Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Laboratorio de Diseño de Circuitos Integrados (DCILab)

Rediseño del módulo de comunicación Serial Peripheral Interface - Bootloader (SPI-Bootloader) en microcontrolador SIWA

Informe de Proyecto de Graduación para optar por el título de Ingeniero en Electrónica con el grado académico de Licenciatura

Jean Carlos Chacón Pérez

Noviembre, 2025
Cartago, Costa Rica

Declaratoria de Autenticidad

Declaro que el presente anteproyecto es una producción original de mi autoría exclusiva. Toda la información técnica y conceptual que aquí se expone ha sido desarrollada por mí o ha sido debidamente referenciada. Me comprometo a no incurrir en prácticas de plagio y a respetar los principios de integridad académica establecidos por el Instituto Tecnológico de Costa Rica. Asumo la total responsabilidad por el presente trabajo y su contenido.

Jean Carlos Chacón Pérez
Cédula: 1 1773 0080
28 de noviembre del 2025
Cartago, Costa Rica

INSTITUTO TECNOLÓGICO DE COSTA RICA

ESCUELA DE INGENIERÍA ELECTRÓNICA

TRABAJO FINAL DE GRADUACIÓN

ACTA DE APROBACIÓN

Defensa del Trabajo Final de Graduación
Requisito para optar por el título de Ingeniero en Electrónica
Grado Académico de Licenciatura
Instituto Tecnológico de Costa Rica

El Tribunal Evaluador aprueba la defensa del Trabajo Final de Graduación denominado Rediseño del módulo de comunicación Serial Peripheral Interface - Bootloader (SPI-Bootloader) en microcontrolador SIWA, realizado por el señor Jean Carlos Chacón Pérez y, hace constar que cumple con las normas establecidas por la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal Evaluador

Ing. Alfonso Chacón Rodríguez
Profesor lector

Ing. Ronny García Ramírez
Profesor lector

Ing. Roberto Molina Robles
Profesor asesor

Cartago, 28 de noviembre, 2025

Resumen

En este trabajo se presenta el rediseño del módulo de comunicación Serial Peripheral Interface – Bootloader (SPI-Bootloader) del microcontrolador Siwa, desarrollado en el Laboratorio de Diseño de Circuitos Integrados (DCILab) del Instituto Tecnológico de Costa Rica. El sistema original empleaba un mecanismo ambiguo para determinar el fin del proceso de arranque, basado en la detección del valor `0xFFFFFFFF` dentro del firmware almacenado en la memoria Flash, lo cual podía provocar la terminación prematura de la carga y comprometer la inicialización del microcontrolador. Para eliminar esta condición de riesgo, se introduce un encabezado estructurado (*header*) embebido en el firmware, que incluye campos de validación y metadatos (MAGIC, VERSION, SIZE y RESERVED). El módulo SPI fue rediseñado para decodificar este encabezado, garantizando un proceso de carga determinista y libre de ambigüedad.

Además, se implementa la reutilización de la memoria Flash como almacenamiento secundario tras finalizar el arranque, habilitando comandos estándar de lectura, escritura y borrado según el protocolo JEDEC. El nuevo diseño incorpora mecanismos de alineación, reconstrucción de palabras, manejo de casos límite y compatibilidad con diferentes formas de programación de la Flash. Los resultados de simulación y síntesis evidencian mejoras en confiabilidad, trazabilidad y robustez funcional del arranque, manteniendo compatibilidad con la arquitectura general de Siwa y sin afectar otros módulos del microcontrolador. Este rediseño fortalece la plataforma para aplicaciones biomédicas implantables, donde la predictibilidad y la integridad del firmware son requisitos esenciales.

Palabras clave: Microcontrolador Siwa; SPI Bootloader; memoria Flash; encabezado de firmware; arranque determinista; RISC-V; diseño VLSI; JEDEC; sistemas embebidos; DCILab.

Abstract

This work presents the redesign of the Serial Peripheral Interface – Bootloader (SPI-Bootloader) communication module of the Siwa microcontroller, developed at the Integrated Circuits Design Laboratory (DCILab) of the Costa Rica Institute of Technology. The original system relied on an ambiguous mechanism to determine the end of the bootloading process, based on detecting the value `0xFFFFFFFF` within the firmware stored in Flash memory. This could lead to premature termination of the loading sequence, compromising the initialization of the microcontroller. To eliminate this risk, a structured firmware header containing validation fields and metadata (MAGIC, VERSION, SIZE, and RESERVED) is introduced. The SPI module was redesigned to decode this header, ensuring a deterministic and unambiguous boot process.

Furthermore, the redesign enables the reuse of the Flash memory as secondary storage after the boot sequence finishes, supporting standard JEDEC read, write, and erase commands. The new architecture incorporates byte-alignment mechanisms, word reconstruction, boundary-case handling, and compatibility with different Flash programming schemes. Simulation and synthesis results demonstrate improvements in reliability, traceability, and robustness of the boot process, while preserving compatibility with the overall Siwa architecture and without affecting other microcontroller modules. This redesign strengthens the platform for implantable biomedical applications, where firmware integrity and predictable system behavior are essential.

Keywords: Siwa microcontroller; SPI Bootloader; Flash memory; firmware header; deterministic boot; RISC-V; VLSI design; JEDEC; embedded systems; DCILab.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo inicial se presenta de forma clara el entorno del problema central que da origen al desarrollo del proyecto. Asimismo, se explica la relevancia de abordar dicha problemática y se introduce la solución propuesta para cumplir con los objetivos planteados.

1.1. Antecedentes y entorno del proyecto

El sector de dispositivos médicos se ha consolidado como un eje estratégico de la economía costarricense, al convertirse en el principal producto de exportación del país y en un motor de generación de empleo y desarrollo tecnológico. De acuerdo con la Coalición Costarricense de Iniciativas de Desarrollo (CINDE), en el año 2017 esta industria brindó empleo directo a 22 399 personas, lo que refleja su creciente relevancia en el mercado laboral [1]. Asimismo, datos de la Promotora del Comercio Exterior de Costa Rica (PROCOMER) señalan que, en ese mismo período, los dispositivos médicos representaron alrededor del 27 % de las exportaciones nacionales de bienes, superando por primera vez a la tradicional producción agrícola. Este hecho marcó un cambio estructural en la composición de la matriz exportadora del país. En años recientes, dicho liderazgo se ha fortalecido al punto de que, para el 2024, Costa Rica se posicionó como el exportador per cápita de dispositivos médicos número uno en América. [2].

La presencia de compañías multinacionales de dispositivos médicos no solo ha favorecido la inversión extranjera directa, sino que además ha posicionado al país como un centro de manufactura avanzada en América Latina. Desempeñando un rol catalizador del desarrollo tecnológico e innovación, que se traduce en mayores oportunidades de investigación e implementación de procesos productivos, que coloca a Costa Rica dentro de los mercados de más alto nivel.

La ingeniería electrónica ha tenido un impacto vital en el campo de la salud, ya que ha permitido avances que favorecen tanto la atención médica como el bienestar de la sociedad. Este aporte se materializa, en gran medida, en el diseño de dispositivos. Sin embargo, también en la optimización y modernización de equipos preexistentes que continúan siendo fundamentales en la práctica médica.

El Tecnológico de Costa Rica (TEC), a través de la Escuela de Ingeniería Electrónica, ha participado en proyectos en la industria médica, mediante las áreas de computación y arquitectura de computadoras. Uno de los laboratorios dedicados a este tipo de iniciativas es el Laboratorio de Diseño de Circuitos Integrados, también conocido por sus siglas como DCI Lab.

El laboratorio es un centro de referencia en microelectrónica a nivel centroamericano, con capacidad para ejecutar el diseño, simulación y la validación de circuitos integrados. Ubicado en el campus central de Cartago, ha liderado proyectos como el microcontrolador Siwa (de la lengua cabécar, significa "sabiduría ancestral") o dispositivos para diagnóstico biomédico por espectroscopía de impedancia. Además de su aporte tecnológico, forma talento humano altamente capacitado e impulsa la autonomía tecnológica del país, posicionando a la Escuela de Ingeniería Electrónica como un actor clave en la industria de semiconductores en la región.

El desarrollo del presente proyecto se realizó sobre Siwa, el primer microcontrolador diseñado completamente en Costa Rica. El prototipo fue enviado a fabricación en Alemania, a través de la empresa XFAB, reconocida por su experiencia en circuitos integrados, como parte de una colaboración con el Departamento de Ingeniería Electrónica (MicroDIE) de la Universidad Católica de Uruguay.

Consiste en un chip que ejecuta instrucciones y controla el funcionamiento de un sistema electrónico. Lo revolucionario no es solo su diseño, sino que fue diseñado en el país, por estudiantes y profesores, utilizando herramientas de diseño electrónico y logrando una fabricación con tecnología de 180 nanómetros, lo cual demuestra que en Costa Rica se puede hacer alta tecnología desde cero y la importancia de atraer empresas que realicen manufactura de este tipo.

El principal propósito del microcontrolador es servir como unidad central de procesamiento de dispositivos médicos implantables, como marcapasos o estimuladores musculares, donde se requieren altos niveles de confiabilidad, bajo consumo energético y control completo sobre el sistema [3].

Está basado en RISC-V, una arquitectura de conjunto de instrucciones (ISA) del tipo RISC (Reduced Instruction Set Computer), que define el conjunto de operaciones que un microprocesador es capaz de ejecutar. Esta especificación se caracteriza por ser abierta, gratuita y altamente personalizable, permitiendo a los diseñadores desarrollar implementaciones adaptadas a los requisitos específicos de un sistema o producto. A diferencia de otras arquitecturas comerciales de carácter cerrado, RISC-V no depende de licencias propietarias, lo cual resulta fundamental en aplicaciones que requieren un control total sobre el hardware.

En términos de microarquitectura, el microcontrolador incorpora un protocolo de comunicación digital denominado Interfaz Periférica Serie o Serial Peripheral Interface (SPI). Se trata de un protocolo síncrono ampliamente utilizado para la comunicación entre un dispositivo controlador y uno o varios periféricos. Su funcionamiento se basa en un reloj generado exclusivamente por el controlador, encargado de sincronizar el intercambio de bits durante cada ciclo de transmisión. SPI utiliza un conjunto mínimo de señales: MOSI, MISO, SCLK y CS/SS; que permiten implementar un esquema full-duplex, en el cual la transmisión y recepción de datos pueden ocurrir de forma simultánea. Gracias a su simplicidad y alta velocidad de transferencia, SPI es una opción habitual para memorias Flash, convertidores ADC/DAC, sensores digitales y otros dispositivos que requieren un intercambio rápido.

Los protocolos de comunicación digital, como SPI, constituyen un elemento fundamental en el diseño de sistemas embebidos, los cuales se caracterizan por estar concebidos para cumplir funciones muy específicas dentro de un sistema o producto de mayor escala. A diferencia de los computadores de propósito general, estos sistemas deben operar con recursos limitados, tanto en capacidad de procesamiento como en memoria y consumo energético. Además, suelen requerir comportamiento en tiempo real, lo que implica que ciertas operaciones deben ejecutarse dentro de ventanas temporales estrictamente definidas, especialmente en aplicaciones críticas como las automotrices, aeroespaciales o biomédicas.

En el caso de Siwa, orientado a aplicaciones biomédicas implantables, protocolos como SPI desempeñan un papel esencial al posibilitar la comunicación con memorias externas y módulos de soporte, asegurando la correcta inicialización y operación del sistema.

Un aspecto clave en este proceso es el bootstrapping (también denominado boot, booteo o arranque). Este término hace referencia a la secuencia de acciones que ejecuta el procesador desde el momento en que recibe alimentación eléctrica hasta que se encuentra en condiciones de cargar y ejecutar el firmware principal. En la práctica, este proceso parte de un conjunto reducido de instrucciones alojadas en una memoria interna mínima, que habilitan la lectura del firmware almacenado en una memoria no volátil externa, generalmente una memoria Flash.

