Universidad Zaragoza
1542

Trabajo Fin de Grado

Implementación de técnicas hardware para la tolerancia a fallos permanentes en las memorias on-chip de aceleradores CNN

Implementation of hardware techniques for permanent fault tolerance in on-chip memories of CNN accelerators

Autora

Alicia Lázaro Huerta

Directores

Alejandro Valero Bresó

Rubén Gran Tejero

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2025

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe remitirse a seceina@unizar.es dentro del plazo de depósito)

D./Dª. Alicia Lázaro Huerta ,

en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de Estudios de la titulación de

Grado en Ingeniería Informática (Título del Trabajo)

Implementación de técnicas hardware para la tolerancia a fallos permanentes en las memorias on-chip de aceleradores CNN

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 26/06/2025

Fdo:

AGRADECIMIENTOS

Quiero expresar mi más sincero agradecimiento a Alejandro Valero y Rubén Gran, mis tutores, así como a Darío Suárez, por su orientación, apoyo y paciencia a lo largo de todo este proyecto. Les agradezco especialmente haberme propuesto las prácticas curriculares que marcaron el inicio de este Trabajo de Fin de Grado. Las reuniones semanales durante todo el año han sido un pilar fundamental para el desarrollo del proyecto.

También quiero mencionar a Samuel Pérez, cuya ayuda ha sido fundamental en el proceso de síntesis y ejecución del código en la FPGA, y que siempre ha estado dispuesto a resolver cualquier duda con gran amabilidad y dedicación.

Implementación de técnicas hardware para la tolerancia a fallos permanentes en las memorias on-chip de aceleradores CNN

RESUMEN

Este Trabajo de Fin de Grado se enmarca en el contexto de la eficiencia energética y la autonomía tecnológica en el diseño de hardware para inteligencia artificial. Su objetivo principal es implementar y validar mecanismos de tolerancia a fallos permanentes en memorias *on-chip* de aceleradores de redes neuronales convolucionales (CNN, por sus siglas en inglés) alimentadas a tensiones agresivas por debajo del nivel de seguridad a efectos de minimizar el consumo energético, garantizando al mismo tiempo la precisión del sistema.

Este trabajo se ha desarrollado en el marco del proyecto Prueba de Concepto denominado RETORNNA y financiado por el Ministerio de Ciencia e Innovación, particularmente como una de sus tareas de investigación orientadas a la transferencia de conocimiento hacia soluciones prácticas, lo que refuerza su valor aplicado y estratégico.

Durante el desarrollo del trabajo se han analizado e implementado dos técnicas microarquitectónicas tolerantes a fallos del estado-del-arte: *flipping* y *patching*. La técnica *flipping* consiste en reordenar los bits de cada activación defectuosa para que los fallos se trasladen a posiciones de memoria menos significativas, donde su impacto en la precisión es limitado o inexistente. Por su parte, *patching* permite recuperar las activaciones más críticas desde una memoria cache auxiliar, libre de fallos, que opera a tensión nominal.

La metodología propuesta ha sido iterativa y jerárquica, comenzando por la implementación de prototipos funcionales en Logisim, que facilitaron la depuración visual y estructural de los diseños. A continuación, se desarrollaron los módulos en SystemVerilog, empleando la herramienta Verilator para su desarrollo y validación. Finalmente, se realizó la síntesis sobre una FPGA ZedBoard, lo que permitió obtener medidas reales de consumo, latencia y ocupación.

Las principales contribuciones de este trabajo incluyen el diseño de la metodología, la implementación completa de la arquitectura, la validación funcional con baterías de pruebas exhaustivas y la evaluación experimental del sistema. En comparación con los datos obtenidos en el artículo original, donde se estima mediante simulación que la técnica Flip-and-Patch aplicada a una memoria de activación de 2 MiB incrementa el consumo estático de 269,8 mW a 288,5 mW, la implementación realizada en este

proyecto presenta una sobrecarga energética significativamente menor, con un consumo total de solo 0,008 W. Esto significa que incrementaría el consumo estático de 269,8 mW a 278,8 mW. Asimismo, mientras que en el estudio original el retardo de acceso aumenta de 2,69 ns a 2,75 ns, en esta implementación el camino crítico mantiene un margen positivo de 0,398 ns para una frecuencia de operación de 100 MHz, cumpliendo con los requisitos de temporización. Estos resultados indican que la versión sintetizada en FPGA no solo es funcional, sino que también resulta notablemente eficiente en términos de consumo y temporización respecto a los valores estimados por simulación.

Índice

Capítulo 1

Introducción

En este capítulo se presenta el contexto general del proyecto, incluyendo la motivación, los objetivos, el alcance y la organización de la memoria. También se introducen aspectos estratégicos relacionados con la producción de chips y la relevancia del trabajo en un marco más amplio de autonomía tecnológica.

1.1. Motivación

En la actualidad, la producción de circuitos integrados y aceleradores para inteligencia artificial se ha convertido en un asunto estratégico a nivel global. Diversos gobiernos están impulsando planes para fomentar la independencia tecnológica, reducir la dependencia de terceros países y reforzar la soberanía digital. Esta situación ha puesto en relieve la necesidad de disponer de capacidades propias de diseño, validación y fabricación de hardware crítico, especialmente en sectores como el procesamiento de datos, defensa o sanidad, entre otros.

En este contexto, los sistemas embebidos que integran aceleradores de redes neuronales cobran una especial relevancia. Estos aceleradores suelen operar en condiciones de energía limitada puesto que estos sistemas cuentan con baterías en lugar de encontrarse conectados a la red eléctrica. De esta manera, existe una gran cantidad de trabajos de investigación centrados en la propuesta de técnicas de ahorro energético en aceleradores hardware, como por ejemplo aquellas que persiguen una reducción agresiva de la tensión de alimentación. Sin embargo, esta estrategia puede provocar fallos permanentes en las memorias *on-chip* del acelerador, las cuales contienen los parámetros (activaciones y pesos) de la red neuronal, comprometiendo la fiabilidad del sistema [1, 2, 3, 4].

Los autores del artículo "*Flip-and-Patch: A fault-tolerant technique for on-chip memories of CNN accelerators at low supply voltage*" proponen dos técnicas microarquitectóricas —*flipping* y *patching*— que permiten mantener la precisión original

de las redes neuronales convolucionales a pesar de operar con una tensión de alimentación (V_{dd}) por debajo del margen de seguridad (V_{min}) [1]. Además, al contrario que trabajos previos, estas técnicas no requieren re-entrenar las redes ni intervención del programador. Su implementación práctica representa una contribución significativa para mejorar la robustez energética de los aceleradores actuales, dentro de una estrategia de autonomía tecnológica y eficiencia energética.

El presente trabajo se ha desarrollado en el marco del proyecto Prueba de Concepto denominado *RETORNNA: RISC-V-based Enhancements Towards Optimised and Resilient machiNe learNing Accelerators* financiado por el Ministerio de Ciencia e Innovación. En concreto, este trabajo conforma una de las tareas de investigación del proyecto orientada a la transferencia de conocimiento hacia soluciones prácticas.

1.2. Objetivos

El objetivo principal de este Trabajo de Fin de Grado es estudiar, implementar y validar los mecanismos de tolerancia a fallos permanentes en memorias on-chip, adaptados a aceleradores de redes neuronales convolucionales (CNN, por sus siglas en inglés), sin afectar al rendimiento del sistema y con bajo coste energético.

Para alcanzar este objetivo principal, se plantean los siguientes objetivos específicos:

- Implementar una arquitectura hardware que combine las técnicas propuestas para mitigar dichos fallos.
- Validar funcionalmente el diseño mediante simulaciones.
- Analizar el consumo energético y la complejidad del hardware mediante síntesis sobre una FPGA real.
- Desarrollar una metodología iterativa de implementación y pruebas que garantice la robustez del sistema.

1.3. Alcance

Este Trabajo de Fin de Grado abarca desde el estudio detallado del comportamiento de las memorias de activación bajo condiciones de baja tensión hasta la implementación y validación de las técnicas de tolerancia a fallos en un entorno de simulación hardware.

El desarrollo del proyecto se ha guiado por una metodología propia, basada en un enfoque iterativo e incremental, diseñada específicamente para facilitar la implementación, prueba y depuración de componentes hardware. Esta estrategia se

detalla en el Anexo Metodología Utilizada y ha demostrado ser eficaz para gestionar la complejidad del diseño hardware, garantizando la fiabilidad y la correcta integración de los módulos desarrollados.

Una vez verificados los bloques funcionales, se procedió a la síntesis del diseño sobre una FPGA, con el objetivo de analizar su impacto en el consumo energético, la frecuencia de funcionamiento y la utilización de recursos hardware.

Cabe destacar que este trabajo forma parte del proyecto RETORNNA, enmarcándose como una tarea concreta de desarrollo e implementación orientada a la mejora de la resiliencia de sistemas embebidos. Esta vinculación refuerza el valor del trabajo como contribución práctica a una línea de investigación activa y alineada con objetivos de transferencia tecnológica.

Para asegurar la trazabilidad, reproducibilidad y mantenimiento del proyecto, se ha organizado todo el desarrollo en una estructura de directorios alojada en un repositorio de Git. Este repositorio contiene tanto los ficheros fuente de las técnicas, como los scripts de pruebas y los ficheros del diseño. El repositorio público del proyecto está disponible en: `https://github.com/alissssia/Implementation-of-Hardware-acceleration-techniques-for-neural-networks.git`.

1.4. Descripción del Documento

El resto del presente documento se organiza como sigue. El Capítulo 2 analiza los conceptos clave y tecnologías relacionadas, así como las técnicas microarquitectónicas a implementar. El Capítulo 3 muestra la metodología empleada, incluyendo las herramientas principales utilizadas. Además, se describe la estrategia de desarrollo seguida durante el proyecto. El Capítulo 4 describe en detalle la arquitectura del sistema, la estructura de los módulos hardware y las técnicas aplicadas para integrar los mecanismos implementados. El Capítulo 5 presenta los resultados obtenidos en términos de consumo energético y uso de recursos. Por último, el Capítulo 6 resume los logros alcanzados y plantea posibles líneas de mejora o ampliación del trabajo en el futuro.

Capítulo 2

Antecedentes

En este capítulo se exploran los conceptos y tecnologías más relevantes que sirven como base para el desarrollo de este Trabajo de Fin de Grado. Se aborda el funcionamiento general de los aceleradores de redes neuronales convolucionales (CNN), prestando especial atención al papel de las memorias on-chip y su vulnerabilidad frente a fallos bajo condiciones de baja tensión. A continuación, se analizan las técnicas actuales de tolerancia a fallos objeto de implementación: *flipping* y *patching*, incluyendo sus implicaciones en términos de eficiencia energética y precisión. Por último, se introduce el contexto del uso de FPGAs como plataformas de validación hardware, dada su relevancia para prototipado y evaluación de arquitecturas digitales.

2.1. Aceleradores de Redes Neuronales Convolucionales

Las redes neuronales convolucionales (CNN) son un tipo de red neuronal ampliamente utilizada en tareas de reconocimiento y procesamiento de imagen, gracias a su capacidad para reconocer patrones en imágenes. La Figura 2.1 muestra una arquitectura típica de red neuronal convolucional, donde se pueden observar las diferentes capas: convolucionales (*Conv*), de agrupamiento (*Pool*), completamente conectadas (*FC*) y de clasificación final (*Softmax*). Este tipo de arquitecturas, como la mostrada en la figura, son representativas de redes clásicas como AlexNet, VGG o Resnet utilizadas ampliamente en tareas de clasificación de imágenes.

El proceso comienza con los datos de entrada (una imagen, por ejemplo) que atraviesa las capas convolucionales mientras se le aplican filtros para detectar características concretas (formas, texturas...). A medida que aumenta la profundidad de la red, más complejas pueden ser estas características, siendo la red capaz de identificar patrones más complejos. Aunque su principal aplicación es la visión por computador, las redes CNN también se utilizan en otros campos como procesamiento de lenguaje natural o

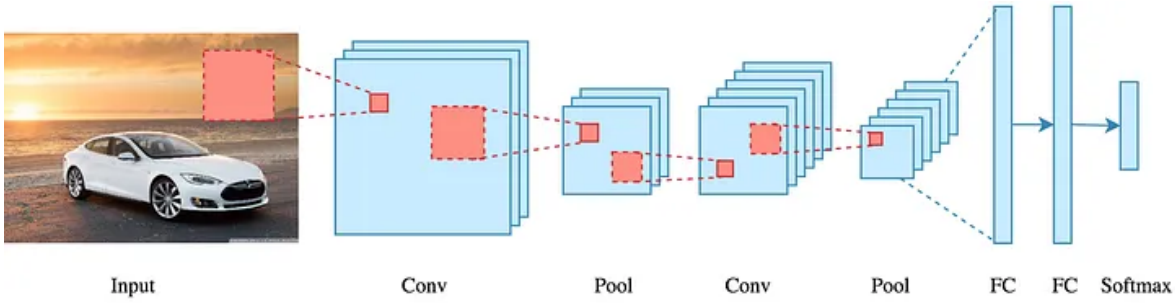


Figura 2.1: Arquitectura típica de una red neuronal convolucional. Figura tomada de [5].

clasificación de señales biomédicas, entre otros.

Sin embargo, la ejecución de las CNN implica un alto coste computacional por su requerimiento de realizar millones de operaciones aritméticas, principalmente multiplicaciones y sumas. Esto ha motivado el desarrollo de aceleradores hardware especializados que permiten ejecutar los modelos de forma más eficiente que los procesadores de propósito general, ya sean CPU o unidades de procesamiento gráfico (GPGPU).

Un acelerador de redes neuronales es un procesador especializado optimizado específicamente para realizar las operaciones típicas de una red neuronal, como las convoluciones. Estos elementos permiten alcanzar un mayor rendimiento con menor consumo energético y ocupación de área, haciéndolos especialmente adecuados para sistemas embebidos o aplicaciones con restricciones de potencia. Un ejemplo destacado son las *Tensor Processing Units* (TPU) de Google, diseñadas para llevar a cabo operaciones matriciales de manera eficiente.

La Figura 2.2 muestra la arquitectura interna de una TPU, donde se pueden observar sus componentes principales, incluyendo las memorias internas (on-chip) dedicadas a almacenar pesos (*Weight FIFO*) y activaciones (*Unified Buffer*), así como la unidad de ejecución matricial (MMU) y las conexiones con el *host*. Esta arquitectura refleja el diseño optimizado de aceleradores para redes neuronales profundas.

Aunque las TPU de Google están orientadas principalmente al cómputo de alto rendimiento en centros de datos, también existen versiones más compactas pensadas para entornos embebidos. Un ejemplo de ellos es la Edge TPU integrada en dispositivos como la Coral Dev Board, véase la Figura 2.3, diseñada para realizar aprendizaje automático en dispositivos con recursos limitados, permitiendo ejecutar modelos de inferencia sin depender del procesamiento en la nube.

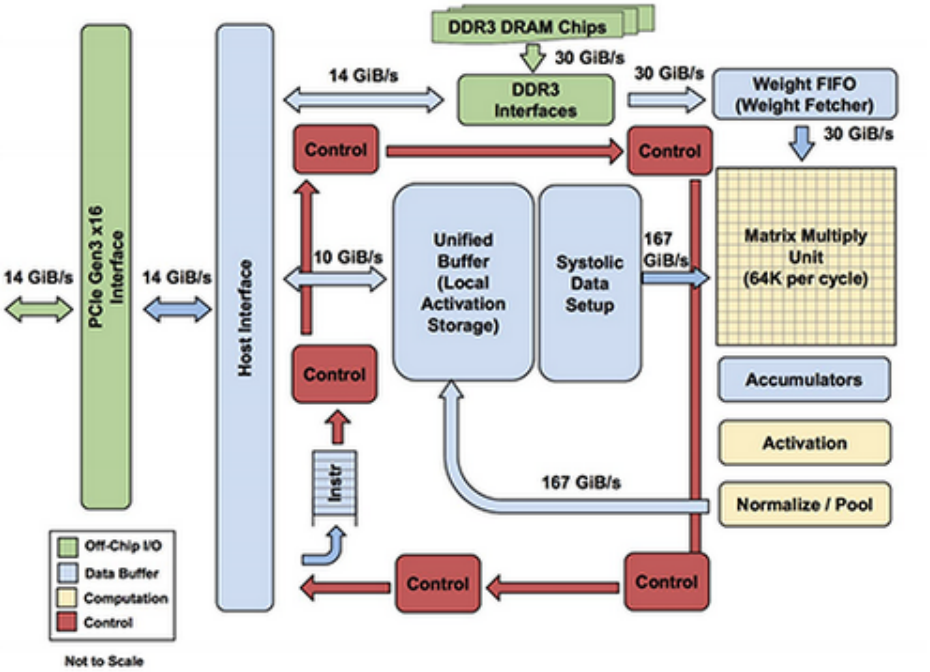


Figura 2.2: Diagrama de bloques de una TPU. Figura tomada de [6].

