este proceso se repite hasta que la FIFO de dicho módulo se vacía, y una vez que esto ocurre, se realiza una nueva lectura secuencial de la CAM0, y todo el proceso se repite hasta que todas las coordenadas de la CAM0, y sus características asociadas hayan sido procesadas, momento en el cual termina el procesamiento de la capa convolucional, y el conjunto CAM1-RAM1 pasa a servir de matriz dispersa de entrada para el procesamiento de la capa posterior.

De esta descripción se pueden concluir los siguientes dos aspectos: en primer lugar, la operación es inherentemente secuencial y en segundo lugar, como se desprende de la Ec. 4.3, cada ciclo de procesamiento de los bloques *IC Phase* y *OC Phase* tiene una duración constante y depende del cuadrado del tamaño del filtro, por lo que la cantidad de ciclos de reloj totales que se necesitan para procesar las coordenadas de una capa convolucional son:

$$T[clk] = \underbrace{i\_coords \times ksize^2}_{T_{OC}[clk]} + \underbrace{o\_coords \times ksize^2}_{T_{IC}[clk]} \tag{4.7}$$

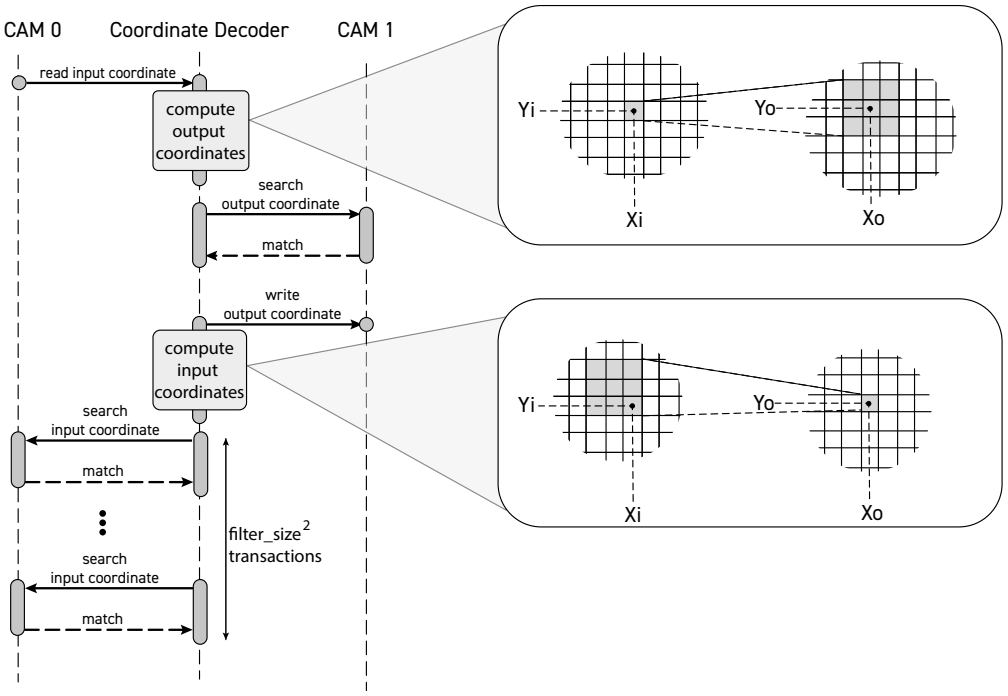



Figura 4.13: Flujo de procesamiento de las coordenadas, que ilustra la lectura secuencial de la memoria CAM.

donde *i_coords* representa la cantidad de coordenadas con características no nulas pertenecientes a la matriz dispersa de entrada, y *o_coords* representa el número de coordenadas que tendrá la matriz dispersa de salida una vez finalizado el procesamiento. El primer término de esta ecuación $T_{OC}[clk]$ representa la cantidad de ciclos de reloj de procesamiento que estará operando el módulo *OC Phase*, mientras que el segundo término $T_{IC}[clk]$, representa la cantidad de ciclos de reloj de procesamiento que estará operando el módulo *IC Phase*.

Esta operación puede acelerarse removiendo los contadores *X Counter* e *Y Counter* de los módulos *IC Phase* y *OC Phase* y replicar hardware para realizar el procesamiento de manera paralela. De esta forma, el factor $ksize^2$ desaparece. En la práctica, como la búsqueda de coordenadas en CAM es secuencial, el ahorro en ciclos de procesamiento se pierde de todas maneras en las operaciones de búsqueda, y en consecuencia no representa una mejora de rendimiento. Naturalmente, $T_{OC}[clk]$ se verá directamente afectado por el grado de dispersión de la matriz de entrada de la siguiente forma:

$$T_{OC}[clk] = (1 - \ell_i) \times (H_i \times W_i) \times ksize^2 \tag{4.8}$$

donde $\ell_i$ es el grado de dispersión de la matriz de entrada, y el producto $(H_i \times W_i)$ representa la cantidad de coordenadas totales. De manera similar, $T_{IC}[clk]$ se puede llevar también a la forma de la Ec. 4.8, pero en este caso depende del grado de dispersión de la matriz de **salida**:

$$T_{IC}[clk] = (1 - \ell_o) \times (H_o \times W_o) \times ksize^2. \tag{4.9}$$

Las dimensiones de la matriz de salida son función de las dimensiones de la matriz de entrada y los parámetros de capa, y pueden calcularse de manera cerrada de la siguiente forma:

$$\begin{aligned} H_o &= \left\lfloor \frac{H_i + 2 \times p - ksize}{\sigma} \right\rfloor \\ W_o &= \left\lfloor \frac{W_i + 2 \times p - ksize}{\sigma} \right\rfloor \end{aligned} \tag{4.10}$$

donde $\lfloor\ \rfloor$ es la función piso, $(H_i, W_i)$ y $(H_o, W_o)$ representan las dimensiones de la matriz de entrada y salida respectivamente, $p$ representa el *padding*, $ksize$ el tamaño de kernel, y $\sigma$ representa el *stride* de la capa a computar. Si no se aplica ninguna técnica especial para mantener el grado de dispersión como la descrita en [35], debido al efecto de dilatación de la operación de convolución, el grado de de dispersión $\ell_o$ de la matriz de salida será siempre menor que el grado de dispersión $\ell_i$ de la matriz de entrada. Sin embargo, cuantitativamente $\ell_o$ es función tanto del grado de dispersión de la matriz de entrada $\ell_i$ como de la **distribución espacial** de las características no nulas.

Para ilustrar este concepto y poder determinar de manera cuantitativa cómo repercuten estos factores en el desempeño del sistema, dado que las Ec. 4.7 a 4.10 permiten calcular de forma precisa los tiempos de procesamiento de coordenadas para un set de parámetros de capa y una matriz de entrada definida, se generan matrices aleatorias de tamaño fijo, en las cuales se manipula de manera independiente el grado de dispersión y la distribución espacial. Para ello se parte de una matriz inicialmente vacía, de dimensiones $(H_i, W_i)$, a la cual se le añaden $(1 - \ell_i) \times (H_i \times W_i)$ características no nulas en ubicaciones únicas, asignadas mediante una distribución de probabilidad que resulta de la combinación ponde-

rada de una distribución uniforme $\mathcal{U}[(0,0),(H_i,W_i)]$, y una distribución normal bivariada $\mathcal{N}(x,y;\mu,\mathbf{\Sigma})$, cuyo valor medio $\mu$ y matriz de covarianza $\Sigma$ están dadas por

$$\begin{aligned}\mu &= (H_i/2, W_i/2)\\ \Sigma &= \begin{bmatrix}(-2/5\rho + 1/2)\times H_i & 0\\ 0 & (-2/5\rho + 1/2)\times W_i\end{bmatrix}\end{aligned} \tag{4.11}$$

donde $0 \leq \rho \leq 1$ es un parámetro que permite modificar la distribución espacial mediante el ajuste del desvío típico de la distribución normal en el rango $[(H_i/10, W_i/10),(H_i/2, W_i/2)]$. Teniendo esto en cuenta, la distribución de probabilidades que resulta de la combinación de la distribución normal $\mathcal{N}$ y la distribución uniforme $\mathcal{U}$ resulta

$$\mathbb{P}(x,y) = e^{-20\rho}\times\mathcal{U} + \frac{e^{3\rho}-1}{e^3-1}\times\mathcal{N} \tag{4.12}$$

donde $\mathbb{P}(x,y)$ es la probabilidad de que la coordenada $(x,y)$ tenga características no nulas, $\mathcal{U}$ es la probabilidad de la coordenada $(x,y)$ asumiendo una distribución uniforme, $\mathcal{N}$ es la probabilidad de la coordenada $(x,y)$ asumiendo una distribución normal, y $0 \leq \rho \leq 1$ es un factor de peso que permite ajustar la distribución de probabilidades de manera que las coordenadas se agrupen en *clusters* para valores de $\rho \approx 1$ gracias a la distribución normal, o se dispersan de manera equiprobable para valores de $\rho \approx 0$ (Fig. 4.14).

Partiendo de esta base, el desempeño del sistema medido en términos de consumo de energía, puede estimarse a partir de conocer el tiempo que demora en ejecutarse cada operación, la cantidad de operaciones, y la potencia consumida por cada módulo interviniente en el cómputo de la capa convolucional, dada una muestra puntual. Para ello, se implementa la microarquitectura de los bloques descritos en la sección 4.1 mediante lenguaje de descripción de hardware, y se sintetiza utilizando una librería de celdas estándar de 65nm. Posteriormente, a partir del netlist post-síntesis generado, se realizan simulaciones con datos aleatorios, y se obtienen las transiciones de cada nodo del circuito. Posteriormente, a partir de esta información, la herramienta de síntesis permite estimar el consumo

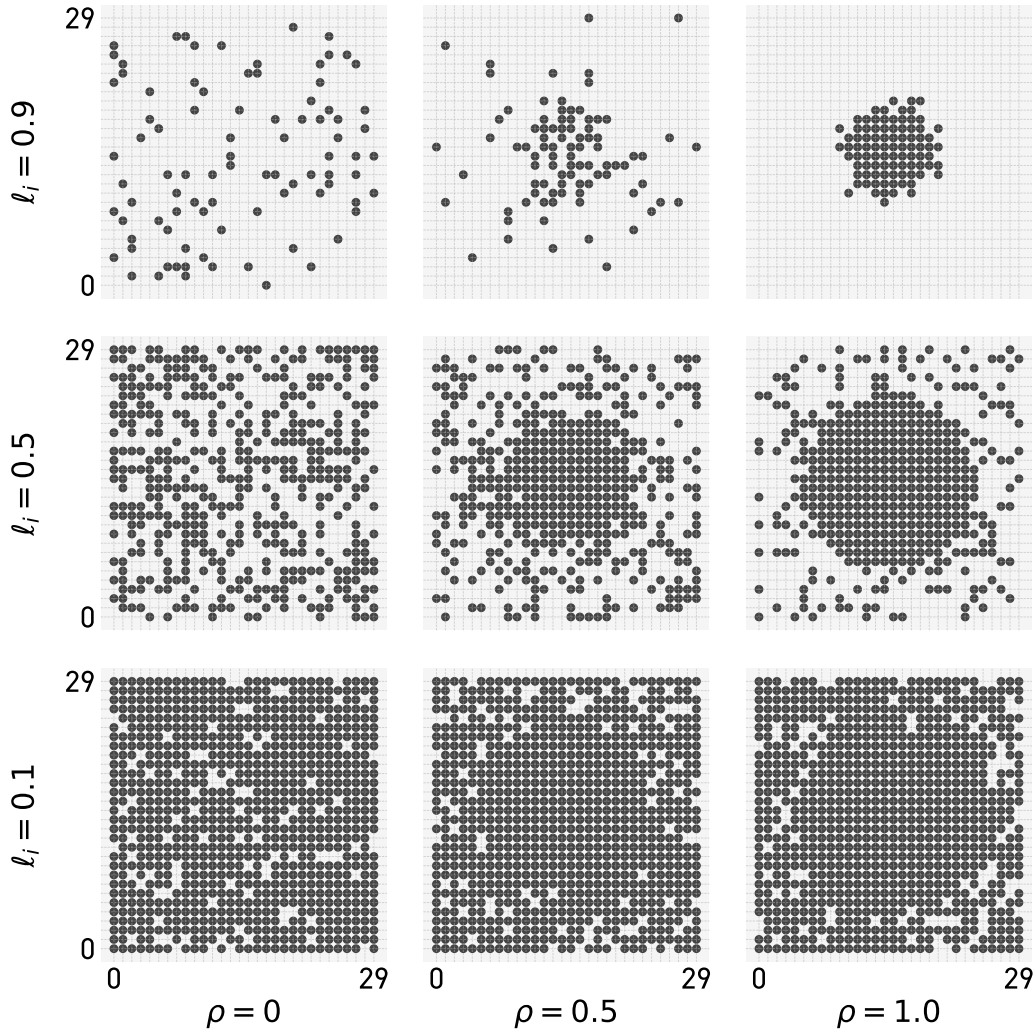



Figura 4.14: Ejemplo de matrices dispersas de $30 \times 30$ con grados de dispersión variando entre $0{,}1 \leq \ell \leq 0{,}9$ y distribución espacial variando entre $0 \leq \rho \leq 1$.

de potencia del circuito mediante la siguiente ecuación

$$\mathbb{P} = \sum P_{cell} = \sum leak_{cell} + \left( f_{clk} \times \sum \overline{p\_dyn_{cell}} \times \alpha_{cell} \right) \tag{4.13}$$

donde $\mathbb{P}$ es la potencia total consumida por el circuito, $P_{cell}$ es la energía consumida por cada compuerta lógica, $leak_{cell}$ es la potencia de *leakage* de cada celda, $\overline{p\_dyn_{cell}}$ es la potencia dinámica promedio de cada celda normalizada en función de la frecuencia, y $\alpha_{cell}$ es un coeficiente adimensional que representa la cantidad de transiciones por ciclo de reloj de cada compuerta lógica, y es obtenido a partir de la información extraída de la simulación. A partir de el netlist post-síntesis generado y la información obtenida de la simulación, se obtienen los resultados de la Tabla 4.2.

Tabla 4.2: Bloques de cómputo y potencia post-síntesis, reportada para una tecnología de 65nm.

| **Bloque de cómputo** | **Potencia [mW]** | **Frecuencia de operación [MHz]** |
|---|---|---|
| ***OC Phase*** | 0.1656 | 100. |
| ***IC Phase*** | 0.1194 | 100. |
| ***PE Array*** | 0.6435 | 100. |

Para la memoria CAM se consideran dos escenarios. Por un lado, se desarrolla una implementación en lenguaje de descripción de hardware, lo cual resulta útil tanto para sistemas implementados en FPGA como en ASIC cuando no se dispone de bloques CAM dedicados. Por otro lado, se analizan los consumos asociados a una implementación dedicada en tecnología de 65 nm. En el caso de las operaciones de búsqueda se han adoptado los valores de consumo energético reportados en [52], en los que se realizó una implementación dedicada en tecnología de 65 nm. Mientras tanto, para las operaciones de lectura y escritura se utilizan los consumos energéticos indicados en [53] para una memoria SRAM dedicada, ya que las celdas CAM incorporan una celda SRAM para almacenar cada bit y, generalmente, las publicaciones referentes a memorias CAM no reportan estos consumos, dado que la operación de búsqueda es la más relevante y demandante en términos energéticos. Estos valores se resumen en la Tabla 4.3.

A partir del Algoritmo 1 se puede desarrollar un modelo que permita estimar

Tabla 4.3: Consumo energético por operación para una memoria CAM sintetizada y una implementación dedicada de 18 kbit en tecnología 65 nm, normalizados a 100 MHz.

| **Operación** | **Energía [pJ]** | | **Frecuencia de operación [MHz]** |
|---|---|---|---|
| | **Síntesis** | **Dedicada** | |
| **Búsqueda** | 369 | 70.9 | 100 |
| **Lectura** | 3.63 | 3.9 | 100 |
| **Escritura** | 45.83 | 4.9 | 100 |

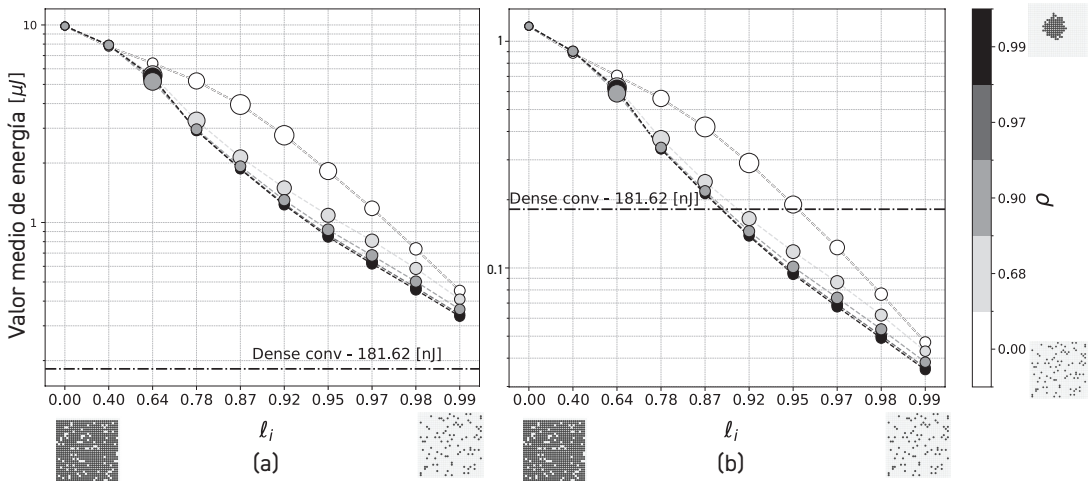



Figura 4.15: Consumo de energía de una capa convolucional de $ksize = 3$, $p = 0$, $\sigma = 1$ para una matriz de entrada de $30 \times 30$ con grados de dispersión $\ell_i$ y distribución espacial $\rho$ variando entre 0 y 0,99 para una arquitectura con (a) memorias CAM sintetizadas y (b) con bloques CAM dedicados, para una tecnología de 65 nm.

la cantidad de operaciones a realizar de cada tipo, dada una matriz dispersa de entrada y un conjunto de parámetros de capa definidos. Con estos datos, conociendo los consumos energéticos de cada operación involucrada se puede estimar el consumo total que tendrá la ejecución del algoritmo de convolución dispersa (Fig. 4.15). Partiendo de este modelo y utilizando como entrada matrices aleatorias de $30 \times 30$ generadas a partir de la distribución de probabilidades definida en la Ec. 4.12, en la Fig. 4.15 se estima el consumo de energía mediante una simulación de Montecarlo. Se exploran distintas combinaciones del grado de dispersión $\ell_i$ y la distribución espacial $\rho$ para un conjunto definido de parámetros de capa, comparando estos consumos con la energía requerida para computar dicha capa mediante el método tradicional (que involucra todas las operaciones de multiplicación y acumulación para obtener la matriz densa).

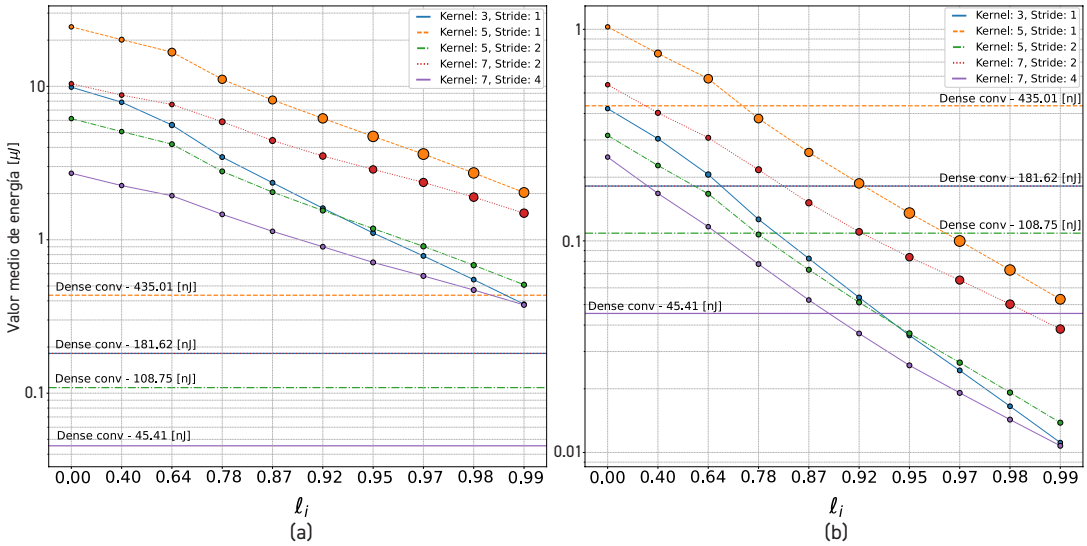



Figura 4.16: Consumo de energía de una capa convolucional para distintas configuraciones de capa considerando una matriz de entrada de $30 \times 30$ con grados de dispersión $0 \leq \ell_i \leq 0{,}99$ para una arquitectura con (a) memorias CAM sintetizadas y (b) con bloques CAM dedicados, para una tecnología de 65 nm.

Cada punto de la gráfica se obtiene a partir de 20 000 muestras; el tamaño del punto refleja el desvío típico, mientras que su tono se asocia con la distribución espacial $\rho$, conforme a la escala indicada a la derecha de la figura. Se observa que, al aproximarse la distribución espacial a una distribución normal (valores de $\rho \approx 1$), el consumo de energía disminuye debido a un menor efecto de dilatación, lo que resulta en una matriz de salida más dispersa en comparación con valores de $\rho \approx 0$.

Dado que el algoritmo depende en gran medida de las operaciones de búsqueda, la eficiencia energética de las memorias CAM es crucial para el desempeño del sistema. En una arquitectura que utiliza memorias CAM sintetizadas (Fig. 4.15a), para una matriz de $30 \times 30$ la convolución densa resulta más eficiente, independientemente del grado de dispersión y de la distribución espacial. En contraste, al sustituir las memorias CAM por bloques dedicados (Fig. 4.15b), la convolución dispersa se vuelve más eficiente que la densa siempre que el grado de dispersión supere aproximadamente 0,9. Esta diferencia se incrementa hasta casi un orden de magnitud para matrices en las que el porcentaje de píxeles con valores no nulos es cercano al 1 % (es decir, $\ell_i \approx 0{,}99$).