La memoria Flash es especialmente importante en este contexto. Se trata de un medio de almacenamiento no volátil, capaz de conservar los datos incluso en ausencia de energía eléctrica, lo que garantiza la disponibilidad permanente del firmware. Su fiabilidad, alta densidad de almacenamiento y bajo consumo energético la convierten en una solución ideal para sistemas embebidos. Más allá de este ámbito, la memoria Flash constituye la base tecnológica de dispositivos de almacenamiento masivo como unidades de estado sólido (SSD), unidades USB y tarjetas SD en sus diferentes variantes.

Esta etapa inicial, aunque fundamental, introduce una serie de desafíos relacionados con la integridad de los datos, la alineación del flujo de lectura y la correcta interpretación del contenido almacenado. Estas limitaciones y posibles puntos de fallo permiten identificar con claridad las problemáticas que aborda el proyecto y justifican la necesidad de una solución.

1.2. Problema existente e importancia de la solución

En el entorno actual del microcontrolador, el módulo SPI es responsable de trasladar, durante la fase de arranque, el código almacenado en una memoria flash externa hacia la memoria de programa interna.

La finalización de este proceso se determina utilizando un patrón de datos específico (`0xFFFFFFFF`) como marca de terminación, ya que al ser encontrado la carga se detiene. Este enfoque introduce una ambigüedad, debido a que dicho patrón puede aparecer legítimamente en un programa, bien sea como número (equivalente a -1 en representación de 32 bits con complemento a dos) o como instrucción válida.

En consecuencia, el microcontrolador confunde un elemento válido con una orden de fin de carga, lo que se traduce en:

1. Terminación prematura del proceso de carga del programa, dejando el sistema en un estado incompleto.
2. Ejecución parcial o corrupta del código durante la inicialización, lo que puede generar fallas funcionales.

Estos riesgos resultan especialmente críticos en aplicaciones donde se requieren comportamientos deterministas y trazabilidad para el correcto funcionamiento, como los dispositivos médicos para los cuales fue concebido este sistema. En este dominio, cualquier indeterminación puede traducirse en fallas funcionales en operación real. Dichas fallas, en el caso de dispositivos biomédicos implantables o de soporte vital, representan un riesgo directo para la seguridad del paciente y, en situaciones extremas, podrían comprometer la vida de las personas.

A pesar de que esta debilidad ha sido identificada en iteraciones anteriores, no se han implementado soluciones en el diseño del módulo SPI. Esto mantiene latente un riesgo funcional cada vez que el microcontrolador es integrado en un sistema.

Adicionalmente, basar el control en un valor de datos contradice buenas prácticas de diseño digital, donde se busca mantener una clara separación entre el plano de control y el plano de datos. Al no respetar esta separación, la controlabilidad se vuelve más compleja y el sistema queda expuesto a comportamientos ambiguos.

Por otra parte, tras finalizar el proceso de arranque, el diseño vigente no incorpora de forma adecuada la transición que habilite el uso de la memoria flash como memoria secundaria. Persisten errores en las operaciones de escritura, lo que impide utilizar la memoria como un medio de almacenamiento seguro y limita la capacidad del sistema.

En síntesis, ambos problemas: la ambigüedad en el arranque y la falta de reutilización de la memoria flash ponen en entredicho la funcionalidad del microcontrolador y restringen su aplicabilidad.

1.3. Requisitos de diseño, restricciones y solución seleccionada

Requisitos de diseño

A partir del análisis del problema descrito, se establecen los requisitos técnicos y de diseño que debe cumplir la solución. Estos criterios buscan garantizar que el rediseño del módulo SPI elimine la ambigüedad en el arranque y permita un uso confiable de la memoria flash tras la inicialización.

Los requisitos principales son los siguientes:

- Reemplazo del mecanismo de fin de carga: Sustituir el mecanismo basado en el valor `0xFFFFFFFF` por un esquema interno de control que permita una finalización no ambigua del proceso de carga.
- Rediseño para soporte de memoria secundaria: Tras finalizar el proceso de arranque, el módulo SPI debe permitir el acceso a la memoria flash en modo lectura y escritura, garantizando su reutilización como unidad de almacenamiento extendido.
- Preservación de la funcionalidad global del sistema: El rediseño debe mantenerse aislado al módulo SPI. No debe alterar el comportamiento, la microarquitectura ni la lógica funcional de otros módulos existentes en el microcontrolador.
- Compatibilidad tecnológica: Asegurar que el rediseño sea completamente sintetizable en tecnología CMOS de 180 nm, evaluado mediante herramientas estándar de síntesis, simulación y verificación de área, consumo y retardo.
- Modularidad y escalabilidad: Diseñar la solución de forma jerárquica y parametrizable, de modo que pueda reutilizarse o adaptarse a variantes futuras del microcontrolador.

Restricciones

Durante el desarrollo del rediseño se deben considerar las siguientes restricciones, las cuales junto con los requisitos de diseño, delimitan el alcance técnico del proyecto y condicionan las posibles soluciones implementables:

- Tamaño del programa desconocido: No es posible conocer de antemano la extensión exacta del programa almacenado en la memoria flash, ya que esta puede variar según el binario cargado. En consecuencia, no se puede establecer un límite fijo de lectura basado únicamente en el contenido.
- Independencia funcional del módulo SPI: El rediseño no debe extraer señales externas de otros módulos del sistema. Todas las modificaciones deben quedar confinadas dentro de la lógica interna del módulo SPI.

Solución seleccionada

Con base en los requisitos y restricciones establecidos, la solución propuesta consiste en un rediseño del módulo SPI, orientado a eliminar la ambigüedad presente en el proceso de arranque y habilitar el uso de la memoria flash como almacenamiento secundario tras la inicialización del sistema.

Eliminación de la ambigüedad en el arranque

Para garantizar una finalización exacta del proceso de carga del programa, se introduce un encabezado (header), un pequeño bloque de datos en el binario almacenado en la memoria flash que contiene metadatos sobre el programa principal. Este encabezado se ubica al inicio del archivo, antes del código de aplicación, y reúne información esencial para validar y estructurar el proceso de arranque. La responsabilidad de generar este encabezado recae sobre el programador durante la construcción del binario, mientras que el módulo SPI fue rediseñado para reconocerlo y utilizarlo como referencia de control durante el arranque.

En la nueva secuencia de inicialización, el módulo SPI realiza una lectura del header y extrae los parámetros necesarios para conducir la transferencia. Entre ellos se encuentra el campo `SIZE`, que indica el tamaño total del firmware que debe copiarse hacia la memoria interna. Con este valor, el controlador limita la transferencia al número exacto de bytes indicados, eliminando así la dependencia de patrones de datos (como `0xFFFFFFFF`) y evitando que el contenido del programa interfiera con la lógica de control del arranque.

El mecanismo funciona de manera sencilla. Primero, el módulo SPI lee el encabezado y verifica que el `MAGIC` sea válido. Si la imagen es correcta, extrae del header el campo `SIZE`, que indica cuántos bytes deben transferirse hacia la memoria interna. Con este valor se fija un límite claro para la copia del firmware. Mientras la lectura avanza, un contador interno lleva el registro de los bytes ya transferidos y, cuando este contador alcanza el valor definido por `SIZE`, el módulo activa una señal de finalización que marca el fin correcto del proceso de arranque.

Este esquema asegura que la condición de fin de carga se defina por un criterio interno y verificable, independientemente del contenido del programa per se en la memoria flash.

Reutilización de la memoria flash como almacenamiento secundario

Una vez finalizada la etapa de arranque, la memoria Flash no debe quedar limitada exclusivamente al rol de origen del firmware, sino habilitada como un recurso de almacenamiento no volátil disponible para el procesador durante la operación normal del sistema. En la microarquitectura vigente, la funcionalidad no está implementada adecuadamente, lo que impide el uso como memoria secundaria.

El rediseño propuesto incorpora una lógica interna que permite alternar de forma controlada entre los modos de operación:

- Modo de arranque: el módulo SPI ejecuta únicamente lecturas secuenciales destinadas a obtener el encabezado y el contenido del firmware. De tal manera que, la comunicación se restringe a operaciones de lectura continua hasta completar la carga del programa en SRAM.
- Modo operativo: habilita los comandos estándar del protocolo SPI para lectura, escritura y borrado, permitiendo al procesador utilizar la memoria como un medio de almacenamiento de datos.

La señal que habilita el modo operativo se deriva directamente de la finalización del proceso de arranque, de modo que la memoria solo se desbloquea cuando el firmware ha sido copiado de forma correcta y verificable. Con ello, se garantiza que la reutilización de la memoria no compromete la integridad del flujo de boot y que el microcontrolador dispone de un dispositivo Flash plenamente funcional durante toda su operación.

Alcances del rediseño

El nuevo enfoque introduce mejoras significativas en la confiabilidad y funcionalidad del sistema:

- La finalización del arranque se determina a partir de un valor explícito del encabezado, eliminando el antiguo criterio ambiguo basado en `0xFFFFFFFF`.
- El proceso de copia del firmware se vuelve determinista y verificable.
- La memoria Flash adquiere un uso dual: primero como origen del firmware y posteriormente como dispositivo de almacenamiento, habilitando operaciones de lectura, escritura y borrado conforme al estándar JEDEC.
- El rediseño se integra de forma aislada dentro del módulo SPI, sin modificar otros bloques del microcontrolador, manteniendo la arquitectura general del sistema.

Capítulo 2

Marco Teórico

En este capítulo se presentan las bases teóricas necesarias para el desarrollo del proyecto. Los fundamentos abordados en esta sección incluyen los principios del protocolo SPI, SPI Bootloader, la memoria IS25WP032D, System Verilog, Diseño VLSI y flujo de diseño e información técnica sobre microcontrolador Siwa.

2.1. Protocolo SPI

El protocolo de comunicación digital denominado Interfaz Periférica Serie, o "Serial Peripheral Interface" (SPI) constituye un mecanismo mediante el cual circuitos integrados intercambian información de forma síncrona, típicamente entre un dispositivo maestro (controlador) y uno o varios esclavos (periféricos) como sensores, memorias o pantallas. A diferencia de los enlaces paralelos, donde los bits viajan simultáneamente por múltiples conductores, SPI transmite la información en serie, bit a bit, sobre líneas dedicadas.

En la configuración más común, el maestro gobierna el reloj de transmisión y coordina todas las transacciones, determinando cuándo iniciar la comunicación y con cuál periférico interactuar. El bus emplea cuatro señales fundamentales:

- MOSI (*Master-Out, Slave-In*): Línea por la cual el maestro envía datos hacia el dispositivo esclavo. Todo bit transmitido por el maestro se coloca sobre MOSI y es muestreado por el periférico siguiendo los flancos del reloj. Esta señal constituye el canal principal de salida del maestro y permite enviar comandos, direcciones o datos según el protocolo específico del dispositivo conectado.
- MISO (*Master-In, Slave-Out*): Línea utilizada por el dispositivo esclavo para retornar información hacia el maestro. Durante cada ciclo de reloj, el esclavo coloca un bit en MISO, el cual es capturado por el maestro de acuerdo con la configuración del modo SPI. Esta señal habilita la comunicación en sentido inverso, permitiendo leer registros, recibir datos de sensores o extraer palabras desde

una memoria Flash.