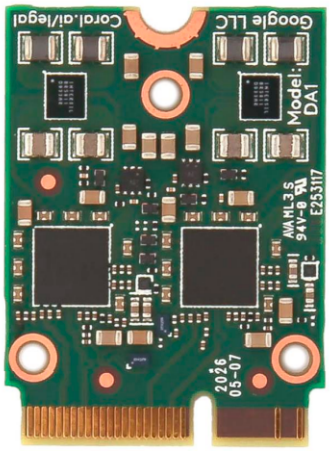


Figura 2.3: Detalle del dispositivo Coral TPU Dual Edge. Figura tomada de [7].

2.2. Memorias On-Chip en Aceleradores

Una memoria on-chip, también conocida como memoria interna, es una memoria que se encuentra integrada en el mismo chip que el procesador, en lugar de estar ubicada en un componente de memoria externo. Este tipo de diseño, no solo contribuye a reducir costes de producción, sino que permite disminuir el consumo energético y mejorar el rendimiento del sistema. Encontrándose estas memorias integradas en el mismo chip, se elimina la necesidad de comunicación entre elementos de memoria externos y separados, reduciendo la latencia. Además, la ausencia o una menor distancia física

entre el procesador y la memoria minimiza los retardos asociados al tiempo de respuesta.

En las redes CNN, las memorias on-chip se utilizan para almacenar los parámetros de la red, esto es, los pesos (*weights*), así como las activaciones (neuronas) intermedias entre las diferentes capas. Por ello, su acceso es crítico para el rendimiento del sistema acelerador. Tómese como ejemplo la Figura 2.2, donde se muestran varias memorias internas de la TPU. El buffer de activaciones (*Unified Buffer*) sirve para almacenar los datos intermedios del procesamiento, mientras que el buffer *Weight FIFO* se encarga de proporcionar de forma eficiente los pesos necesarios para las operaciones de multiplicación en la matriz principal.

2.3. Reducción de Tensión

La reducción dinámica de la tensión de alimentación (*Dynamic Voltage Scaling*) es un método de reducción del consumo medio en un sistema basado en computador, incluyendo a los sistemas embebidos y dentro de estos, a los aceleradores CNN. Esto se consigue ajustando dinámicamente la frecuencia y la tensión del circuito [8].

Típicamente, esta técnica se emplea en dispositivos que utilizan baterías, ya que el ahorro de energía y la duración de las baterías son primordiales. También se utiliza en entornos con múltiples procesadores en los que el ahorro de energía es importante por razones térmicas.

La reducción de la tensión tiene un impacto cuadrático en el consumo de energía, mientras que la frecuencia afecta linealmente. Esta relación se expresa mediante la siguiente fórmula:

$$P = \frac{1}{2} C V^2 f$$

Donde P es la potencia consumida, C es la capacitancia de carga, V es la tensión de alimentación y f la frecuencia de conmutación. Por lo tanto, disminuir ambos tiene un efecto muy significativo en la reducción del consumo total. Sin embargo, reducir la frecuencia implica aumentar el tiempo de ciclo del procesador, lo que cual se traduce en un mayor tiempo de ejecución para completar una tarea. Esto significa que aunque se consuman menos vatios por segundo, el procesador puede estar activo durante más tiempo, aumentando el consumo de energía total.

2.4. Tipos de Fallos en Memorias On-Chip

Cuando se aplican técnicas de reducción drástica de la tensión de alimentación (V_{dd}) por debajo del margen de seguridad (V_{min}), como parte de estrategias para mejorar la eficiencia energética, las memorias on-chip comienzan a experimentar un número

considerable de fallos permanentes. Estos fallos son debidos a variaciones inherentes al proceso de fabricación en tecnologías CMOS avanzadas.

Las celdas SRAM de 6 transistores utilizadas en las memorias on-chip son especialmente susceptibles a estos fallos bajo condiciones donde $V_{\text{dd}} < V_{\text{min}}$. En este contexto, un fallo permanente se traduce habitualmente en un bit que tiene un valor lógico fijo, sin capacidad de cambiar de estado independientemente del valor escrito.

El impacto de estos fallos depende en gran medida de la posición de los bits defectuosos dentro de las palabras de 16 bits (asumiendo un acelerador de 16 bits). En [1], se estudia la reducción de V_{dd} en las memorias de activación. Por tanto, se distinguen tres tipos de activaciones defectuosas:

- *Low-Order* (LO): fallos únicamente en el byte menos significativo. Representan el 0.45 % de las activaciones. Su impacto en la precisión de la red es despreciable.

- *High-Order* (HO): fallos únicamente en el byte más significativo. También suponen el 0.45 % del total, pero pueden degradar severamente la precisión.

- *Low- & High-Order* (L&HO): fallos simultáneos en ambos bytes. Son más infrecuentes (0.0035 %), pero extremadamente perjudiciales para la exactitud del modelo.

Estas estadísticas están basadas en una experimentación real sobre FPGAs fabricadas en tecnología de 32 nm [9], y sirven como base para las técnicas propuestas.

2.5. Técnicas de Tolerancia a Fallos

Las activaciones de tipo LO no afectan significativamente a la precisión de la red neuronal, mientras que las HO sí pueden comprometerla de forma considerable. Por este motivo, el propósito de las dos técnicas propuestas —*flipping* y *patching*— es minimizar el impacto de los bits defectuosos. La primera estrategia consiste precisamente en convertir las activaciones HO en LO, dado que los fallos en los bits de menor peso tienen una repercusión mucho más limitada.

El método *flipping* se basa en la siguiente idea. Suponiendo una representación *little-endian* y un tamaño de activación de 16 bits, el bit en la posición i-ésima de la palabra pasará a ocupar la posición $16 - i - 1$, tal como se detalla en la figura 2.4. De este modo, se invierte el orden de los bits y, por tanto, los fallos presentes en los bits más significativos se trasladan a las posiciones de menos significativas, reduciendo su impacto en el valor final de la activación.

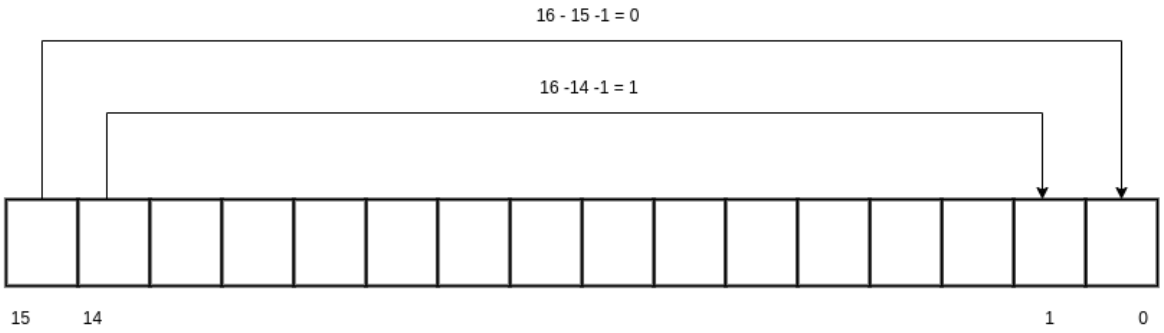


Figura 2.4: Esquema de funcionamiento de *flipping*.

Para identificar las activaciones HO, las cuales deben ser modificadas mediante *flipping*, se utiliza un bit de control por palabra denominado *f*. Este bit se establece mediante tests de memoria realizados en una etapa posterior a la fabricación del circuito pero con anterioridad a comercializarlo. Concretamente, se aplica la reducción de tensión y con los tests se identifican aquellas activaciones de tipo HO, marcando el bit f a '1' en este caso, o a '0' en caso contrario.

No obstante, el mecanismo *flipping* no resuelve el problema de las activaciones clasificadas como L&HO, en las que existen fallos tanto en los bits altos como en los bajos. Para estos casos, se propone la técnica *patching*, que emplea una pequeña memoria cache que opera a tensión nominal y por tanto se encuentra libre de fallos. Esta cache de respaldo permite almacenar una copia fiable de aquellas activaciones identificadas como L&HO. Dado que esta memoria es mucho más pequeña (2,5 KiB) que la memoria on-chip que almacena todas las activaciones (2 MiB), su impacto en el consumo energético total es mínimo.

Al igual que en el mecanismo *flipping*, es necesario marcar qué activaciones deben recuperarse desde la cache de respaldo. Para ello, se emplea un bit de control p por palabra, también establecidos mediante tests de memoria anteriores a la comercialización del dispositivo.

La Figura 2.5 muestra ambas técnicas incorporadas al puerto de lectura de una memoria on-chip de activaciones. El valor de cada activación a leer en cada operación de lectura, compuesta por un bloque de 16 activaciones, se realiza mediante un multiplexor 4:1. Este multiplexor elige entre tres posibles fuentes: la activación normal leída directamente desde la memoria on-chip (selección 0 en el mux); la versión corregida mediante *flipping* (selección 1); o la activación recuperada desde la memoria cache de respaldo (selección 2). Una vez transformada la activación según el caso, el bloque de activaciones se transmite al vector de elementos de procesamiento o *Processing Element Array*.

Cabe destacar que no es posible identificar una activación que sea simultáneamente

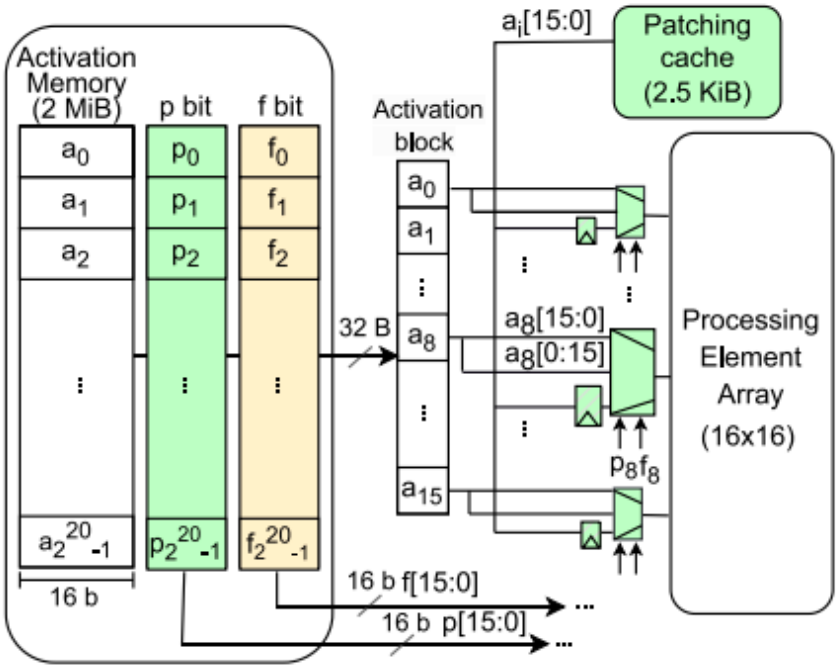


Figura 2.5: Puerto de lectura de una memoria on-chip de activación incluyendo las dos técnicas *flipping* (en amarillo) y *patching* (en verde). Cada lectura implica leer un bloque de activaciones de 16 bits. Figura tomada de [1].

de tipo LO y L&HO, por lo que no existen casos en los que los bits de f y p estén activos al mismo tiempo (selección 3 en el mux).

Se refiere al lector al artículo original [1] para más detalles acerca del diseño conceptual de las técnicas *flipping* y *patching*.

Capítulo 3

Metodología

Este capítulo describe las estrategias empleadas en el desarrollo de las técnicas, incluyendo la configuración del entorno y la metodología seguida para la consecución del proyecto.

3.1. Entorno de Desarrollo

Durante el desarrollo de las técnicas se han empleado diferentes herramientas, seleccionadas según las necesidades específicas de cada etapa del proyecto. Estas han incluido desde editores de texto para la escritura del código hasta simuladores para validar el funcionamiento de los módulos hardware.

Se ha optado por herramientas que facilitaran tanto la implementación como la prueba de los mecanismos desarrollados, permitiendo una integración fluida entre el código en SystemVerilog y los entornos de simulación y test. La combinación de herramientas gráficas y en línea de comandos ha permitido trabajar de forma eficiente en el diseño, depuración y validación del sistema.

La Tabla 3.1 recoge las principales herramientas utilizadas junto con sus versiones:
A continuación se explica brevemente el uso principal de cada herramienta:

- **Visual Studio Code**: utilizado como editor principal para la escritura y organización del código en SystemVerilog y C++.
- **C++**: lenguaje usado para los *testbenches* y *scripts* de prueba en conjunto con *Verilator*.
- **Verilator**: herramienta de simulación que ha permitido comprobar el funcionamiento de los módulos de SystemVerilog mediante pruebas escritas en C++.
- **Logisim**: se ha usado principalmente para diseñar y validar el prototipo funcional de la memoria cache. Su interfaz gráfica ha facilitado la depuración del diseño.

Herramienta	Versión
Visual Studio Code [10]	1.100.2
C++ [11]	13.3.0
Verilator [12]	5.027
Logisim [13]	2.7.1
Ubuntu [14]	24.04
Git [15]	2.43.0
Vivado [16]	2024.2
ZedBoard (FPGA) [17]	-

Tabla 3.1: Herramientas con sus versiones usadas en el desarrollo de las técnicas.

- **Ubuntu**: sistema operativo utilizado durante el desarrollo por su estabilidad y compatibilidad con herramientas de código abierto.
- **Git**: sistema de control de versiones empleado para gestionar el código y mantener un seguimiento de los cambios realizados durante el desarrollo.
- **AMD Vivado Design Suite**: herramienta de diseño utilizada para la síntesis, implementación y análisis de los resultados del proyecto.
- **FPGA ZedBoard Zynq-7000**: plataforma hardware de implementación.

3.2. Metodología de Diseño Digital

Para el desarrollo del proyecto se ha propuesto una **metodología propia** basada en un enfoque iterativo e incremental, especialmente adaptada a las necesidades del diseño hardware. Esta estrategia consiste en implementar primero bloques funcionales simples, verificarlos mediantes pruebas unitarias y reutilizarlos para construir módulos más complejos. Esta metodología ha permitido una integración progresiva del sistema, asegurando que cada componente estuviera probado antes de su integración.

Los detalles técnicos específicos de esta metodología se desarrollan con mayor profundidad en el Anexo A.

Capítulo 4

Propuesta de Implementación Hardware de los Mecanismos *Flipping* y *Patching*

Este capítulo recoge el proceso completo de implementación hardware de las técnicas de tolerancia a fallos propuestas, desde el diseño inicial hasta la validación funcional en entorno simulado. En primer lugar, se describe en detalle el desarrollo del mecanismo *flipping*, tanto a nivel de activación individual como de bloque, incluyendo la incorporación de biestables y su programa de pruebas. A continuación, se aborda la implementación de la memoria caché de respaldo necesaria para la técnica *patching*, comenzando por un prototipo en Logisim, el diseño de su unidad de control y su posterior adaptación y codificación en SystemVerilog. Seguidamente, se explica cómo se ha implementado el mecanismo *patching* y cómo se han integrado ambos mecanismos en un único sistema funcional. Finalmente, se presentan las baterías de pruebas de validación desarrolladas para garantizar la fiabilidad del sistema y la correcta interacción de todas las partes.

Para más detalles sobre la implementación, se puede consultar el siguiente repositorio `https://github.com/alissssia/Implementation-of-Hardware-acceleration-techniques-for-neural-networks.git`.