Para facilitar la comparación del consumo energético en diferentes configuraciones de capa, en la Fig. 4.16 se han promediado las curvas correspondientes

a diversas distribuciones espaciales para mejorar la claridad del gráfico. Es importante destacar que, al igual que en la Fig. 4.15, para matrices de tamaño $30 \times 30$ la arquitectura con memorias CAM sintetizadas (Fig. 4.16a) nunca supera a la convolución densa, sin importar el grado de dispersión. En contraste, en la arquitectura con bloques de memoria CAM dedicados se observa un punto de cruce —el umbral a partir del cual el algoritmo de convolución dispersa resulta energéticamente más eficiente que el algoritmo de convolución densa tradicional— que varía según el grado de dispersión, dependiendo de la combinación de $ksize$ y $\sigma$.

## 4.3. Conclusiones

En este Capítulo se identificaron las operaciones fundamentales involucradas en el cálculo de los algoritmos presentados en el Capítulo 3 y se diseñaron bloques de hardware específicos para su implementación. Estos bloques se interconectaron mediante un bus, constituyendo el sistema mínimo necesario para ejecutar los algoritmos de integración de eventos y convolución. Además, se desarrolló un algoritmo para generar matrices dispersas con distintos grados de dispersión y distribución espacial, lo que permitió modelar la interacción entre los bloques de hardware y sus tiempos de cómputo, y así comprender la relación entre las características de la matriz de entrada (como la cantidad de coordenadas no nulas, su distribución espacial, el número de canales y los parámetros de capa —por ejemplo, el tamaño del filtro $ksize$, el stride y el padding—) y el rendimiento del sistema. Utilizando como referencia una tecnología de 65 nm, se estimaron los consumos energéticos de cada operación involucrada en el cálculo del algoritmo de convolución dispersa. Los resultados revelaron que la eficiencia energética del algoritmo disperso depende en gran medida de las operaciones de búsqueda, siendo determinante una implementación eficiente de los bloques CAM.

# Capítulo 5

# DigineuronV3b - Acelerador basado en CAM

En el capítulo anterior se identificaron y modelaron los bloques de hardware esenciales para la ejecución de los algoritmos expuestos en el Capítulo 3, caracterizando cada operación en términos de tiempo de ejecución y consumo energético para una tecnología de 65 nm; a partir de esta caracterización, se desglosó el Algoritmo 1 en dichas operaciones para estimar su consumo energético total y, finalmente, se comparó este valor con la energía requerida por la implementación de una convolución densa tradicional, lo que permitió evaluar y contrastar la eficiencia energética de ambas aproximaciones.

El nombre *DIGINEURON* proviene de un proyecto homónimo iniciado en 2021 en los laboratorios de *Silicon Austria Labs*. Su objetivo original fue crear una plataforma que permitiera el prototipado rápido de aceleradores para redes neuronales. Dado que estos aceleradores requieren un flujo continuo de datos y mecanismos de configuración y control, resultaba imprescindible desarrollar una arquitectura modular y flexible que facilitara su integración con el entorno. Con este fin, se diseñó un sistema basado en un microprocesador y periféricos de propósito general, comunicados con los aceleradores a través de un bus AMBA[1] AHB[2]-Lite configurable. Se desarrollaron varias iteraciones que fueron fabricadas en tecnología de 65nm, las cuales fueron incrementando en complejidad y

[1] *Advanced Microcontroller Bus Architecture* (Arquitectura avanzada de Controladores de Bus)

[2] *Advanced High-performance Bus* (Bus avanzado de alto rendimiento)

prestaciones.

La versión inicial [22], DigineuronV1 (Fig. 5.1), incorporó un acelerador convolucional optimizado para filtros de tamaño $3 \times 3$ y un acelerador simplicial simétrico. Ambos módulos se interconectaron mediante un bus AMBA AHB-Lite y quedaron bajo el control de un procesador ARM Cortex-M0+ de bajo consumo. Para completar la plataforma, se añadieron periféricos genéricos —GPIO, UART, temporizadores y un watchdog— procedentes del kit Corstone de ARM, accesible gracias al programa ARM Academic Access. Este acceso también permitió emplear librerías de celdas estándar y memorias SRAM durante el tapeout. Adicionalmente, se diseñaron dos bloques de hardware fundamentales: un controlador QSPI para gestionar memorias externas y el módulo spi2ahb, que brinda capacidades de depuración mediante un puerto SPI, facilitando la lectura y escritura en cualquier dirección de la memoria del ASIC a través del bus AHB-Lite. En la

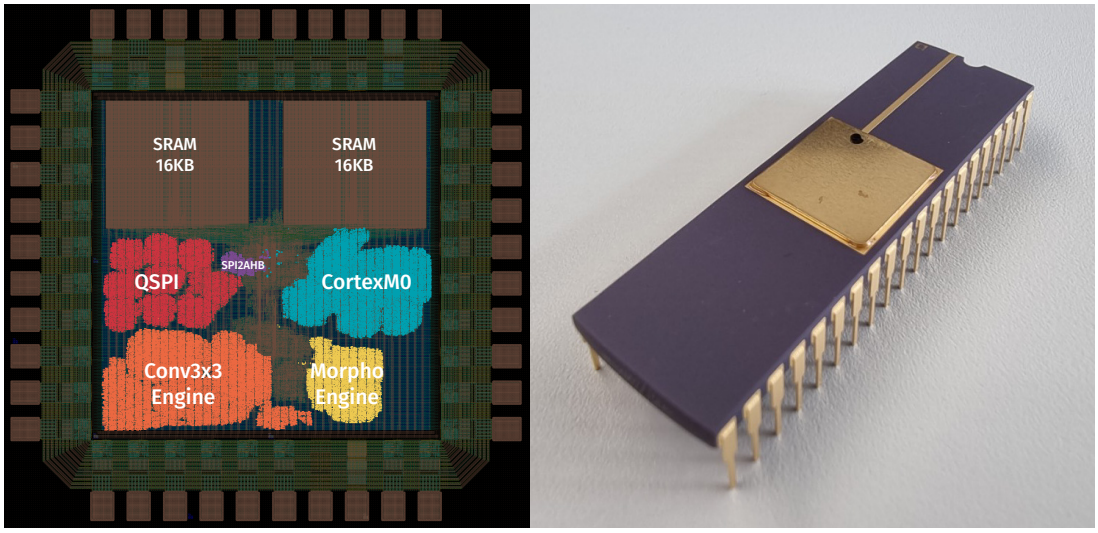



Figura 5.1: Digineuron V1. (a) Layout del circuito integrado, (b) ASIC recibido en un encapsulado DIL 40.

segunda versión [23], DigineuronV2 (Fig. 5.2), se sustituyó el ARM Cortex-M0+ por el procesador open-source RISC-V cv32e40p, desarrollado por el *OpenHW Group*, y se integró un controlador DMA para optimizar las transferencias de datos sin intervención directa del microprocesador.

La tercera generación mantiene el mismo espíritu de mejora continua de la infraestructura básica y se presenta en dos variantes. DigineuronV3a revisa y perfecciona el acelerador simplicial simétrico de las versiones anteriores, mientras

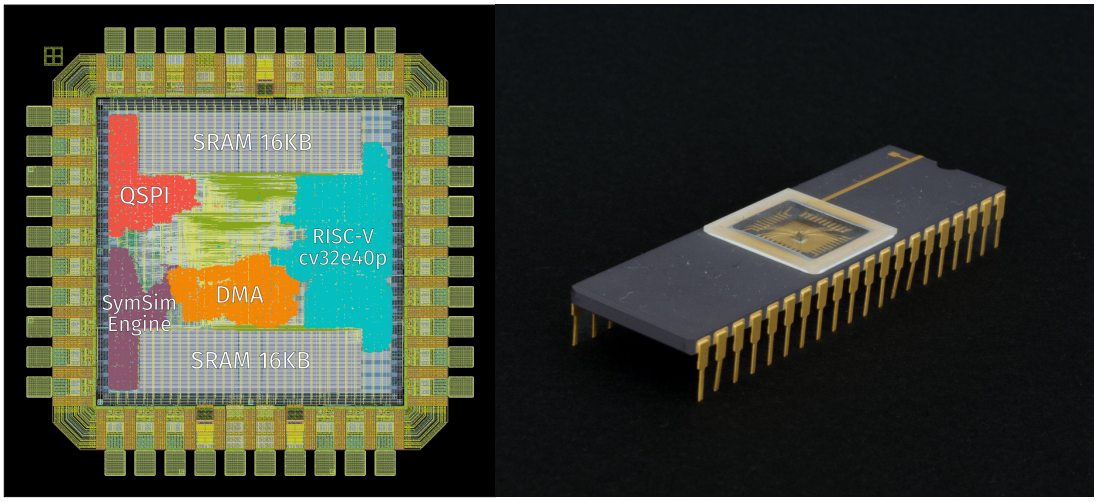



Figura 5.2: Digineuron V2. (a) Layout del circuito integrado, (b) ASIC recibido en un encapsulado DIL 40.

que DigineuronV3b —objeto de este capítulo— aprovecha los cimientos previos para desplegar una arquitectura de hardware aún más flexible. Esta última versión incluye los bloques de procesamiento necesarios para ejecutar los algoritmos propuestos en esta tesis, y abre la puerta a la exploración de nuevas técnicas de procesamiento más allá del alcance de este trabajo.

## 5.1. Arquitectura

La implementación de una arquitectura para adquisición y procesamiento de eventos en tiempo real tal y como se plantea en el Capítulo 3 conlleva el desafío de integrar los eventos entrantes mientras en simultáneo se realiza el procesamiento de la red en base a la matriz dispersa de la ventana de integración del instante anterior. Este proceso, si bien es perfectamente realizable, requiere contar con controladores independientes tanto para la operación de integración, como para el procesamiento de las capas de la red neuronal, y en simultáneo, un CPU que supervise y sincronice la operación de ambos controladores. Este enfoque resulta muy eficiente cuando se desea implementar una red neuronal en particular, entrenada para realizar una tarea específica; es poco apropiado si se busca una arquitectura flexible capaz de ejecutar distintos algoritmos. Por estas razones se optó por sacrificar el funcionamiento en tiempo real y utilizar

un único controlador: el procesador RISC-V open source **cv32e40p**, ya probado en el **DigineuronV2**. El procesador RISC-V de 32 bits gobierna, por medio de un bus AMBA AHB-Lite de 64 bits, un sistema que integra diversos periféricos, entre ellos dos memorias CAM de 1024 entradas cada una, implementadas según la microarquitectura expuesta en la Sección 4.1.1.

Para estas CAM se fijó un ancho de palabra de **18 bits** dedicado a la dirección—9 bits para la coordenada $y$ y 9 bits para la coordenada $x$—más **6 bits** de banderas, alcanzando así un total de **24 bits (3 bytes)**. La elección responde a las características de la cámara por eventos **DAVIS346 Color** disponible en el laboratorio, cuyo sensor contiene $260 \times 346$ píxeles.

La implementación de un arreglo CAM convencional basado en celdas de 10 transistores [48] implicaría una complejidad considerable; por ello, se aceptó un mayor consumo de área y energía y se optó por un diseño con celdas estándar, en el que cada celda combina un registro de 1 bit con una compuerta XOR.

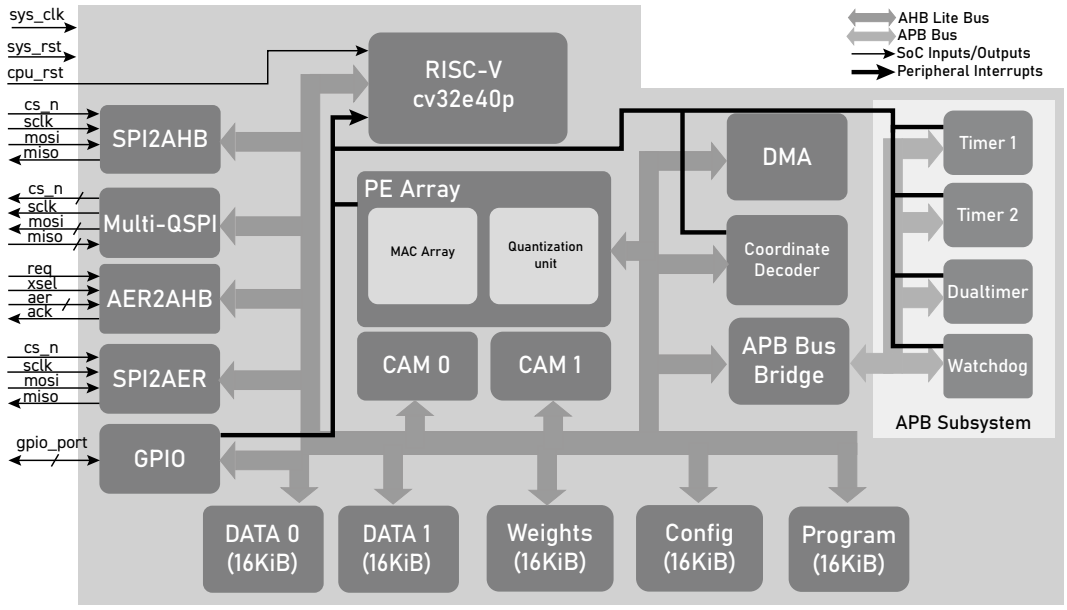



Figura 5.3: Diagrama en bloques de la arquitectura implementada en el ASIC DigineuronV3b.

Por otro lado, el diseño incorpora un decodificador de coordenadas capaz de calcular las coordenadas de las matrices dispersas de entrada y salida para capas convolucionales y de pooling con filtros de tamaño entre $1 \times 1$ y $7 \times 7$, implementado según la microarquitectura definida en la Sección 4.1.3. En esta implementación, para soportar filtros de hasta $7 \times 7$, las FIFOs se diseñaron con

un ancho de palabra de 6 bits y 49 registros, y el banco de registros de doble puerto, al almacenar direcciones, se diseñó con un ancho de palabra de 32 bits y 49 registros. Las señales de control y datos se mapean a registros que se acceden mediante un puerto AHB-Lite de tipo periférico de 64 bits.

Se ha incorpora un arreglo de 8 unidades de procesamiento MAC de 8 bits (**PE Array** en la Fig. 5.3), implementado según la microarquitectura definida en la Sección 4.1.4. Cada unidad MAC posee un registro de acumulación de 22 bits que permite almacenar los 16 bits resultantes de la operación de multiplicación, y acumular hasta 64 sumas parciales. Las salidas de estas 8 unidades MAC, alimentan por un lado a las unidades de truncado y redondeo que reducen los 22 bits de entrada a 8 bits de salida, y por otro lado a un sumador serie de 22 bits que acumula estas 8 palabras en un registro de 25 bits conectado a una unidad de truncado y redondeo dedicada, que al igual que las anteriores, reduce el resultado a 8 bits. Este bloque, al igual que el decodificador de coordenadas, tiene sus señales de control, y datos mapeadas a registros, que son accedidas mediante un puerto AHB-Lite periférico de 64 bits.

El sistema posee dos bloques de adquisición de eventos. El bloque de adquisición principal, denominado `aer2ahb`, implementa la recepción de eventos mediante el protocolo AER siguiendo la microarquitectura descrita en 4.1.2. Este bloque se diseñó de manera que sea compatible con el puerto AER presente en la DAVIS346 Color, por lo que la FIFO se diseñó con un ancho de palabra de 19 bits, para poder almacenar los 18 bits de coordenada, más 1 bit extra para almacenar la polaridad del evento. En cuanto a la cantidad de registros presentes en la FIFO, dada la tasa de entrada de 1 mega evento por segundo (1 MEPS), que el fabricante estipula, se determinó mediante simulación, que 256 registros FIFO son suficientes para que el procesador trabajando a una frecuencia de 50 MHz pueda leer y actualizar la matriz dispersa almacenada en CAM-RAM, sin que ésta se llene y comience a rechazar transacciones. Al igual que los periféricos anteriores, este bloque mapea sus señales de control y datos a un puerto AHB-Lite periférico de 64 bits.

El bloque de adquisición secundario (`spi2aer` en la Fig. 5.3) incorpora un puerto de entrada de eventos que sigue el protocolo SPI y se añade por dos mo-

tivos complementarios: primero, para proporcionar redundancia en caso de que el bloque principal falle, y segundo, para dotar al sistema de mayor flexibilidad. Esta última se aprovecha cuando la **DAVIS346 Color**, además de su puerto AER, se conecta por USB a una PC mediante la API en Python que acompaña a la cámara; tal conexión permite obtener los eventos por USB, aplicarles algoritmos de preprocesamiento en la PC y reinyectarlos en el ASIC a través de una FPGA[3] que interconecta ambos dispositivos mediante SPI. El módulo `spi2aer` recibe y decodifica las señales en *modo 1*[4], resuelve la sincronización entre el dominio de reloj del SPI y el del sistema, y alberga cuatro registros de configuración de 8 bits, accesibles exclusivamente con el reloj SPI, útiles para verificar la lógica de recepción y detectar posibles problemas en el cruce de dominios. Cada evento recibido se almacena inmediatamente en una FIFO de 256 entradas de 19 bits. Las transacciones SPI, de longitud arbitraria, comparten una estructura común compuesta por un *opcode* de 2 bits (se admiten tres operaciones) seguido de los datos: en las operaciones de lectura o escritura de registros, se transmiten a continuación los cuatro bytes del registro correspondiente, mientras que en la operación de escritura de eventos se envía primero la coordenada $y$ y, seguidamente, todas las coordenadas $x$ de los eventos generados en esa misma fila, con el fin de maximizar el número de eventos por transacción.

El sistema además incorpora bloques que no son específicos para el procesamiento de eventos y matrices dispersas, pero que son esenciales para el manejo de datos e interacción con el exterior, entre los cuales se encuentran un bloque GPIO (del inglés *General Purpose Input Output*) de 32 entradas-salidas. Este bloque se compone simplemente de un banco de registros que configuran la dirección de cada pin (entrada o salida), recibe o envía señales digitales con baja velocidad, y proveen interrupciones independientes al procesador al recibir un nuevo dato. Este periférico resulta particularmente útil cuando por ejemplo se necesita señalizar instantes de tiempo donde el sistema se encuentra realizando operaciones de interés.

Por otro lado, cuenta con un subsistema AMBA APB (del inglés *Amba Pe-*

[3] *Field Programmable Gate Array*

[4] Modo 1: `CPOL`=0, `CPHA`=1; las señales se muestrean en el flanco descendente y se actualizan en el flanco ascendente del reloj SPI.

*ripheral Bus*) de menor velocidad que el bus AHB principal, que cuenta con 2 timers un dualtimer y un watchdog. Los timers esencialmente permiten ejecutar interrupciones programadas al microprocesador. Es decir, son contadores programables que al cabo de una cierta cantidad de ciclos de reloj determinada por un registro de configuración, generan una interrupción para que el microprocesador ejecute cierta subrutina, como por ejemplo cesar la adquisición de eventos al cabo de cierto tiempo. Por otro lado, el watchdog permite resetear el sistema de manera automática si por alguna razón éste entra en estado de atasco o bloqueo (*stall*).

El sistema cuenta también con un módulo DMA[5] cuyo propósito es mover bloques de datos dentro y fuera del sistema. Posee 2 puertos AMBA AHB controladores de 64 bits, de los cuales uno es de escritura y el otro de lectura de manera de poder paralelizar ambos tipos de transacciones, y un puerto de configuración de tipo periférico, mapeado en el subsistema APB. Este bloque DMA acepta transacciones disparadas por software mediante el microprocesador, pero además incorpora señales hardware de request provenientes de otros periféricos, tales como el Multi-QPSI controlador (Multi Quad SPI) que también se incorpora.

Este periférico Multi-QPSI, está compuesto por 8 módulos QPSI controladores, independientes y altamente configurables, que comparten el mismo puerto AMBA AHB-Lite periférico. El objetivo de este módulo es poder aprovechar al máximo el bus de 64 bits del que se dispone. Por ende, cada módulo QSPI consta de dos FIFOs de 64 registros de 8 bits cada una, las cuales se conectan a las señales de datos provenientes de un solo Byte del Bus, lo que permite enviar o recibir datos a una tasa máxima de 8 Bytes cada 2 ciclos de reloj SPI.

El sistema incorpora 80 KiB de memoria SRAM repartidos en 5 memorias de tamaño 16KiB, con 2048 registros de 64 bits cada una, que pueden ser accedidas de manera independiente. Estas memorias, sin embargo, tienen funciones específicas asociadas. La primera es la memoria de programa o instrucciones, que es donde se almacena el *firmware* que ejecutará el microprocesador. La segunda es una memoria dedicada a parámetros de configuración, como pueden ser

[5] *Direct Memory Access*

los parámetros de capa de la red neuronal, o distintas configuraciones de transacciones DMA. La tercer y cuarta memoria denominadas DATA0 y DATA1, asociadas a las CAM0 y CAM1 respectivamente, tienen como funcionalidad almacenar las características asociadas a las matrices dispersas de entrada y salida, cuyas coordenadas se almacenan en las CAM0 y CAM1. Por último, la quinta memoria almacena los pesos o filtros de las capas de la red neuronal a procesar. Estas memorias se mapean en direcciones de memoria consecutivas, de manera de flexibilizar los tamaños. Por ejemplo, si de pronto el programa necesita más cantidad de memoria que los 16 KiB destinados a tal fin, y la memoria de configuración no se utiliza en su totalidad, el programa podría extenderse sin problema alguno, ocupando parte de la memoria de configuración.

El microprocesador cv32e40p dispone de 32 entradas de interrupción según estipula el set de insctrucciones (ISA[6]) RISC-V, de las cuales solamente 18 se encuentran disponibles para su uso, y tienen un orden de prioridad específico en el cual son atendidas. Por este motivo, y siendo que la arquitectura del sistema a implementar cuenta con múltiples entradas de interrupción que superan con creces las entradas disponibles, se diseñó un controlador de interrupciones que permite mapear en "páginas" estas interrupciones y así flexibilizar los periféricos que el procesador ve en sus entradas de interrupción y el orden de prioridad, en función de la aplicación a ejecutar.