- SCLK (*Serial Clock*): Señal de reloj generada exclusivamente por el maestro, encargada de sincronizar tanto la transmisión como la recepción de datos. Cada flanco de SCLK indica cuándo los datos deben ser estables y válidos para su captura. El manejo centralizado del reloj simplifica el diseño del bus y asegura que todos los dispositivos conectados operen con la misma referencia temporal.
- CS (*Chip Select*): Señal que habilita al dispositivo esclavo con el cual el maestro desea comunicarse. Por convención, el nivel activo suele ser bajo. Cuando CS está desactivada, el periférico ignora cualquier actividad en MOSI, MISO y SCLK; al activarse, queda habilitado para interpretar comandos y participar en la transferencia de datos. Este mecanismo permite que múltiples dispositivos compartan el mismo bus SPI sin interferencias.

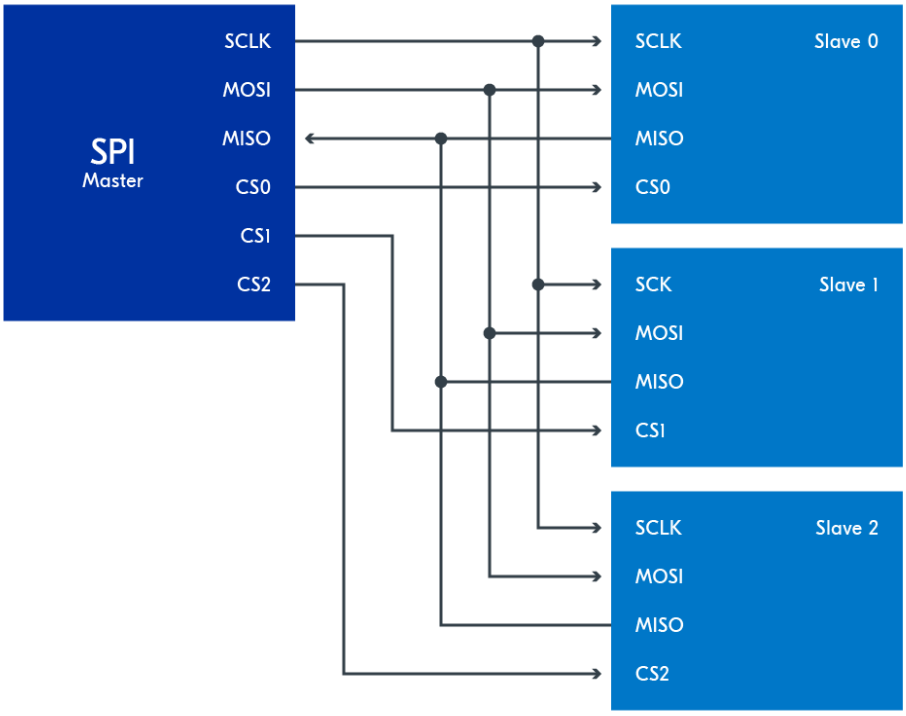

Figura 2.1. Ejemplo de topología SPI: un maestro y tres periféricos. Fuente: [4]

2.2. SPI Bootloader

El proceso de arranque mediante interfaz SPI, o *SPI Bootloader*, constituye el mecanismo a través del cual un sistema digital carga su firmware desde una memoria Flash externa utilizando el protocolo SPI como medio de transferencia. A diferencia de la comunicación SPI convencional, cuyo objetivo es únicamente el intercambio de datos entre un maestro y sus periféricos, el SPI Bootloader implementa una secuencia estructurada de lectura, validación e interpretación de información crítica para que el microcontrolador pueda inicializarse correctamente tras cada encendido. Este proceso permite separar físicamente la memoria de programa del núcleo de procesamiento, otorgando mayor flexibilidad y modularidad al diseño del sistema.

En un escenario típico, el microcontrolador actúa como maestro SPI desde los primeros ciclos posteriores a la alimentación, y se encarga de recuperar un bloque mínimo de instrucciones denominado boot code o código de arranque. Dicho código se encuentra almacenado en una región fija y conocida de la memoria Flash, y está diseñado para habilitar la lectura del firmware completo desde la misma memoria no volátil. El funcionamiento del SPI Bootloader sigue una secuencia definida: primero activa al dispositivo Flash mediante la línea CS, luego transmite comandos SPI específicos para lectura, y finalmente recibe los datos del firmware a través de MISO, sincronizados por la señal de reloj SCLK.

Durante este proceso, el Bootloader no solo transfiere datos, sino que también debe interpretar estructuras internas como encabezados (headers), que contienen información esencial para validar la imagen almacenada. Entre estos datos se incluyen identificadores como:

- El magic number.
- La versión del firmware.
- El tamaño del bloque a cargar.
- Posibles campos reservados para integridad o compatibilidad.

La correcta decodificación de estos campos asegura que el contenido recuperado desde la memoria Flash sea coherente y válido antes de ser copiado a la memoria de programa interna.

Una vez verificada la integridad del encabezado, el SPI Bootloader procede a leer el payload del firmware, generalmente organizado en palabras o bloques de longitud fija. Estos datos se transfieren secuencialmente desde la Flash hacia la SRAM o memoria ejecutable del sistema, donde finalmente podrán ser interpretados y ejecutados por el procesador. Gracias al uso de SPI, este proceso puede realizarse con un número reducido de señales y a velocidades suficientemente altas como para minimizar el tiempo de arranque, manteniendo al mismo tiempo una implementación efectiva en hardware.

2.2.1. Encabezado del Firmware (Header)

El encabezado del firmware, o header, constituye la estructura inicial que precede al contenido principal del programa almacenado en la memoria Flash. Su función es proporcionar al sistema información esencial para validar, interpretar y cargar correctamente la imagen de firmware durante el proceso de arranque. A diferencia del payload, que contiene las instrucciones y datos ejecutables, el encabezado actúa como un bloque de metadatos estructurados que orientan el funcionamiento del Bootloader y garantizan la coherencia de la imagen almacenada.

En términos generales, el header se encuentra ubicado en una dirección fija dentro de la memoria no volátil, de modo que el Bootloader pueda acceder a él inmediatamente después de activar la interfaz SPI y ejecutar las primeras instrucciones del proceso de lectura. Al recuperar este bloque inicial de datos, el sistema obtiene parámetros fundamentales como el identificador de formato, la versión del firmware, el tamaño total del contenido a cargar y posibles campos reservados para futuras extensiones o validaciones adicionales. Esta separación entre metadatos y contenido ejecutable permite implementar mecanismos de compatibilidad, control de versiones y verificación, favoreciendo el ciclo de arranque. Dentro de los campos más relevantes del encabezado se encuentran:

- Magic Number: Valor constante predefinido que funciona como identificador del formato de firmware esperado. Su presencia permite al sistema detectar errores graves, tales como corrupción de datos, escritura incompleta o la lectura accidental de información no válida. Actúa como una firma que confirma que el contenido de la memoria Flash corresponde a una imagen compatible.
- Size: Campo que especifica el tamaño total, en bytes, del payload del firmware. Esta información es indispensable para que el Bootloader determine cuántos datos debe leer desde la memoria Flash y evitar lecturas por fuera de los límites definidos. Garantiza, además, que el proceso de copia del firmware a la memoria interna se realice de forma segura y completa.
- Version: Indica la versión del firmware almacenado y permite gestionar la compatibilidad entre distintas revisiones del software del sistema. Este parámetro facilita la detección de versiones obsoletas, incompatibles o no autorizadas, contribuyendo a la trazabilidad y mantenimiento del sistema embebido.
- Reserved: Espacio reservado para futuras extensiones o funcionalidades adicionales del formato de firmware. Estos campos permiten incorporar características nuevas sin necesidad de alterar la estructura general del encabezado, garantizando la compatibilidad hacia adelante.

Durante el proceso de lectura mediante SPI, el header puede no coincidir exacta ni naturalmente con los límites de palabra utilizados por el hardware, motivo por el cual es habitual que el Bootloader deba aplicar mecanismos de alineación y reensamblado de bytes. Esto incluye la reconstrucción de palabras de 32 bits a partir de secuencias recibidas en flujos de 8 bits, así como la corrección de endianness o desplazamientos introducidos durante la escritura del firmware. La correcta interpretación del header depende, por tanto, tanto de su estructura lógica como de la capacidad del sistema para reconstruirlo fielmente a partir de los datos suministrados por la memoria Flash.

2.3. Memoria IS25WP032D

La IS25WP032D [5] es una memoria Flash serial de 32 Mbit (4 MB) diseñada para sistemas embebidos que requieren almacenamiento no volátil de alta confiabilidad y acceso mediante protocolos SPI avanzados. Soporta una amplia variedad de modos de operación: *SPI*, *Fast Read*, *Dual I/O*, *Quad I/O*, versiones DTR y el protocolo QPI. En el contexto de este trabajo, se emplean únicamente los modos SPI estándar, específicamente en los modos 0 y 3, lo cual coincide con el soporte implementado en el módulo SPI maestro del sistema. La memoria opera a frecuencias de hasta 50 MHz en lectura normal y 133 MHz en lectura acelerada, proporciona lectura continua en bloques de 8, 16, 32 y 64 bytes, y garantiza una vida útil superior a 100 000 ciclos de escritura y borrado. Desde el punto de vista eléctrico, funciona con tensiones entre 1.65 V y 1.95 V, presentando corrientes típicas de 8 µA en suspensión y 4 mA durante la lectura activa.

El protocolo definido por el fabricante organiza el funcionamiento de la memoria en torno a comandos JEDEC específicos. Estos comandos permiten leer configuraciones internas, cargar datos hacia la memoria, borrar sectores o bloques completos, y comprobar el estado actual del dispositivo:

- Write Enable (WREN, 8'h06): habilita el latch interno de escritura. Sin este comando previo, la memoria no acepta comandos de programación ni borrado.
- Page Program (PP, 8'h02): permite escribir hasta 256 bytes dentro de una página. Si se escriben múltiples bytes consecutivos, la dirección se autoincrementa internamente.
- Normal Read (READ, 8'h03): habilita la lectura secuencial a partir de una dirección de 24 bits. La memoria incrementa automáticamente el puntero interno para permitir flujos de lectura continuos mientras la señal `SCS` permanezca activa.

Este comportamiento determina la estructura temporal de las transacciones SPI: cada operación de lectura o escritura comienza con un comando de un byte, seguido de tres bytes de dirección, y continúa con un flujo de datos cuya longitud depende del tipo de operación. La autoincrementación de direcciones y la granularidad de página imponen restricciones que el diseño del Bootloader debe considerar para evitar violaciones del protocolo.

Con el fin de reproducir todas estas características durante la verificación, se desarrolló el módulo `spi_flash_stub`, un modelo comportamental que emula de manera fiel la funcionalidad de la IS25WP032D, el cual se abarca en el siguiente capítulo.

2.4. SystemVerilog

SystemVerilog es un lenguaje que se utiliza para diseñar hardware digital, es decir, circuitos electrónicos que después se convertirán en chips reales. La idea principal es que, así como un programador escribe código para que una computadora ejecute instrucciones, un diseñador de hardware escribe código en SystemVerilog para que un circuito funcione de cierta manera una vez que esté fabricado. La gran diferencia es que este código no se ejecuta en un procesador, más bien se transforma en circuitos físicos, como compuertas lógicas, registros, memorias y señales eléctricas.

A diferencia de lenguajes tradicionales como C, Python o Java, que están pensados para crear programas, SystemVerilog está hecho para describir estructuras de hardware. Por ejemplo, si se necesita un contador, un sumador una máquina de estados que controle un periférico o incluso un pequeño procesador, todo eso se puede describir en SystemVerilog, donde el diseñador indica cómo deben moverse los datos, cuándo deben cambiar las señales internas y qué debe ocurrir en cada ciclo de reloj.