4.1. Visión General

La Figura 4.1 ilustra una visión general a alto nivel del funcionamiento conjunto de los mecanismos de tolerancia a fallos desarrollados en este proyecto. El esquema presentado se ha obtenido a partir de Logisim. Partiendo de una activación original proveniente de la memoria on-chip, se pueden aplicar dos posibles técnicas de corrección, en función de si la activación es de tipo HO o L&HO. Recuérdese que esta clasificación

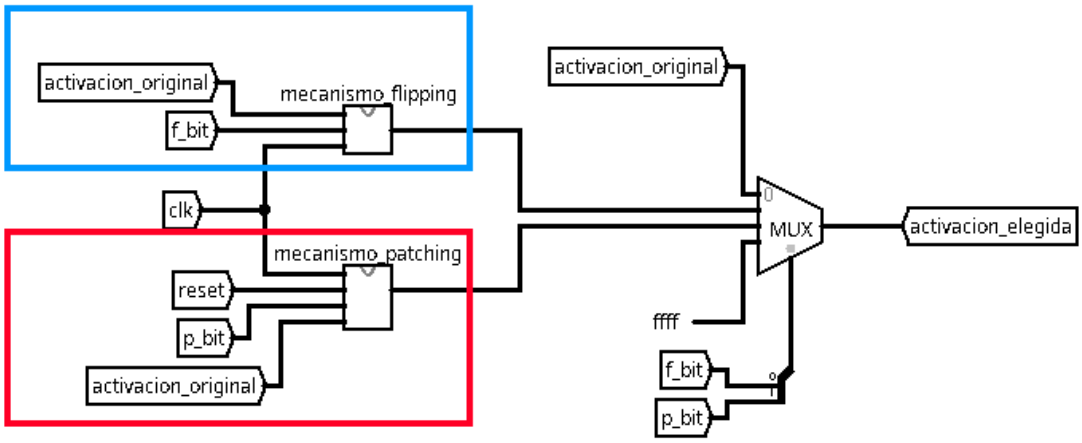


Figura 4.1: Esquema general de los mecanismos. Los colores azul y rojo resaltan los mecanismos de *flipping* y *patching*, respectivamente.

se determina mediante los bits de control f y p, que en un entorno real se obtendrían de forma individual para cada memoria mediante un test de arranque, en el cual se comprobaría el comportamiento de todas las activaciones a diferentes tensiones. En el contexto de este proyecto, dichos bits se han generado aleatoriamente para simular su disponibilidad previa.

Como se puede observar en la Figura 4.1, ambos mecanismos de corrección se aplican en paralelo, y la selección final de la activación que será utilizada en la red neuronal se realiza mediante un multiplexor 4:1 [1].[1] Esta lógica de selección garantiza el uso de la versión más fiable de la activación en función del tipo de fallo detectado, priorizando la robustez del sistema. A continuación se describe con detalle el proceso de implementación de ambos mecanismos.

La Figura 4.1 muestra en azul el encapsulado del mecanismo *flipping*. Sus entradas son la activación original y el bit f, además del reloj; mientras que su salida es la activación intercambiada o la activación original, dependiendo del valor de f. Su implementación se describe con detalle en la Sección 4.2. Por su parte, en rojo se puede observar el encapsulado del mecanismo *patching*. Sus entradas son la activación original y el bit p, además del reloj y la señal de reset; mientras que su salida será la activación original o la guardada en la cache, dependiendo del valor de p. Su implementación se describe con detalle en la Sección 4.3.3. En la figura 4.2 se recuerda el diseño del que se parte para la implementación de los dos mecanismos.

[1] Nótese que la selección 3 en el multiplexor, originalmente sin uso, tiene asociada una entrada fija a 0xFFFF.

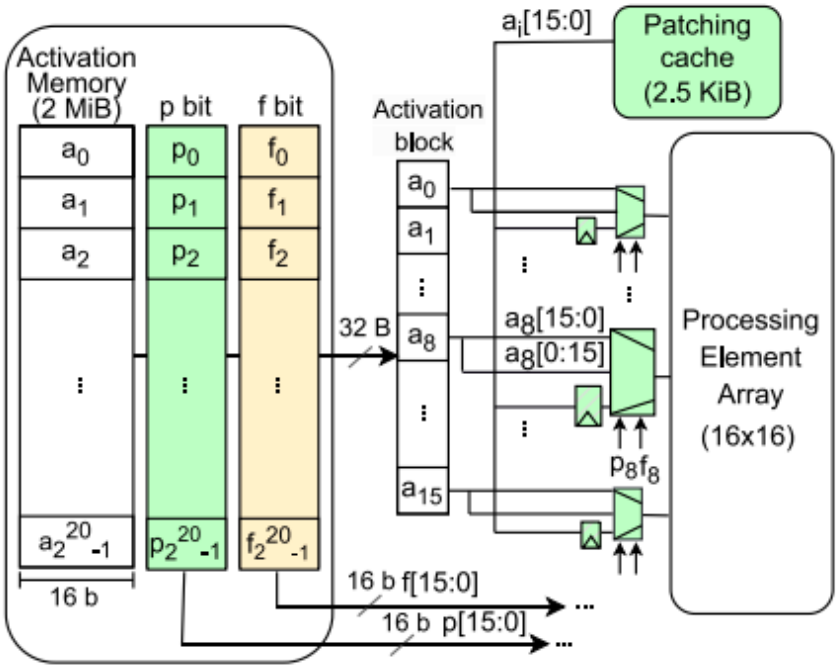


Figura 4.2: A partir del diseño mostrado en esta figura, se desarrolla su implementación hardware en las siguientes secciones. Figura tomada de [1].

4.2. Implementación del Método *Flipping*

Como se detalla en la Sección 2.5, la técnica de *flipping* permite intercambiar el bit en la posición i de una activación de 16 bits por el bit en la posición $16 - i - 1$. El mecanismo de *flipping* se implementa primero para una sola activación, utilizando un multiplexor 2:1 en SystemVerilog que utiliza como entradas seleccionables el bit i y el bit $16 - i - 1$ de la activación original. Dependiendo del valor del bit de control f, se elige una u otra entrada seleccionable. Empleando tantos multiplexores como bits para representar una activación (16 bits), se obtiene un vector que representa a una activación en su estado original, es decir, en caso de ser una activación de tipo LO ($f = 0$), o a una activación volteada o *flippeada* en caso de ser de tipo HO ($f = 1$). La Figura 4.3 muestra la implementación de la técnica en Logisim para los tres bits de mayor peso de una activación.

Tomando en consideración que los accesos a la memoria on-chip del acelerador se realizan con una granularidad de bloque de 16 activaciones, el siguiente paso en el diseño consiste en ampliar el uso de la técnica a un bloque de 16 activaciones. Gracias al diseño incremental, es posible instanciar 16 veces el módulo encargado de aplicar la técnica de *flipping* sobre una única activación, permitiendo así su ejecución en paralelo sobre las 16 activaciones del bloque.

El último paso de este diseño consiste en incorporar biestables para gestionar las

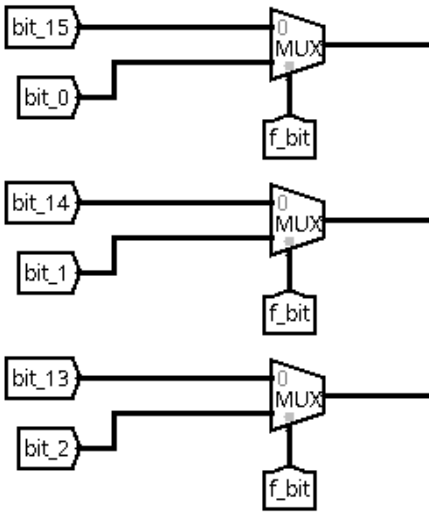


Figura 4.3: Detalle del diseño de la técnica *flipping* para los tres bits de mayor peso de una activación de 16 bits.

entradas y salidas del módulo *flipping*. Se desarrolla de forma escalable, comenzando por un flip-flop de un solo bit y extendiéndolo a módulos capaces de almacenar palabras de N bits y conjuntos de M palabras. Esta estructura permite almacenar tanto los bits de control f como los bloques completos de activaciones, facilitando su integración modular en el sistema.

Una vez disponibles todos los biestables, se puede implementar el programa de prueba del mecanismo *flipping* completo, y se programa la batería de tests detallada en la Sección 4.6.

4.3. Implementación de la Memoria Cache de Respaldo

En el artículo original se especifica que la memoria cache debe ser asociativa por conjuntos. En este diseño se ha adoptado una asociatividad de 5 vías, lo que permite evitar conflictos durante el acceso a datos. No obstante, al ser un sistema completamente parametrizable, este valor puede modificarse fácilmente si cambian los requisitos. El tamaño y la asociatividad de la cache se han dimensionado suponiendo una memoria on-chip de activaciones de 2 MiB. Si el tamaño de esta memoria creciera, también lo haría el número de fallos potenciales, por lo que sería necesario ajustar en consecuencia el tamaño de la cache.

El resto de la sección se organiza de la siguiente manera. En primer lugar, se detalla la primera iteración del diseño en Logisim. A continuación, se describe el diagrama de estados necesario para el control de la cache. Finalmente, se introduce el diseño del

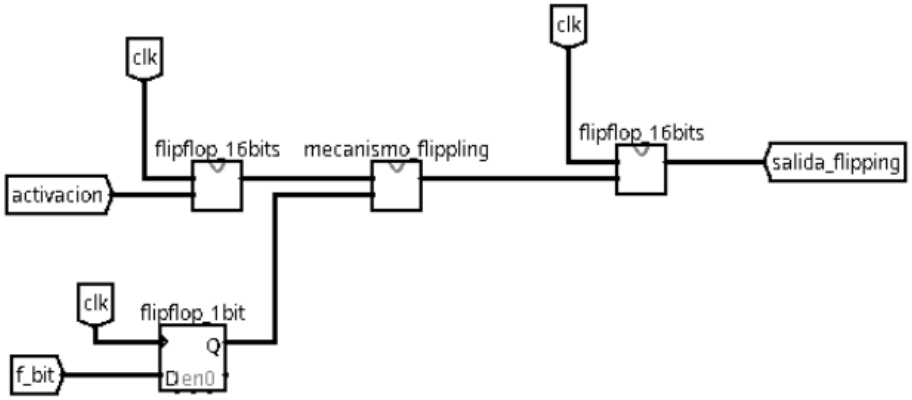


Figura 4.4: Esquema del mecanismo *flipping* con los biestables

circuito en SystemVerilog.

4.3.1. Diseño en Logisim

El diseño original de la cache se concibió para que se pudiera acceder en paralelo al *array* de etiquetas y datos. De esta manera, se permite que la verificación de un acierto y la obtención de las activaciones ocurra en el mismo ciclo de reloj, mejorando el tiempo medio de acceso a la cache. Además, se evita desperdiciar un ciclo sólo para leer las etiquetas y otro adicional para los datos, simplificando la máquina de estados de la unidad de control de la cache y, consecuentemente, aumentando el rendimiento del sistema.

Se consideran dos memorias RAM por cada vía, como se puede constatar en la Figura 4.5: una para almacenar las etiquetas y otra para las activaciones. Como entrada al bloque de la cache se dispone de una dirección, que se descompone en etiqueta y conjunto. El conjunto indica el índice de la fila dentro de la cache a la que pertenece la dirección, es decir, permite seleccionar el grupo de vías donde puede estar almacenada la activación correspondiente a la dirección. Por otro lado, la etiqueta corresponde a los bits más significativos de la dirección y se utiliza para verificar si el bloque almacenado en la cache coincide con la dirección solicitada, es decir, es su identificador. Se puede ver con detalle la división de los bits de la dirección de entrada en la Figura 4.6.

El conjunto se usa como entrada de lectura en las dos memorias RAM de las 5 vías. De esta manera, se obtendrán 5 etiquetas diferentes y otras tantas activaciones diferentes. Dependiendo de la etiqueta indicada en la dirección de entrada, se elegirá la vía correcta. Así, no es necesario esperar a que se haya leído el valor de la etiqueta para que se pueda leer la vía correcta de las activaciones, sino que se hace a la vez, ahorrando ciclos de lectura como se comentaba anteriormente. Tras la lectura de 5 activaciones correspondientes al conjunto destino, un multiplexor elige la activación demandada.

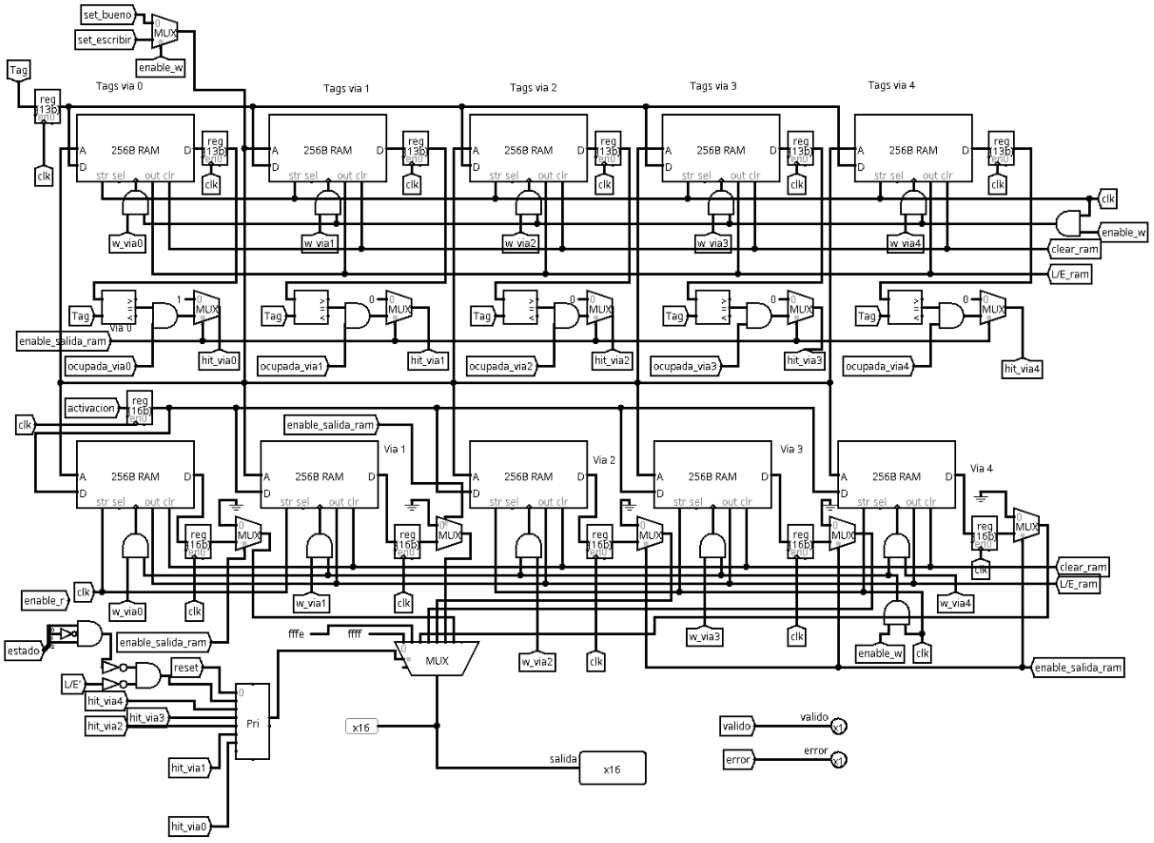


Figura 4.5: Esquema del acceso en paralelo a las etiquetas y a los datos de cada vía.

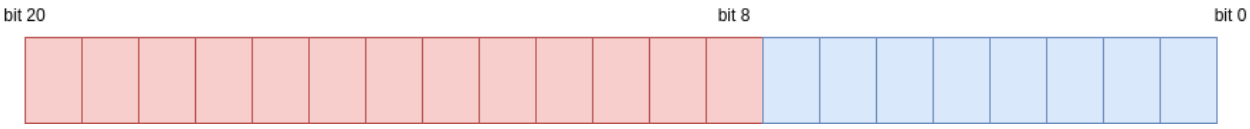


Figura 4.6: División de los bits de entrada de la dirección de 21 bits en etiqueta (*tag*), en rojo; y conjunto (*set*), en azul.

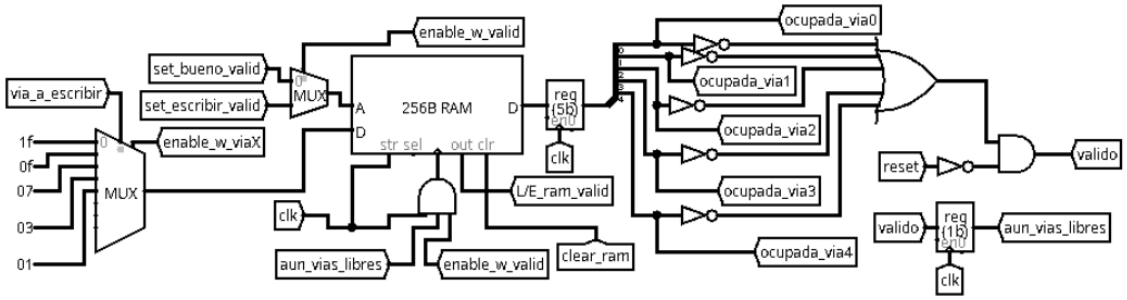


Figura 4.7: Lógica de lectura y escritura de la RAM de validez y salida *valido*.