El diseño de la arquitectura no incorpora una memoria ROM con instrucciones que indiquen al microprocesador cómo inicializar el sistema y los periféricos, por lo tanto se incorpora un bloque adicional denominado `spi2ahb` el cual permite, mediante un puerto spi periférico, acceder al Bus AMBA AHB-Lite del sistema, y así poder leer y escribir cualquier dirección de memoria interna del ASIC. Por tal motivo, este bloque es el encargado de escribir el programa que será ejecutado por el microprocesador, enviar la configuración de la red neuronal a ejecutar, y configurar el bus para evitar colisiones entre transacciones emitidas por los distintos puertos AHB controladores que posee el sistema. Esto se logra a través de bloques de hardware conocidos como árbitros. Cada puerto AHB periférico, posee un árbitro que otorga el acceso a transacciones provenientes de un

[6] *Instruction Set Architecture*

puerto controlador u otro, en función de la prioridad que cada puerto controlador tiene sobre este puerto periférico en específico. Estos niveles de prioridad están determinados por un banco de registros de configuración que son accedidos por el bloque spi2ahb a través de un puerto AHB periférico dedicado. Este banco de registros, con su respectivo puerto conforma un bloque definido como **gconf** (del inglés *General Configuration*). Para poder garantizar la correcta configuración de las prioridades de acceso y poder escribir el programa, el microprocesador dispone de una señal de reset independiente a la del resto del sistema. Esto garantiza que el spi2ahb pueda operar el sistema sin riesgo de colisión o intervención alguna por parte del microprocesador.

Con todos los dimensionamientos antes mencionados, y empleando una tecnología de 65nm de la empresa taiwanesa TSMC[7], se sintetizó e implementó el sistema, obteniendo las dimensiones de la Tabla 5.1. La Fig. 5.4 resume la in-

Tabla 5.1: Resultados de área post síntesis de cada bloque.

| **Bloque** | **Area** $[\mu m^2]$ | **Cantidad de Celdas** |
|---|---|---|
| **CAM 0** | 467 974 | 112 736 |
| **CAM 1** | 467 974 | 112 736 |
| **AER2AHB** | 47 823 | 9 682 |
| **SPI2AER** | 47 939 | 9 804 |
| **COORDINATES DECODER** | 49 140 | 10 328 |
| **PE ARRAY** | 52 265 | 13 672 |
| **APB SUBSYSTEM** | 9 158 | 2 445 |
| **AHB MATRIX** | 37 294 | 13 163 |
| **CPU** | 68 604 | 20 311 |
| **GCONF** | 17 074 | 3 781 |
| **SRAM (Total)** | 517 162 | 2 105 |
| **MQSPI** | 102 375 | 23 907 |
| **DMA** | 28 811 | 8 339 |
| **GPIO** | 4 664 | 1 219 |
| **SPI2AHB** | 3 639 | 1 176 |
| **IRQ CONTROLLER** | 2 490 | 535 |

formación de la Tabla 5.1 en forma de diagrama de torta de manera de tener una representación porcentual de las contribuciones de los distintos elementos

[7] *Taiwan Semiconductor Manufacturing Company*

del sistema respecto al área total.

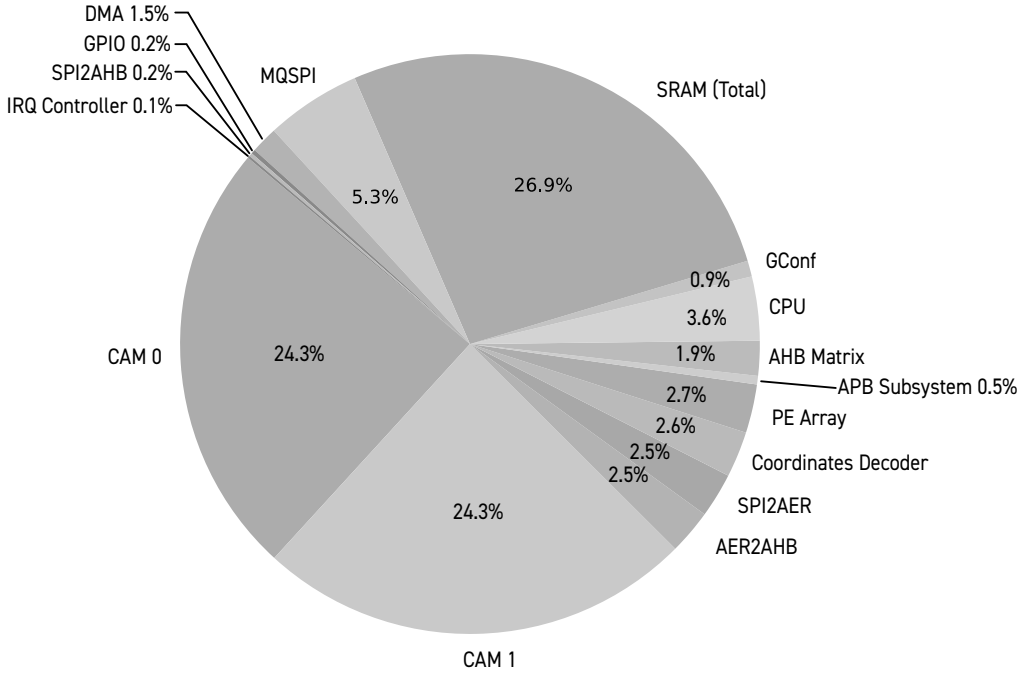



Figura 5.4: Distribución del área total post síntesis.

Tras completar la síntesis y la simulación, se inicia la implementación física del ASIC a partir del netlist a nivel de compuertas. El proceso comienza con la planificación de piso (*floorplanning*), donde se fijan las dimensiones del *die* y se distribuyen pads y bloques funcionales; en el caso de DigineuronV3b, la altura estaba limitada a 2 mm por requerimientos del fabricante, de modo que se adoptó un tamaño final de 4,5 $mm \times 2$ $mm$. A continuación se diseña la red de distribución de potencia: se colocan los anillos de alimentación y se despliega una malla que distribuye la energía de forma uniforme a todas las celdas. Luego se realizan, de manera iterativa, el posicionamiento de celdas, la síntesis del árbol de reloj y el ruteo, repitiendo los ajustes necesarios hasta cumplir las restricciones de temporización especificadas por el diseñador. Verificado que el circuito satisface todas las métricas de rendimiento, se obtiene el *layout* definitivo que se envía a fabricación (Figs. 5.5 y 5.6).

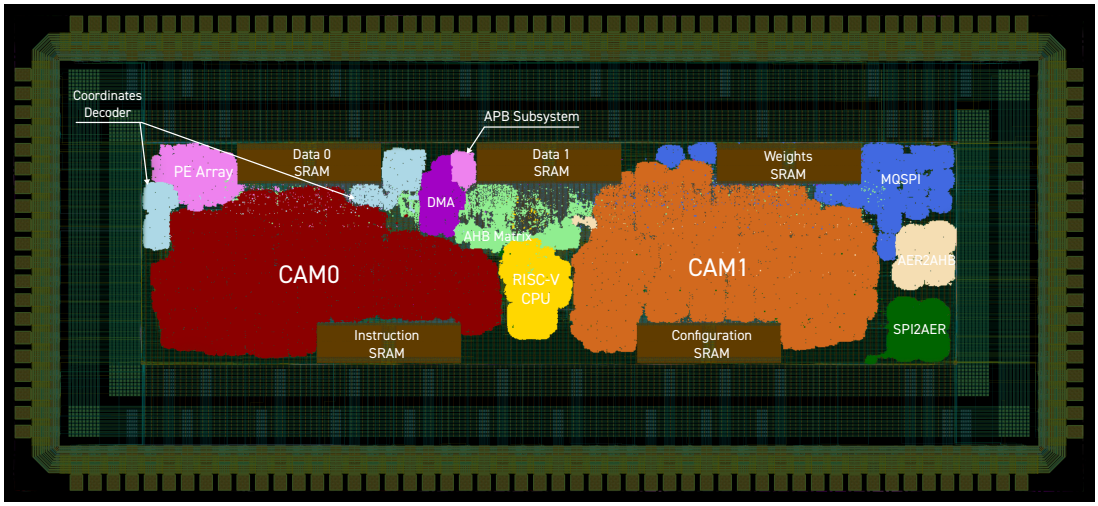



Figura 5.5: Layout del chip Digineuron_v3b.

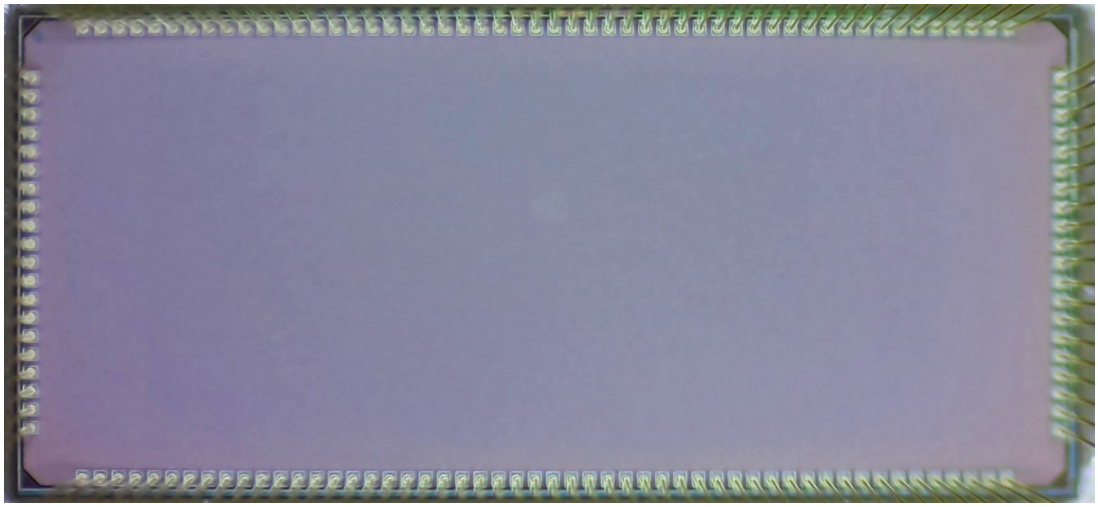


Figura 5.6: Fotografía del die de Digineuron_v3b una vez fabricado.

## 5.2. Resultados experimentales

El proceso de evaluación se organiza en etapas progresivas de creciente complejidad para agilizar la detección de fallos. Primero se comprueba la "vitalidad" de cada periférico conectado al bus AMBA AHB-Lite del ASIC mediante pruebas puramente funcionales. Si se detectan anomalías, se corrigen mediante bloques de hardware redundantes o, en su defecto, mediante soluciones de firmware que suplan la funcionalidad afectada. Superada esta fase, se verifica la operación principal del ASIC—la integración de eventos y el procesamiento de capas de convolución dispersa—y, finalmente, se realizan mediciones experimentales de rendimiento para identificar los puntos de operación que maximizan el desempeño.

### 5.2.1. Evaluación Funcional

El procedimiento de evaluación avanza de forma incremental: se empieza por los bloques más críticos y, a medida que crece la complejidad, se incorpora solo un bloque a la vez para limitar la aparición de variables nuevas. Para cada función del sistema se escribe un modelo funcional de referencia, a partir del cual se generan estímulos aleatorios que ejercitan el mayor espacio de estados posible y se comparan las salidas observadas en silicio con las predicciones del modelo, lo que permite detectar discrepancias con rapidez. A continuación se describen, en el mismo orden en que se probaron, los bloques verificados y las pruebas funcionales realizadas.

#### SPI2AHB

Se trata de uno de los bloques más críticos del sistema, pues permite la carga del firmware y proporciona las funciones de *debug*. El módulo presenta dos dominios de reloj: la etapa de recepción opera en el dominio SPI, mientras que el controlador del bus AMBA se sincroniza con el reloj principal del ASIC. La verificación comienza con la etapa de recepción SPI: se realizan operaciones de lectura y escritura sobre una serie de registros internos—residentes en el dominio SPI—mientras el CPU y el resto del sistema permanecen en reset. Una vez con-

firmada la comunicación entre la FPGA y el chip, se mantiene el procesador en reset, se habilita el resto del sistema y se efectúan ráfagas de escritura y lectura hacia las memorias internas. Scripts en Python comparan los datos enviados y recibidos, validando así la integridad del bus y el funcionamiento correcto del bloque.

### Memorias CAM

Para verificar las memorias CAM se preparan un conjunto de coordenadas únicas, de las cuales una parte serán escritas en la memoria CAM, y otra parte serán utilizadas sólo para búsqueda. Se procede a escribir las coordenadas destinadas para ello mediante la operación de escritura secuencial, y se realiza una ráfaga de operaciones de búsqueda con coordenadas presentes y no presentes en la CAM, y se comparan los resultados obtenidos. Una vez realizado esto, se procede a realizar la lectura de las coordenadas almacenadas, mediante la operación de lectura secuencial y se verifican los resultados obtenidos.

### Recepción de eventos

Para verificar el bloque de adquisición de eventos primario `aer2ahb` se prepara un conjunto de eventos aleatorios, y mediante el bloque transmisor **AERtx** de la FPGA, se envían uno a uno, y por medio del `spi2ahb` se lee el evento recibido, y se compara con el enviado, pudiendo verificar de esta manera el correcto funcionamiento del bloque receptor. Siguiendo la misma metodología, mediante el bloque transmisor spi implementado en la FPGA, se envían eventos al bloque receptor secundario `spi2aer`, y de la misma manera, se leen los eventos recibidos mediante el bloque `spi2ahb` y se comparan, para completar el proceso de verificación del bloque.

### CPU

Para comprobar el correcto funcionamiento del núcleo RISC-V se elaboró una suite de *firmware* de complejidad creciente. Cada programa se valida primero en simulación y, tras superar esta fase, se ejecuta en el banco experimental. A continuación se describen las pruebas realizadas:

1. **Fibonacci**. El CPU genera de manera secuencial la serie de Fibonacci en la memoria de datos. Una vez concluido el proceso, se coloca al CPU en *reset* y la secuencia generada se lee a través del bloque `spi2ahb`. Esta prueba confirma que el procesador arranca, ejecuta y finaliza el *firmware* sin errores.

2. **Fibonacci con GPIO**: Se reutiliza el código anterior, pero se configura la GPIO como salida y se emite la serie término a término por dicho puerto, verificando así la lógica de este módulo.

3. **Coordinate Decoder**. Mediante `spi2ahb` se escribe la configuración de capa y se inicializa la primera CAM con coordenadas únicas aleatorias. El *firmware* calcula todas las coordenadas de salida haciendo uso del módulo decodificador de coordenadas (`Coordinates Decoder`) y las almacena en la segunda CAM; completado el cómputo, el CPU vuelve a *reset* y las coordenadas resultantes se extraen vía `spi2ahb` para luego ser contrastadas con un modelo en lenguaje Python.

4. **Procesamiento de características**. Se cargan datos y parámetros aleatorios en las memorias SRAM y, mediante un *firmware*, el CPU habilita las ocho unidades MAC que conforman el arreglo de elementos de procesamiento (`PE Array`). El bloque se configura para que el cómputo se ejecute de forma autónoma tan pronto como los datos y parámetros estén disponibles en las FIFO de entrada. Al finalizar el procesamiento, los resultados se vuelcan de nuevo en la SRAM y se extraen por el `spi2ahb` para su verificación posterior.

5. **Integración de eventos**. Este *firmware* recibe los eventos y los acumula en una matriz dispersa de dos canales, donde cada canal contabiliza el número de eventos de polaridad positiva y negativa, respectivamente. El diagrama de flujo se muestra en la Fig. 5.7.

   Para validar el algoritmo se emplearon secuencias del conjunto de datos N-MNIST [54], capturadas con una cámara ATIS. Cada dígito se adquiere mediante tres movimientos sacádicos (Fig. 5.8, izquierda). Los eventos se

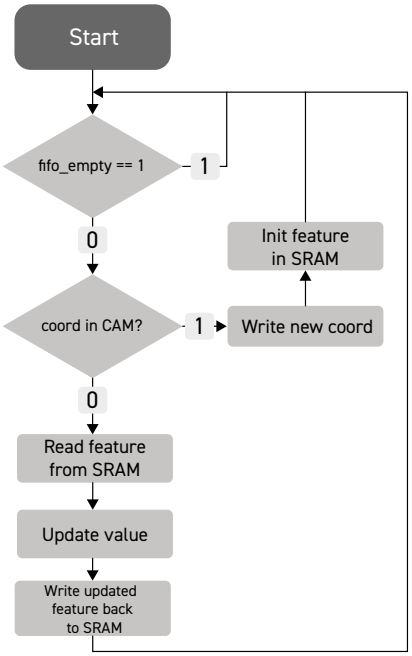



Figura 5.7: Diagrama de flujo del *firmware* de acumulación de eventos ejecutado por el CPU.

inyectan en el chip por el puerto AER; el *firmware* los integra en la matriz dispersa, registrando para cada coordenada la cuenta de polaridades 0 y 1. Tras procesar todos los eventos, el CPU se reinicia y la matriz resultante se extrae a través de `spi2ahb`. La Fig. 5.8 (derecha) muestra la matriz obtenida, que se contrasta con un modelo funcional en Python, confirmando la correcta operación.

6. **Convolución dispersa**. Este *firmware* implementa el algoritmo de convolución dispersa descrito en el Cap. 3, haciendo uso de los bloques dedicados del chip. El flujo de operación se ilustra en la Fig. 5.9.

   Como verificación funcional, se selecciona una muestra de cada clase de N-MNIST, se integra en una matriz dispersa de dos canales y se aplica un filtro de desenfoque gaussiano mediante una convolución con *kernel* $3 \times 3$ (Fig. 5.10). Las matrices dispersas se cargan en el ASIC a través de `spi2ahb`, se procesan con el CPU y los bloques de hardware de convolución, y el resultado se lee nuevamente por `spi2ahb`. Finalmente, la salida se compara con la implementación de referencia en PyTorch, obteniéndose coincidencia bit a bit.

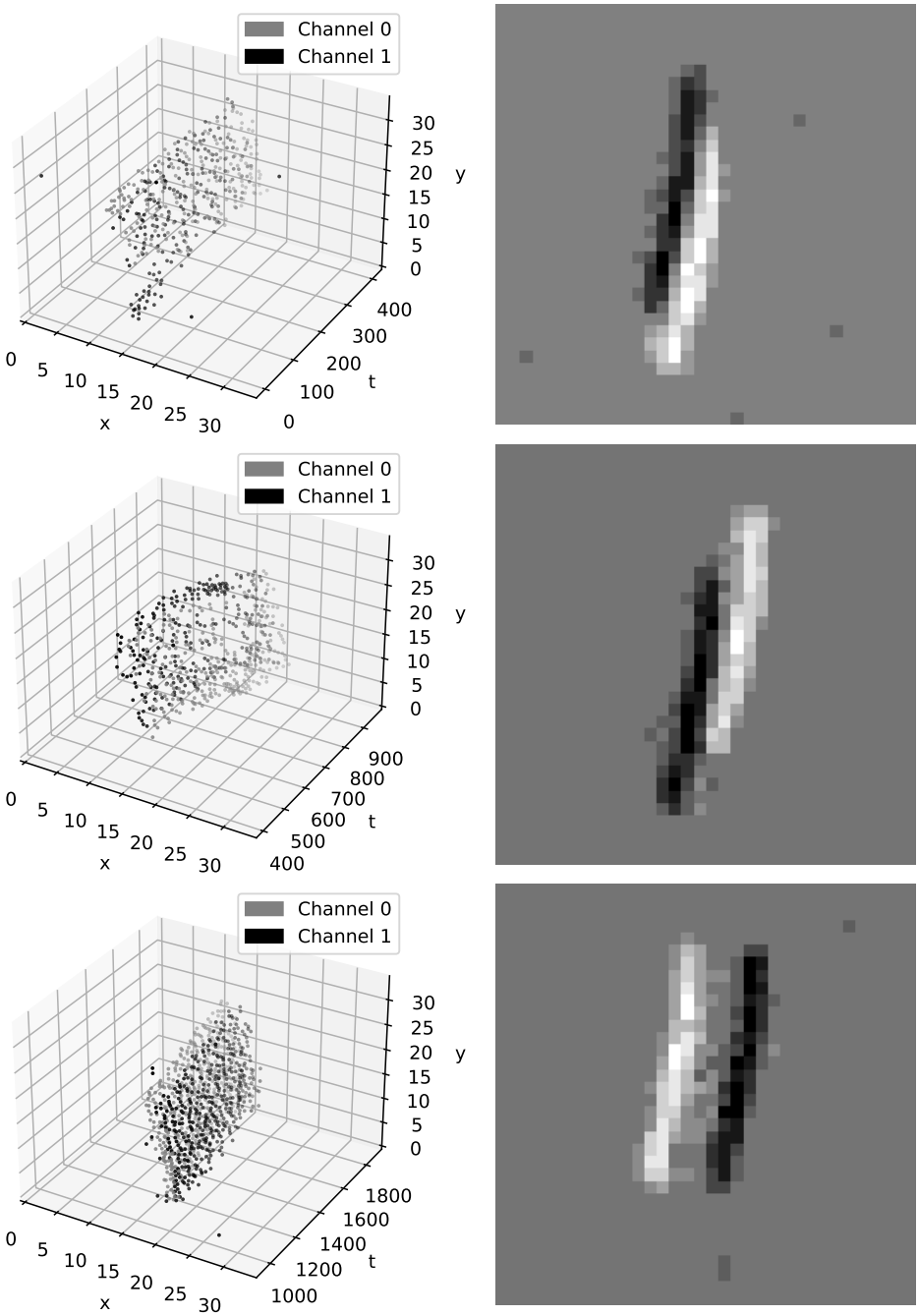



Figura 5.8: Integración de eventos para los tres movimientos sacádicos de una muestra de la clase “1” del dataset N-MNIST: (izq.) nubes de eventos de entrada; (der.) matriz dispersa de dos canales generada por el ASIC.

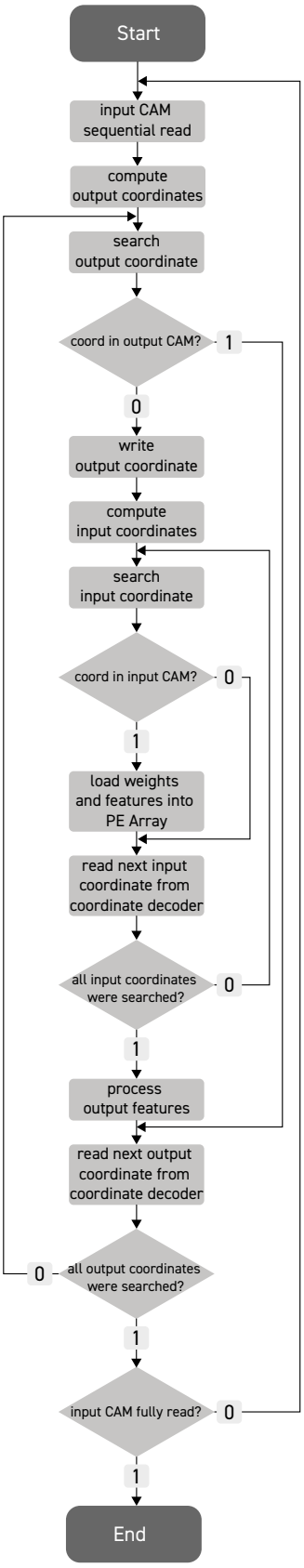



Figura 5.9: Diagrama de flujo de la operación de convolución que combina las CAM, el decodificador de coordenadas y el `PE Array`.