En la práctica, cuando se trabaja con SystemVerilog, el diseñador describe el comportamiento del hardware a nivel de registros y transferencias de datos (nivel RTL). Por ejemplo, puede indicar que cada vez que llegue un pulso de reloj, un registro debe guardar cierto valor, o que una señal debe activarse cuando se cumpla una condición. Todo esto queda escrito de manera textual, pero representa directamente circuitos reales. Luego, este código se pasa por un proceso llamado síntesis, donde herramientas especializadas lo traducen a compuertas lógicas de verdad, siguiendo la tecnología en la que se va a fabricar el chip.

```
1 module inversor (
2     input  logic a,    // Entrada
3     output logic y     // Salida
4 );
5
6     // Asignacion continua: la salida siempre es el inverso de la entrada
7     assign y = ~a;
8
9 endmodule
```

Código 2.1: Ejemplo básico de un inversor escrito en SystemVerilog

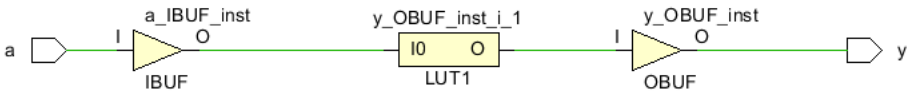

Figura 2.2. Inversor de código 2.1 sintetizado en SystemVerilog.

Este flujo permite que, antes de fabricar un circuito físico, pueda simularse, probarse y corregirse tantas veces como sea necesario, todo desde el código. Justamente por eso SystemVerilog se ha convertido en uno de los lenguajes más utilizados para diseñar microprocesadores, microcontroladores y sistemas completos dentro de un chip.

2.5. Diseño VLSI y flujo de diseño

El diseño VLSI, por sus siglas en inglés Very-Large-Scale Integration, describe el proceso mediante el cual un circuito electrónico complejo pasa de ser una idea a convertirse en hardware real fabricado en silicio. El término VLSI se refiere a la integración de millones de transistores dentro de un solo circuito integrado, lo cual permite que los dispositivos modernos sean compactos, rápidos y altamente eficientes.

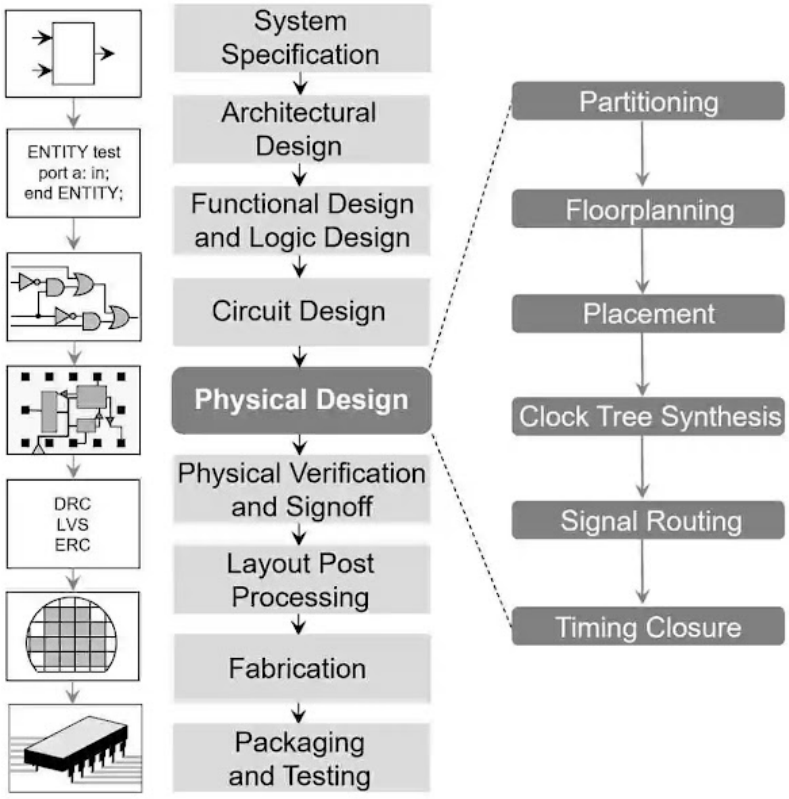

Figura 2.3. Diagrama de flujo del diseño VLSI. Fuente: [6]

Como se observa en la figura anterior, el flujo inicia con la especificación del sistema, etapa donde se definen los objetivos generales del circuito: qqué funciones debe realizar, qué rendimiento debe alcanzar, qué dimensiones físicas tendrá y en qué tecnología de fabricación será implementado. Esta especificación funciona como una hoja de ruta y establece las bases para todas las decisiones de diseño.

Posteriormente, el proceso avanza hacia el diseño arquitectónico, donde se determina la organización general del circuito. Se decide qué bloques compondrán el chip, cómo se comunicarán entre sí y qué periféricos o módulos especializados incluirá. Luego, se desarrolla el diseño funcional y lógico, que describe el comportamiento de cada bloque (RTL). El flujo continúa con el diseño de circuitos, etapa donde se ajustan y optimizan los elementos generados por la síntesis.

En el diseño físico, los componentes se ubican dentro del área del chip y se conectan mediante rutas metálicas siguiendo las reglas del proceso tecnológico. Esta etapa incluye tareas como la partición del diseño, la planificación del área (floorplanning), la colocación de celdas estándar, la distribución de

potencia y tierra, la síntesis del árbol de reloj y las diferentes fases de ruteo. El diseño físico influye directamente en el desempeño, área, consumo energético y confiabilidad del chip, por lo que requiere optimización mediante herramientas CAD especializadas.

Antes de que el circuito sea fabricado, el diseño pasa por verificación física, donde se comprueba que la implementación respeta las reglas del proceso de fabricación y que coincide funcionalmente con el diseño lógico. Las verificaciones más comunes incluyen la revisión de reglas geométricas (DRC), la comparación entre el esquema y el layout (LVS), la extracción de parasitajes y métricas eléctricas avanzadas.

Una vez aprobada la verificación, el diseño final se envía a una fundición para su fabricación mediante fotolitografía sobre obleas de silicio. Tras este proceso, los chips individuales se cortan, se encapsulan en paquetes como DIP, PGA o BGA, y pasan por etapas de pruebas eléctricas para asegurar que cumplen con las especificaciones originales.

Este flujo, ampliamente aceptado en la industria, permite transformar una descripción funcional escrita en HDL en un circuito integrado real, optimizado y confiable. Gracias a este proceso sistemático es posible fabricar chips complejos que combinan millones de transistores con un desempeño adecuado para sus aplicaciones.

2.6. Microcontrolador Siwa

El microcontrolador Siwa es un circuito integrado desarrollado con el propósito de funcionar como núcleo de control en dispositivos biomédicos implantables. Su diseño se basa en la arquitectura RISC-V, utilizando específicamente la variante RV32I, la cual define un conjunto compacto de instrucciones enteras de 32 bits, libre de licencias y adecuada para aplicaciones donde simplicidad y eficiencia energética resultan prioritarias [7, 8].

En este contexto, un microcontrolador basado en RISC-V es un circuito digital que implementa un conjunto de instrucciones estandarizado, mientras que su microarquitectura puede adaptarse libremente a las necesidades del proyecto. Esta filosofía abierta permite reducir área y consumo energético al incluir únicamente los bloques estrictamente necesarios. Como resultado, arquitecturas como RV32I resultan adecuadas para sistemas pequeños o embebidos.

La microarquitectura del Siwa se caracteriza por un diseño no segmentado (non-pipelined) y controlado mediante una máquina de estados finitos. Este enfoque reduce considerablemente tanto el área como la energía consumida, ya que evita la lógica adicional necesaria en arquitecturas segmentadas. El núcleo está compuesto por cuatro bloques principales: la Unidad de Control, el Register File, la Unidad Aritmética y Lógica (ALU) y el Decodificador de Instrucciones. Estos módulos trabajan conjuntamente para ejecutar cada instrucción de la ISA RV32I, procesar datos y gestionar saltos, comparaciones y operaciones lógicas y aritméticas [7].

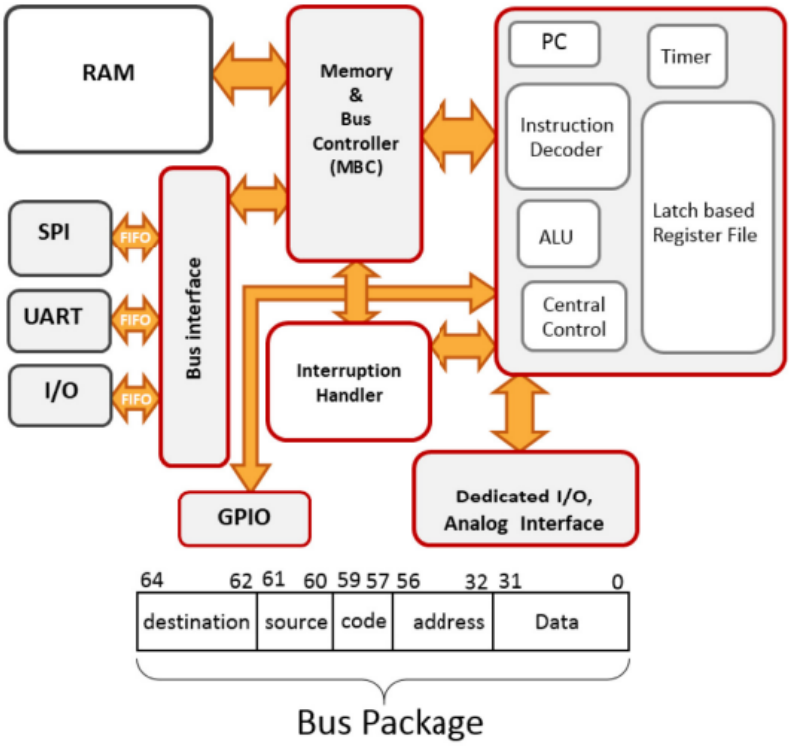

Figura 2.4. Descripción de alto nivel de Siwa. Fuente: [8]

El Register File está conformado por 32 registros de propósito general de 32 bits. En Siwa, este bloque fue implementado empleando latches en lugar de flip-flops, lo que permitió reducir su área en aproximadamente un 30 % y su consumo energético en un 25 %. Esta optimización es especialmente valiosa en dispositivos implantables, donde cada microvatio ahorrado contribuye a prolongar la vida útil del sistema.

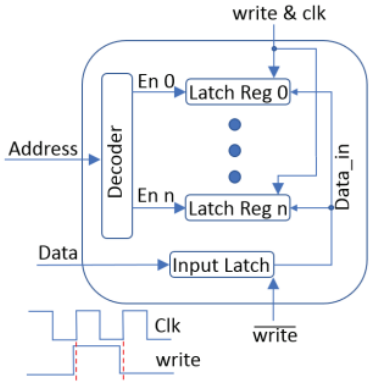

Figura 2.5. Descripción general del Register File basado en latches utilizado en Siwa. Fuente: [7]

La ALU, por su parte, realiza operaciones fundamentales como sumas, restas, desplazamientos, comparaciones y operaciones lógicas, todo bajo el control de la Unidad de Control, que determina la secuencia de acciones necesarias para completar cada instrucción [7].

Para permitir la comunicación entre la CPU y los periféricos, Siwa utiliza un bus interno de 64 bits con arbitraje distribuido. Este bus opera mediante paquetes de datos que contienen información sobre el origen, destino y tipo de mensaje, lo que permite una comunicación flexible entre los módulos del chip. El diseño basado en paquetes brinda tolerancia ante fallos en la comunicación interna y facilita la expansión o modificación futura del sistema [8].