El diseño propuesto incluye otra memoria RAM en la simulación que almacena unos bits de validez, véase la Figura 4.7. Se utilizan de modo que, si tienen valor '1' en la posición i-ésima, la vía i-ésima del conjunto objetivo estará ocupada y, por lo tanto, si se quiere escribir en el conjunto, será necesario hacerlo en la siguiente vía, es decir, la

vía $i + 1$. Nótese que la memoria con los bits de validez está físicamente separada del resto de memorias de la cache. Esto se debe a una mayor facilidad en el diseño a la hora de realizar un reset de la memoria cache completa simplemente estableciendo a '0' todos los bits de la cache de validez. Además, se puede comprobar la siguiente vía libre consultando una sola memoria RAM, en vez de consultar las 5 RAMs para comprobar si alguna está vacía, de manera que se ahorran cuatro accesos a memoria. La lógica para calcular la siguiente vía a escribir dados los bits de válido se muestra en la Figura 4.8.

Con el propósito de ahorrar energía, también se establece que, en caso de ciclos en de reposo en los cuales no se desea ni leer ni escribir en la cache, ésta no lea ni escriba nada. En aras de reducir aún más el consumo energético, se fuerza a que el último conjunto accedido no cambie y sea el mismo que el último ciclo en el que se realizó un acceso a la cache. Esta lógica se puede observar en la Figura 4.9.

4.3.2. Autómata de la Unidad de Control

Para manejar todas las señales de lectura, escritura, reset, etc, de la memoria cache en los ciclos correctos, es necesario un autómata y una unidad de control. Para esta última, se utiliza un diseño habitual, formado por un registro y dos memorias ROM correspondientes a la tabla de transición y tabla de salidas. La Figura 4.10 detalla el diseño de la unidad de control en Logisim, incluyendo todas las salidas de la unidad. Las Tablas 4.1 y 4.2 explican las entradas y salidas utilizadas tanto en el autómata como en el diseño de Logisim.

La unidad de control se describe mediante el autómata representado en la Figura 4.11. El autómata cuenta con 6 estados descritos a continuación:

- *Reset*: estado inicial al que se dirigen todos los estados si en algún momento la señal de entrada *reset* se activa. De esta manera, es posible realizar un reset de todas las memorias RAM a la vez cuando se inicie la ejecución de una red neuronal. Partiendo de este estado, sólo se podrá transitar a otro si la señal de reset se desactiva para comenzar una nueva ejecución.

- *Nothing*: es un estado para permitir que no se ejecute nada, ya que puede suceder que en algunos ciclos no sea necesario realizar ninguna acción sobre la cache: ni lectura ni escritura; por ejemplo, en aquellos ciclos donde el bit p sea nulo, es deseable mantener la cache sin leer ni escribir para no consumir energía. Para ello se dispone de una señal llamada *request*, que permite que si su valor es '0', se pueda dejar la cache en modo reposo.

- *Error*: estado para establecer ejecuciones en la cache no permitidas. Si se llega a

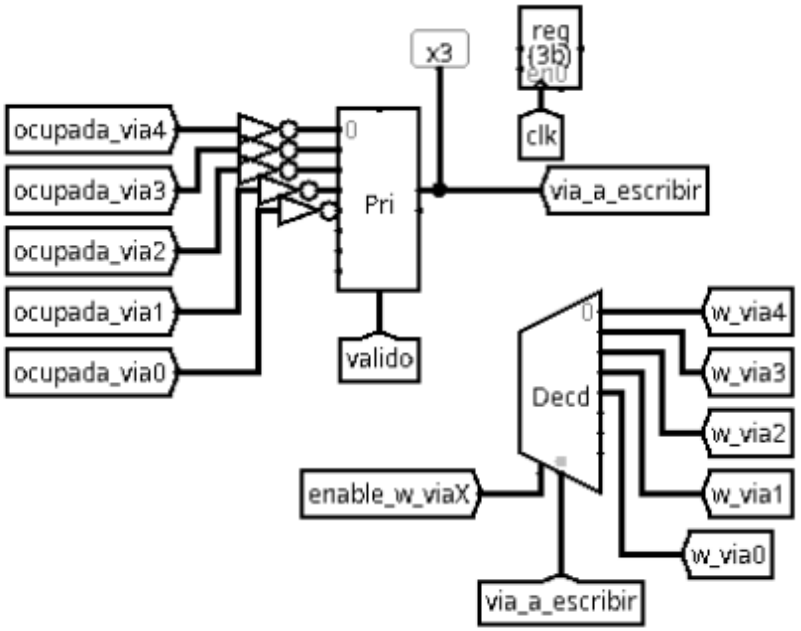


Figura 4.8: Lógica de cálculo de la siguiente vía a escribir y *enables* de escritura *valido*.

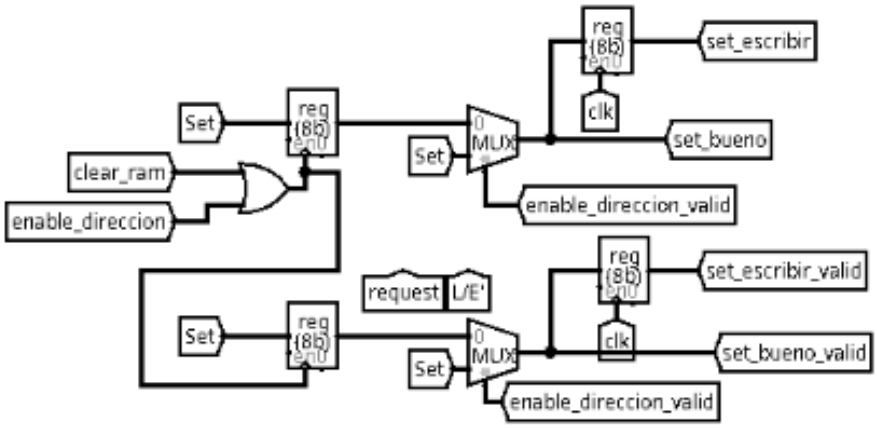


Figura 4.9: Lógica de mantenimiento del conjunto accedido el ciclo anterior a un ciclo de reposo.

Nombre	**Uso**
reset	Hace que el autómata vuelva al estado inicial y activa el reinicio de todos los contenidos de la cache
L/E'	Elige entre lectura o escritura según la acción que se quiera realizar
request	Indica si hay datos a escribir o a leer de forma efectiva. Si está a '0' la cache podrá entrar en modo reposo

Tabla 4.1: Entradas del autómata con sus usos.

este estado se activará una salida llamada *error* que indica al programador que ha habido un error. Al ser una cache diseñada especialmente para este propósito,

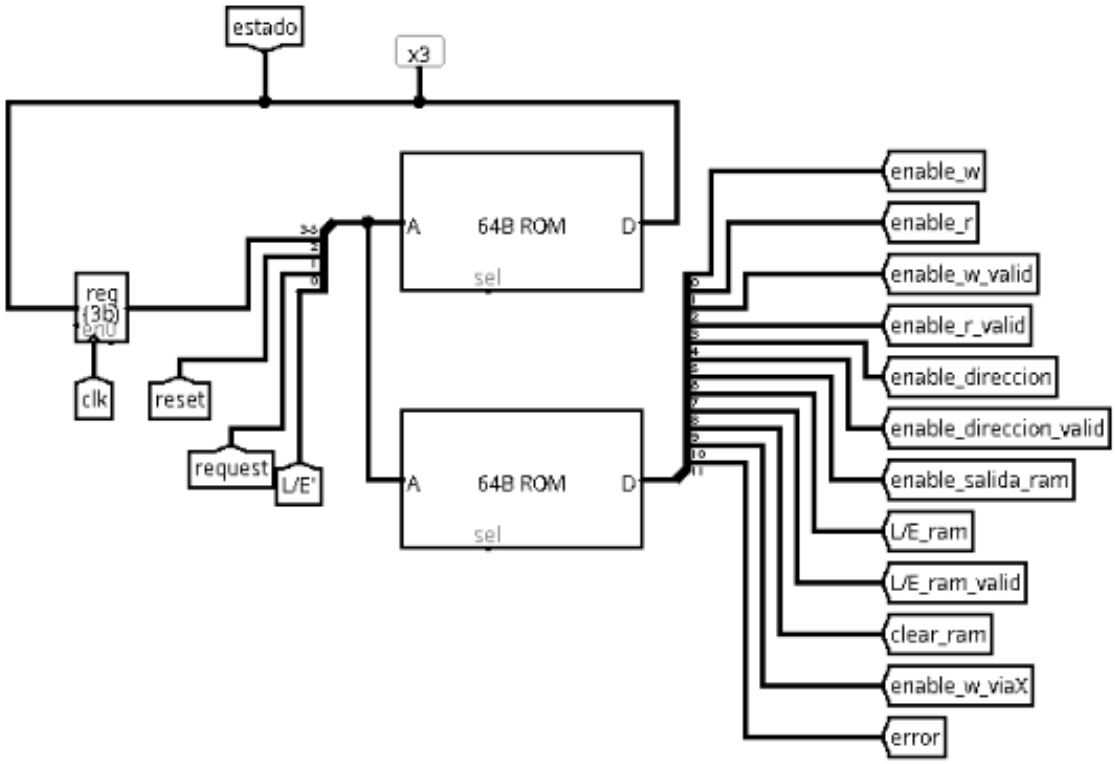


Figura 4.10: Diseño de la unidad de control.

existen algunas acciones que no están permitidas para simplificar el diseño de la cache:

- Leer antes de escribir: no se permite leer antes de haber escrito nada; si al estado de *Reset* llega una petición de lectura con la señal *request*=‘1’, transitará directamente al estado de error.
- Escribir después de leer: al tratarse de una cache que escribe todas las activaciones L&HO mientras se produce la siguiente capa de la red, al acabar de escribir todas estas activaciones no se permite escribir de nuevo, y sólo se utilizará la cache en modo lectura para consumir las activaciones L&HO de la capa calculada anteriormente. Así que si nos encontramos en el estado de lectura y llega una escritura con el bit de *request* = ‘1’, transitaremos al estado de error.
- Leer en mitad de una escritura: escribir tiene una latencia de dos ciclos. Por ello, si se cambia a modo lectura y *request*= ‘1’ después de haber empezado una escritura, pero sin haber hecho el segundo ciclo, transitamos al estado de error.

Nombre	Uso
clear_ram	Cuando está activa pone a '0' todos los contenidos de las RAMs que conforman la cache
error	Indica si el autómata se encuentra en el estado de Error
enable_w_viaX	Activa el decodificador donde se selecciona la vía en la que se va a escribir
L/E_ram_valid	Acción que se quiere realizar en la RAM de validez (lectura o escritura)
L/E_ram	Acción que se quiere realizar en las RAMs de etiquetas y datos
enable_salida_ram	Elige la salida válida de las RAM o tierra para ahorrar energía
enable_direccion_valid	Activa la selección de la dirección que entra en la RAM de validez o la dirección almacenada para ahorrar la energía en ciclos de reposo
enable_direccion	Activa la selección de la dirección que entra en las RAMs de etiquetas y datos o la dirección almacenada para ahorrar energía en ciclos de reposo
enable_r_valid	Control del reloj de lectura en la RAM de validez
enable_r	Control del reloj de lectura en las RAMs de etiquetas y datos
enable_w_valid	Control del reloj de escritura en la RAM de validez
enable_w	Control del reloj de escritura en las RAMs de etiquetas y datos

Tabla 4.2: Salidas del autómata con sus usos.

– *Estados de escritura*: existen dos estados de escritura, *Read_Valid* y *Write*. Al leer la RAM de validez en el primer ciclo de escritura (*Read_Valid*), podemos conocer qué vías del conjunto están ocupadas y decidir la siguiente a ocupar, que será siempre la que tenga el índice menor. En el segundo ciclo (*Write*), ya se conoce la vía a escribir, por lo que se puede iniciar la escritura, tanto de la etiqueta, activación y bit de validez de la nueva vía ocupada. Cuando estamos en alguno de los estados de escritura, la salida de la activación tiene un valor fijo de 0xFFFF para indicar que no se ha leído nada útil. Además, el diseño dispone de una salida llamada *valido* que muestra un '1' si aún quedan vías libres en el conjunto destino o muestra un '0' si ya no hay vías libres. Si se intentara escribir en un conjunto completo (*valido*='0'), automáticamente transitamos al estado de error, mostrando un '1' en la salida *error*, y solo sería posible recuperar el sistema con la entrada *reset* activa.

– *Reading*: en el estado de lectura se leen todas las RAMs de las etiquetas y de las activaciones a la vez, y se eligen las correctas con un multiplexor. Además, se lee

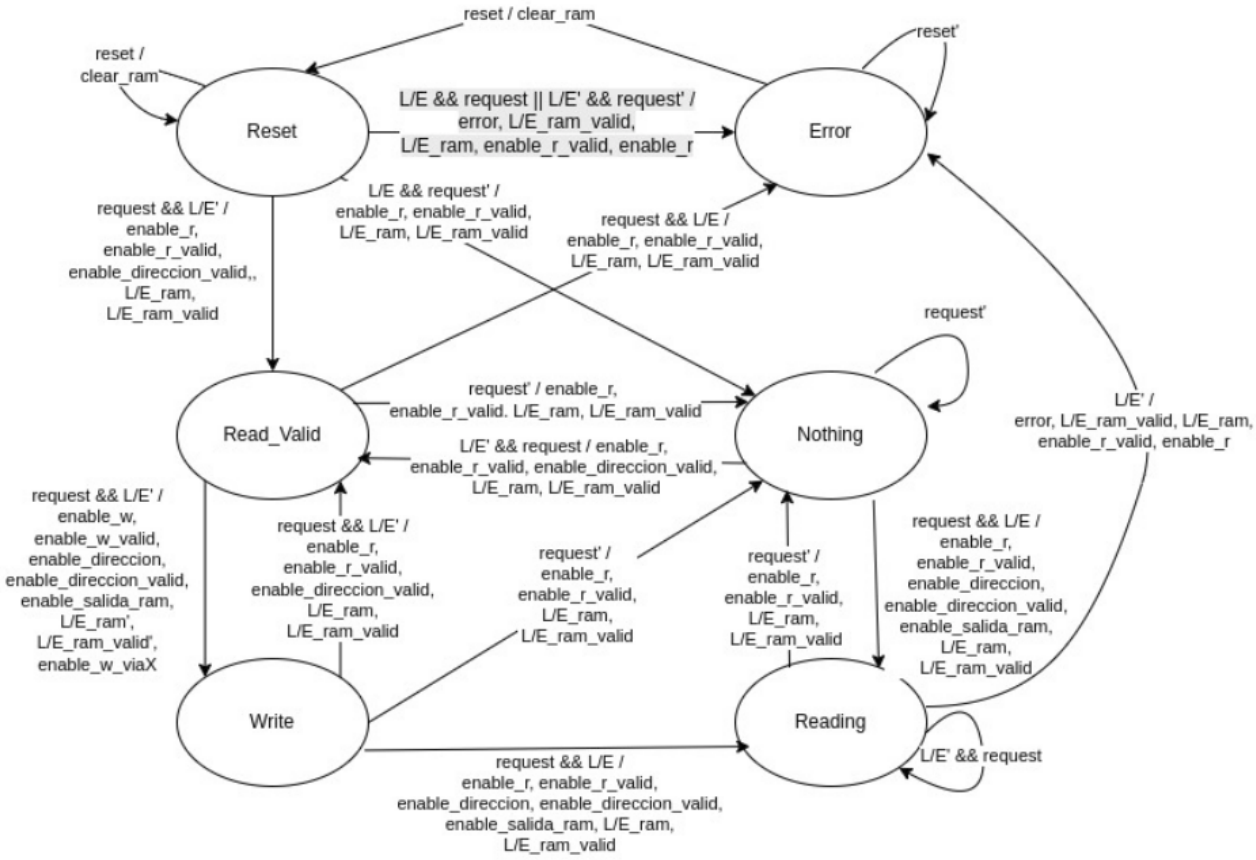


Figura 4.11: Autómata de la unidad de control para el diseño en Logisim.

la RAM de los bits de validez y se activa la salida de *valido*.

4.3.3. Diseño en SystemVerilog

Una vez validado el diseño funcional en simulación con Logisim, se adapta el sistema para su implementación en SystemVerilog, teniendo en cuenta las particularidades del hardware objetivo en la FPGA, ya que se quiere sintetizar el circuito allí y obtener los resultados de consumo y comportamiento temporal.

El principal problema es que la FPGA no dispone de memorias RAM genéricas como las utilizadas en simulación. En su lugar, las únicas memorias disponibles son BRAMs, las cuales se diferencian de las RAM simples en que incorporan un registro en la salida de datos. Este registro introduce un ciclo de retardo en la lectura, y no puede ser deshabilitado. Por lo tanto, es imprescindible considerar este retardo adicional en el diseño del autómata, adaptando su lógica de control para que funcione correctamente en el entorno de hardware real.