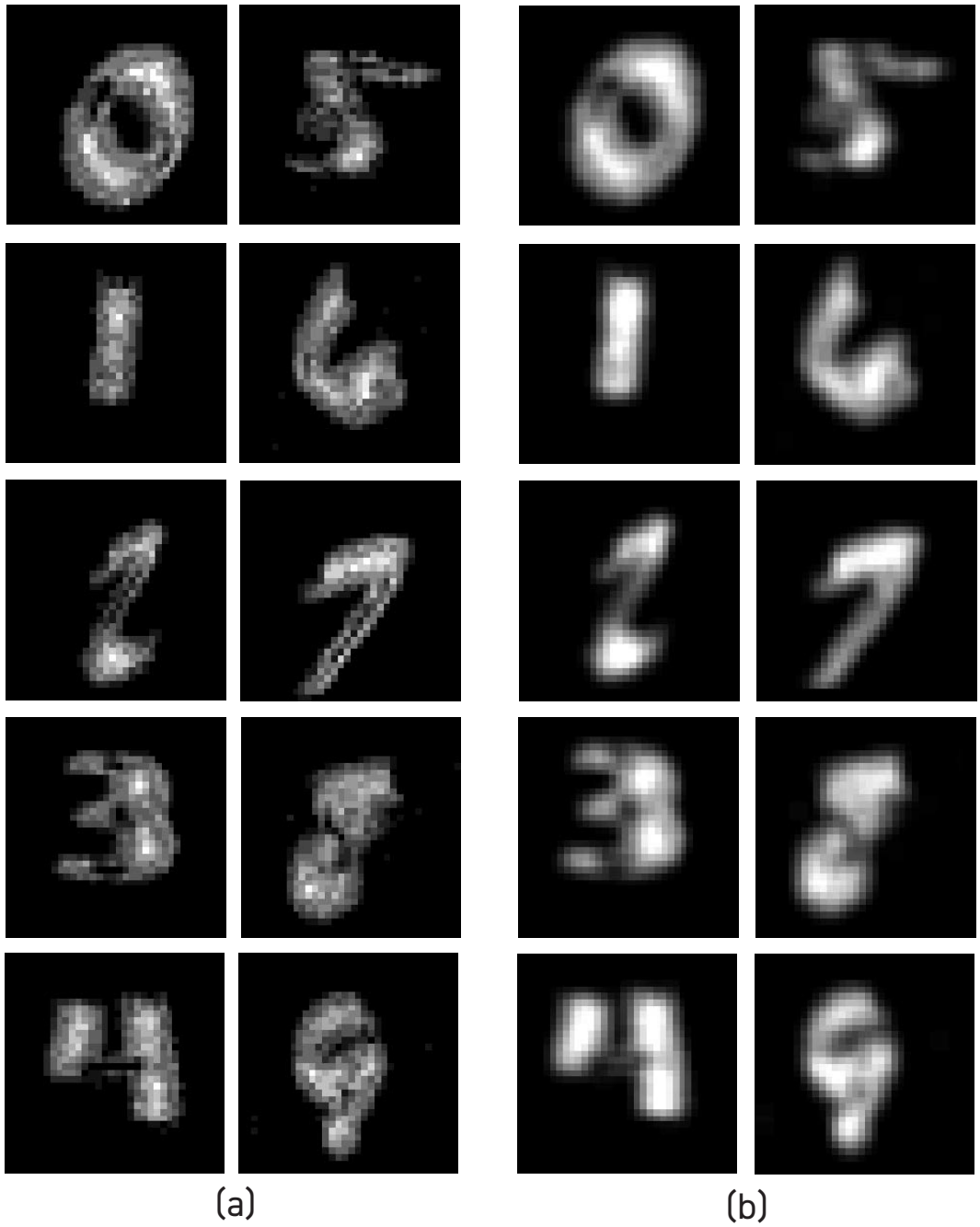



Figura 5.10: Aplicación de un filtro gaussiano $3 \times 3$ a los dígitos de N-MNIST: (a) imagen dispersa de entrada; (b) resultado procesado por el ASIC.

### 5.2.2. Caracterización energética

Verificada la funcionalidad del ASIC, se procede a cuantificar su consumo energético. Para ello se emplea el kit **X-NUCLEO-LPM01A** de *ST Microelectronics*, diseñado expresamente para medir la potencia de sistemas embebidos y utilizado como referencia en el *benchmark* MLPerf [55]. A fin de obtener resultados reproducibles, la caracterización se realiza de forma estadística, siguiendo la misma metodología expuesta en el Cap. 4.

El flujo de medida se esquematiza en la Fig. 5.11. A partir del modelo generador de matrices dispersas desarrollado en el Cap. 4, se crean matrices dispersas aleatorias que se envían, por un lado, al chip (donde se ejecuta el *firmware* de convolución) y, por otro, a un modelo funcional en Python que valida el resultado y calcula el número exacto de operaciones de multiplicación/acumulación. Combinando ese recuento con la energía medida se obtiene la eficiencia en MAC/W.

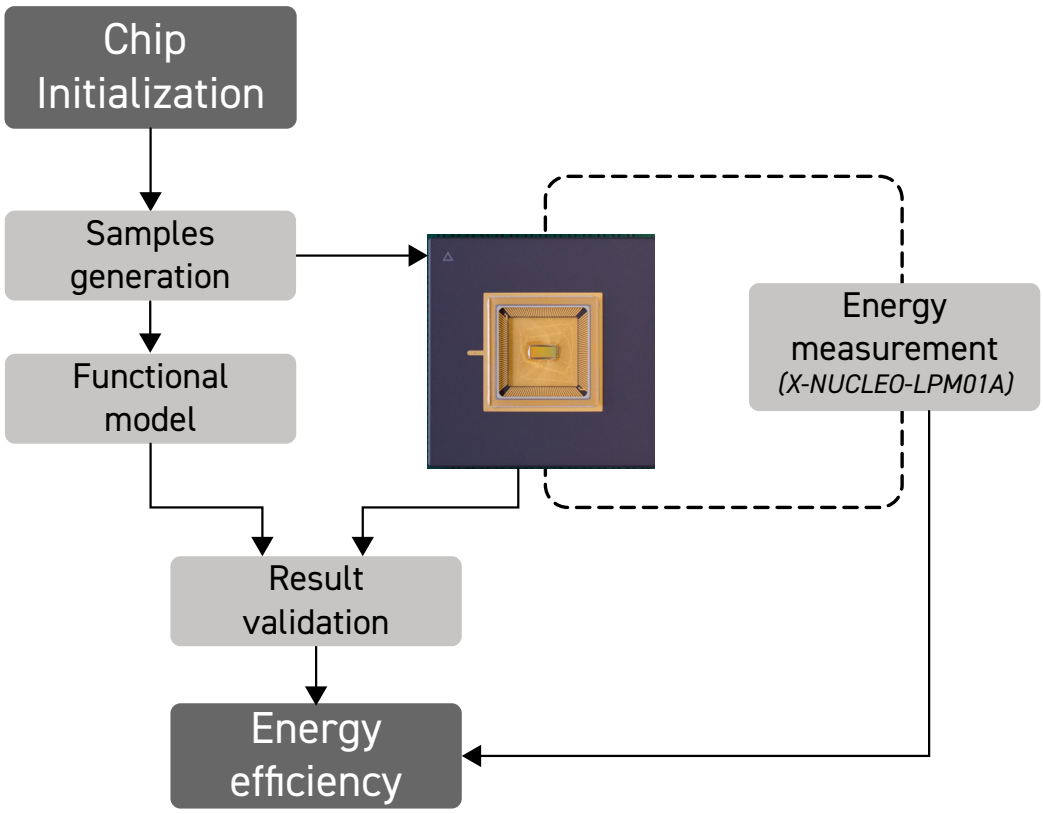



Figura 5.11: Proceso de caracterización energética del ASIC.

El kit suministra al ASIC una tensión de 1,8 V y muestrea la corriente a 100 kSa/s (cien mil muestras por segundo). La Fig. 5.12 muestra la traza de

corriente de un ciclo completo de medida, donde se distinguen tres fases:

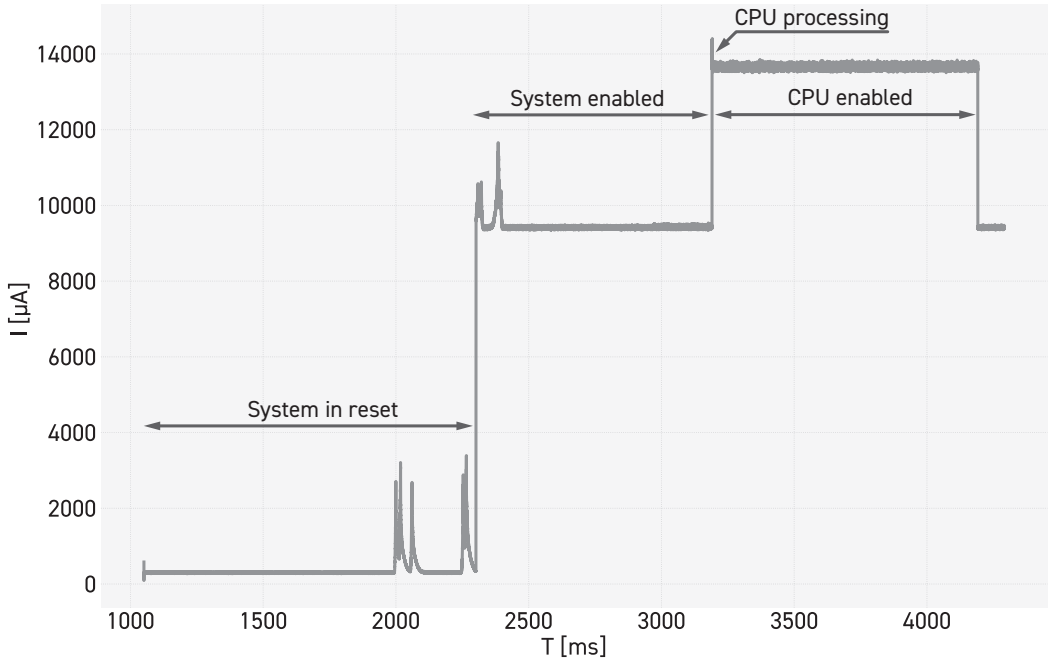



Figura 5.12: Corriente instantánea durante el procesamiento de una capa convolucional.

(I) **Reset *(System in reset).*** El ASIC parte en *reset*. A los 2 s se habilita el reloj, lo que provoca los picos de corriente al final de esta región.

(II) **Sistema habilitado *(System enabled).*** Con todos los bloques encendidos excepto el CPU, se cargan a través de `spi2ahb` la configuración del bus, la matriz dispersa de entrada, los filtros y el *firmware*. Estos accesos generan los picos iniciales de la fase *System enabled*.

(III) **Procesamiento del CPU *(CPU processing).*** Tras un retardo de seguridad se libera el *reset* del procesador, que ejecuta la convolución (intervalo amplificado en la Fig. 5.13). Terminado el cálculo, el CPU pasa a estado de reposo (*Idle*) hasta que se vuelve a colocar en *reset* y se extrae la matriz de salida.

**Consumo en reposo.** El nivel de corriente en *Idle* es elevado porque el núcleo `cv32e40p` sólo dispone de una celda de *clock gating* situada en el controlador de

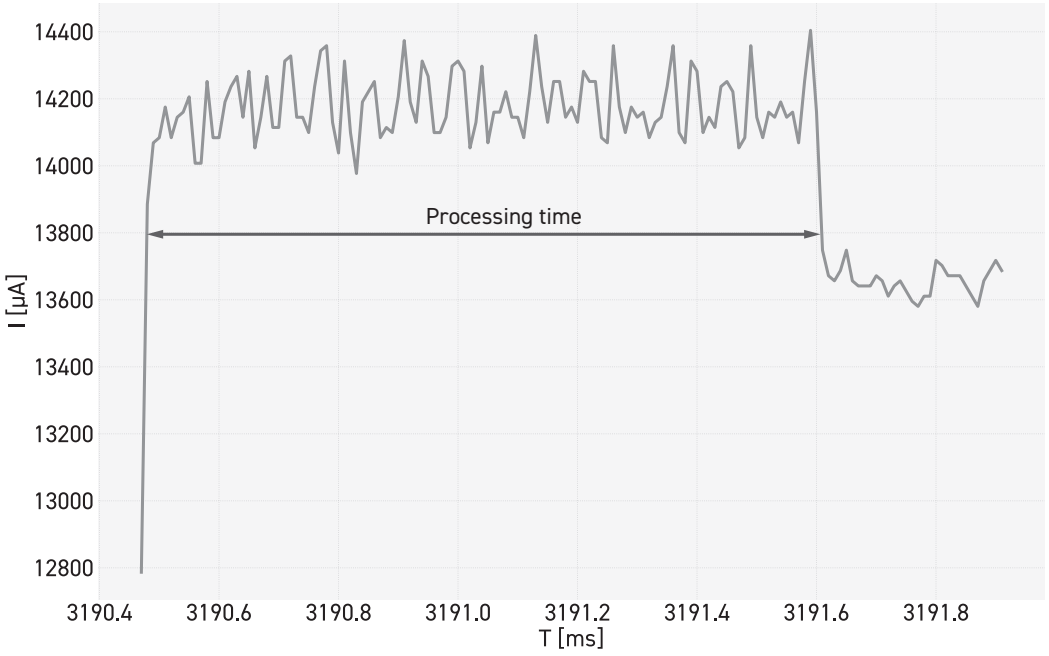



Figura 5.13: Detalle de la fase *CPU processing* para un filtro $3 \times 3$, *stride*=1 y una entrada dispersa de 10 coordenadas.

*sleep*; esta celda se activa mediante la instrucción WFI (*wait for interrupt*), que detiene el reloj del CPU hasta recibir una interrupción. Aunque el SoC integra un controlador de interrupciones, cada servicio implica guardar el contexto, lo que añade unos ∼80 ciclos de latencia. A 50 MHz y con una tasa de eventos máxima de 1 MHz, sólo hay margen de 50 ciclos para la captura completa: introducir interrupciones haría inviable el proceso de adquisición sin pérdida de eventos. Por ello, el reloj del CPU permanece activo durante el periodo de espera, lo que explica el piso de consumo observado.

**Caracterización Energética** Para caracterizar el consumo de energía, se extrae la corriente promedio y la duración de la fase de *CPU processing*. Se generan 50 muestras de $30 \times 30 \times 2$ por cada punto de medida (para garantizar que la matriz densa entre en la memoria CAM del chip y así realizar la caracterización completa), variando el grado de dispersión $\ell_i$ y la distribución espacial $\rho$, para distintas combinaciones de configuración de capa. El tamaño muestral se eligió con base en los tiempos de medición: cada ciclo de medida completo tarda alrededor de 10 s, debido al reinicio total del ASIC entre mediciones. Así, 50 muestras

resultan representativas y permiten obtener resultados en un tiempo razonable. La figura 5.14 muestra el consumo energético medido durante el procesamien-

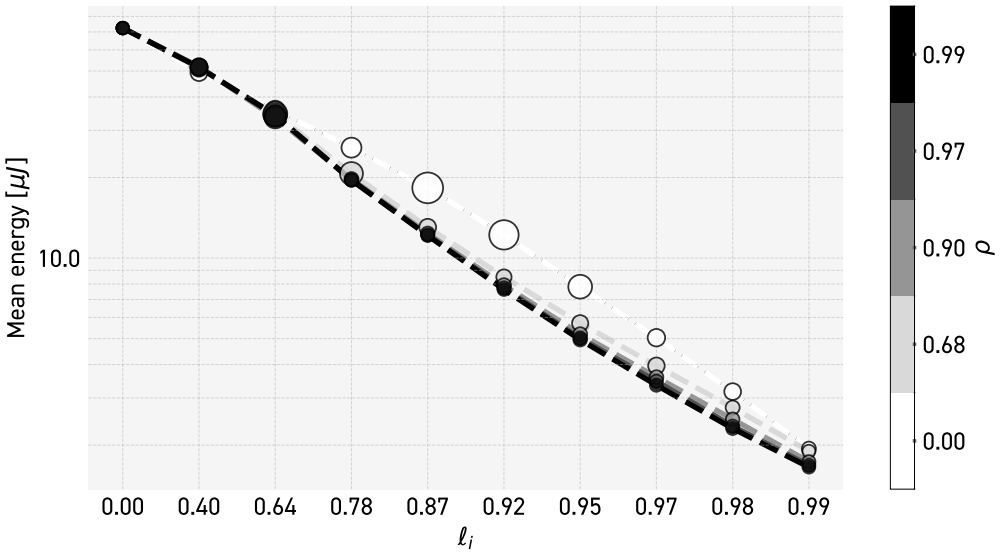



Figura 5.14: Consumo de energía durante el procesamiento de una capa convolucional con $ksize = 3$, $p = 0$, $\sigma = 1$, para una entrada de $30 \times 30$ pixeles. Se varían $\ell_i$ y $\rho$ en el rango $[0, 0{,}99]$, con alimentación de 1,8 V a 100 MHz.

to de una capa convolucional con tamaño de kernel $ksize = 3$, padding $p = 0$ y desviación $\sigma = 1$. Se aprecia un comportamiento análogo al observado en la figura 4.15b, obtenida mediante simulación de Monte Carlo basada en el modelo energético derivado de simulaciones post-síntesis. No obstante, las mediciones experimentales presentan valores de consumo superiores, lo cual se explica principalmente por la tensión de alimentación empleada (1,8 V), que corresponde al límite inferior del kit de medida, frente a la tensión nominal de 1,2 V asumida en la simulación. Para comparar el consumo energético entre distintas configuraciones de capa, se promediaron las mediciones a lo largo del factor de distribución espacial $\rho$ (Fig. 5.15). Los resultados coinciden con las predicciones del modelo presentado en el Cap. 4: para un mismo grado de dispersión $\ell_i$, el consumo viene determinado por la combinación del tamaño del kernel y del *stride*. Dado que evaluar exhaustivamente todas las posibles combinaciones de parámetros de capa requiere un tiempo de procesamiento elevado, se consideraron únicamente las configuraciones más habituales en arquitecturas de redes convolucionales del estado del arte. Además del consumo, y usando el modelo del Cap. 4, se calcula el número de operaciones MAC efectuadas en cada ciclo de medida, obteniendo

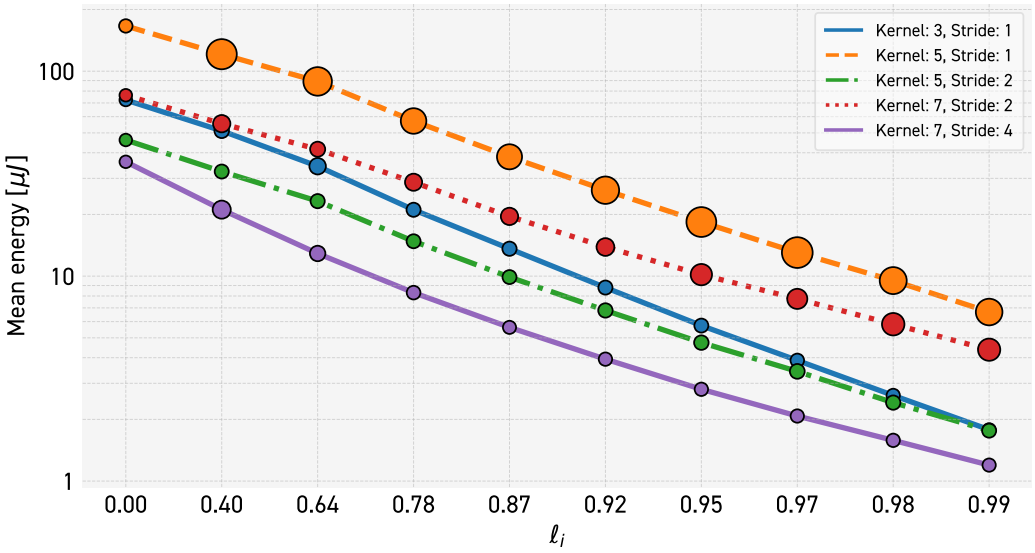



Figura 5.15: Consumo de energía para distintas configuraciones de capa convolucional en una entrada de $30 \times 30$, variando $\ell_i \in [0, 0{,}99]$, con 1,8 V y 100 MHz.

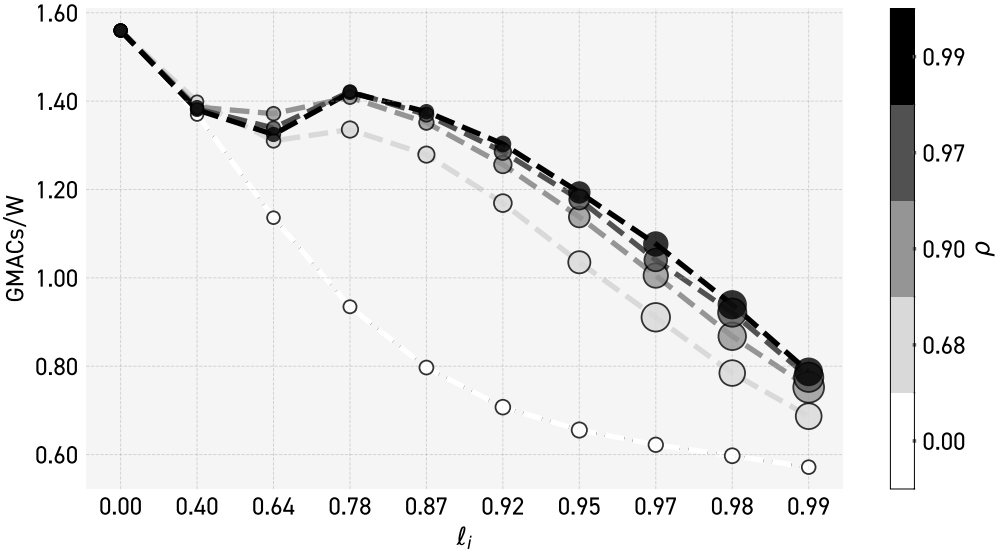



Figura 5.16: Eficiencia energética (MACs/W) para una capa convolucional con $ksize = 3$, $p = 0$, $\sigma = 1$, entrada $30 \times 30$, variando $\ell_i \in [0, 0{,}99]$, a 1,8 V y 100 MHz.

así una métrica de eficiencia energética (MACs/W). La Fig. 5.16 muestra esta eficiencia para un kernel $3 \times 3$ y *stride* 1. Destacan dos aspectos:

1. La eficiencia decrece con el aumento de $\ell_i$, al basarse en las operaciones *reales* ejecutadas, en lugar de las *virtuales* necesarias para una convolución densa sobre todos los píxeles.