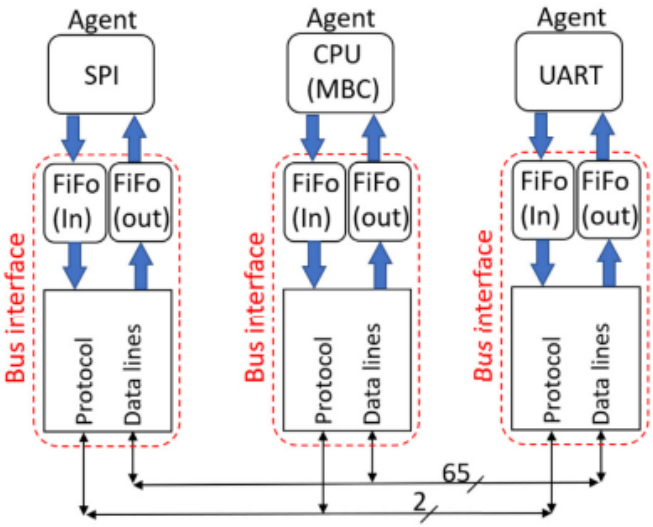

Figura 2.6. Microarquitectura del bus de datos de Siwa. Fuente: [8]

Además del núcleo de procesamiento, Siwa integra varios periféricos esenciales: un puerto UART, ocho líneas GPIO, un módulo SPI y una interfaz de estimulación para tejido biológico. Este último periférico es de relevancia particular, ya que permite realizar funciones de sensado y estimulación sin la necesidad de integrar circuitos adicionales externos, lo que contribuye a disminuir el tamaño y el consumo del sistema final.

El proceso de arranque del microcontrolador está implementado directamente en hardware. Cuando el sistema se enciende, un bloque de arranque toma control del módulo SPI para leer el firmware almacenado en una memoria Flash externa. Dicho firmware se copia a la memoria SRAM interna de 8 kB, tras lo cual inicia la ejecución del programa. Esta estrategia evita la necesidad de incluir una ROM interna de gran tamaño y reduce el área del chip. [8].

Los resultados experimentales presentados en los estudios técnicos muestran que Siwa alcanza un consumo aproximado de 48 pJ por ciclo y un consumo promedio cercano a 70 µW, cifras competitivas en comparación con otros microcontroladores utilizados en dispositivos médicos implantables. Este desempeño se atribuye a su microarquitectura simple, a la implementación optimizada del File y a la integración eficiente de sus módulos internos en un proceso CMOS de baja fuga [7].

Capítulo 3

Procedimiento metodológico

3.1. Módulo `top_spi`: señales internas y funciones auxiliares

El módulo `top_spi` constituye la unidad de control central encargada de coordinar la lectura del encabezado (header), la transferencia del firmware, las operaciones de borrado (ERASE) y el copiado interno flash–flash. Debido a la complejidad de su FSM y a la variedad de rutas de datos involucradas, esta sección presenta una descripción exhaustiva y organizada de todas las señales internas y funciones auxiliares definidas en el módulo, previo a detallar el comportamiento de la máquina de estados en las secciones siguientes.

Para mayor claridad, las señales se agrupan según su función lógica dentro del diseño.

3.1.1. Señales asociadas a la interfaz FIFO

Tabla 3.1. Descripción de señales del interfaz FIFO del módulo `top_spi`

Señal	Tipo	Descripción
`pndgn`	input	Indica que existe un comando disponible en la FIFO.
`D_pop`	input [63:0]	Palabra extraída de la FIFO, contiene comandos e información.
`pop`	output	Pulso para extraer un elemento de la FIFO.
`D_push`	output [63:0]	Palabra enviada hacia la FIFO (escrituras de SRAM o END_BST).
`push`	output	Pulso para insertar un elemento en la FIFO.

Constituyen la interfaz de comunicación con la arquitectura superior. Los comandos relevantes incluyen: `START_BST`, `WRITE_4BYTE`, `WRITE_2BYTE`, `WRITE_BYTE`, así como los comandos de borrado.

3.1.2. Señales de control del SPI Maestro

Tabla 3.2. Señales internas asociadas al IP SPI integrado

Señal	Tipo	Descripción
`tspi_data_config`	reg [15:0]	Configuración del IP SPI (modo, CPOL, CPHA, etc.).
`tspi_config_en`	reg	Pulso para aplicar una nueva configuración en el IP.
`tspi_data_tx`	reg [63:0]	Frame SPI a transmitir (opcode, dirección, dummy).
`tspi_send_data`	reg	Pulso que inicia la operación SPI.
`data_read`	wire [31:0]	Palabra recibida desde la Flash.
`end_send`	wire	Señal del IP indicando fin de transacción.
`MISO/MOSI/SCLK/SCS`	puertos	Líneas estándar del bus SPI.

Permiten construir y enviar tramas SPI tanto para el header como para el firmware y los comandos de borrado/copiado.

3.1.3. Señales para el manejo del encabezado

Tabla 3.3. Señales internas utilizadas durante la decodificación del encabezado

Señal	Tipo	Descripción
`hdr_cnt`	reg [1:0]	Fase del header (MAGIC, VERSION, SIZE, RESERVED).
`h_magic`	reg [31:0]	Campo MAGIC leído del header.
`h_version`	reg [31:0]	Campo VERSION leído.
`h_size`	reg [31:0]	Tamaño del firmware en bytes.
`h_reserved`	reg [31:0]	Campo reservado.
`dr_prev`	reg [31:0]	Palabra anterior para formar la ventana de 64 bits.
`have_prev_hdr`	reg	Indica si `dr_prev` es válido.
`align_sel`	reg [2:0]	Modo de reordenamiento de bytes (swizzle).
`hdr_off_locked`	reg [2:0]	Offset donde se encontró el MAGIC.

Realizan la identificación del MAGIC en una ventana de 64 bits, permitiendo reconstruir el header aun si su alineación o endianness cambian

3.1.4. Señales para el manejo del firmware

Tabla 3.4. Señales internas utilizadas durante la transferencia del firmware

Señal	Tipo	Descripción
`prev_word`	reg [31:0]	Historial para reconstrucción de palabras del payload.
`bytes_fw`	reg [31:0]	Total de bytes ya copiados a SRAM.
`w_adj`	reg [31:0]	Palabra alineada según `FW_OFFSET`.
`tail_byte`	reg [7:0]	Byte residual cuando SIZE no es múltiplo de 4.

Se actualizan en los estados `FW_CMD`, `FW_WAIT`, `FW_TAIL_1B` y determinan qué tipo de escritura (1, 2 o 4 bytes) debe generarse hacia la FIFO.

3.1.5. Señales para la funcionalidad de copiado Flash–Flash

Tabla 3.5. Señales internas utilizadas en la copia Flash→Flash mediante Page Program

Señal	Tipo	Descripción
`copy_active`	reg	Indica que la FSM está realizando una copia interna.
`copy_src`	reg [23:0]	Dirección origen del bloque.
`copy_dst`	reg [23:0]	Dirección destino.
`copy_bytes_left`	reg [31:0]	Bytes restantes de la operación.
`copy_word`	reg [31:0]	Palabra alineada lista para Page Program.
`cpy_prev_word`	reg [31:0]	Historial de lectura para la etapa de copia.

Usa el mismo mecanismo de lectura del firmware, pero aplica comandos `WREN` y `PP` entre lecturas.

3.1.6. Señales asociadas a operaciones de borrado (ERASE)

Tabla 3.6. Señales internas para el soporte de borrado JEDEC

Señal	Tipo	Descripción
`erase_addr`	reg [23:0]	Dirección objetivo del borrado.
`erase_opcode`	reg [7:0]	Opcode JEDEC: SE4K, BE32K, BE64K o CE.

Se utilizan en los estados `ERASE_WREN`, `ERASE_CMD` y sus respectivos estados de espera.

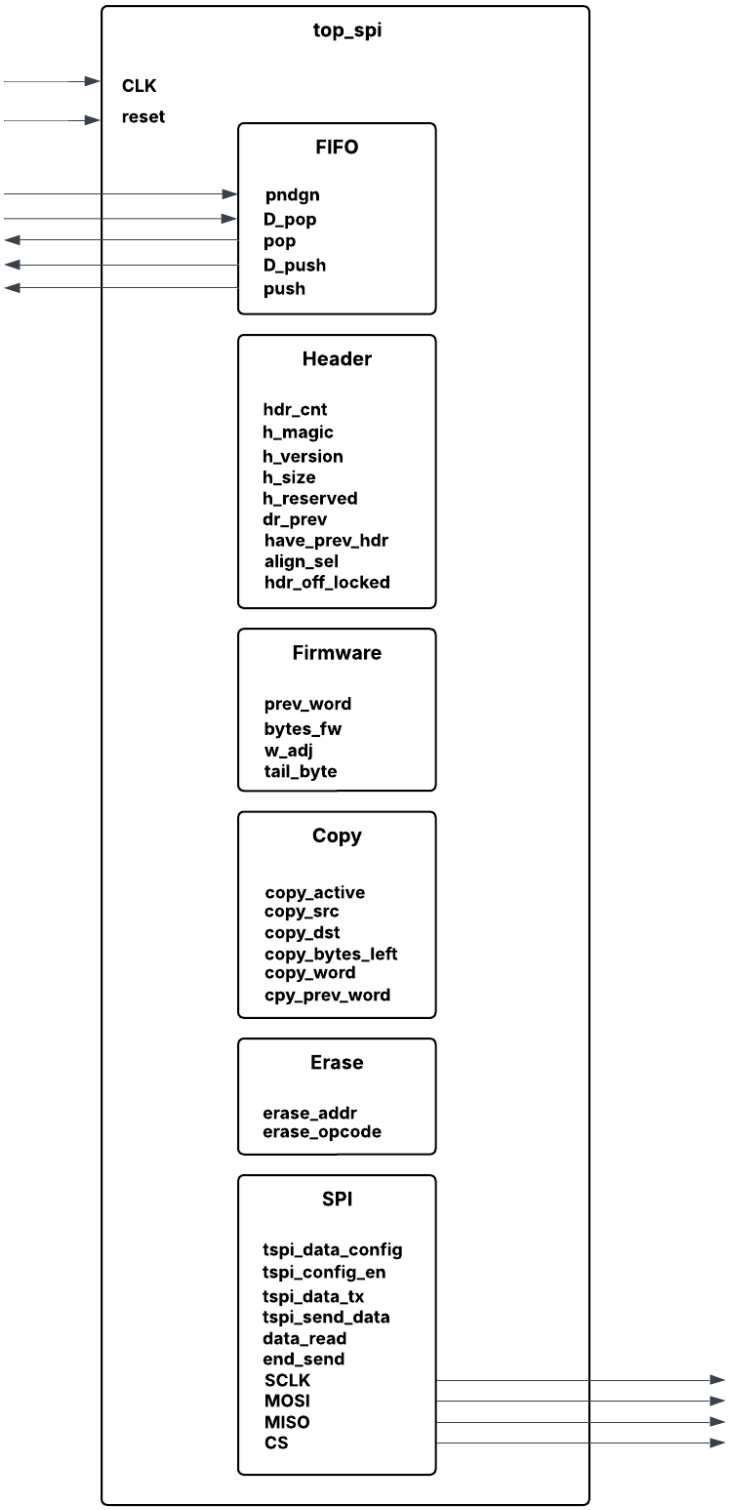

Figura 3.1. Organización interna del módulo `top_spi`.

3.1.7. Funciones internas del módulo

El módulo define tres funciones centrales que permiten manejar alineación y reconstrucción de palabras:

- `swizzle(x, sel)`: reorganiza los bytes de `x` según el modo `sel`. Es fundamental para adaptar la lectura del header a distintos endianness posibles.
- `hdr_extract32(prev, curr, off, swz)`: forma una ventana de 64 bits `{prev, curr}` y extrae una palabra en el offset dado, aplicando luego `swizzle`.
- `capture32(now, prev, cap)`: reconstruye una palabra de 32 bits cuando los datos llegan desalineados o deben combinarse entre lecturas consecutivas. Se utiliza principalmente en la fase de firmware y copiado.

Estas funciones permiten que el diseño sea tolerante a diferencias en alineamiento, desplazamientos no múltiplos de cuatro bytes y variaciones en la forma en que fue programada la flash.