Esta limitación fuerza la incorporación de un estado extra en el autómata para la lectura. La Figura 4.12 muestra el diagrama de estados en el contexto SystemVerilog,

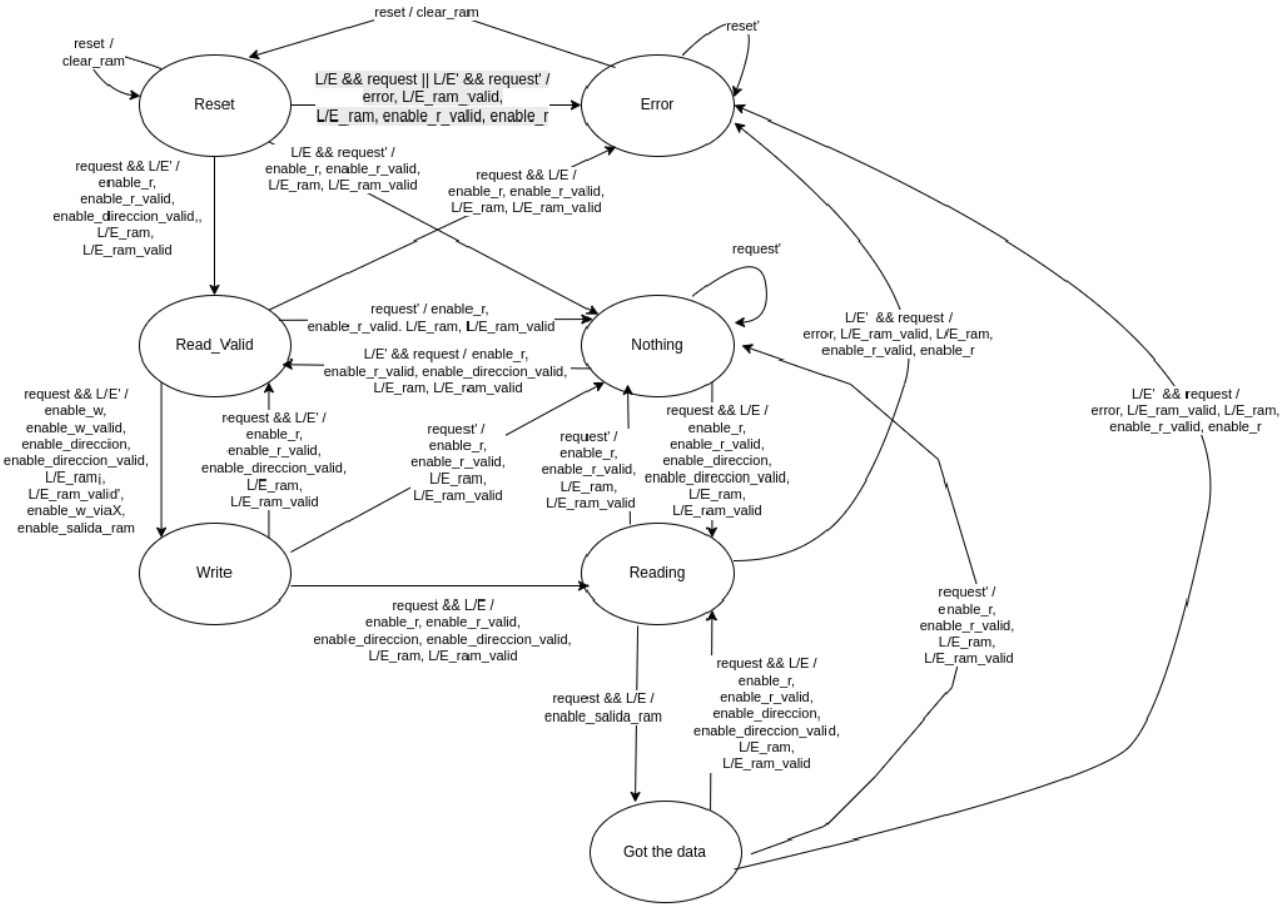


Figura 4.12: Autómata de la unidad de control para el diseño en SystemVerilog.

incorporando el nuevo estado de lectura *Got the data*. En un primer ciclo se leen las BRAMs (*Reading*), mientras que en un segundo ciclo (*Got the data*) se selecciona la activación válida mediante de acuerdo con el acierto en la etiqueta correspondiente. En cambio, la escritura no requiere ciclos adicionales, ya que la lectura anticipada del bit de validez puede solaparse con el segundo ciclo de escritura. En la figura 4.13 se detalla la evolución de los estados y las diferentes señales del autómata ciclo a ciclo simulando un reset, una escritura y una lectura. Se han omitido algunas señales menos importantes por claridad.

El nuevo comportamiento es validado replicando el diseño en Logisim con las mismas pruebas anteriores, asegurando una transición segura al entorno SystemVerilog.

Como se especifica en la sección de metodología (3.2, el siguiente paso en la implementación es traducir diferentes bloques de Logisim a módulos en diferentes ficheros en SystemVerilog:

– *address_decoder*: dada una dirección de entrada de 21 bits, la divide de manera que adjudica al conjunto los 8 bits de menos peso, y a la etiqueta los 13 bits restantes de mayor peso.

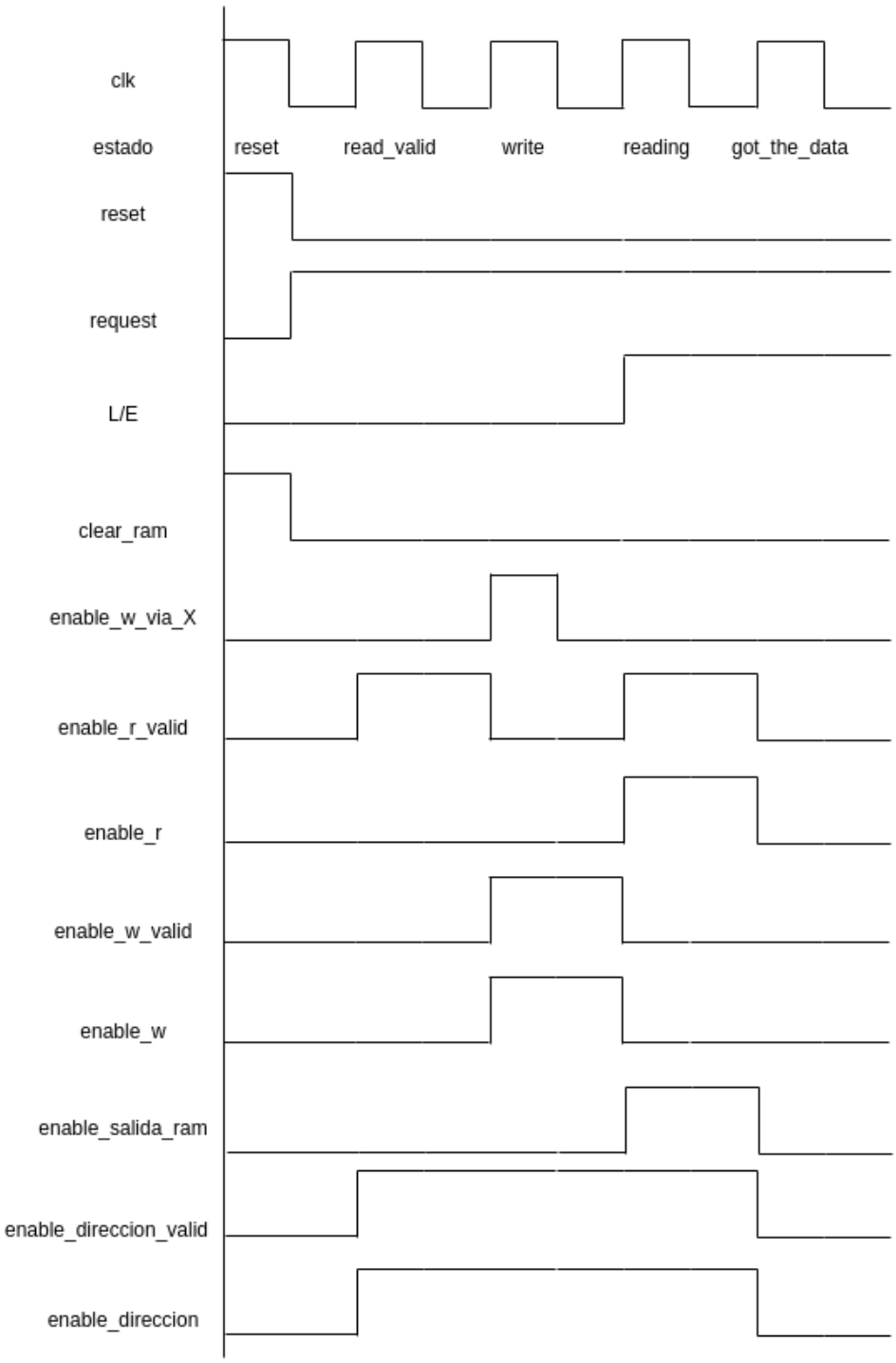


Figura 4.13: Cronograma ciclo a ciclo de una escritura seguida de una lectura

– *bram*: es el fichero de la memoria BRAM que se utiliza en la FPGA.

– *cache_way*: inicializa todas las BRAMs tanto de las etiquetas como de las activaciones, incluidas las señales que las controlan.

– *write_way_selector*: selecciona la primera vía libre a escribir en el conjunto destino y marca el nuevo bit de validez.

– *write_enable_generator*: una vez seleccionada la nueva vía a escribir, genera las máscaras de escritura para permitir la acción sólo en la BRAM de la vía elegida.

– *hit_logic*: compara las etiquetas obtenidas de las BRAMs y devuelve la vía que ha habido acierto. Además, se asegura de que esa vía esté realmente escrita comprobando los bits de validez.

– *activation_output_selector*: con las activaciones de las BRAMs, se elige la vía correcta averiguada gracias al módulo anterior. Funciona con un multiplexor 5:1 implementado en otro módulo.

Todos estos bloques se integran en el módulo principal de la cache, controlado por las señales del autómata. Una vez ensamblado, se valida su funcionamiento mediante pruebas unitarias y de transición de estados.

4.4. Implementación del Método *Patching*

Una finalizada la implementación de la cache, se aborda el mecanismo *patching*, cuya lógica es sencilla: seleccionar entre la activación original o la *cacheada* en función del valor del bit p.

Al igual que con el mecanismo de *flipping*, se desarrolla en primer lugar una versión para una sola activación, seguida de su extensión al bloque completo. Una vez validados ambos módulos, se diseña un conjunto de pruebas para verificar su funcionamiento.

4.5. Integración Conjunta de *Flipping* y *Patching*

Con todos los mecanismos funcionando en sus respectivos módulos, se realiza un programa de pruebas para comprobar que juntos funcionan correctamente.

En este programa de pruebas es necesario instanciar los distintos mecanismos ya diseñados, además de un nuevo módulo que funciona como un multiplexor para seleccionar la salida adecuada, dependiendo de los bits f y p.

También se desarrolla un pequeño autómata para controlar las señales necesarias durante los tests (como *request* o *read_write*) en función de los diferentes casos evaluados. Aunque el programa de pruebas está diseñado para generar automáticamente las señales necesarias, también se implementa una interfaz alternativa que permite manejar manualmente las señales de la cache, por si se desea realizar pruebas más controladas o específicas desde el propio entorno de test.

4.6. Pruebas de Validación

La pruebas de validación diseñadas para este proyecto han sido rigurosas, abarcando tanto situaciones típicas como casos límite. Se ha comprobado de forma exhaustiva el comportamiento de cada componente por separado, así como su integración conjunta, asegurando la correcta funcionalidad del sistema en todos los posibles escenarios de uso.

El detalle completo de estas pruebas se recoge en el Anexo D.

4.7. Síntesis en FPGA

Una vez validados los distintos mecanismos del sistema, se procede a la síntesis completa del sistema en la FPGA. El objetivo principal de esta etapa es comprobar la viabilidad del diseño en hardware real, así como cuantificar el uso de recursos y el consumo energético. Para todas las pruebas se utiliza una frecuencia de reloj objetivo de 100 MHz.

En el Anexo E se detallan los primeros intentos de sintetizar el diseño. Para cumplir con las limitaciones de entrada y salida de la FPGA, y evitar sobrecargas innecesarias y descartes de módulos por parte del sintetizador, se re-diseña una versión optimizada del sistema que procesa una única activación por ciclo optando por utilizar directamente las versiones unitarias de los mecanismos de *flipping* y *patching*, en lugar de los módulos diseñados para procesar bloques de M activaciones para evitar duplicidad de lógica y simplificar el diseño final. Como paso final y como parte de la metodología detallada en el Anexo A, se realizan nuevas pruebas de verificación sobre el módulo re-diseñado, asegurando su correcto funcionamiento antes de intentar la síntesis y ejecución en hardware. El siguiente capítulo muestra los resultados obtenidos a partir de esta última iteración en el diseño.

Capítulo 5

Análisis de Resultados

Tras realizar la síntesis completa del diseño en la FPGA, el presente capítulo evalúa tanto el uso de recursos como el consumo energético, así como la temporización y distribución física de los módulos implementados. Los resultados obtenidos permiten confirmar la viabilidad del diseño final y validar las decisiones tomadas durante su desarrollo.

5.1. Uso de Recursos

Como se observa en la Figura 5.1, el módulo principal (tm) integra tanto el mecanismo *flipping* como *patching*. El componente más costoso en lógica es claramente la memoria cache (patch_cache), que emplea 749 LUTs, 1283 registros y 469 *slices*. Este consumo es especialmente elevado debido a las múltiples vías y a la lógica asociada al control de lectura y escritura. No obstante, gracias a su diseño modular y a una implementación optimizada, su impacto global sobre el uso de recursos en la FPGA se reduce significativamente.

Por ejemplo, puede verse que en la FPGA se han empleado únicamente 5 BRAMs en vez de 10 (5 vías para datos + 5 vías para etiquetas), lo cual se consigue gracias a que se implementa una sola BRAM por vía, compartida entre datos y etiquetas. Esta optimización permite que la lógica ocupe solo un 1,4 % de las LUTs disponibles (749 de 53200) y un 1,2 % de los registros (1283 de 106400), haciendo la implementación mucho más eficiente y escalable.

Por su parte, el mecanismo de *flipping* tiene un peso muy reducido en la lógica total utilizada, lo que es coherente con su funcionalidad acotada a la manipulación de bits individuales. En comparación con la cache, solo utiliza 33 registros, 16 LUTs y 8 *slices*.

Para mejorar la claridad visual de la Figura 5.1, se omiten las columnas correspondientes a los recursos Bonded IOPADs y BUFGCTRL, ya que todos sus valores son cero para los módulos sintetizados. Esto es esperable, ya que el diseño aún

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)
tm (design_1_alicia_tfg_axi_0_0_test_mecha	765	1316	170	85	477	765	0	5
flip_inst (design_1_alicia_tfg_axi_0_0_me	16	33	0	0	8	16	0	0
u0 (design_1_alicia_tfg_axi_0_0_flipflo	8	16	0	0	8	8	0	0
u1 (design_1_alicia_tfg_axi_0_0_flipflo	0	1	0	0	1	0	0	0
u3 (design_1_alicia_tfg_axi_0_0_flipflo	8	16	0	0	7	8	0	0
patch_inst (design_1_alicia_tfg_axi_0_0_t	749	1283	170	85	469	749	0	5
patch_cache (design_1_alicia_tfg_axi_(	749	1283	170	85	469	749	0	5
hit_logic_inst (design_1_alicia_tfg_a	58	0	0	0	34	58	0	0

Figura 5.1: Recursos hardware utilizados para la implementación en la FPGA.

no ha sido conectado físicamente a entradas/salidas externas de la FPGA —aunque esto sería posible en caso de querer realizar pruebas funcionales adicionales, simplemente conectando un programa en C++ a los puertos de entrada y salida del mecanismo conjunto—, ni se utilizan buffers globales de reloj (BUFGCTRL).

5.2. Consumo Energético

Según el informe de consumo detallado en la Figura 5.2, el diseño completo presenta un consumo dinámico estimado de 0,008 W, con todos los módulos contribuyendo de forma mínima y uniforme (menos de 0,001 W cada uno). Este dato resalta la eficiencia energética del sistema, especialmente considerando que gran parte de la lógica está implementada en BRAMs, lo que reduce notablemente el gasto respecto a un diseño completamente en lógica programable.

Además, el hecho de que el consumo esté equilibrado y no existan picos en módulos concretos indica una gestión eficiente de la actividad interna, sin componentes infrautilizados o mal dimensionados.