2. Aparece un máximo local alrededor de $\ell_i \approx 0{,}78$ cuando $\rho$ tiende a una distribución normal. Esto se explica por un incremento de operaciones MAC reales debido al agrupamiento de eventos, y por la no linealidad de la generación de datos definida en la Ecuación (4.12).

## 5.3. Análisis comparativo de desempeño con otras implementaciones

En esta sección se muestran sistemas reportados en la literatura con el objetivo de realizar una comparación de desempeño. Al respecto, es común reportar el desempeño en términos de operación de multiplicación y acumulación (MACS) de una cantidad de bits estándar (típicamente, 8 bits). Otros sistemas suelen utilizar Operaciones por segundo (OPS), pero no hay consenso en cuanto a qué constituye una operación, pudiéndose encontrar una gran dispersión de criterios. Existen dos familias de SoC para procesamiento de redes neuronales que pueden utilizarse para efectuar una comparación de desempeño. En primer lugar, se encuentra la familia de SoC neuromórficos diseñados específicamente para procesar eventos. Este tipo de chips realizan operaciones de tipo LIF (basados en modelos de neuronas digitales inspiradas en la biología), que usualmente consisten en sumas condicionales. Dada la diferencia en complejidad entre este tipo de operaciones y las multiplicaciones y acumulaciones de 8-bits que DigineuronV3b realiza, aquí hay una gran discrepancia en lo que se define como una OP. De hecho, algunos SoC dentro de esta familia, directamente reportan valores de eficiencia sin especificar qué operación se consideró y bajo qué condiciones, por lo que tampoco pueden ser considerados. Por otro lado están los SoC diseñados para trabajar con imágenes estándar que se optimizan para el procesamiento de redes

convolucionales dispersas, utilizando técnicas como *prunning*. En estos casos, al igual que Digineuron_v3b, realizan el cómputo mediante unidades MACs, por lo que la comparación puede hacerse directamente.

Los chips de la literatura analizados [56, 57, 58, 59] presentan una arquitectura de elementos de procesamiento de tipo MAC, interconectados entre sí mediante una NoC. Los diseños reportados en [56, 57] han sido fabricados en tecnología de 65nm de TSMC, mientras que [58] ha sido fabricado en tecnología de 16nm, pero también reporta resultados de síntesis en 65nm de la misma arquitectura a modo comparativo. Por otro lado, en [59] se reportan resultados de síntesis e implementación en 55nm pero sin validación en silicio. Para la eficiencia, en esta tesis se adopta como métrica la cantidad de operaciones MAC por segundo, en relación con la potencia que consume el SoC, expresada en **MACs/W**. En la literatura revisada, se adoptan las **OPS/W**, donde se consideran que 2 "operaciones" (OPS) equivalen a una operación MAC de 8-bits. Dado que la operación de multiplicación de dos operandos de 8-bits y la operación de suma no tienen la misma complejidad, se traducen los resultados reportados en términos de **MACs/W**. Los tres SoC presentados en [56, 57, 58], presentan métricas de eficiencia para distintos niveles de dispersión, donde reportan el cálculo utilizado para el cómputo de eficiencia en el procesamiento de convoluciones con matrices densas, pero no reportan las operaciones consideradas para operar con matrices dispersas. Se deduce que para estos casos, la eficiencia se calcula a partir de considerar el total de las operaciones que deberían efectuarse si no se contara con hardware optimizado para el cálculo con matrices dispersas, en lugar de considerar las operaciones que se realizan de forma efectiva al operar solamente las características no nulas, lo que genera valores de eficiencia "virtuales" altísimos, completamente artificiales. Por tal motivo, se calcula la eficiencia en base a las operaciones **reales** efectuadas (Fig. 5.16), que corresponde con los valores reportados para el cálculo de convoluciones con matrices densas.

Los resultados comparativos se recopilan en la Tabla 5.2. Para DigineuronV3b presentamos dos valores de eficiencia: el medido experimentalmente a 1,8 V, y una versión escalada a 1,2 V para estimar el desempeño en condiciones de operación normales. De este modo, garantizamos una evaluación rigurosa y directamente

Tabla 5.2: Comparación con otras implementaciones.

| | **Digineuron_v3b** | [56] | [58] | [59] | [57] |
|---|---|---|---|---|---|
| **Proceso** | 65nm | 65nm | 65nm | 55nm | 65nm |
| **Frecuencia [MHz]** | 100 | 200 | 250 | 200 | 100 |
| **# Multiplicadores** | 8 | 256 | 252 | 256 | 64 |
| **n-bits de entradas** | 8 | 8 | 8 | 8 | 8/4 |
| **Memoria en chip [KiB]** | 80 | 170 | 280.6 | 308.6 | 196 |
| **Área de core [mm²]** | 1,9 | 7,8 | 9,32 | 7,5 | 7,8 |
| **Potencia máxima [mW]** | 26,3@1.8V | 284,4 | 500 | 198 | 99 |
| **Eficiencia [GMAC/W]** | 1.56@1.8V<br>3.19@1.2V | 205 | 125 | 260 | 65 |

comparable con los chips del estado del arte.

De la Tabla 5.2 se desprenden varios puntos clave. En primer lugar, DigineuronV3b opera a una frecuencia máxima de 100 MHz, frente a los 200 MHz de sus competidores, y dispone únicamente de ocho unidades MAC, dos órdenes de magnitud por debajo de otros diseños. Esta configuración responde a la ausencia de controladores internos y al uso de tan solo dos memorias CAM, lo cual limita la paralelización espacial del procesamiento. Como se analizó en el Cap. 4, la síntesis de las memorias CAM ejerce un impacto crítico en la eficiencia energética, hecho que se refleja directamente en el valor de MAC/W presentado en la tabla.

## 5.4. Conclusiones

En este capítulo se ha descrito DigineuronV3b, un SoC para procesamiento de redes neuronales convolucionales basado en RISC-V, que integra dos periféricos de adquisición de eventos, dos memorias CAM sintetizadas (capaces de almacenar matrices dispersas de hasta $511 \times 511$ con 1024 coordenadas cada una), 80 KiB de memoria interna, decodificación doble de coordenadas y un arreglo de ocho unidades MAC. El diseño, implementado en tecnología CMOS de 65 nm con un área total de $9\,\mathrm{mm}^2$, fue validado funcionalmente en múltiples niveles y caracterizado energéticamente mediante medidas experimentales a 1,8 V y simulaciones post-síntesis a 1,2 V.

Los resultados muestran que, aunque Digineuron_v3b queda por debajo de los chips del estado del arte analizados en términos de MACs/W —debido principalmente a las memorias CAM sintetizadas y a la falta de controladores dedicados que permitan mayor paralelización—, la sustitución de estas CAM por memorias CAM dedicadas, tal y como se propuso en el Cap. 4, permitiría reducir el consumo en un orden de magnitud y mejorar notablemente la eficiencia energética.

# Capítulo 6

# Conclusiones y trabajos futuros

En esta tesis, a partir del estado del arte realizado en el Capítulo 2, se propuso un mecanismo novedoso de procesamiento de redes convolucionales dispersas empleando memorias CAM, que posibilita la adquisición de eventos, generación y actualización de la representación matricial en tiempo real, y posterior cómputo de la red convolucional iterando las memorias CAM, lo que derivó en la presentación de una solicitud de patente [24]. A partir de los algoritmos propuestos en el Capítulo 3, se identificaron las operaciones principales y se diseñaron bloques de hardware dedicados para su ejecución. En función de esto, en el Capítulo 4, se propuso un diseño de arquitectura y se desarrolló un modelo de alto nivel detallado, que permitió identificar los aspectos claves que impactan en el desempeño y la eficiencia energética. De este análisis surge que si bien se reducen las operaciones MAC a las mínimas necesarias, se introduce un número considerable de operaciones de búsqueda en CAM, lo que implica que el diseño eficiente de este bloque es crítico para el desempeño y la eficiencia energética del sistema.

A partir de los algoritmos desarrollados, en el Capítulo 5, se ha presentado el diseño e implementación del sistema en un chip eficiente de $9mm^2$ en tecnología CMOS de 65nm. Se validó el sistema completo, comprobando el correcto funcionamiento de todos los bloques internos. Se caracterizó estadísticamente el SoC en términos de consumo y eficiencia, alcanzando valores máximos de 1.56 GMACs/W, medido a una tensión de alimentación de 1,8 V y 100 MHz, y 3.19 GMACs/W a la tensión nominal de 1,2 V y a la misma frecuencia de trabajo, escalado a partir de las mediciones. Se han realizado pruebas experimentales

donde se contrastaron los resultados obtenidos contra valores de eficiencia reportados por otros SoC del estado del arte que fueron analizados y comparados [56, 57, 58, 59], los cuales fueron diseñados en la misma tecnología (excepto [59] que fue diseñado en el nodo de 55nm).

Este esquema resulta especialmente eficiente para datos altamente dispersos procedentes de sensores por eventos. Así, particularmente en el Capítulo 4, se demostró cuantitativamente que utilizando memorias CAM dedicadas, esta arquitectura resulta más eficiente que el enfoque tradicional de procesamiento de matriz densa, para grados de dispersión mayores al 90 %.

El SoC implementado se diseñó dando prioridad a la flexibilidad y programabilidad por encima del rendimiento, con el fin de minimizar la aparición de errores de implementación críticos. Este trabajo sienta las bases para continuar desarrollando esta línea de investigación hacia la implementación en hardware de algoritmos más especializados con un enfoque en mejorar el rendimiento. Una de las oportunidades de mejora viene dada por la incorporación de memorias CAM dedicadas, un incremento en la frecuencia de operación, y la replicación de hardware, ya que actualmente cada matriz dispersa (entrada y salida) reside en una única CAM, lo que impide paralelizar búsquedas y actualizaciones. Dividiendo espacialmente la matriz en sectores más pequeños, cada uno atendido por su propia CAM y unidad de cómputo, sería posible procesar eventos de múltiples regiones simultáneamente, acelerando de forma significativa todo el flujo de trabajo. A su vez, este nuevo sistema podría dividirse en múltiples dominios de tensión de alimentación, lo que, por intermedio de un controlador de ultra bajo consumo, podría físicamente apagar los bloques que no estén siendo utilizados y mejorar aún más la eficiencia energética.

Por lo expuesto, puede concluirse que el presente trabajo establece un punto de partida sólido para el desarrollo de sistemas de procesamiento que explotan la dispersión inherente a los datos de sensores por eventos; haciendo posible el desarrollo de nuevas aplicaciones en visión artificial y sistemas embebidos de ultra-bajo consumo.

# Anexo A

# Mapas de Memoria

Este apéndice incluye los mapas de memoria de la arquitectura del chip Digineuron_v3b, y los principales bloques de cómputo, que actúan como modelo de programación para la interacción con dichos bloques a través del programa que ejecutará el procesador. Además, se proporciona una descripción cualitativa de la funcionalidad de cada registro. Se comienza con la lista de periféricos y el tipo de árbitro que poseen, y luego se adjunta el mapa de memoria general del sistema. Las filas representan los periféricos y las columnas los controladores, donde cada celda contiene el rango de direcciones de cierto periférico.

Tabla A.1: Mapa de memoria del subsistema AMBA APB.

| **Periférico** | **Dirección Base** |
|---|---|
| Timer 0 | 0x00000 |
| Timer 1 | 0x10000 |
| Dualtimer | 0x20000 |
| Watchdog | 0x30000 |
| DMA Config | 0x40000 |

Tabla A.2: Periféricos del bus AHB-Lite y esquema de arbitraje.

| **Periférico** | **Tipo de árbitro** |
|---|---|
| instr_mem | simple priority |
| config_mem | simple priority |
| weight_mem | simple priority |
| data0_mem | simple priority |
| data1_mem | simple priority |
| cam0 | weighted round robin |
| cam1 | weighted round robin |
| aer2ahb | weighted round robin |
| spi2aer | weighted round robin |
| gconf | weighted round robin |
| irq_controller | weighted round robin |
| gpio | weighted round robin |
| qspi | weighted round robin |
| pe_array | weighted round robin |
| coord_decoder | weighted round robin |
| apb_subsystem | weighted round robin |

Tabla A.3: Mapa de memoria del bus AHB-Lite del chip Digineuron_v3b.

| | **spi2ahb** | **cpu_instr** | **cpu_data** | **dma_rd** | **dma_wr** |
|---|---|---|---|---|---|
| **instr_mem** | 0x00000000<br>0x00003FFF | 0x00000000<br>0x00003FFF | 0x00000000<br>0x00003FFF | 0x00000000<br>0x00003FFF | 0x00000000<br>0x00003FFF |
| **config_mem** | 0x00004000<br>0x00007FFF | 0x00004000<br>0x00007FFF | 0x00004000<br>0x00007FFF | 0x00004000<br>0x00007FFF | 0x00004000<br>0x00007FFF |
| **weight_mem** | 0x00008000<br>0x0000BFFF | 0x00008000<br>0x0000BFFF | 0x00008000<br>0x0000BFFF | 0x00008000<br>0x0000BFFF | 0x00008000<br>0x0000BFFF |
| **data0_mem** | 0x0000C000<br>0x0000FFFF | 0x0000C000<br>0x0000FFFF | 0x0000C000<br>0x0000FFFF | 0x0000C000<br>0x0000FFFF | 0x0000C000<br>0x0000FFFF |
| **data1_mem** | 0x00010000<br>0x00013FFF | 0x00010000<br>0x00013FFF | 0x00010000<br>0x00013FFF | 0x00010000<br>0x00013FFF | 0x00010000<br>0x00013FFF |
| **cam0** | 0x10000000<br>0x1FFFFFFF | | 0x10000000<br>0x1FFFFFFF | 0x10000000<br>0x1FFFFFFF | 0x10000000<br>0x1FFFFFFF |
| **cam1** | 0x20000000<br>0x2FFFFFFF | | 0x20000000<br>0x2FFFFFFF | 0x20000000<br>0x2FFFFFFF | 0x20000000<br>0x2FFFFFFF |
| **aer2ahb** | 0x30000000<br>0x3FFFFFFF | | 0x30000000<br>0x3FFFFFFF | 0x30000000<br>0x3FFFFFFF | 0x30000000<br>0x3FFFFFFF |
| **spi2aer** | 0x40000000<br>0x4FFFFFFF | | 0x40000000<br>0x4FFFFFFF | 0x40000000<br>0x4FFFFFFF | 0x40000000<br>0x4FFFFFFF |
| **gconf** | 0x50000000<br>0x50007FFF | | 0x50000000<br>0x50007FFF | | |
| **irq_controller** | 0x50008000<br>0x5000FFFF | | 0x50008000<br>0x5000FFFF | | |
| **gpio** | 0x60000000<br>0x6FFFFFFF | | 0x60000000<br>0x6FFFFFFF | 0x60000000<br>0x6FFFFFFF | 0x60000000<br>0x6FFFFFFF |
| **qspi** | 0x70000000<br>0x7FFFFFFF | | 0x70000000<br>0x7FFFFFFF | 0x70000000<br>0x7FFFFFFF | 0x70000000<br>0x7FFFFFFF |
| **pe_array** | 0x80000000<br>0x8FFFFFFF | | 0x80000000<br>0x8FFFFFFF | 0x80000000<br>0x8FFFFFFF | 0x80000000<br>0x8FFFFFFF |
| **coord_decoder** | 0x90000000<br>0x9FFFFFFF | | 0x90000000<br>0x9FFFFFFF | 0x90000000<br>0x9FFFFFFF | 0x90000000<br>0x9FFFFFFF |
| **apb_subsystem** | 0xA0000000<br>0xAFFFFFFF | | 0xA0000000<br>0xAFFFFFFF | 0xA0000000<br>0xAFFFFFFF | 0xA0000000<br>0xAFFFFFFF |

## A.1. Módulo de adquisición de eventos AER to AHB-Lite

Tabla A.4: Mapa de memoria del bloque de adquisición de eventos **aer2ahb**.

| **Nombre** | **Dirección** | **Sentido** | **Bits** | **Descripción** |
|---|---|---|---|---|
| fifo_empty_flag | 0x001 | RD | [0] | Bandera de FIFO vacía. |
| fifo_full_flag | 0x001 | RD | [1] | Bandera de FIFO llena. |
| fifo_almost_empty_flag | 0x001 | RD | [2] | Bandera cuya activación ocurre al cruzar el umbral vacío programable. |
| fifo_almost_full_flag | 0x001 | RD | [3] | Bandera cuya activación ocurre al cruzar el umbral lleno programable. |
| fifo_full_prog_threshold | 0x004 | RW | [7:0] | Umbral lleno programable. |
| fifo_empty_prog_threshold | 0x004 | RW | [23:16] | Umbral vacío programable. |
| fifo_read_data | 0x008 | RD | [23:0] | Lectura de eventos almacenados en la FIFO. |
| irq_fifo_full_en | 0x00C | RW | [0] | Habilitación de interrupción por FIFO llena. |
| irq_fifo_almost_full_en | 0x00C | RW | [1] | Habilitación de interrupción por FIFO al cruzar umbral lleno programable. |
| events_en | 0x00C | RW | [2] | Habilitación de adquisición de eventos. |
| wait_count | 0x00C | RW | [31:24] | Umbral del contador de espera. |

## A.2. Módulo de adquisición de eventos SPI

Tabla A.5: Opcodes del bloque de adquisición de eventos SPI.

| **Mnemónico** | **Opcode** | **Descripción** |
|---|---|---|
| RD_INT_REG | 0b00 | Lectura registros internos. Se envían los 2 bits de opcode y retorna los 4 Bytes almacenados |
| WR_INT_REG | 0b01 | Escritura de registros internos. Se envían los 2 bits de opcode seguidos de los 4 Bytes que se quieren escribir. |
| WR_AHB_DATA | 0b10 | Escritura de eventos. Se envían los 2 bits de opcode, seguidos de al menos 3 Bytes, que contienen la primer coordenada **y** seguida de al menos 1 coordenada **x** con su respectiva polaridad |

Tabla A.6: Mapa de memoria del bloque de adquisición de eventos **spi2aer**.

| **Nombre** | **Dirección** | **Sentido** | **Bits** | **Descripción** |
|---|---|---|---|---|
| fifo_empty_flag | 0x001 | RD | [0] | Bandera de FIFO vacía. |
| fifo_full_flag | 0x001 | RD | [1] | Bandera de FIFO llena. |
| fifo_almost_empty_flag | 0x001 | RD | [2] | Bandera cuya activación ocurre al cruzar el umbral vacío programable. |
| fifo_almost_full_flag | 0x001 | RD | [3] | Bandera cuya activación ocurre al cruzar el umbral lleno programable. |
| spi_working | 0x002 | RD | [0] | Bandera que indica que el bloque receptor SPI se encuentra actualmente recibiendo datos. |
| fifo_full_prog_threshold | 0x004 | RW | [7:0] | Umbral lleno programable. |
| fifo_empty_prog_threshold | 0x005 | RW | [7:0] | Umbral vacío programable. |
| fifo_read_data | 0x008 | RD | [23:0] | Lectura de eventos almacenados en la FIFO. |
| irq_fifo_full_en | 0x00C | RW | [0] | Habilitación de interrupción por FIFO llena. |
| irq_fifo_almost_full_en | 0x00C | RW | [1] | Habilitación de interrupción por FIFO al cruzar umbral lleno programable. |

## A.3. Content Addressable Memory

Tabla A.7: Mapa de memoria del bloque CAM.

| **Nombre** | **Dirección** | **Sentido** | **Bits** | **Descripción** |
|---|---|---|---|---|
| CAM_random_access | 0x000 - 0x1FF | RW | - | Acceso aleatorio a cada registro CAM. |
| search_in | 0x200 | WR | [17:0] | Puerto de búsqueda. |
| search_result | 0x200 | RD | [9:0] | Resultado de búsqueda (dirección). |
| hit | 0x200 | RD | [31] | Señal de dato encontrado. |
| sequential_cam_write | 0x204 | WR | [17:0] | Escritura secuencial de la CAM. |
| sequential_data_write | 0x204 | WR | [29:24] | Escritura secuencial de los registros de banderas. |
| last_written_addr | 0x204 | RD | [9:0] | puntero de escritura. |
| cam_en | 0x208 | RW | [0] | Habilitación de escritura de la CAM. |
| data_en | 0x208 | RW | [1] | Habilitación de escritura de los registros de bandera. |
| rst_wr_ptr | 0x208 | RW | [2] | Reset del puntero de escritura. |
| rst_rd_ptr | 0x208 | RW | [3] | Reset del puntero de lectura. |
| en_irq_cam_full | 0x208 | RW | [4] | Habilitación de interrupción por CAM llena. |
| en_irq_cam_empty | 0x208 | RW | [5] | Habilitación de interrupción por CAM vacía. |
| en_irq_cam_fully_read | 0x208 | RW | [6] | Habilitación de interrupción por CAM completamente leída. |
| cam_full_reset | 0x20C | WR | [0] | Reset de arreglo CAM. |
| sequential_cam_read | 0x20C | RD | [17:0] | Lectura secuencial de la CAM. |
| sequential_data_read | 0x20C | RD | [29:24] | Lectura secuencial de los registros de banderas. |

## A.4. Módulo decodificador de coordenadas

Tabla A.8: Mapa de memoria del bloque decodificador de coordenadas.