3.2. Eliminación de la ambigüedad en el arranque

En el diseño original del microcontrolador, el proceso de arranque dependía de un criterio impropio para determinar el final de la transferencia del firmware desde la memoria Flash SPI: se asumía que la aparición del valor `0xFFFFFFFF` marcaba la finalización del programa. Este enfoque generaba una ambigüedad crítica, ya que dicho patrón podía aparecer de forma legítima dentro del binario del sistema, ya fuese como dato, valor por defecto o incluso como instrucción válida. Como consecuencia, el mecanismo podía interrumpir prematuramente la carga del programa, dejando secciones incompletas y comprometiendo la inicialización del sistema.

Para resolver esta deficiencia, se introdujo un encabezado estructurado (header) al inicio del firmware almacenado en la memoria Flash. Dicho encabezado incorpora la información necesaria para que el módulo SPI determine de manera determinista la extensión real del programa a cargar. El formato contempla los siguientes campos: un Magic Number para validar la imagen, una palabra SIZE que especifica la longitud total del payload en bytes, un campo de VERSION y un campo RESERVED destinado a futuras extensiones.

El módulo SPI fue rediseñado para ejecutar la siguiente secuencia durante el arranque: leer el header mediante operaciones estándar del protocolo SPI, validar la imagen mediante la comparación del *MAGIC*, interpretar el campo SIZE para determinar el número exacto de bytes que deben ser copiados y transferir a la memoria interna únicamente los bytes especificados en el encabezado.

Con este esquema, el fin de carga ya no depende del contenido del programa sino de un criterio explícito y verificable, lo que suprime totalmente la ambigüedad y establece una separación clara entre el plano de datos y el plano de control. Esto permite un arranque trazable del firmware durante la inicialización.

3.2.1. Máquina de estados del módulo `top_spi` para el manejo del encabezado

En esta sección se describe en detalle el funcionamiento de la máquina de estados finitos (FSM) del módulo `top_spi` en lo relativo al proceso de arranque y, en particular, al manejo del encabezado (*header*) de firmware almacenado en la memoria Flash SPI. Se explica cómo se inicia la lectura, cómo se reconstruyen las palabras de 32 bits a partir del flujo serie, cómo se detecta y valida el campo `MAGIC` y cómo se extraen los campos `VERSION`, `SIZE` y `RESERVED` que gobiernan el resto del proceso de carga.

Visión general de la FSM

El proceso de arranque se organiza en las siguientes etapas:

- **Estados de configuración del IP SPI:**
  `CFG_SET`, `CFG_HOLD1`, `CFG_HOLD2`, `CFG_WAIT1`, `CFG_WAIT2`.
- **Estados asociados al manejo del encabezado:**
  `HDR_SETUP`, `HDR_CMD`, `HDR_WAIT`, `HDR_VALIDATE`.
- **Estados para la transferencia del payload de firmware:**
  `FW_CMD`, `FW_WAIT`, `FW_TAIL_1B`, `FW_DONE`.
- **Estados de copiado Flash–Flash y borrado (ERASE):**
  Estas operaciones no introducen una FSM independiente, sino que reutilizan la infraestructura ya empleada en la lectura del encabezado y del firmware:
  - Las lecturas de datos durante el copiado reutilizan exactamente `FW_CMD` y `FW_WAIT`.
  - Las escrituras (`WREN` y `PP`) usan la misma lógica de transmisión que en la etapa de `HDR_CMD`.
  - ERASE utiliza el mismo patrón de configuración emitido en `CFG_SET` → `CFG_WAIT2`.

El proceso de arranque comienza cuando el módulo recibe, a través de la FIFO externa, la orden `START_BST`, que habilita la secuencia de inicialización. A partir de ese momento, el top configura el IP SPI en modo maestro y establece las condiciones necesarias para iniciar la comunicación con la memoria Flash. Una vez habilitado el enlace, el sistema procede a leer el encabezado ubicado al inicio del binario y ejecuta validaciones destinadas a confirmar su estructura y su alineación en memoria.

Después de validar el encabezado, incluyendo la detección del `MAGIC`; la interpretación del formato y la obtención del campo `SIZE`, el módulo queda en condiciones de realizar la transferencia del firmware. El valor de `SIZE` determina cuántos bytes deben copiarse hacia la memoria interna, reemplazando así el antiguo mecanismo basado en patrones de terminación y eliminando cualquier ambigüedad asociada a datos del programa que pudieran confundirse con un fin de transmisión.

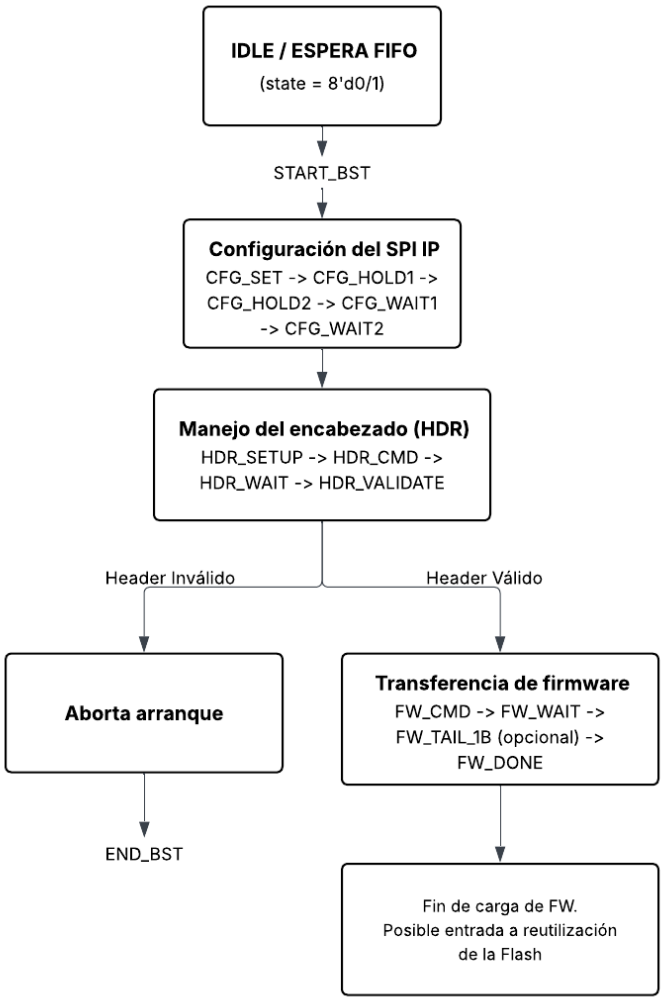

Figura 3.2. Flujo general de la FSM del SPI.

Variables y funciones auxiliares para el encabezado

El proceso de detección y decodificación del encabezado utiliza un conjunto especializado de registros internos y funciones auxiliares que permiten reconstruir palabras de 32 bits a partir de tramas SPI y compensar variaciones de endianness o alineamiento.

Señales asociadas al encabezado:

- `h_magic`, `h_version`, `h_size`, `h_reserved`: Registros de 32 bits donde se almacenan los cuatro campos principales del encabezado.
- `hdr_cnt`: Contador de fase para el proceso de lectura del header:
  - `hdr_cnt = 0`: fase especial para detectar `MAGIC` y extraer `VERSION`.
  - `hdr_cnt = 2`: lectura de `SIZE` y `hdr_cnt = 3`: lectura de `RESERVED`.
- `dr_prev`: Registra la palabra SPI previa, permitiendo construir una ventana de 64 bits al concatenarla con la palabra actual.
- `win_hdr`: Ventana de 64 bits que permite evaluar las ocho posiciones posibles en las que podría estar alineado el `MAGIC`.
- `align_sel` (`swz_t`): Selector del esquema de reordenamiento de bytes (*swizzle*) que permitirá interpretar todo el encabezado con la misma convención.
- `hdr_off_locked`: Desplazamiento en bytes (0–7) dentro de la ventana donde se encontró el `MAGIC`. Garantiza consistencia en la extracción de `VERSION`.
- `have_prev_hdr`: Indica si ya se ha almacenado la primera palabra del header para formar la ventana inicial de 8 bytes.

Funciones asociadas al encabezado:

- `swizzle(x, sel)`: Reordena los bytes de `x` según el modo `sel`. Cada variante compensa una posible forma en que los datos hayan sido programados en la Flash (big-endian, little-endian, rotaciones de 8 o 24 bits, combinaciones con byte-swap).
- `hdr_extract32(prev_w, curr_w, off, swz)`: Construye la ventana de 64 bits {prev_w, curr_w}, extrae 32 bits a partir del desplazamiento `off` y luego aplica `swizzle`. Esta función permite detectar el `MAGIC` incluso si los datos no están alineados a 32 bits, si el encabezado cruza una frontera de palabra, o si la Flash fue programada en big-endian, little-endian o con un empaquetado no estándar.

Gracias a este conjunto de mecanismos, el módulo no depende de ningún supuesto rígido sobre cómo fueron escritos los bytes en la Flash.

Estados CFG

A continuación se muestra el diagrama de flujo para los estados de configuración del IP SPI.

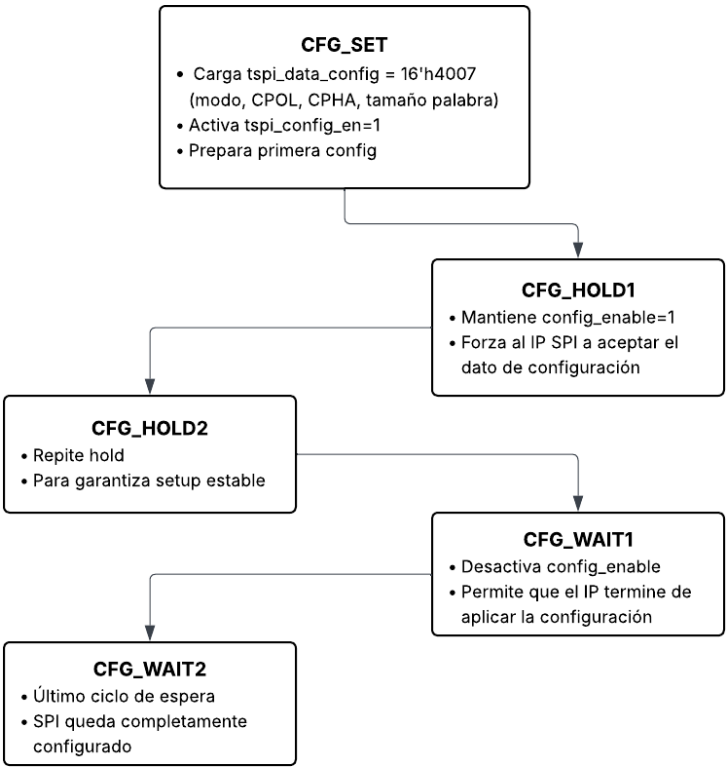

Figura 3.3. Flujo para los estados de configuración del IP SPI.

Estado `CFG_SET`: carga de la configuración del IP SPI En este estado, `top_spi` inicializa la configuración del IP SPI maestro. Para ello, se carga en el registro `tspi_data_config` la palabra `16'h4007`, que define parámetros como el modo de operación, la polaridad y fase del reloj (CPOL/CPHA) y el tamaño de palabra utilizado en las transacciones SPI. A la vez, la FSM activa la señal `tspi_config_en`, indicando al bloque SPI que la trama de configuración presente en `tspi_data_config` es válida y debe ser aplicada.

Estados `CFG_HOLD1` y `CFG_HOLD2`: mantenimiento de la configuración La FSM activa la señal `tspi_config_en` durante varios ciclos de reloj consecutivos. El valor previo no se modifica, de modo que el IP SPI dispone de tiempo suficiente para muestrear y almacenar de manera estable los parámetros de configuración. Este esquema responde a los requisitos específicos del bloque SPI instanciado, el cual espera una ventana mínima de ciclos con la señal de habilitación activa. Tras completar estos ciclos de retención, la FSM desactiva la configuración.