Para mejorar la claridad visual de la Figura 5.2, se han omitido las columnas correspondientes a los recursos relacionados con interfaces externas y bloques específicos de la FPGA, concretamente Memory Interface, I/O Interface, PLLs y AXI. En todos los casos, sus valores eran inferiores a 0.001 W, indicando un uso prácticamente nulo. Esto era esperable, ya que el diseño no se encuentra conectado a buses AXI ni hace uso de periféricos, memorias externas o lógica programable avanzada como PLLs, al tratarse de una simulación centrada únicamente en el consumo de los mecanismos internos del sistema.

5.3. Temporización

La Figura 5.3 muestra el resumen de la temporización generada durante la implementación. El parámetro *Worst Negative Slack* mide cuánto margen hay entre el

	Name	Clocks (W)	Signals (W)	Data (W)	Clock Enable (W)	Set/Reset (W)	Logic (W)	BRAM (W)	PS7 (W)	Processor (W)
0.008 W (1% of total)	tm (design_1_alicia_tfg_	0.004	0.004	0.004	<0.001	<0.001	<0.001	0.001	<0.001	<0.001
0.008 W (1% of total)	patch_inst (design_1_ali	0.003	0.004	0.004	<0.001	<0.001	<0.001	0.001	<0.001	<0.001
0.008 W (1% of total)	patch_cache (design_1_	0.003	0.004	0.004	<0.001	<0.001	<0.001	0.001	<0.001	<0.001
0.007 W (1% of total)	Leaf Cells (1647)									
<0.001 W (<1% of total	ways[1].way_instance (d	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total	ways[2].way_instance (d	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total	ways[3].way_instance (d	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total	ways[4].way_instance (d	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total	ways[0].way_instance (d	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total	hit_logic_inst (design_1_	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total)	flip_inst (design_1_alicia	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total)	u3 (design_1_alicia_tfg_	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total)	u0 (design_1_alicia_tfg_	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total)	u1 (design_1_alicia_tfg_	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001

Figura 5.2: Consumo de la implementación en la FPGA.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,398 ns	Worst Hold Slack (WHS):	0,090 ns	Worst Pulse Width Slack (WPWS):	4,020 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	8275	Total Number of Endpoints:	8275	Total Number of Endpoints:	2938

All user specified timing constraints are met.

Figura 5.3: Resumen de la temporización de la implementación en la FPGA.

retardo del camino más lento del circuito (camino crítico) y el tiempo de ciclo (10 ns por ciclo). Un valor positivo es bueno porque significa que el camino se completa a tiempo y que el diseño cumple los requisitos de temporización para el tiempo de *setup* (es decir, los datos llegan a tiempo antes del flanco de reloj).

El parámetro *Worst Hold Slack* indica el margen mínimo entre el momento en que una señal llega a una celda secuencial (como un flip-flop) y el flanco de reloj que la captura. Este margen garantiza que la señal no llegue demasiado pronto, lo cual podría provocar que el dato anterior aún no se haya capturado correctamente. Un valor positivo implica que todos los caminos cumplen con el retardo mínimo necesario para que la señal se mantenga estable antes de ser registrada. Además, asegura que la sincronización del diseño es segura en ese aspecto.

Por su parte, el parámetro *Worst Pulse Width Slack* asegura que los pulsos del reloj sean suficientemente anchos como para activar los registros. Este valor es positivo y amplio, lo cual indica que existe un margen de seguridad amplio.

Finalmente, el mensaje *All user specified timing constraints are met* confirma que el diseño cumple todos los requisitos de temporización, asegurando que el diseño es estable, fiable y funcional a 100 MHz.

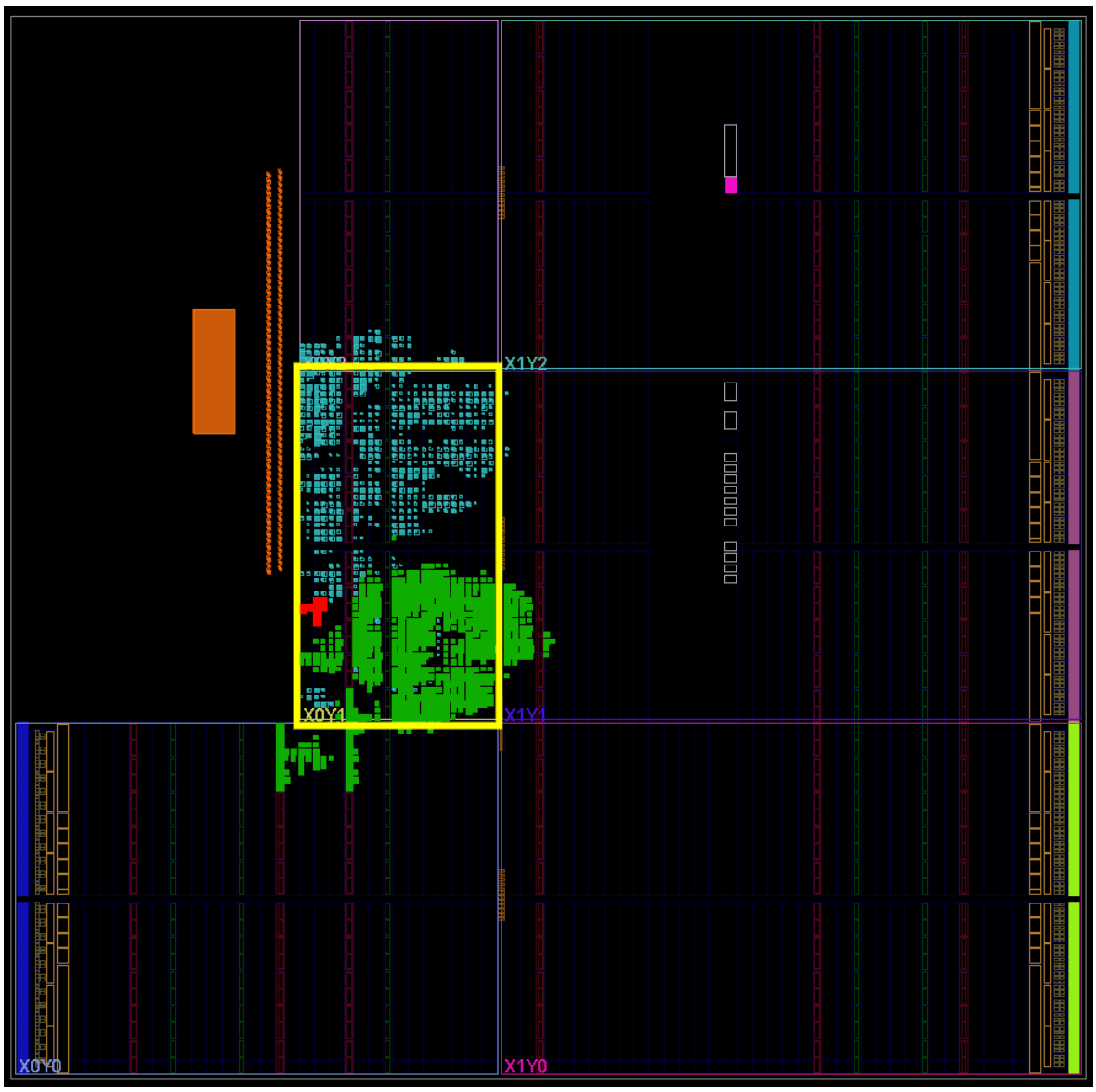


Figura 5.4: Distribución física de la implementación en la FPGA.

5.4. Distribución Física en la FPGA

La Figura 5.4 muestra la distribución física del diseño tras la implementación en la FPGA. Se puede apreciar que la mayoría del área ocupada corresponde a la memoria cache (en color verde), concentrada en una región bien definida. Mucho más modesta, como cabría esperar, la lógica de control (en rojo) queda situada en las inmediaciones de la cache, lo que permite minimizar las distancias de interconexión y los retardos asociados.

Se pueden observar también las regiones identificadas con coordenadas como X0Y0, X0Y1, etc., que corresponden a divisiones físicas de la FPGA. Estas etiquetas indican bloques de área lógica dentro del chip, organizados en una cuadrícula. Cada región contiene recursos como LUTs, registros, BRAMs o interfaces de entrada/salida. En

este caso, se aprecia que la mayor parte de la lógica se ha ubicado en la región X0Y1. En la figura se resalta esa región para evidenciar la zona principal donde se ubica la lógica sintetizada. Esto es debido a que dicha zona concentra suficientes recursos lógicos disponibles, sin restricciones especiales, lo cual permite al sintetizador y al colocador ubicar ahí la mayor parte del diseño de forma eficiente. Además, al estar próximas unas instancias a otras, se reducen los retardos de enrutamiento, mejorando el rendimiento general.

Finalmente, la red de interconexión con el exterior de la FPGA se representa en color celeste. Su distribución en los bordes de la zona activa refleja el mapeo hacia los pines físicos del dispositivo, y su densidad confirma la necesidad de gestionar múltiples señales de entrada y salida, incluso en esta versión con bloques de solamente una activación.

Capítulo 6

Conclusiones

El desarrollo de este proyecto ha demostrado la viabilidad de aplicar técnicas de tolerancia a fallos permanentes en memorias on-chip de aceleradores CNN como medio efectivo para reducir el consumo energético sin comprometer la funcionalidad del sistema. Las técnicas *flipping* y *patching* se han integrado correctamente en una arquitectura funcional y su implementación hardware ha sido validada mediante simulación y síntesis sobre una FPGA real.

Entre las principales aportaciones del presente trabajo se encuentra el diseño modular y escalable de los mecanismos en SystemVerilog, la validación funcional del comportamiento lógico del sistema, la evaluación del diseño mediante síntesis en FPGA, así como la creación y adopción de una metodología iterativa para su implementación y la depuración eficaz de los distintos componentes. También se ha prestado especial atención a la organización del proyecto, la trazabilidad del desarrollo y la documentación del flujo de pruebas, con el objetivo de asegurar la reproducibilidad del sistema.

Desde el punto de vista de la evaluación, la implementación sintetizada cumple los requisitos de temporización para una frecuencia objetivo de 100 MHz, sin violaciones de *setup* ni *hold*. El análisis temporal muestra que el camino crítico del diseño dispone de un margen positivo de 0,398 ns respecto al tiempo de ciclo (10 ns), lo que garantiza una operación fiable. En cuanto al consumo, se ha verificado que el sistema completo introduce una sobrecarga energética mínima, con una potencia estimada de tan solo 0,008 W (8 mW), lo que lo hace especialmente adecuado para su integración en plataformas embebidas de bajo consumo. En comparación con los resultados presentados en el trabajo original que propone las técnicas Flip-and-Patch, donde se estima que el consumo estático pasaría de 269,8 mW a 288,5 mW y el retardo de acceso de 2,69 ns a 2,75 ns [1], los valores obtenidos en esta implementación muestran una eficiencia mayor, manteniéndose funcional a menor potencia y con márgenes de temporización holgados.

Como líneas de trabajo futuro, se propone llevar a cabo la integración de estos mecanismos en un sistema de inferencia completo que permita validar su efecto en la

precisión del modelo. Además, se podría estudiar la viabilidad de una implementación en un circuito integrado de aplicaciones específicas (ASIC, por sus siglas en inglés), un tipo de chip diseñado para realizar una tarea concreta de forma muy eficiente, en contraste con los dispositivos reconfigurables como las FPGAs. Este enfoque permitiría evaluar el comportamiento real del sistema en un entorno industrial más cercano al producto final, optimizando tanto el consumo energético como el rendimiento en aplicaciones reales. También sería interesante automatizar el proceso de caracterización de fallos y marcado de bits de control, lo que facilitaría su integración en flujos de diseño más amplios.

En conjunto, este Trabajo de Fin de Grado constituye una primera aproximación práctica al diseño de hardware tolerante a fallos para aceleradores de CNNs, apoyado en una metodología iterativa por niveles y en el uso coordinado de herramientas como Logisim, SystemVerilog, Verilator y Vivado. Este enfoque sienta las bases para desarrollos más complejos orientados a sistemas embebidos robustos y energéticamente eficientes.

Finalmente, este trabajo se ha desarrollado en el marco del proyecto RETORNNA, como parte de sus tareas de transferencia tecnológica hacia soluciones aplicadas en el ámbito del diseño de hardware resiliente, lo que refuerza su relevancia dentro de una línea estratégica de investigación y desarrollo.

Capítulo 7

Bibliografía

[1] Yamilka Toca-Díaz, Reynier Hernández Palacios, Rubén Gran Tejero, and Alejandro Valero. Flip-and-patch: A fault-tolerant technique for on-chip memories of cnn accelerators at low supply voltage. *Microprocessors and Microsystems*, 106:105023, 2024.

[2] Christopher Wilkerson, Hongzhong Gao, Abdulrahman R. Alameldeen, Zohaib Chishti, Muhammad Khellah, and Shih-Lien Lu. Trading off cache capacity for reliability to enable low voltage operation. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, pages 203–214. IEEE/ACM, 2008.

[3] Jongwoo Kim, Nikos Hardavellas, Kevin Mai, Babak Falsafi, and James Hoe. Multi-bit error tolerant caches using two-dimensional error coding. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 197–209. IEEE/ACM, 2007.

[4] Jiawei Tan, Qian Wang, Kai Yan, Xiaowei Wei, and Xin Fu. Saca-fi: A microarchitecture-level fault injection framework for reliability analysis of systolic array based cnn accelerators. *Future Generation Computer Systems*, 147:251–264, 2023.

[5] Bootcamp AI. Redes neuronales convolucionales. `https://bootcampai.medium.com/redes-neuronales-convolucionales-5e0ce960caf8`, 2019. Bootcamp AI.

[6] Google Cloud. An in-depth look at Google's first Tensor Processing Unit (TPU). `https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit`, 2017. Google Cloud Blog.

[7] Coral (distribuido por Mouser Electronics). G650-06076-01. https://www.mouser.es/ProductDetail/Coral/G650-06076-01?qs=W%2FMpXkg%252BdQ6LZJp2eyeh4w%3D%3D&mgh=1&vip=1&utm_id=19098080631&utm_source=google&utm_medium=cpc&utm_marketing_tactic=emeacorp&gad_source=1&gad_campaignid=19105070087&gclid=EAIaIQobChMImYXV9KSAjgMV6WxBAh2bozBcEAYYASABEgJRKfD_BwE, 2025. Mouser Electronics.

[8] Trevor Pering, Trent Burd, and Robert Brodersen. Dynamic voltage scaling and the design of a low-power microprocessor system. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 96–101. ACM/IEEE, 1998.

[9] Rajeev Balasubramonian, Andrew B. Kahng, N. Murali Manohar, Ali Shafiee, and Vignesh Srinivas. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):21:1–21:25, 2017.

[10] Microsoft. Visual Studio Code. https://code.visualstudio.com, 2025. Accessed: 2025.

[11] Free Software Foundation. GCC, the GNU Compiler Collection. https://gcc.gnu.org, 2025. Boston, MA; Accessed: 2025.

[12] Veripool. Verilator. https://www.veripool.org/verilator, 2025. Accessed: 2025.

[13] Circuit. Circuit Simulator. https://sourceforge.net/projects/circuit/, 2025. SourceForge; Accessed: 2025.

[14] Canonical Ltd. Ubuntu Linux Operating System. https://ubuntu.com, 2025. Accessed: 2025.

[15] Git. Git SCM. Source Code Management. https://git-scm.com, 2025. Accessed: 2025.

[16] AMD. Vivado. https://www.amd.com, 2025. Santa Clara, CA; Accessed: 2025.

[17] Digilent Inc. *ZedBoard Zynq™ Evaluation and Development Hardware User's Guide*. Digilent Inc., January 2012. Ver 1.1.

Lista de Figuras

Lista de Tablas

Anexos

Anexos A

Metodología Utilizada

Durante el desarrollo del proyecto se ha seguido una metodología propia, diseñada específicamente para facilitar la implementación, prueba y depuración de los componentes hardware. Esta metodología se basa en un enfoque iterativo e incremental, en el que se contruyen bloques funcionales simples como primer paso, y se avanza progresivamente hacia estructuras más complejas.

- Diseño de bloques simples: se comienza implementando módulos básicos con funcionalidades aisladas y bien definidas.

- Pruebas unitarias: tras implementar cada bloque, se desarrollan testbenches específicos para comprobar su correcto funcionamiento mediante simulación.

- Validación funcional: si el módulo supera las pruebas unitarias, se considera apto para ser utilizado como componente en bloques superiores.

- Construcción jerárquica: los bloques validados se integran en unidades funcionales de mayor complejidad, que a su vez son sometidas al mismo proceso de prueba.

- Reutilización y verificación continua: si en etapas avanzadas se detectan errores, se retrocede al nivel correspondiente para corregir y volver a validar. Esto garantiza la robustez del sistema y evita propagar fallos a niveles superiores.