| **Nombre** | **Dirección** | **Sentido** | **Bits** | **Descripción** |
|---|---|---|---|---|
| filter_size | 0x000 | RW | [2:0] | Tamaño de filtro (máximo 7). |
| stride | 0x000 | RW | [4:3] | Stride expresado como potencia de 2. |
| padding | 0x000 | RW | [6:5] | Padding. |
| output_size_y | 0x000 | RW | [16:7] | Tamaño de la matriz de salida (H). |
| output_size_x | 0x000 | RW | [26:17] | Tamaño de la matriz de salida (W). |
| oc_fifo_full_flag | 0x001 | RD | [0] | Bandera de FIFO de coordenadas de salida. |
| oc_fifo_empty_flag | 0x001 | RD | [1] | Bandera de FIFO de coordenadas de salida. |
| oc_done | 0x001 | RD | [2] | Cálculo de coordenadas de salida finalizado. |
| kc_fifo_full_flag | 0x001 | RD | [3] | Bandera de FIFO de coordenadas de kernel. |
| kc_fifo_empty_flag | 0x001 | RD | [4] | Bandera de FIFO de coordenadas de kernel. |
| ic_fifo_full_flag | 0x001 | RD | [5] | Bandera de FIFO de coordenadas de entrada. |
| ic_fifo_empty_flag | 0x001 | RD | [6] | Bandera de FIFO de coordenadas de entrada. |
| ic_done | 0x001 | RD | [7] | Cálculo de coordenadas de entrada finalizado. |
| en_ic_done_irq | 0x001 | RD | [8] | Habilitación de interrupción. |
| en_oc_done_irq | 0x001 | RD | [9] | Habilitación de interrupción. |
| manual_oc_search | 0x001 | RD | [10] | Habilitación de búsqueda manual (Ver Capítulo 4). |
| search_ic_y | 0x002 | RD | [8:0] | Coordenada de entrada al bloque *OC Phase*. |
| search_ic_x | 0x002 | RD | [17:9] | Coordenada de entrada al bloque *OC Phase*. |
| start_search_oc | 0x002 | WR | [18] | Start *OC Phase*. |

| **Nombre** | **Dirección** | **Sentido** | **Bits** | **Descripción** |
|---|---|---|---|---|
| output_coord_y | 0x004 | RW | [8:0] | RD: Salida del decodificador *IC Phase*. WR: Escritura de coordenada manual |
| output_coord_x | 0x004 | RW | [17:9] | RD: Salida del decodificador *IC Phase*. WR: Escritura de coordenada manual |
| read_new_oc | 0x004 | WR | [18] | Siguiente coordenada de salida |
| start_ic | 0x004 | WR | [19] | Calcular coordenadas de entrada |
| input_coord_y | 0x005 | RD | [8:0] | Coordenada de salida del bloque *IC Phase* |
| input_coord_x | 0x005 | RD | [17:9] | Coordenada de salida del bloque *IC Phase* |
| kernel_coord_y | 0x005 | RD | [20:18] | Coordenada de salida del kernel. |
| kernel_coord_x | 0x005 | RD | [23:21] | Coordenada de salida del kernel. |
| next_kernel_coord | 0x005 | RD | [24] | Siguiente coordenada del kernel. |
| kernel_coord_mem_addr | 0x007 | RD | [31:0] | Dirección de memoria de la coordenada del kernel. |
| kernel_address | 0x007 - 0x038 | RW | [31:0] | Direcciónes de memoria de cada coordenada de kernel posible desde la (0, 0) hasta la (6, 6), organizadas por fila. |

# Anexo B

# Protocolo Address Event Representation

Cuando hablamos de sensores de imágenes por eventos, debido a la naturaleza asincrónica de la información que estos producen, fue necesaria la implementación de un protocolo que satisfaga dicha condición. Uno de los protocolos más adoptados, el *Address Event Representation* (AER), originalmente propuesto por [60], [61] se ha convertido con el tiempo en el estándar de comunicación para desarrollos de circuitos neuromórficos.

El protocolo AER es un protocolo asincrónico unidireccional desde un emisor, típicamente un sensor, hacia un receptor, típicamente un circuito de procesamiento [62]. La comunicación entre emisor y receptor consiste en señales de control entre las cuales tenemos una señal de *request* controlada por el emisor, y una señal de *acknowledge*, controlada por el receptor, para indicar la correcta recepción de los datos. Por otro lado, los datos se transmiten de manera paralela. La implementación específica del protocolo depende del fabricante. En este caso se adoptó la implementación propuesta por la compañía IniVation [49]. En la Fig. B.1 se presenta un diagrama temporal que ilustra el protocolo.

Cada transacción comienza con un flanco descendente de la señal de *request* por parte del emisor. En este punto, los datos en el bus *data[9:0]* pueden ser considerados válidos y almacenados por el receptor, quien confirma emitiendo un flanco descendente en la señal de *acknowledge*. Los sensores de IniVation [49] violan la asunción que todas las señales de datos son inmediatamente válidas

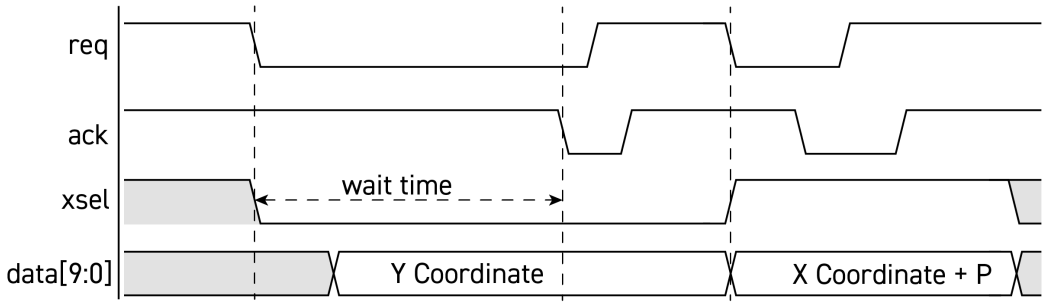



Figura B.1: Diagrama temporal del protocolo AER

luego del flanco descendente de la señal de *request*, por lo que el receptor debe incorporar un delay de al menos 50ns para garantizar la estabilidad de los datos antes de almacenarlos. Así mismo, esta implementación incorpora una señal adicional *xsel* que indica si el dato presente en el bus se trata de la coordenada **y** o la coordenada **x** más la polaridad del evento.

# Anexo C

# Esquema de bondeo y empaquetado.

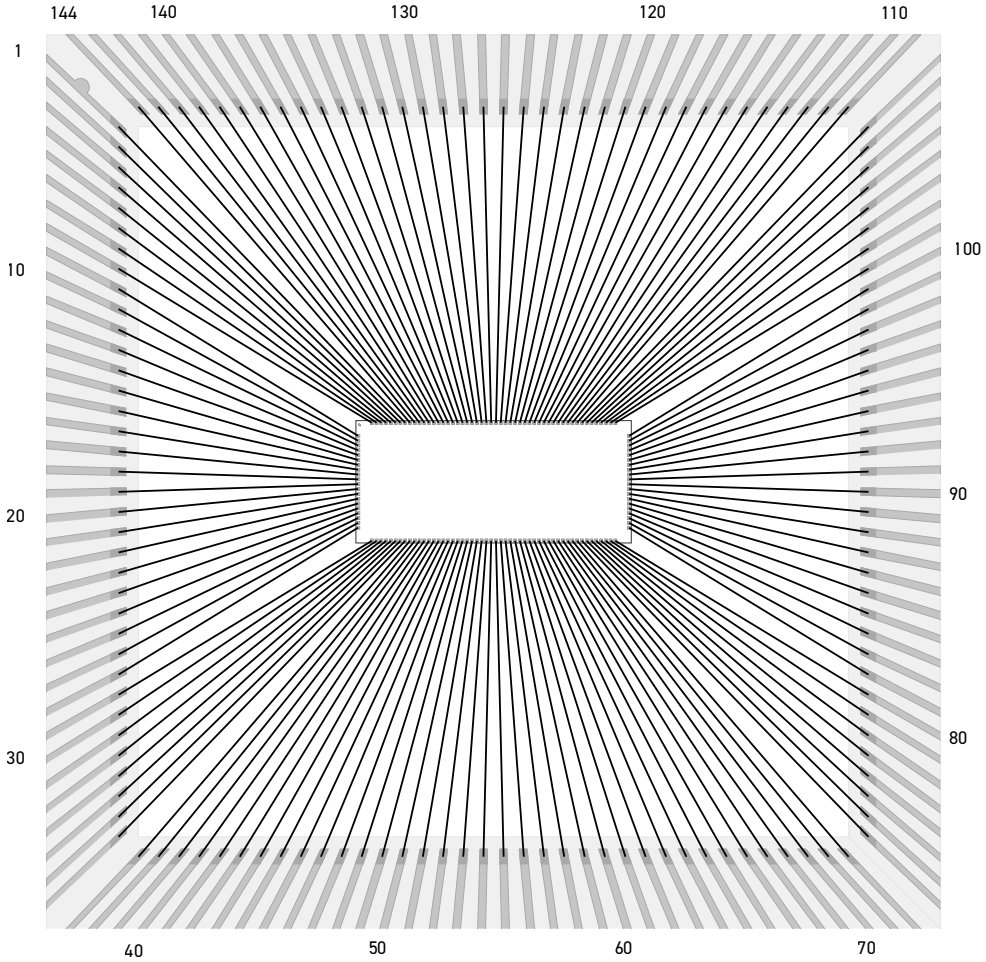



Figura C.1: Diagrama de bondeo del SoC Digineuron_v3b en un encapsulado CPGA144.

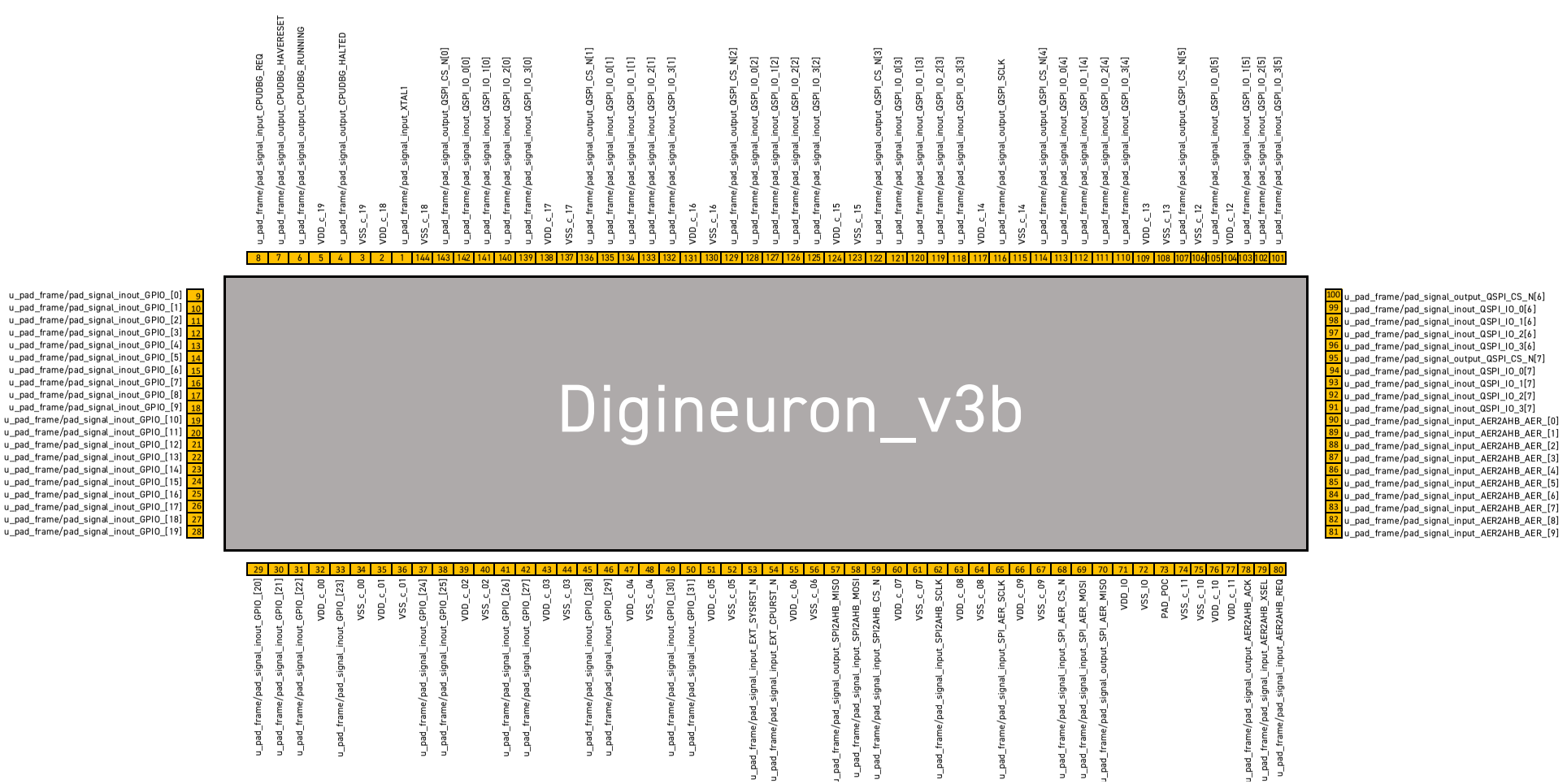
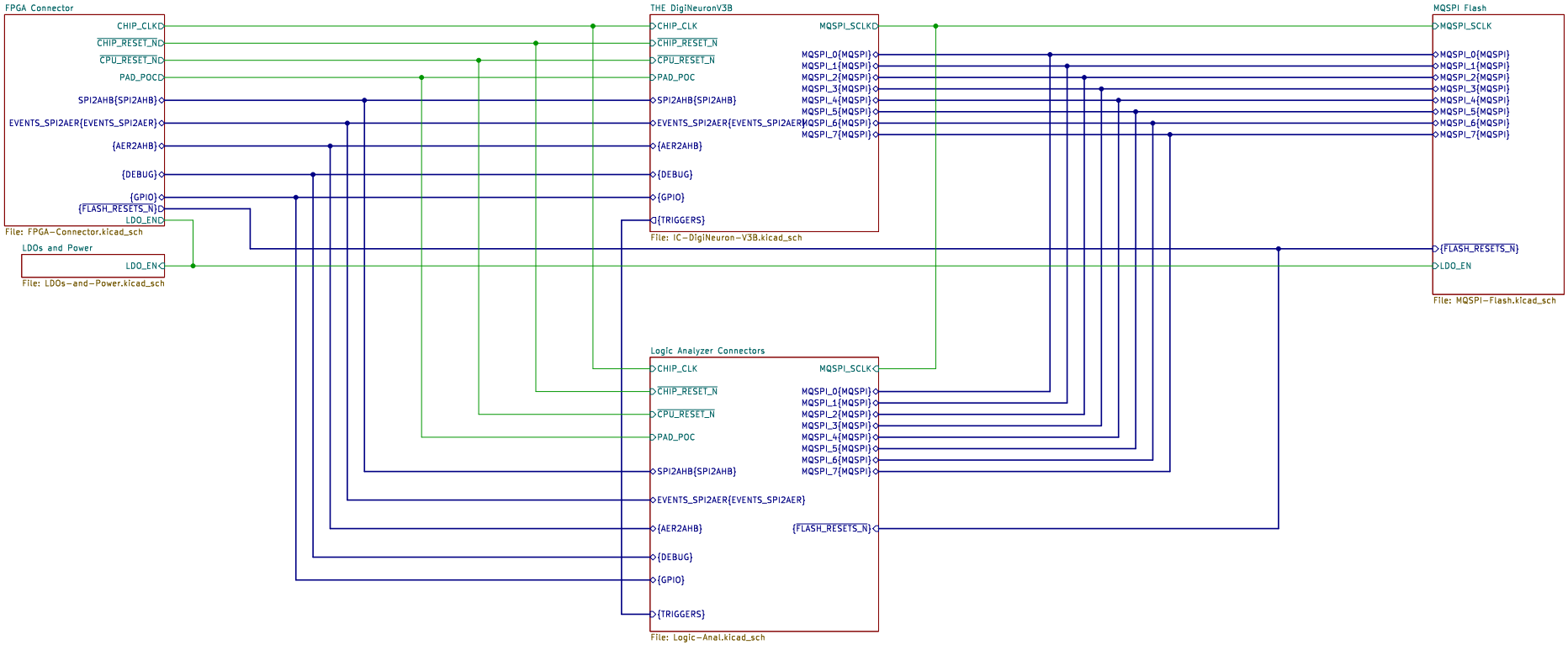



Figura C.2: Pinout del SoC.