Estados `CFG_WAIT1` y `CFG_WAIT2`: estabilización posterior a la configuración Aquí `tspi_config_en` permanece desactivada, y no se realizan nuevas escrituras sobre `tspi_data_config`. El objetivo de esta fase es proporcionar un margen de reloj para que el SPI complete internamente cualquier proceso asociado a la nueva configuración, evitando así posibles conflictos entre la etapa de configuración y los siguientes estados. Al finalizar, la máquina de estados da por concluida la configuración del SPI y transfiere el control al estado `HDR_SETUP`, donde comienza el manejo del encabezado del firmware almacenado en la memoria Flash.

Estados HDR

El diagrama de flujo de los estados encargados de procesar el encabezado del firmware es:

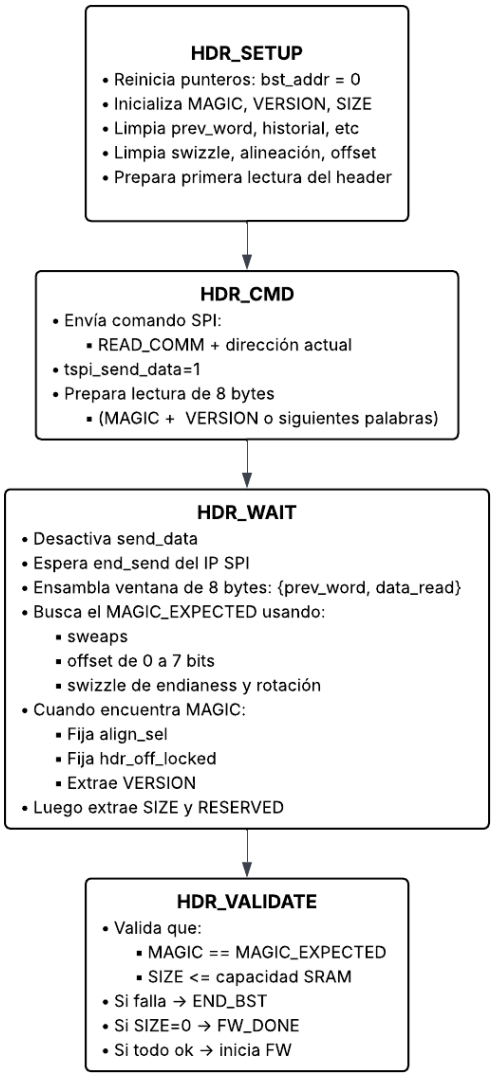

Figura 3.4. Flujo de los estados encargados de procesar el encabezado del firmware.

Estado `HDR_SETUP`: inicialización del encabezado Cuando la FSM entra en `HDR_SETUP`, se limpia toda la estructura interna asociada al header:

- `bst_addr` se fija en `24'h000000`.
- `bytes_fw` se reinicia a cero.
- `hdr_cnt` se reinicia a `0`.
- `h_magic`, `h_version`, `h_size`, `h_reserved` se limpian.
- `align_sel` se ajusta a `SWZ_ID`, como modo base.
- `hdr_off_locked` se reinicia.
- `have_prev_hdr` se limpia para forzar la captura de la primera palabra del header.

El estado deja al sistema preparado para iniciar las lecturas reales del encabezado, pasando a `HDR_CMD`.

Estado `HDR_CMD`: emisión del comando de lectura SPI En este estado, el maestro SPI arma la trama: {`READ_COMM`, `bst_addr`, 32'hFFFF_FFFF} y activa la señal `tspi_send_data`. Al terminar la emisión del comando, la FSM se mueve a `HDR_WAIT`.

Estado `HDR_WAIT`: reconstrucción y detección del `MAGIC` Este es el estado más complejo. Aquí se espera a que termine la transacción SPI (`end_send`); se registre la palabra recibida (`data_read`); se busca el `MAGIC` probando ocho desplazamientos y múltiples modos de `swizzle`; se extrae `VERSION` dentro de la misma ventana y posteriormente lee `SIZE` y `RESERVED`. El comportamiento exacto depende de `hdr_cnt`, tal que:

Fase `hdr_cnt = 0`: búsqueda del `MAGIC` y obtención de `VERSION` .

En la primera iteración con `hdr_cnt = 0`:

1. Cuando `have_prev_hdr` = 1, la FSM ya dispone de dos palabras consecutivas (`dr_prev` y `data_read`). A partir de ellas se forma la ventana `win_hdr` ← `dr_prev, data_read`, de 64 bits.

   Sobre esta ventana se realiza una búsqueda del `MAGIC`. El procedimiento es el siguiente:

- Se recorren todos los desplazamientos posibles de 0 a 7 bytes.
- Para cada desplazamiento `off_i`:
  a) Se seleccionan los 32 bits correspondientes (`slice32`).
  b) Se prueban todas las variantes de `swizzle` (`SWZ_ID`, `SWZ_BSWAP`, `SWZ_ROT8`, etc.).
  c) Si alguna combinación `slice32_swz` coincide con `MAGIC_EXPECTED`, entonces:
    - `best_swz` se fija en el swizzle que produjo la coincidencia,
    - `best_off` toma el valor de `off_i`,
    - `found_local` se marca en 1.

Si después de explorar todos los offsets y swizzles no se encuentra el `MAGIC`, el contenido de la Flash se considera inválido. La FSM entonces genera un `END_BST` hacia la FIFO, limpia los registros internos usados para el análisis del encabezado y retorna al estado `8'd1`, quedando a la espera de nuevas órdenes.

En caso de hallarse el `MAGIC`, se fija de inmediato la alineación del encabezado:

- `swz_hdr` = `best_swz` (o el valor forzado por `USE_FIXED_ALIGN`, si aplica),
- `align_sel` = `swz_hdr`,
- `hdr_off_locked` = `best_off`.

Una vez fijado el alineamiento, se procede a extraer los campos del encabezado que también residen dentro de la misma ventana de 64 bits:

- Se obtiene el `MAGIC` consolidado mediante: `h_magic` ← `hdr_extract32(dr_prev, data_read, best_off, swz_hdr)`.
- Se calcula el offset de `VERSION` como: `off_ver` ← (`best_off` + 4) & 3'b111, aprovechando que el header coloca `MAGIC` y `VERSION` en palabras consecutivas dentro de los mismos 8 bytes.
- Se extrae `h_version`: `h_version` ← `hdr_extract32(dr_prev, data_read, off_ver, swz_hdr)`.

Con `MAGIC` y `VERSION` ya capturados, la FSM establece `hdr_cnt = 2` y avanza `bst_addr` en 4, de modo que la siguiente lectura (`HDR_CMD`) corresponda al campo `SIZE`.

Fase `hdr_cnt = 2, 3`: lectura de `SIZE` y `RESERVED` .

Una vez superada la fase de detección del `MAGIC`, el resto del encabezado se interpreta de forma más directa. Para `hdr_cnt = 2` y `hdr_cnt = 3`:

- La palabra `data_read` recibida se pasa por la función `swizzle` con el `align_sel` ya fijado `w_raw_hdr` ← `swizzle(data_read, align_sel)`.
- Si `hdr_cnt = 2`, se asume que `w_raw_hdr` corresponde al campo `SIZE`: texttth_size ← `w_raw_hdr`.
- Si `hdr_cnt = 3`, se captura el campo `RESERVED`: `h_reserved` ← `w_raw_hdr`.

Después de leer `SIZE`, la FSM incrementa `hdr_cnt` y `bst_addr` para avanzar al siguiente campo. Una vez que `hdr_cnt = 3` ha sido procesado, la FSM pasa al estado `HDR_VALIDATE`.

Estado `HDR_VALIDATE`: validación del encabezado En `HDR_VALIDATE` se realiza la comprobación final de coherencia del header. S verifica que `h_magic` coincida con el valor esperado `MAGIC_EXPECTED`, se comprueba que `h_size` no exceda la capacidad de la memoria interna, comparándolo contra `SRAM_CAP_BYTES`.

Las posibles ramas son:

1. Header inválido: Sucede cuando (`h_magic` es distinto a `MAGIC_EXPECTED` o `h_size` mayor que `SRAM_CAP_BYTES`), entonces se genera un comando `END_BST` hacia la FIFO y a FSM vuelve a un estado de espera (`8'd1`), sin transferir ningún byte de firmware.
2. Header válido con `SIZE = 0`: No hay payload que transferir. Se pasa directamente a `FW_DONE`, donde se emite `END_BST` (y eventualmente se entra a los modos de copia o postarranque, según corresponda).
3. Header válido con `SIZE >0`: Se fija `bst_addr` en `FW_OFFSET`, que es la dirección a partir de la cual comienza el payload de firmware real, se resetean los historiales de palabra (`prev_word`, `cpy_prev_word`) usados para la captura del payload. Si el modo de captura no está "congelado" (`cap_locked = 0`), se inicializa `cap_sel` a `CAP_NOW`. La FSM pasa al estado `FW_CMD`, iniciando la fase de transferencia del firmware propiamente dicho.

3.2.2. Relación del encabezado con la transferencia de firmware

Aunque el foco de esta sección es el header, es relevante mencionar brevemente cómo la información extraída se utiliza en la fase de transferencia de firmware (`FW_CMD`, `FW_WAIT`, `FW_TAIL_1B`).

El campo `SIZE` (`h_size`) se interpreta como el número total de bytes a copiar desde la Flash hacia la memoria interna. La FSM mantiene un contador. De ahí, en cada palabra recibida durante la fase de

firmware, el módulo aplica primero la función `capture32` (para reconstruir palabras a partir de pares de lecturas, si es necesario) y luego `swizzle` usando el `align_sel` fijado durante la detección del header. La FSM genera comandos de escritura a la SRAM (modelados como `WRITE_4BYTE`, `WRITE_2BYTE`, `WRITE_BYTE` en la interfaz FIFO), cuidando que la suma `bytes_fw` no exceda nunca el `h_size` previamente validado.

De esta manera, el encabezado actúa como única fuente de verdad sobre la cantidad de información a transferir. La FSM ya no depende de patrones de datos dentro del payload (como `0xFFFFFFFF`) para concluir la carga del programa, sino de un límite explícito derivado del campo `SIZE` y validado mediante el `MAGIC`. Esto elimina la ambigüedad de versiones previas y permite un arranque determinista y verificable.