Esta estrategia modular permite detectar errores de forma temprana, facilita la identificación de errores y reduce el riesgo de introducir fallos en frases avanzadas del proyecto. Además, ha demostrado ser eficaz para gestionar la complejidad del diseño hardware, especialmente cuando se trabaja con lenguajes de descripción y herramientas de simulación.

A.1. Aplicación Práctica de la Metodología

Un ejemplo representativo de esta metodología es el desarrollo del mecanismo *flipping*. Inicialmente, se diseño para operar sobre una única activación, y tras comprobar su correcto funcionamiento, se amplía fácilmente para trabajar con bloques de 16 activaciones, replicando el módulo base sin necesidad de modificar su lógica central.

Este mismo enfoque se aplica al diseño del autómata de control, la memoria cache y el mecanismo *patching*. Cada uno de estos componentes se prueba individualmente antes de integrarse en el sistema, garantizando así la estabilidad del diseño en cada etapa.

A.2. Uso de herramientas visuales en fases tempranas

Para facilitar la depuración de bloques complejos como la memoria cache de respaldo, se emplea la herramienta Logisim en las fases iniciales. Aunque se uso estaba previsto únicamente para representar el esquema general, se opta por implementar la cache completa en Logisim. Esto permite realizar pruebas visuales intuitivas, detectar errores estructurales en el diseño y validar la lógica del autómata de forma temprana.

La implementación en Logisim no solo mejora la comprensión del sistema, sino que también sirve como base fiable para su posterior traducción a SystemVerilog, consolidando así una metodología visual y funcionalmente robusta.

A.3. Del Diseño a la Implementación Física

Es importante aclarar que la implementación en SystemVerilog no implica su ejecución directa en hardware. Este lenguaje se utiliza como medio para describir y simular el sistema, permitiendo una validación exhaustiva antes de su síntesis.

Para obtener resultados reales —como consumo energético o latencia— es necesario sintetizar el diseño en una **FPGA ZedBoard Zynq-7000**. Por ello, la programación en SystemVerilog contituye una etapa esencial previa a la implementación física, asegurando que el diseño lógico estuviera completamente verificado antes de su despliegue en hardware.

Anexos B

Gestión del Proyecto

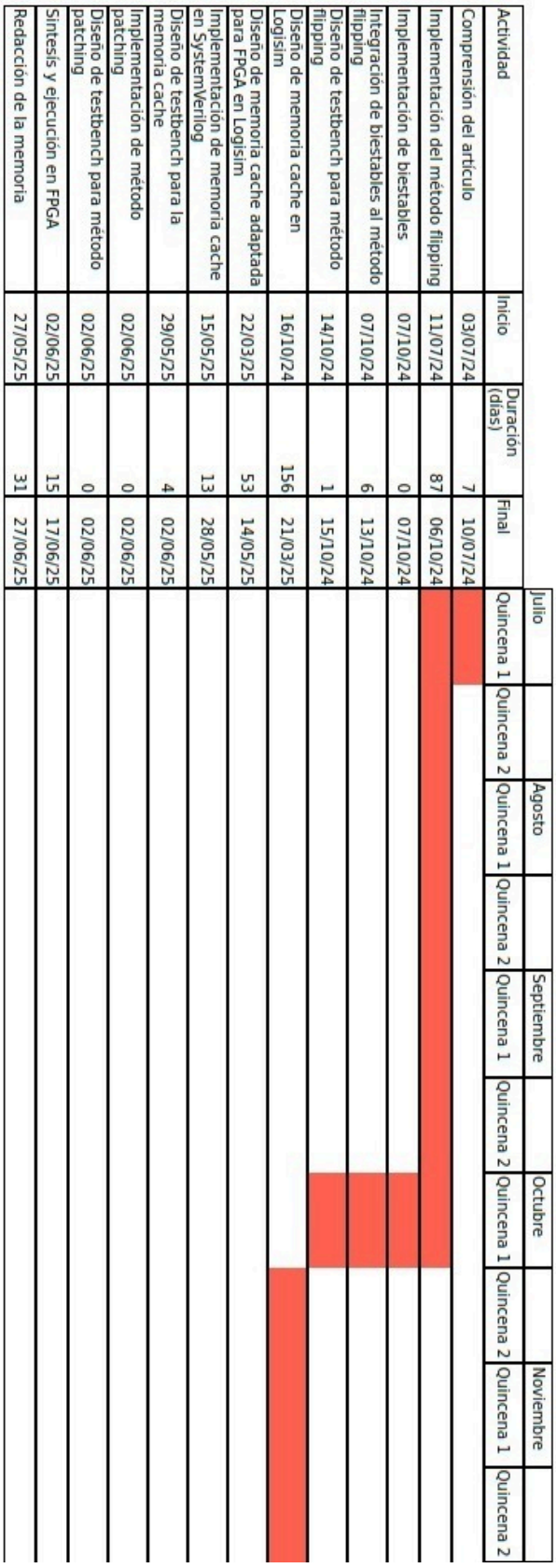
El desarrollo de los mecanismos se estructuró en varias fases clave:

- **Comprensión del artículo:** el primer paso consistió en la lectura y entendimiento del artículo del que se partía, ya que era fundamental comprender todos los conceptos usados y el funcionamiento de los mecanismos a implementar.
- **Implementación del mecanismo *flipping*:** como el mecanismo *flipping* no era arquitecturalmente complicado, se decidió implementarlo directamente en SystemVerilog. Primero se intentó implementarlo directamente con bloques de 16 activaciones, pero se vio que daba lugar a muchos errores y, por ello, se optó por un diseño partiendo de bloques muy sencillos que juntos formaban el mecanismo más complicado.
- **Diseño del autómata de la memoria cache:** una vez el primer mecanismo funcionó se pudo empezar a implementar el mecanismo *patching*, para el cual se necesitaba una memoria cache con características muy específicas. Para comenzar a diseñarla se creó una máquina de estados que definía su funcionamiento.
- **Diseño de la memoria cache en Logisim:** para asegurar el correcto funcionamiento del autómata creado y facilitar la implementación de la cache usando bloques sencillos se decidió crear un prototipo funcional en Logisim. Se eligió este programa por la facilidad de uso y el hecho de poder tener un esquema de todo lo necesario a implementar más adelante.
- **Implementación de la cache en SystemVerilog:** una vez finalizado el prototipo, sólo era necesario traducir a SystemVerilog desde cada uno de los bloques colocados en Logisim.
- **Implementación del mecanismo *patching*:** con la cache funcionando, se pudo comenzar con la implementación del mecanismo *patching* empezando por

versiones más simples y luego implementando la versión completa partiendo de estas. Además, se implementaron programas de pruebas.

- **Integración de los dos mecanismos:** una vez estuvieron los dos mecanismos funcionando, se pudo proceder a su integración para el funcionamiento simultáneo.
- **Síntesis del código en FPGA:** para obtener datos sobre el consumo y retardos del proyecto se decidió sintetizar y ejecutar los mecanismos en una FPGA.

B.1. Diagrama de Gantt del Proyecto



Actividad	Inicio	Duración (días)	Final
Comprensión del artículo	03/07/24	7	10/07/24
Implementación del método flipping	11/07/24	87	06/10/24
Implementación de biestables	07/10/24	0	07/10/24
Integración de biestables al método flipping	07/10/24	6	13/10/24
Diseño de testbench para método flipping	14/10/24	1	15/10/24
Diseño de memoria cache en Logisim	16/10/24	156	21/03/25
Diseño de memoria cache adaptada para FPGA en Logisim	22/03/25	53	14/05/25
Implementación de memoria cache en SystemVerilog	15/05/25	13	28/05/25
Diseño de testbench para la memoria cache	29/05/25	4	02/06/25
Implementación de método patching	02/06/25	0	02/06/25
Diseño de testbench para método patching	02/06/25	0	02/06/25
Síntesis y ejecución en FPGA	02/06/25	15	17/06/25
Redacción de la memoria	27/05/25	31	27/06/25

Figura B.1: Parte 1 del diagrama de Gantt del proyecto.

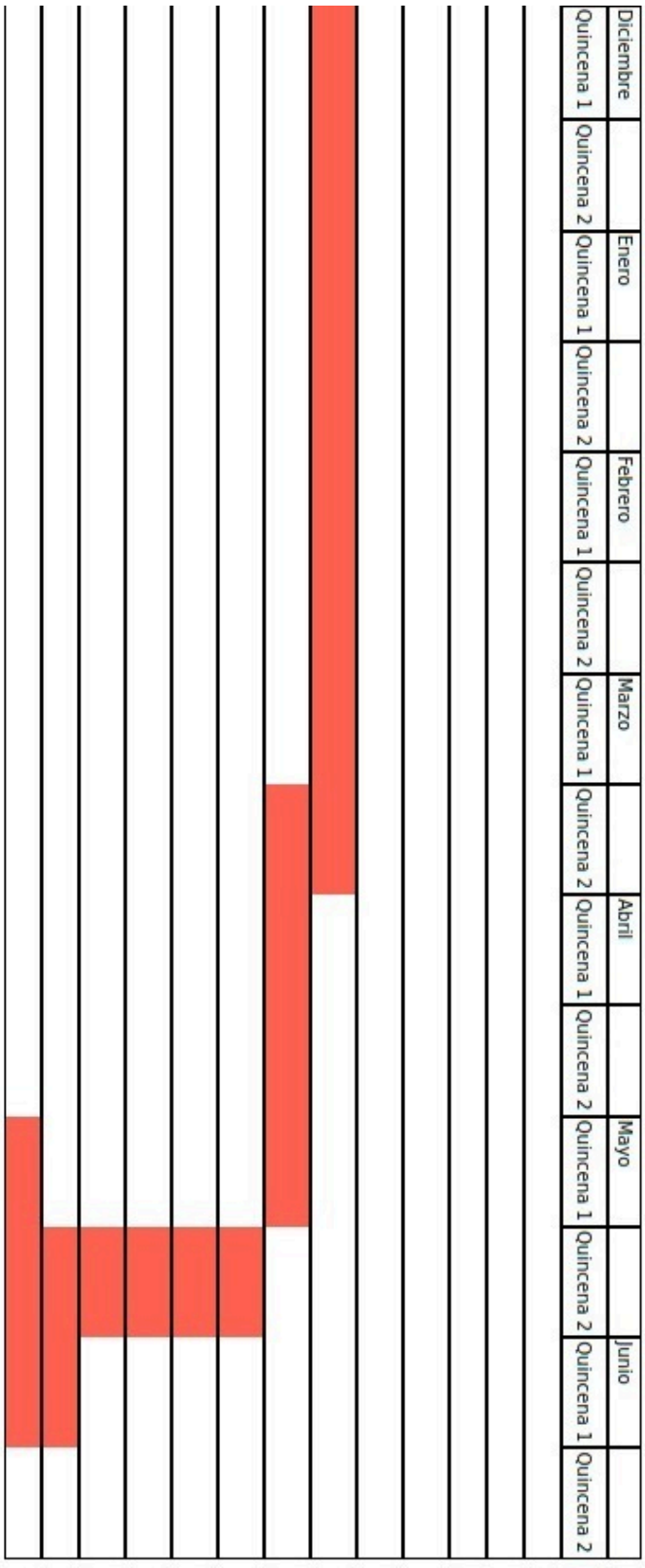


Figura B.2: Parte 2 del diagrama de Gantt del proyecto.

Anexos C

Organización del Proyecto

El desarrollo del proyecto se ha llevado a cabo utilizando el programa Visual Studio Code (VSC) como entorno principal de programación. El código está organizado en un directorio que agrupa los distintos subdirectorios que funcionan como subsistemas, y se encargan de las diferentes partes del sistema total.

Los subsistemas existentes son:

- **mecanismo_flipping**: Este subdirectorio gestiona todo lo relacionado con el método *flipping*, incluidas las pruebas realizadas para comprobar su correcto funcionamiento.

- **cache**: Aquí se colocan todos los bloques necesarios para el funcionamiento de la cache diseñada para el método *patching*. Además, contiene las pruebas de funcionamiento.

- **mecanismo_patching**: Este subdirectorio gestiona todo lo relacionado con el método *patching* y usa la cache implementada en el subdirectorio anterior. También contiene todas sus pruebas de funcionamiento.

Por último, y fuera de los subdirectorios, encontramos el módulo que integra todas las partes del sistema, así como sus pruebas, para comprobar el correcto funcionamiento de todos los subsistemas a la vez.

A continuación, la figura C.1 representa las dependencias entre los diferentes módulos del proyecto, mostrando cómo interactúan entre sí.

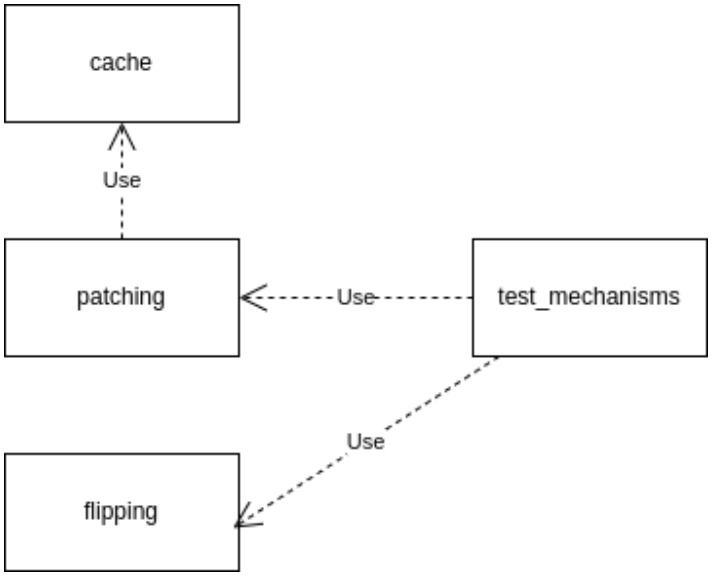


Figura C.1: Esquema de las relaciones entre los subsistemas del proyecto

Anexos D

Pruebas de Validación

La validación funcional de los mecanismos se basa en baterías de tests diseñadas para abarcar tanto casos típicos como situaciones límite. Esta sección detalla los escenarios evaluados.

Las pruebas realizadas para verificar el correcto funcionamiento del mecanismo *flipping* son las que siguen:

- Comprobación con $f = 0$, $f = 1$, valores alternos de f y valores de activaciones diversas como constantes, capicúas o aleatorios.

- Comprobación flanco a flanco de la obtención de los resultados.

Para la memoria cache de respaldo se decide que la mejor manera de comprobar el correcto funcionamiento es comprobar cada una de las transiciones del autómata y corroborar que ocurre el comportamiento esperado. A continuación se detallan las diferentes pruebas:

- Se escriben todas las vías de varios conjuntos seguidos de la cache. Para mayor facilidad en la verificación, se escribe la activación con valor 0 en la etiqueta 0, la activación con valor 1 en la etiqueta 1 y así sucesivamente. Esto permite leer las activaciones una tras otra y encontrar potenciales fallos con facilidad.

- Una vez terminado el proceso de barrido completo en la cache con escrituras, se realiza lo propio con las lecturas, verificando cada valor almacenado.

- Para comprobar que no se permita volver a escribir después de leer sin transitar por el estado de reset, se fuerza una escritura nada más acabar de leer y se comprueba que se transita al estado de error.

- Se verifica que no se puede abandonar el estado de error de ninguna otra manera intentando escribir o leer desde este estado.

– Se resetea la cache y se comprueba que tras esta acción de pueda escribir de nuevo.

– Se prueba mantener varios ciclos sin realizar ninguna acción transitando y manteniéndonos en el estado de *nothing*.

– Se resetea la cache y se intenta leer antes de escribir, lo que conlleva transitar al estado de error.

– Después de volver a resetear se comprueba que si se ocupan todas las vías de un conjunto no se puede volver a escribir en el mismo. Si esto se realiza, el sistema queda en un estado de error y sólo se podrá abandonar este estado con un reset.

– Se resetea de nuevo y se prueba a escribir de nuevo en un mismo conjunto para comprobar que tras el reset se borra (invalida) todo el contenido escrito en la memoria cache.

Por su parte, el mecanismo *patching* ha resultado mucho más sencillo de verificar, ya que su correcto funcionamiento depende únicamente de la selección apropiada de una activación previamente almacenada en la cache. Una vez validado que el mecanismo de lectura y escritura de la cache es fiable, se ha podido comprobar este módulo simplemente evaluando si selecciona correctamente la activación correspondiente según el parámetro de control. La Figura D.1 muestra los resultados esperados en pantalla para distintas configuraciones del parámetro p y diferentes valores almacenados en la cache.

Con todas las pruebas anteriores realizadas satisfactoriamente, el último paso es combinar los dos mecanismos, realizando pruebas de integración consistentes es evaluar todas las posibles combinaciones de los bits de control:

– Se comprueba que si están los bits de f y p en '1' a la vez se produce un error.