# Anexo D

# Esquemático de la PCB de Testeo

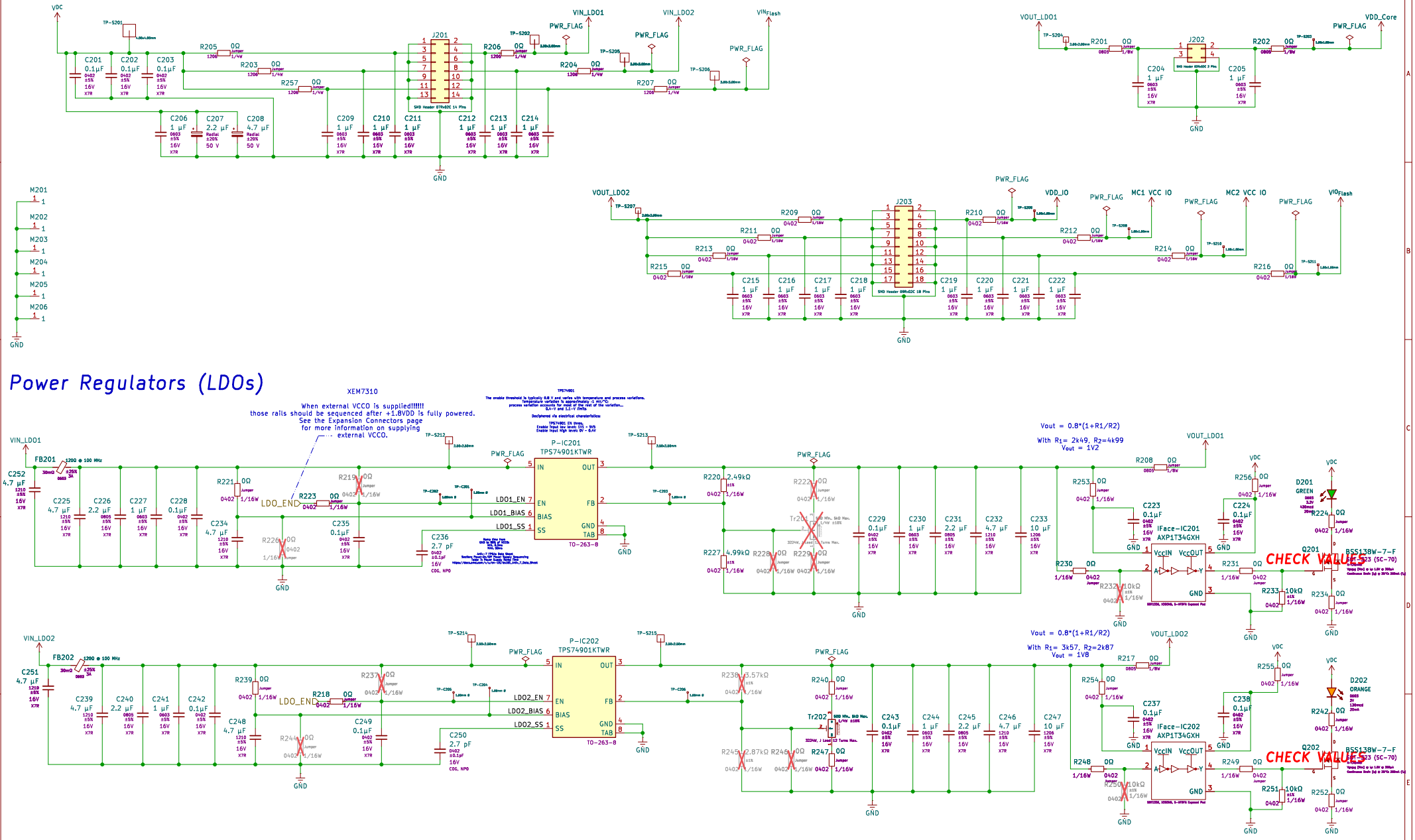


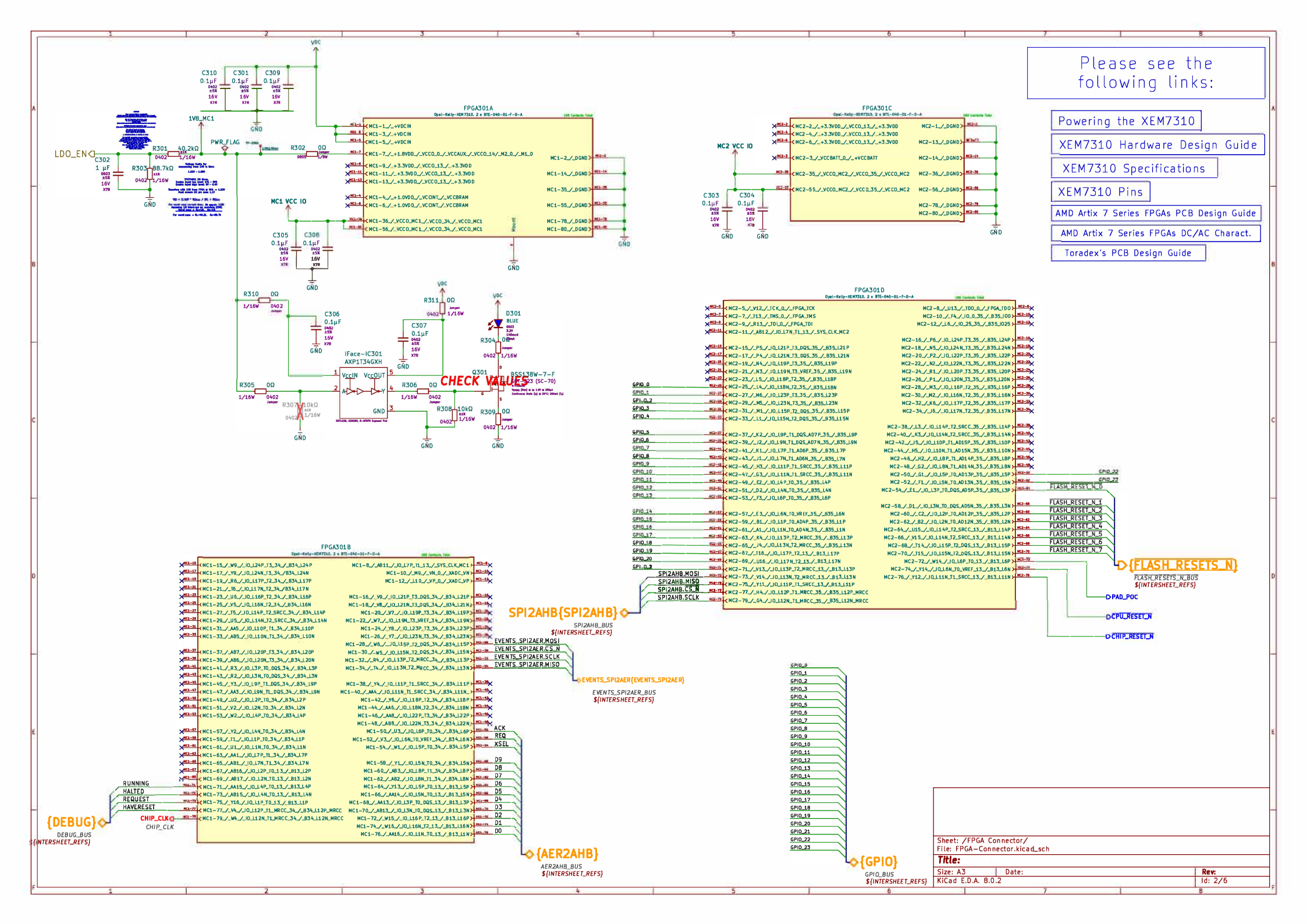

**Silicon Austria Labs GmbH**

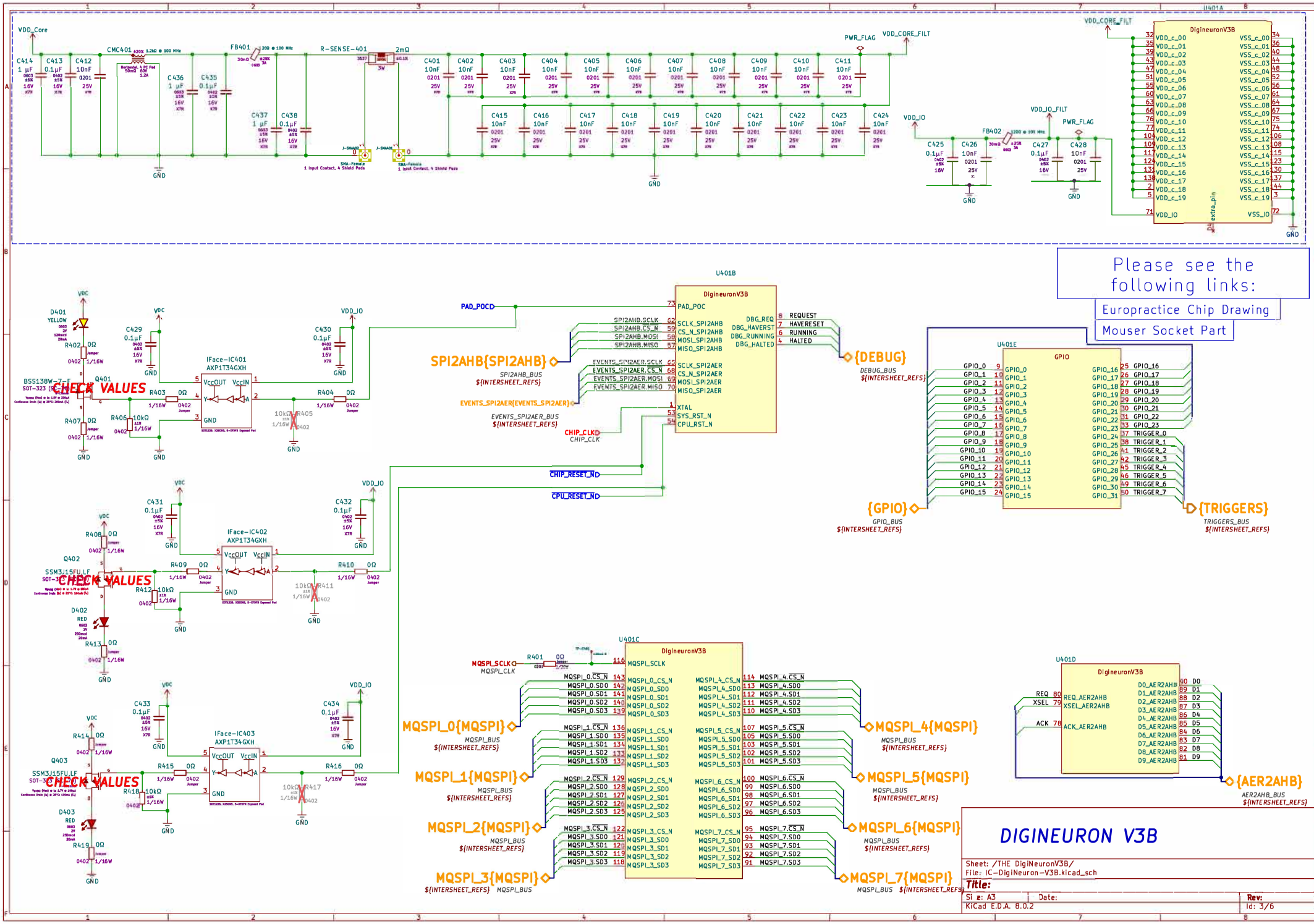
Sheet: /
File: DigiNeuron-V3B.kicad_sch

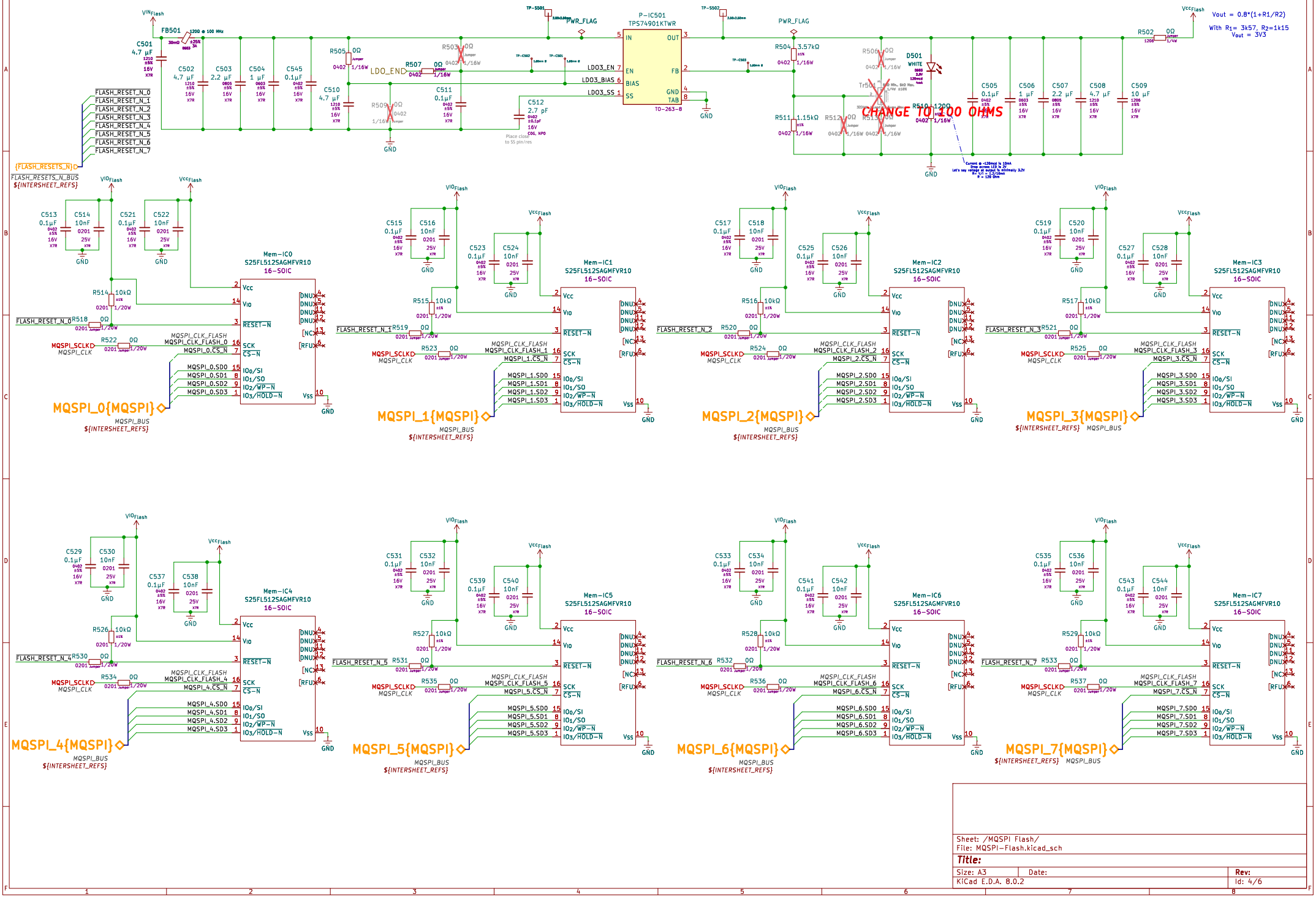
***Title: Digineuron_v3b_BringUp_XEM7310***

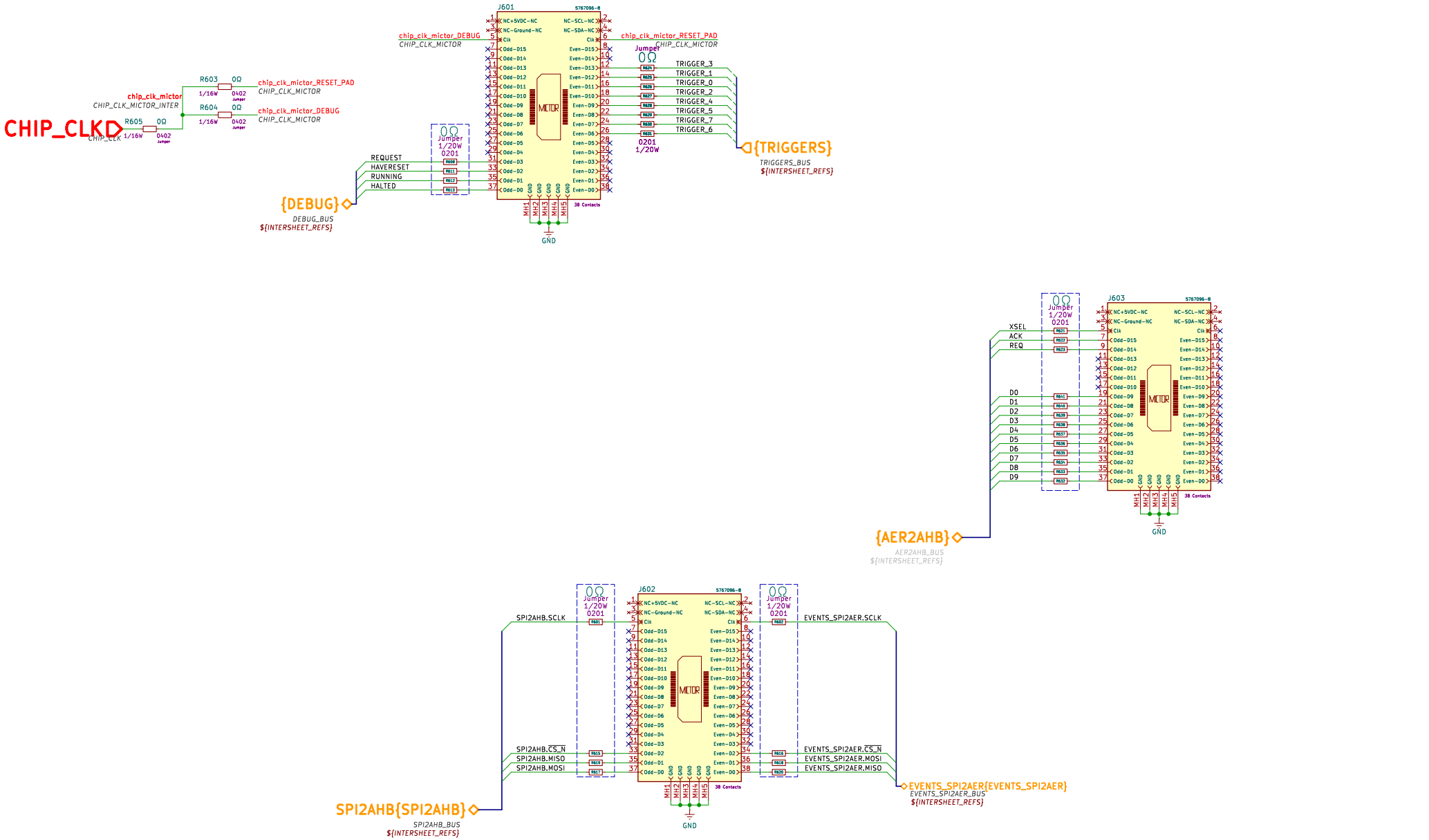
| Size: A3 | Date: 2024-06-10 | **Rev: 0.1** |
|---|---|---|
| KiCad E.D.A. 8.0.2 | | Id: 0/6 |


Power Regulators (LDOs)
XEM7310
When external VCCO is supplied!!!!!!
those rails should be sequenced after +1.8VDD is fully powered.
See the Expansion Connectors page
for more information on supplying
external VCCO.
VIN_LDO1
VIN_LDO2
VOUT_LDO1
VOUT_LDO2
VDD_Core
VDD_IO
MC1 VCC IO
MC2 VCC IO
VIO_Flash
P-IC201
P-IC202
TPS74901KTWR
TO-263-8
LDO_EN
LDO1_EN
LDO1_BIAS
LDO1_SS
LDO2_EN
LDO2_BIAS
LDO2_SS
Vout = 0.8*(1+R1/R2)
With R1= 2k49, R2=4k99
Vout = 1V2
With R1= 3k57, R2=2k87
Vout = 1V8
IFace-IC201
IFace-IC202
AXP1T34GXH
CHECK VALUES
D201
GREEN
D202
ORANGE
Q201
Q202
BSS138W-7-F
GND
PWR_FLAG
J201
J202
J203
M201
M202
M203
M204
M205
M206
Sheet: /LDOs and Power/
File: LDOs-and-Power.kicad_sch
Title:
Size: A3
Date:
Rev:
KiCad E.D.A. 8.0.2
Id: 1/6


Please see the following links:
Powering the XEM7310
XEM7310 Hardware Design Guide
XEM7310 Specifications
XEM7310 Pins
AMD Artix 7 Series FPGAs PCB Design Guide
AMD Artix 7 Series FPGAs DC/AC Charact.
Toradex's PCB Design Guide
FPGA301A
FPGA301B
FPGA301C
FPGA301D
MC1 VCC IO
MC2 VCC IO
CHECK VALUES
IFace-IC301
AXP1T34GXH
Q301
BSS138W-7-F
D301
BLUE
SPI2AHB{SPI2AHB}
SPI2AHB_BUS
${INTERSHEET_REFS}
EVENTS_SPI2AER{EVENTS_SPI2AER}
EVENTS_SPI2AER_BUS
{DEBUG}
DEBUG_BUS
{AER2AHB}
AER2AHB_BUS
{GPIO}
GPIO_BUS
{FLASH_RESETS_N}
FLASH_RESETS_N_BUS
PAD_POC
CPU_RESET_N
CHIP_RESET_N
Sheet: /FPGA Connector/
File: FPGA-Connector.kicad_sch
Title:
Size: A3
Date:
Rev:
KiCad E.D.A. 8.0.2
Id: 2/6


VDD_Core
CMC401
FB401
R-SENSE-401
2mΩ
PWR_FLAG
VDD_CORE_FILT
U401A
DigineuronV3B
VDD_IO
VDD_IO_FILT
FB402
GND
Please see the following links:
Europractice Chip Drawing
Mouser Socket Part
U401B
PAD_POCD
SPI2AHB{SPI2AHB}
SPI2AHB_BUS
${INTERSHEET_REFS}
EVENTS_SPI2AER{EVENTS_SPI2AER}
EVENTS_SPI2AER_BUS
CHIP_CLKD
CHIP_CLK
CHIP_RESET_ND
CPU_RESET_ND
{DEBUG}
DEBUG_BUS
U401E
GPIO
{GPIO}
GPIO_BUS
{TRIGGERS}
TRIGGERS_BUS
U401C
MQSPI_SCLK
MQSPI_CLK
MQSPI_0{MQSPI}
MQSPI_1{MQSPI}
MQSPI_2{MQSPI}
MQSPI_3{MQSPI}
MQSPI_4{MQSPI}
MQSPI_5{MQSPI}
MQSPI_6{MQSPI}
MQSPI_7{MQSPI}
MQSPI_BUS
U401D
{AER2AHB}
AER2AHB_BUS
IFace-IC401
IFace-IC402
IFace-IC403
AXP1T34GXH
CHECK VALUES
Q401
Q402
Q403
D401
D402
D403
YELLOW
RED
BSS138W-7-F
SSM3J15FU,LF
DIGINEURON V3B
Sheet: /THE DigiNeuronV3B/
File: IC-DigiNeuron-V3B.kicad_sch
Title:
Size: A3
Date:
Rev:
KiCad E.D.A. 8.0.2
Id: 3/6


VINFlash
FB501
120Ω @ 100 MHz
C501 4.7 µF
C502 4.7 µF
C503 2.2 µF
C504 1 µF
C545 0.1µF
C510 4.7 µF
C511 0.1µF
C512 2.7 pF
R505 0Ω
R507 0Ω
R503 0Ω
R509 0Ω
LDO_EN
PWR_FLAG
P-IC501
TPS74901KTWR
IN
OUT
EN
FB
BIAS
GND
SS
TAB
TO-263-8
LDO3_EN
LDO3_BIAS
LDO3_SS
GND
R504 3.57kΩ
R506 0Ω
R511 1.15kΩ
R512 0Ω
R513 0Ω
R510 120Ω
D501 WHITE
CHANGE TO 100 OHMS
C505 0.1µF
C506 1 µF
C507 2.2 µF
C508 4.7 µF
C509 10 µF
R502 0Ω
VccFlash
Vout = 0.8*(1+R1/R2)
With R1= 3k57, R2=1k15
Vout = 3V3
FLASH_RESET_N_0
FLASH_RESET_N_1
FLASH_RESET_N_2
FLASH_RESET_N_3
FLASH_RESET_N_4
FLASH_RESET_N_5
FLASH_RESET_N_6
FLASH_RESET_N_7
{FLASH_RESETS_N}
FLASH_RESETS_N_BUS
${INTERSHEET_REFS}
VIOFlash
C513 0.1µF
C514 10nF
C521 0.1µF
C522 10nF
R514 10kΩ
R518 0Ω
R522 0Ω
MQSPI_CLK_FLASH
MQSPI_CLK_FLASH_0
MQSPI_0.CS_N
MQSPI_SCLK
MQSPI_CLK
MQSPI_0.SD0
MQSPI_0.SD1
MQSPI_0.SD2
MQSPI_0.SD3
Mem-IC0
S25FL512SAGMFVR10
16-SOIC
Vcc
Vio
RESET-N
SCK
CS-N
IO0/SI
IO1/SO
IO2/WP-N
IO3/HOLD-N
Vss
DNU
NC
RFU
MQSPI_0{MQSPI}
MQSPI_BUS
C515 0.1µF
C516 10nF
C523 0.1µF
C524 10nF
R515 10kΩ
R519 0Ω
R523 0Ω
MQSPI_1.CS_N
MQSPI_1.SD0
MQSPI_1.SD1
MQSPI_1.SD2
MQSPI_1.SD3
Mem-IC1
MQSPI_1{MQSPI}
C517 0.1µF
C518 10nF
C525 0.1µF
C526 10nF
R516 10kΩ
R520 0Ω
R524 0Ω
MQSPI_2.CS_N
MQSPI_2.SD0
MQSPI_2.SD1
MQSPI_2.SD2
MQSPI_2.SD3
Mem-IC2
MQSPI_2{MQSPI}
C519 0.1µF
C520 10nF
C527 0.1µF
C528 10nF
R517 10kΩ
R521 0Ω
R525 0Ω
MQSPI_3.CS_N
MQSPI_3.SD0
MQSPI_3.SD1
MQSPI_3.SD2
MQSPI_3.SD3
Mem-IC3
MQSPI_3{MQSPI}
C529 0.1µF
C530 10nF
C537 0.1µF
C538 10nF
R526 10kΩ
R530 0Ω
R534 0Ω
MQSPI_4.CS_N
MQSPI_4.SD0
MQSPI_4.SD1
MQSPI_4.SD2
MQSPI_4.SD3
Mem-IC4
MQSPI_4{MQSPI}
C531 0.1µF
C532 10nF
C539 0.1µF
C540 10nF
R527 10kΩ
R531 0Ω
R535 0Ω
MQSPI_5.CS_N
MQSPI_5.SD0
MQSPI_5.SD1
MQSPI_5.SD2
MQSPI_5.SD3
Mem-IC5
MQSPI_5{MQSPI}
C533 0.1µF
C534 10nF
C541 0.1µF
C542 10nF
R528 10kΩ
R532 0Ω
R536 0Ω
MQSPI_6.CS_N
MQSPI_6.SD0
MQSPI_6.SD1
MQSPI_6.SD2
MQSPI_6.SD3
Mem-IC6
MQSPI_6{MQSPI}
C535 0.1µF
C536 10nF
C543 0.1µF
C544 10nF
R529 10kΩ
R533 0Ω
R537 0Ω
MQSPI_7.CS_N
MQSPI_7.SD0
MQSPI_7.SD1
MQSPI_7.SD2
MQSPI_7.SD3
Mem-IC7
MQSPI_7{MQSPI}
GND
Sheet: /MQSPI Flash/
File: MQSPI-Flash.kicad_sch
Title:
Size: A3
Date:
Rev:
KiCad E.D.A. 8.0.2
Id: 4/6




# LOGIC ANALYZER CONNECTORS

Sheet: /Logic Analyzer Connectors/
File: Logic-Anal.kicad_sch

**Title:**

| Size: A3 | Date: | **Rev:** |
|---|---|---|
| KiCad E.D.A. 8.0.2 | | Id: 5/6 |

# Anexo E

# Sistema de Prueba

Concluido el diseño del ASIC, fue necesario disponer de una plataforma capaz de excitar todas sus señales una vez fabricado y encapsulado. Para ello se desarrolló, sobre una FPGA, una arquitectura de prueba con periféricos específicos que permiten controlar cada entrada y monitorizar cada salida del chip. Entre las alternativas del mercado, las placas de Opal Kelly—basadas en FPGAs AMD (antes Xilinx)—resultaron idóneas: combinan conectores de alta densidad, que facilitan su acoplamiento a una PCB ad hoc para el ASIC, con la infraestructura *FrontPanel*. Esta expone una serie de registros, llamados *Endpoints*, accesibles a alta velocidad desde un PC mediante USB y una API en Python, simplificando la generación de estímulos y la captura de respuestas. Existen tres tipos de Endpoints: *Wires*, para fijar o leer estados lógicos; *Triggers*, que lanzan eventos como la inicialización de una máquina de estados; y *Pipes*, destinados a transferencias multibyte de gran volumen.

La arquitectura de verificación (Fig. E.1) se organiza en torno a un bus AMBA AHB-Lite. El bloque controlador **ok2AHB** recibe comandos desde la interfaz *Pipes* de FrontPanel; estos se almacenan temporalmente en una FIFO y, a continuación, se traducen en transacciones AHB que el puerto maestro envía a los periféricos conectados al ASIC. La plataforma incluye dos controladores SPI: el primero dialoga con el bloque **spi2ahb** del chip y el segundo con el receptor de eventos **spi2aer**. Se integra asimismo un GPIO de 32 líneas—la misma IP que aloja el ASIC—para intercambiar datos o señalizar hitos de ejecución con la conmutación de un bit. Finalmente, se diseñó un emisor de eventos AER conectado

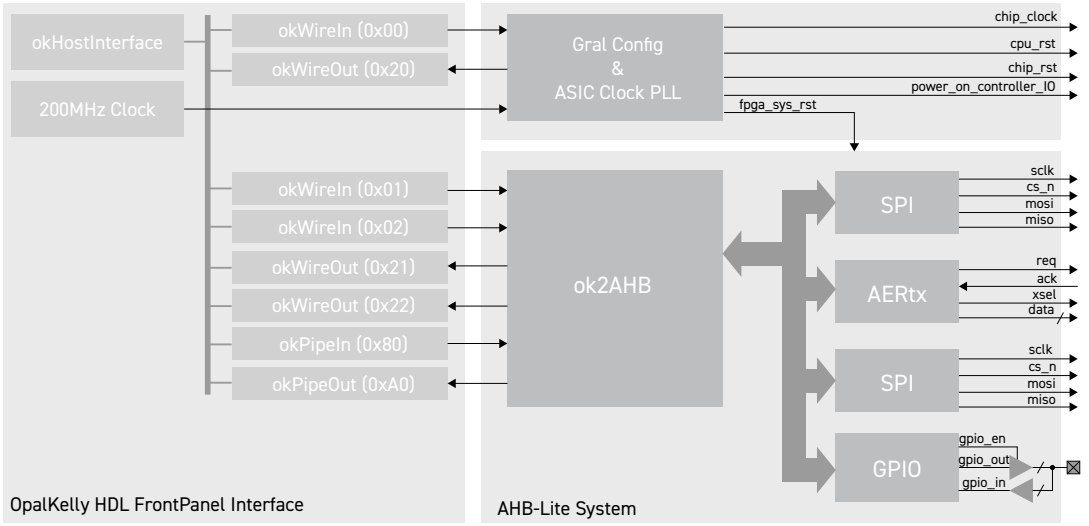



Figura E.1: Diagrama en bloques de la arquitectura implementada en la FPGA OpalKelly para verificar el funcionamiento del ASIC DigineuronV3b.

al puerto de entrada del bloque **aer2ahb**. Aunque dicho puerto está destinado a la cámara DAVIS346 Color, durante la verificación resulta imprescindible inyectar patrones de datos controlados, que luego se leen mediante **spi2ahb** y se contrastan con los valores originales.

El reloj del ASIC lo suministra la FPGA: un bloque PLL toma la referencia de 200 MHz proveniente de un cristal en la PCB de la propia FPGA y la multiplica hasta 400 MHz. Esta señal de alta velocidad llega a un divisor programable cuyo número de ciclos en nivel bajo y en nivel alto se ajusta dinámicamente mediante Endpoints tipo *Wire*; así se genera un reloj más lento, con frecuencia y ciclo de trabajo configurables, que se aplica al pin de reloj del ASIC. Además del reloj, el chip requiere tres líneas de control: el reset asincrónico global, el reset asincrónico dedicado al microprocesador, y la señal **power-on controller** exigida por TSMC, que, mientras permanece en nivel alto, habilita la lógica de pads para permitir la transmisión y recepción de datos.

La interconexión física entre la placa FPGA y el chip se resuelve con una PCB ad hoc (Fig. E.2). Dos conectores Samtec de alta densidad enlazan la Opal-Kelly con la tarjeta, mientras que ocho memorias Flash QSPI Infineon s25fl512s de 512 Mb se cablean directamente a los ocho buses del subsistema multi-QSPI del ASIC. Para igualar las longitudes de pista y evitar errores de temporización, cuatro memorias se montan en la cara frontal y las otras cuatro, simétricamente,

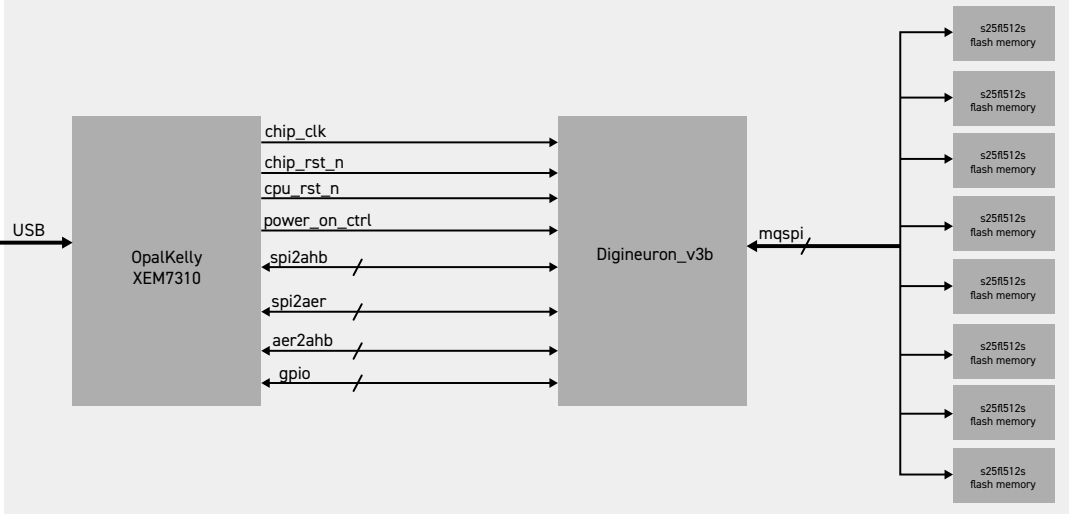



Figura E.2: Diagrama en bloques del interconexionado del ASIC DigineuronV3b con el ecosistema de prueba.

en la trasera, a la misma distancia del encapsulado. En la cara frontal (Fig. E.3) se añade una resistencia shunt de 0,2 Ω de alta precisión en la línea de alimentación del chip; la caída de tensión sobre ella permite medir el consumo con un osciloscopio. Un conjunto de jumpers habilita o desconecta tres reguladores situados en la cara posterior: uno fija los 1.2 V de alimentación (65 nm TSMC), otro ajusta la tensión de E/S (1.8 V, 2.5 V o 3.3 V) y el tercero suministra los 3.3 V requeridos por las memorias Flash.

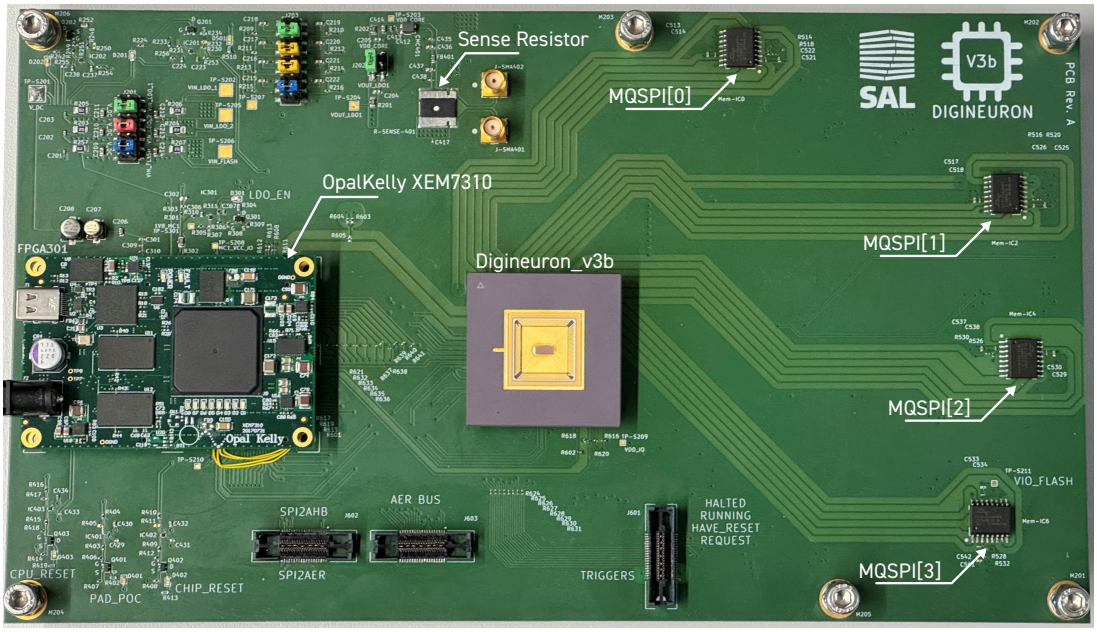



Figura E.3: Vista frontal de la placa PCB diseñada para la verificación del ASIC DigineuronV3b.

# Bibliografía


[1] G. Gallego, T. Delbrück, G. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. J. Davison, J. Conradt, K. Daniilidis, and D. Scaramuzza, "Event-based vision: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 1, pp. 154–180, 2022.

[2] J. H. Lin, P. O. Pouliquen, A. G. Andreou, A. C. Goldberg, and C. G. Rizk, "Flexible readout and integration sensor (FRIS): a bio-inspired, system-on-chip, event-based readout architecture," *Infrared Technology and Applications XXXVIII*, pp. 83 531N–83 531N–16, 2012.

[3] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[4] B. Graham, "Sparse 3d convolutional neural networks," 2015. [Online]. Available: https://arxiv.org/abs/1505.02890

[5] C. Schuldt, I. Laptev, and B. Caputo, "Recognizing human actions: a local svm approach," in *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, vol. 3, 2004, pp. 32–36 Vol.3.

[6] K. Soomro, A. R. Zamir, and M. Shah, "Ucf101: A dataset of 101 human actions classes from videos in the wild," 2012. [Online]. Available: https://arxiv.org/abs/1212.0402

[7] M. A. Mahowald and C. Mead, "The silicon retina," *Scientific American*, vol. 264, no. 5, pp. 76–82, may 1991.

[8] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128× 128 120 db 15 $\mu$s latency asynchronous temporal contrast vision sensor," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 2, pp. 566–576, 2008.

[9] C. Posch, D. Matolin, and R. Wohlgenannt, "A qvga 143 db dynamic range frame-free pwm image sensor with lossless pixel-level video compression and time-domain cds," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 259–275, 2011.

[10] C. Brandli, R. Berner, M. Yang, S.-C. Liu, and T. Delbruck, "A 240 × 180 130 db 3 µs latency global shutter spatiotemporal vision sensor," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 10, pp. 2333–2341, 2014.

[11] iniVation. (2023) inivation - neuromorphic vision systems. [Online]. Available: https://inivation.com/

[12] ——. (2024) prophesee - metavision technologies. [Online]. Available: https://www.prophesee.ai/

[13] D. Falanga, K. Kleber, and D. Scaramuzza, "Dynamic obstacle avoidance for quadrotors with event cameras," *Science Robotics*, vol. 5, no. 40, p. eaaz9712, 2020. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.aaz9712

[14] B. Forrai, T. Miki, D. Gehrig, M. Hutter, and D. Scaramuzza, "Event-based agile object catching with a quadrupedal robot," 2023. [Online]. Available: https://arxiv.org/abs/2303.17479

[15] A. M. I. Maqueda, A. Loquercio, G. Gallego, N. García, and D. Scaramuzza, "Event-based vision meets deep learning on steering prediction for self-driving cars," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5419–5427, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:4610262

[16] G. Chen, F. Wang, W. Li, L. Hong, J. Conradt, J. Chen, Z. Zhang, Y. Lu, and A. Knoll, "Neuroiv: Neuromorphic vision meets intelligent vehicle towards safe driving with a new database and baseline evaluations," *IEEE*

*Transactions on Intelligent Transportation Systems*, vol. 23, no. 2, pp. 1171–1183, 2022.

[17] E. Perot, P. de Tournemire, D. Nitti, J. Masci, and A. Sironi, "Learning to detect objects with a 1 megapixel event camera," 2020. [Online]. Available: https://arxiv.org/abs/2009.13436

[18] P. Bhattacharyya, J. Mitton, R. Page, O. Morgan, B. Menzies, G. Homewood, K. Jacobs, P. Baesso, D. Trickett, C. Mair, T. Muhonen, R. Clark, L. Berridge, R. Vigars, and I. Wallace, "Helios: An extremely low power event-based gesture recognition for always-on smart eyewear," 2024. [Online]. Available: https://arxiv.org/abs/2407.05206

[19] D. G. Ivanovich, G. Vilches, P. Julián, and G. F. Moroni, "Noise characterization of an event-based imager," in *2021 Argentine Conference on Electronics (CAE)*, 2021, pp. 32–39.

[20] D. G. Ivanovich, C. Xu, and P. Julián, "Event-based hand shadow recognition with varied light intensity and background subtraction," in *2021 55th Asilomar Conference on Signals, Systems, and Computers*, 2021, pp. 1121–1124.

[21] S. F. Müller-Cleve, V. Fra, L. Khacef, A. Pequeño-Zurro, D. Klepatsch, E. Forno, D. G. Ivanovich, S. Rastogi, G. Urgese, F. Zenke, and C. Bartolozzi, "Braille letter reading: A benchmark for spatio-temporal pattern recognition on neuromorphic hardware," *Frontiers in Neuroscience*, vol. 16, 2022. [Online]. Available: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2022.951164

[22] N. Rodríguez, M. Villemur, D. Klepatsch, D. G. Ivanovich, and P. Julián, "System on chip testbed for deep neuromorphic neural networks," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2023, pp. 1–5.

[23] N. Rodríguez, D. G. Ivanovich, M. Villemur, and P. Julián, "Risc-v based soc platform for neural network acceleration," in *2024 Argentine Conference on Electronics (CAE)*, 2024, pp. 142–147.

[24] D. Gigena Ivanovich and P. Julián, "Method for data processing of information data, in particular measurement data from an event-based image sensor, and device for this purpose," European Union Patent EP24 201 923.0, 9 24, 2024.

[25] X. Lagorce, G. Orchard, F. Galluppi, B. E. Shi, and R. B. Benosman, "Hots: A hierarchy of event-based time-surfaces for pattern recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 7, pp. 1346–1359, 2017.

[26] R. Benosman, C. Clercq, X. Lagorce, S.-H. Ieng, and C. Bartolozzi, "Event-based visual flow," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 2, pp. 407–417, 2014.

[27] Y. Sekikawa, K. Hara, and H. Saito, "Eventnet: Asynchronous recursive event processing," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[28] D. Gehrig and D. Scaramuzza, "Low-latency automotive vision with event cameras," *Nature*, vol. 629, no. 8014, pp. 1034–1040, 2024.

[29] T. Delbruck and C. A. Mead, "Time-derivative adaptive silicon photoreceptor array," in *Infrared Sensors: Detectors, Electronics, and Signal Processing*, T. S. J. Jayadev, Ed., vol. 1541, International Society for Optics and Photonics. SPIE, 1991, pp. 92 – 99. [Online]. Available: https://doi.org/10.1117/12.49323

[30] E. Culurciello, R. Etienne-Cummings, and K. Boahen, "A biomorphic digital image sensor," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 2, pp. 281–294, 2003.

[31] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[32] C. Mouton, J. C. Myburgh, and M. H. Davel, "Stride and translation invariance in cnns," in *Artificial Intelligence Research*, A. Gerber, Ed. Cham: Springer International Publishing, 2020, pp. 267–281.

[33] B. Graham, "Spatially-sparse convolutional neural networks," 2014. [Online]. Available: https://arxiv.org/abs/1409.6070

[34] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986.

[35] B. Graham, M. Engelcke, and L. v. d. Maaten, "3d semantic segmentation with submanifold sparse convolutional networks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 9224–9232.

[36] N. Messikommer, D. Gehrig, A. Loquercio, and D. Scaramuzza, "Event-based asynchronous sparse convolutional networks," in *Computer Vision – ECCV 2020*, A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, Eds. Cham: Springer International Publishing, 2020, pp. 415–431.

[37] M. A. Qureshi and A. Munir, "Sparse-pe: A performance-efficient processing engine core for sparse convolutional neural networks," *IEEE Access*, vol. 9, pp. 151 458–151 475, 2021.

[38] X. Wang, J. Yu, C. Augustine, R. Iyer, and R. Das, "Bit prudent in-cache acceleration of deep convolutional neural networks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 81–93.

[39] H. Caminal, K. Yang, S. Srinivasa, A. K. Ramanathan, K. Al-Hawaj, T. Wu, V. Narayanan, C. Batten, and J. F. Martínez, "Cape: A content-addressable processing engine," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 557–569.

[40] E. Garzón, A. Teman, M. Lanuzza, and L. Yavits, "Aida: Associative in-memory deep learning accelerator," *IEEE Micro*, vol. 42, no. 6, pp. 67–75, 2022.

[41] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 319–333. [Online]. Available: https://doi.org/10.1145/3352460.3358275

[42] O. Moreira, A. Yousefzadeh, F. Chersi, A. Kapoor, R.-J. Zwartenkot, P. Qiao, G. Cinserin, M. Khoei, M. Lindwer, and J. Tapson, "Neuronflow: A hybrid neuromorphic – dataflow processor architecture for ai workloads," in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2020, pp. 01–05.

[43] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. 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*, vol. 34, no. 10, pp. 1537–1557, 2015.

[44] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M. D. Flickner, and D. S. Modha, "Convolutional networks for fast, energy-efficient neuromorphic computing," *Proceedings of the National Academy of Sciences*, vol. 113, no. 41, p. 11441–11446, Sep. 2016. [Online]. Available: http://dx.doi.org/10.1073/pnas.1604850113

[45] A. Amir, B. Taba, D. Berg, T. Melano, J. McKinstry, C. Di Nolfo, T. Nayak, A. Andreopoulos, G. Garreau, M. Mendoza, J. Kusnitz, M. Debole, S. Esser, T. Delbruck, M. Flickner, and D. Modha, "A low power, fully event-based gesture recognition system," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 7388–7397.

[46] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[47] L. Bamberg, A. Pourtaherian, L. Waeijen, A. Chahar, and O. Moreira, "Synapse compression for event-based convolutional-neural-network accelerators," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 4, pp. 1227–1240, 2023.

[48] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (cam) circuits and architectures: a tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2006.

[49] IniVation, "DAVIS 346 AER," Datasheet, Nov. 2022, retrieved from https://inivation.com/wp-content/uploads/2023/07/DAVIS346-AER.pdf.

[50] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello, "Snowflake: An efficient hardware accelerator for convolutional neural networks," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[51] N. Hurley and S. Rickard, "Comparing measures of sparsity," *IEEE Transactions on Information Theory*, vol. 55, no. 10, pp. 4723–4741, 2009.

[52] A. T. Do, C. Yin, K. Velayudhan, Z. C. Lee, K. S. Yeo, and T. T.-H. Kim, "0.77 fJ/bit/search Content Addressable Memory Using Small Match Line Swing and Automated Background Checking Scheme for Variation Tolerance," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 7, pp. 1487–1498, 2014.

[53] B. Rooseleer, S. Cosemans, and W. Dehaene, "A 65 nm, 850 MHz, 256 kbit, 4.3 pJ/access, ultra low leakage power memory using dynamic cell stability and a dual swing data link," *2011 Proceedings of the ESSCIRC (ESSCIRC)*, pp. 519–522, 2011.

[54] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor, "Converting static image datasets to spiking neuromorphic datasets using saccades," *Frontiers in neuroscience*, vol. 9, p. 437, 2015.

[55] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[56] Z. Yuan, Y. Liu, J. Yue, Y. Yang, J. Wang, X. Feng, J. Zhao, X. Li, and H. Yang, "Sticker: An energy-efficient multi-sparsity compatible accelerator for convolutional neural networks in 65-nm cmos," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 2, pp. 465–477, 2020.

[57] Y. Yang, Y. Liu, Z. Yuan, W. Sun, R. Liu, J. Wang, J. Yue, X. Feng, Z. Yuan, X. Li, and H. Yang, "A 65-nm energy-efficient interframe data reuse neural network accelerator for video applications," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 8, pp. 2574–2585, 2022.

[58] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, "Snap: An efficient sparse neural acceleration processor for unstructured sparse deep neural network inference," *IEEE Journal of Solid-State Circuits*, vol. 56, no. 2, pp. 636–647, 2021.

[59] C. Lin, Y. Liu, and D. Shang, "Orsas: An output row-stationary accelerator for sparse neural networks," *IEEE Access*, vol. 11, pp. 44 123–44 135, 2023.

[60] M. A. Sivilotti, *Wiring considerations in analog VLSI systems, with application to field-programmable networks.* California Institute of Technology, 1991.

[61] M. Mahowald, “Vlsi analogs of neuronal visual processing: a synthesis of form and function,” Ph.D. dissertation, California Institute of Technology Pasadena, 1992.

[62] “The address-event representation communcation protocol,” Intitute of Neuroinformatics, ETH Zurich, Standard, Feb. 1993, retrieved from https://tilde.ini.uzh.ch/ amw/scx/std002.pdf.