– Se comprueba que cuando los bits de f y p están a '0' se obtiene la activación original proveniente de la memoria on-chip.

– Se comprueba que cuando el bit de f es '1' y p es '0' se obtiene la activación flippeada con respecto a la activación original en la memoria on-chip.

– Se comprueba que cuando el bit de f es '0' y p es '1' se obtiene la activación almacenada en la memoria cache de respaldo.

En la Figura D.2 se muestra un resumen de los resultados obtenidos al aplicar simultáneamente las técnicas de *flipping* y *patching* con diferentes valores de activación. Para ello, se ha simulado una activación original proveniente de la memoria on-chip,

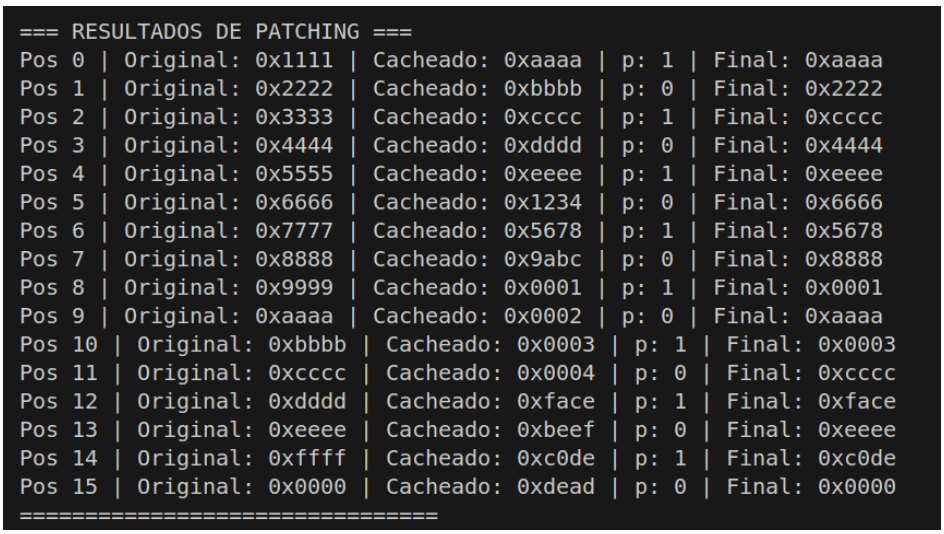
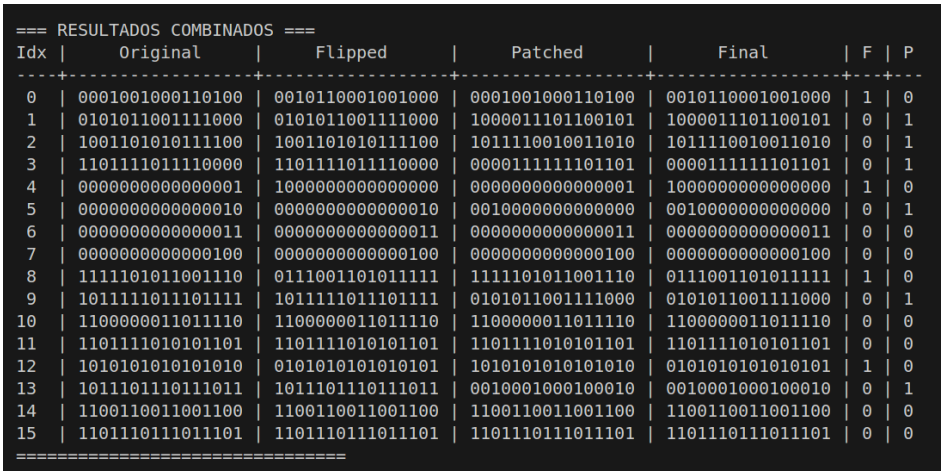


Figura D.1: Pantallazo con los resultados de las pruebas del método *patching*.

junto con distintos valores de los bits de control f y p. Además, se ha rellenado la memoria cache con valores diferentes para comprobar las decisiones de selección en cada caso. Por ejemplo, en la primera línea, con índice `0`, se observa que el bit f está a `1`, por lo que la salida debería ser la activación original, pero con sus bits invertidos (ordenados de derecha a izquierda). Como puede verse, la salida final corresponde efectivamente a esa transformación, confirmando el funcionamiento correcto del mecanismo.



```
=== RESULTADOS COMBINADOS ===
Idx |     Original     |     Flipped      |     Patched      |      Final       | F | P
----+------------------+------------------+------------------+------------------+---+---
 0  | 0001001000110100 | 0010110001001000 | 0001001000110100 | 0010110001001000 | 1 | 0
 1  | 0101011001111000 | 0101011001111000 | 1000011101100101 | 1000011101100101 | 0 | 1
 2  | 1001101010111100 | 1001101010111100 | 1011110010011010 | 1011110010011010 | 0 | 1
 3  | 1101111011110000 | 1101111011110000 | 0000111111101101 | 0000111111101101 | 0 | 1
 4  | 0000000000000001 | 1000000000000000 | 0000000000000001 | 1000000000000000 | 1 | 0
 5  | 0000000000000010 | 0000000000000010 | 0010000000000000 | 0010000000000000 | 0 | 1
 6  | 0000000000000011 | 0000000000000011 | 0000000000000011 | 0000000000000011 | 0 | 0
 7  | 0000000000000100 | 0000000000000100 | 0000000000000100 | 0000000000000100 | 0 | 0
 8  | 1111101011001110 | 0111001101011111 | 1111101011001110 | 0111001101011111 | 1 | 0
 9  | 1011111011101111 | 1011111011101111 | 0101011001111000 | 0101011001111000 | 0 | 1
10  | 1100000011011110 | 1100000011011110 | 1100000011011110 | 1100000011011110 | 0 | 0
11  | 1101111010101101 | 1101111010101101 | 1101111010101101 | 1101111010101101 | 0 | 0
12  | 1010101010101010 | 0101010101010101 | 1010101010101010 | 0101010101010101 | 1 | 0
13  | 1011101110111011 | 1011101110111011 | 0010001000100010 | 0010001000100010 | 0 | 1
14  | 1100110011001100 | 1100110011001100 | 1100110011001100 | 1100110011001100 | 0 | 0
15  | 1101110111011101 | 1101110111011101 | 1101110111011101 | 1101110111011101 | 0 | 0
================================
```

Figura D.2: Pantallazo con los resultados de las pruebas de los dos mecanismos a la vez.

Anexos E

Iteraciones de Síntesis Fallidas

Una vez validados los distintos mecanismos, se procede a la síntesis del diseño completo en la FPGA ZedBoard. El objetivo es comprobar su viabilidad en hardware real, evaluando tanto el uso de recursos como el consumo energético. La frecuencia objetivo es de 100 MHz.

E.1. Síntesis con Bloques de 16 Activaciones

El primer intento de síntesis se realiza con bloques de 16 activaciones procesadas en paralelo. Sin embargo, este enfoque presenta problemas importantes. El más acuciante es el número excesivo de puertos de entrada y salida necesarios, que supera ampliamente la capacidad disponible en la FPGA [17] (más información de la FPGA usada en la sección 3.1). Concretamente, se requieren más de 1300 puertos, mientras que el límite de la FPGA utilizada es de tan sólo de 200, tal como se puede apreciar en la Figura E.1.

En cuanto al consumo energético, la Figura E.2 muestra que más del 90 % de la energía total estimada se destina a los puertos de entrada y salida, resultando en 0,190 W. Esto implica que el gasto energético no está relacionado con el procesamiento lógico interno, sino con la transmisión de datos, haciendo que la estimación no sea representativa del comportamiento del sistema real.

E.2. Síntesis con Bloques de 2 y 1 Activación

Para abordar el problema anterior, se reduce el tamaño del bloque a 2 activaciones. Esta modificación permite disminuir el número de puertos, aunque sigue sin ser suficiente para cumplir con las limitaciones de la FPGA tal como muestra la Figura E.3, donde el número de puertos necesarios queda establecido en 331. No obstante, el consumo energético total mejora significativamente como muestra la Figura E.4, reduciéndose de 0,32 W (Figura E.2) a 0,163 W. En esta nueva configuración, el peso relativo de la

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Bonded IOB (200)	BUFGC TRL
test_mechanisms	731	796	32	16	1315	1
patch_inst (top_patching_final)	592	256	32	16	0	0
flip_inst (mecanismo_flipping_flipflop)	128	528	0	0	0	0

Figura E.1: Estimación de uso de componentes hardware de la FPGA.

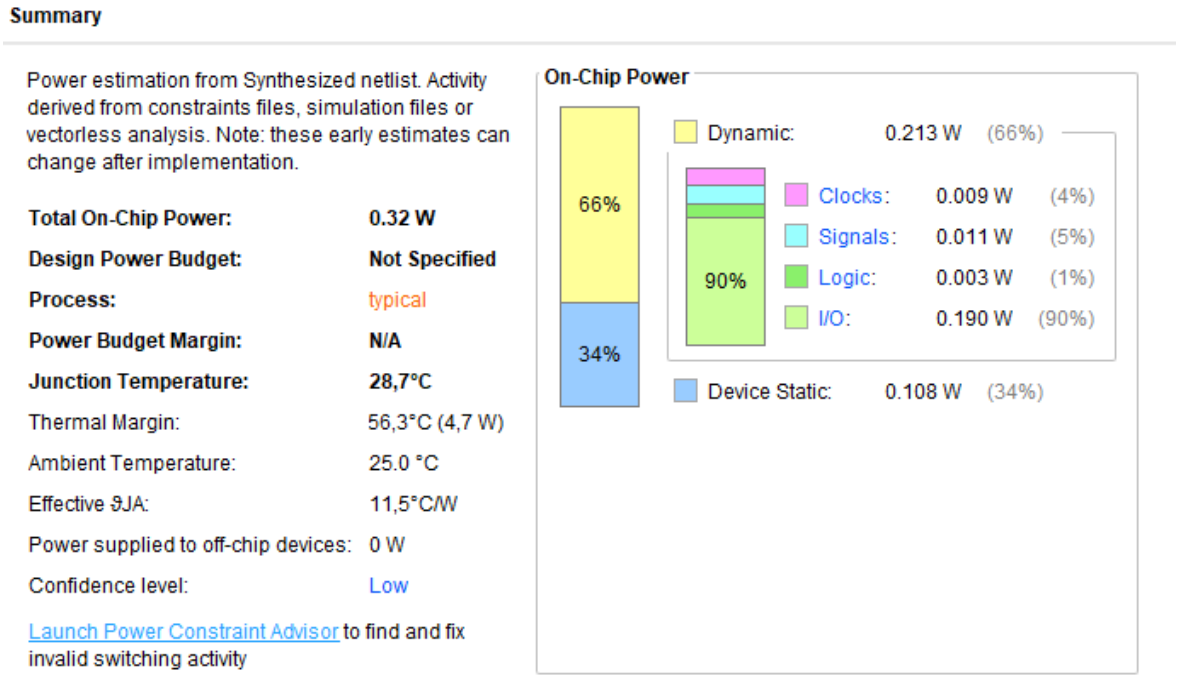


Figura E.2: Estimación de la energía consumida por el diseño en la FPGA.

Name	Slice LUTs (53200)	Slice Registers (106400)	Bonded IOB (200)	BUFGCTRL (32)
test_mechanisms	184	202	331	1
flip_inst (mecanismo_flipping_flipflop)	32	132	0	0

Figura E.3: Estimación de uso de componentes hardware en la FPGA con bloques de 2 activaciones.

energía dinámica —es decir, aquella consumida por los componentes activos durante el funcionamiento— disminuyó respecto al total. Este valor es más representativo del consumo real del sistema, al reducirse el impacto de la transmisión de datos externos.

Finalmente, se prueba el diseño con bloques de una sola activación. En esta configuración sí se logra cumplir con las restricciones de entrada/salida como se puede ver en la Figura E.5, donde los puertos resultan ser 85, completando la síntesis con éxito.

En la figura E.6 se muestra la estimación de energía consumida por el diseño con bloques de una activación. Como puede observarse, el consumo se ha reducido

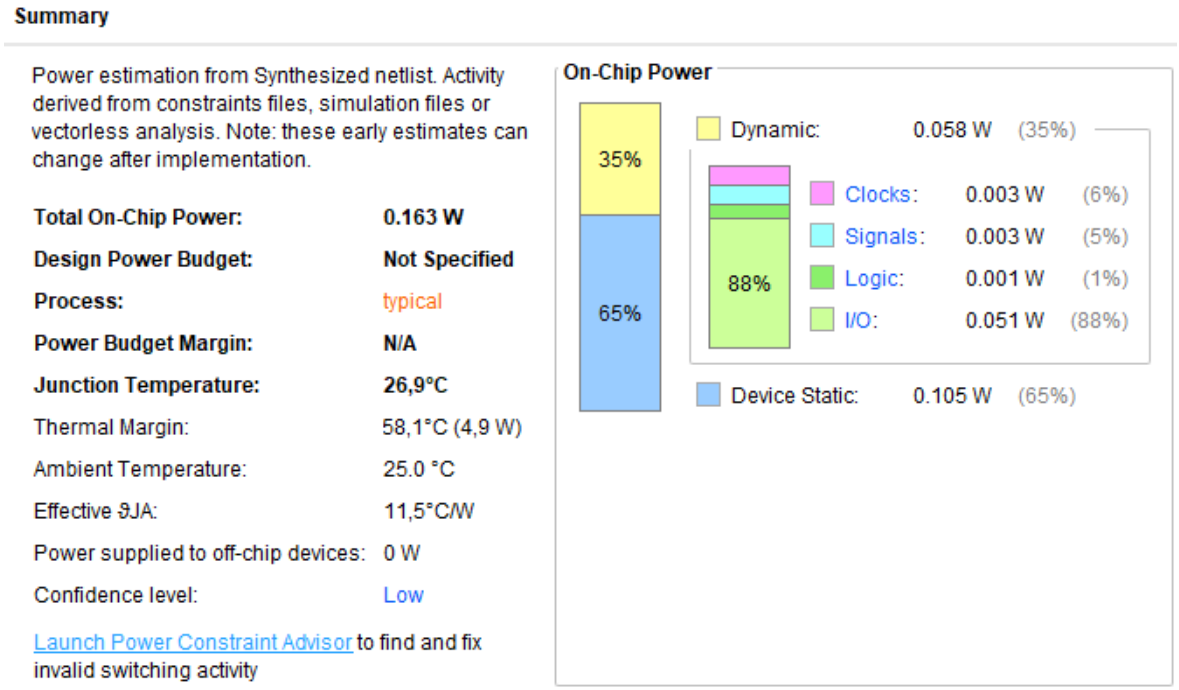


Figura E.4: Estimación de energía consumida en la FPGA con bloques de 2 activaciones.

Name	Slice LUTs (53200)	Slice Registers (106400)	Bonded IOB (200)	BUFGCTRL (32)
top_test_mechanisms	42	53	85	1
dut (test_mechanisms)	42	53	0	0

Figura E.5: Estimación de uso de componentes hardware en la FPGA con bloques de una activación.

considerablemente respecto a configuraciones anteriores, alcanzando un valor de tan solo 0,014 W. Esta disminución es esperable debido al uso de una única activación y al menor número de puertos de entrada y salida implicados en el diseño, lo que reduce significativamente la actividad interna del sistema durante su funcionamiento.

No obstante, al revisar el diseño sintetizado, se comprueba que la FPGA no había sintetizado la memoria cache. Esto puede deberse a varias razones comunes en el flujo de síntesis. La más probable es que el módulo de la cache no esté conectado a ninguna salida observable o ninguna parte del circuito que realmente afecte al comportamiento final del sistema. En ese caso, el sintetizador detecta que ese bloque no tiene efecto en el funcionamiento del diseño y lo elimina automáticamente para ahorrar recursos. Este comportamiento se observa tanto en la configuración con una activación como en la de 2 y 16 activaciones, lo que indica que el módulo de cache fue descartado sistemáticamente por no considerarse funcionalmente relevante desde la perspectiva del sintetizador.

Utilization	Name	Clocks (W)	Signals (W)	Data (W)	Clock Enable (W)	Logic (W)	I/O (W)
0.014 W (12% of total)	top_test_mechanisms						
0.012 W (10% of total)	Leaf Cells (86)						
0.002 W (2% of total)	dut (test_mechanisms)	0.001	<0.001	<0.001	<0.001	<0.001	<0.001
0.001 W (1% of total)	flip_inst (mecanismo_flip	0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (1% of total)	patch_inst (top_patching_	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total)	selector_inst (mux_selec	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total)	Leaf Cells (9)						

Figura E.6: Estimación de energía consumida en la FPGA con bloques de una activación.