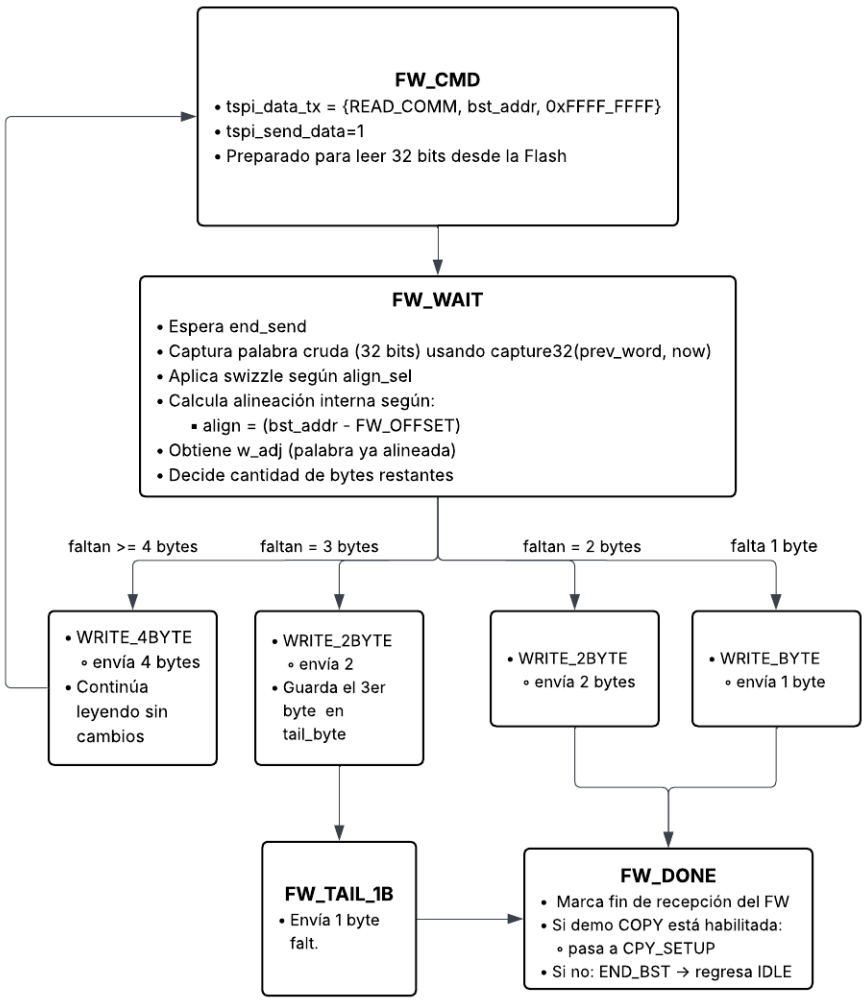

Figura 3.5. Flujo sobre la relación del encabezado con la transferencia de firmware.

3.3. Reutilización de la memoria flash como almacenamiento secundario

El diseño previo del módulo SPI únicamente contemplaba el uso de la memoria Flash durante el arranque, sin embargo, presentaba impedimentos para su utilización como memoria secundaria.

El rediseño introduce una transición explícita entre dos modos de operación:

- **Modo de arranque:** El módulo SPI se restringe a leer el encabezado y transferir el programa hacia la memoria interna del sistema. Únicamente se permite la lectura secuencial del firmware.
- **Modo operativo:** Una vez completada la carga del programa, el módulo SPI habilita el conjunto completo de comandos JEDEC estándar, permitiendo:
  - Lecturas: `READ`, `FASTREAD`.
  - Escrituras: `WREN` (Write Enable), `PP` (Page Program).
  - Borrados: `SE4K`, `BE32K`, `BE64K`, `CE`.

La activación del modo operativo se controla internamente a partir de la bandera generada cuando se completa la transferencia del firmware. De esta forma, la memoria Flash queda disponible como unidad de almacenamiento persistente sin requerir modificaciones externas ni cambios en otros módulos del microcontrolador.

Este mecanismo habilita la reutilización total de la memoria tras el arranque, ampliando significativamente la funcionalidad del sistema y permitiendo la implementación de registros, configuraciones o almacenamiento general no volátil.

3.3.1. Máquina de estados y mecanismos internos para la reutilización de la Flash

En el diseño se habilita el uso de la memoria Flash SPI como almacenamiento secundario una vez finalizada la etapa de arranque. Esto se materializa en dos formas de manejo de la memoria no volátil: mediante una rutina de copiado Flash–Flash que se ejecuta inmediatamente después de completar la transferencia de firmware (`FW_DONE`), siempre que `DO_COPY_AFTER_BOOT = 1` y el tamaño de firmware `h_size` sea distinto de cero, utilizando los comandos estándar JEDEC de escritura mediante página (`WREN` + `PP`) para replicar el firmware ya cargado en una región distinta de la misma memoria Flash y, la otra, por medio de un conjunto de comandos de borrado por bloques (`SE4K`, `BE32K`, `BE64K`, `CE`) que permiten al sistema invalidar regiones completas de la Flash y prepararlas para nuevos datos.

Ambos mecanismos reutilizan la misma infraestructura de transmisión SPI ya empleada en el arranque (comandos `READ` + burst de datos), pero ahora orientada explícitamente a soportar operaciones típicas de memoria secundaria: lectura, escritura y borrado de bloques de información no volátil.

En particular, el parámetro DO_COPY_AFTER_BOOT controla si, tras completar la carga del firmware desde la Flash hacia la SRAM interna, el módulo ejecuta automáticamente una copia del binario hacia otra dirección dentro de la Flash (base COPY_DST_BASE). Este flujo sirve como demostrador de que el módulo puede operar sobre la memoria ya no sólo como fuente de código de arranque, sino también como destino de escritura persistente.

De forma complementaria, el soporte para comandos de borrado (SE4K, BE32K, BE64K, CE) permite limpiar sectores, bloques o el chip completo, lo que constituye la operación fundamental previa a cualquier reprogramación de datos en memorias Flash NOR.

A diferencia de la fase de encabezado, donde la FSM opera en torno a los estados HDR_SETUP, HDR_CMD, HDR_WAIT y HDR_VALIDATE, en el caso de la reutilización de la Flash intervienen dos grupos específicos de estados:

- **Estados de copiado Flash–Flash**:
  - FW_DONE: punto de entrada al flujo de copia tras finalizar la carga de firmware.
  - CPY_SETUP: configuración inicial de los punteros y longitud de copia.
  - CPY_READ_CMD, CPY_READ_WAIT: lectura del bloque origen mediante READ_COMM.
  - CPY_WREN, CPY_WREN_WAIT: habilitación de escritura en la Flash mediante WREN_COMM.
  - CPY_PP, CPY_PP_WAIT: programación de una palabra de 32 bits en la dirección destino usando PP_COMM.
  - CPY_NEXT: actualización de punteros y conteo de bytes pendientes.
  - CPY_DONE: finalización de la copia y notificación a la FIFO mediante END_BST.
- **Estados de borrado (ERASE)**:
  - ERASE_WREN, ERASE_WREN_WAIT: preparación de la Flash para recibir un comando de borrado.
  - ERASE_CMD, ERASE_CMD_WAIT: emisión del opcode de borrado (SE4K_COMM, BE32K_COMM, BE64K_COMM, CE_COMM) junto con la dirección afectada.
  - ERASE_DONE: retorno al estado de espera global sin afectar el flujo de boot ni de copia.

Señales internas para la copia y el borrado .

El flujo de reutilización de la Flash se apoya en un conjunto específico de señales internas:

- **Señales de copia Flash–Flash**:
  - copy_active: indica que hay una operación de copia en curso. Mientras sea 1, el estado 8'd1 ignora nuevos comandos START_BST procedentes de la FIFO, evitando solapar arranques adicionales durante una operación de mantenimiento de la Flash.

- `copy_src`, `copy_dst`: punteros de 24 bits con la dirección origen y destino de la copia dentro de la Flash. El origen se inicializa en `FW_OFFSET` (comienzo del payload de firmware) y el destino en `COPY_DST_BASE`.
- `copy_bytes_left`: contador de 32 bits con el número de bytes aún pendientes por copiar. Se inicializa con `h_size` truncado al múltiplo de 4 inmediatamente inferior, garantizando que la copia se realiza siempre en palabras completas de 32 bits.
- `copy_word`: registro de 32 bits que almacena la palabra ya alineada y "swizzleada" que será programada en la región destino mediante `PP_COMM`.
- `cpy_prev_word`: historial separado del flujo `prev_word` usado para el firmware principal. Permite reutilizar la función `capture32` en la fase de copia sin interferir con el registro de historial del canal de arranque.

- **Parámetros de configuración**:
  - `DO_COPY_AFTER_BOOT`: bit de configuración que habilita o deshabilita el flujo de copia una vez completada la etapa de arranque.
  - `COPY_DST_BASE`: dirección base de 24 bits a partir de la cual se almacenará la copia del firmware dentro de la misma Flash.
  - `FW_OFFSET`: dirección base de inicio del payload de firmware en la Flash, usada tanto en el arranque como en la copia.
- **Señales de borrado**:
  - `erase_addr`: dirección de 24 bits sobre la que se aplica el comando de borrado. Se deriva directamente del campo de dirección de la palabra recibida desde la FIFO en el estado `8'd2`.
  - `erase_opcode`: opcode JEDEC de 8 bits que codifica el tipo de borrado solicitado. Se selecciona mediante un `case` en función del índice de ERASE recibido (`ERASE_SE4K`, `ERASE_BE32K`, `ERASE_BE64K`, `ERASE_CHIP`).

Flujo de copiado Flash–Flash .

El flujo de copiado comienza en el estado `FW_DONE`, que actúa como transición entre la fase de arranque y la fase de reutilización. Si `DO_COPY_AFTER_BOOT = 1` y `h_size` es distinto de cero, el módulo no genera todavía el comando `END_BST`. En su lugar activa la señal `copy_active` y transfiere el control al estado `CPY_SETUP`. En cambio, si la copia está deshabilitada o el tamaño es cero, `FW_DONE` se comporta como en un boot tradicional: se emite `END_BST` y se retorna al estado `8'd1`.

En el estado `CPY_SETUP` se prepara la operación de copia Flash–Flash. Lo primero que hace la FSM es calcular el valor `trunc_bytes = h_size[31:2], 2'b00`, que corresponde al tamaño del firmware redondeado hacia abajo al múltiplo de cuatro más cercano. A continuación se inicializan los punteros: `copy_src` se fija en `FW_OFFSET`, indicando la dirección del payload dentro de la Flash, mientras que `copy_dst` se establece en `COPY_DST_BASE`, que es la dirección de destino donde se

almacenará la copia. El contador copy_bytes_left se carga con el valor de trunc_bytes, y cpy_prev_word se limpia a cero para iniciar el historial de captura sin residuos previos.

Si el valor de trunc_bytes resulta ser cero, no existe ninguna palabra completa que copiar, por lo que la máquina avanza directamente hacia CPY_DONE. De lo contrario, comienza el ciclo principal de copiado pasando al estado CPY_READ_CMD, donde se emite la primera lectura SPI del bloque origen.

En CPY_READ_CMD se emite un comando de lectura: tspi_data_tx ← {READ_COMM, copy_src, 32'hFFFF_FFFF} con el que se solicita una palabra de 32 bits desde dirección actual copy_src.

En el estado CPY_READ_WAIT, la FSM permanece detenida hasta que se cumpla que la línea SCS haya sido observada en nivel bajo al menos una vez, lo que confirma la participación efectiva del dispositivo Flash en la transacción, y que el IP SPI indique end_send = 1, señalando que la operación de lectura ha finalizado. Una vez atendidas estas condiciones, la palabra data_read obtenida desde la Flash se somete a una secuencia de reconstrucción y alineación. Primero, el sistema emplea la función capture32 para combinar la palabra recién recibida con cpy_prev_word, formando así una palabra "cruda" w_raw_fw_cpy capaz de manejar adecuadamente situaciones en que los datos cruzan fronteras de palabra.

Posteriormente, esta palabra reconstruida se somete al mismo esquema de reordenamiento de bytes (swizzle) que fue determinado durante la etapa de análisis del encabezado; dicho reordenamiento puede emplear align_sel o, si el sistema está configurado en modo fijo, SWZ_FORCE. El resultado de esta operación es la palabra w_now_cpy. A continuación, la lógica ajusta la alineación en función de la posición relativa del puntero de lectura respecto del inicio del payload, calculando un desplazamiento align_cpy = (copy_src - FW_OFFSET) & 2'b11. Este valor determina cómo debe reacomodarse la palabra para producir w_adj_cpy, de modo que los datos queden perfectamente alineados antes de ser escritos en la región de destino.

Finalmente, la palabra ajustada w_adj_cpy se almacena en copy_word, que será programada en la memoria Flash destino durante la siguiente etapa de la FSM. De esta forma, cada lectura realizada durante la operación de copiado se normaliza, se alinea y se prepara cuidadosamente para garantizar una escritura coherente y libre de desplazamientos indeseados.

Una vez preparado copy_word, la FSM pasa a CPY_WREN, donde se envía el comando de habilitación de escritura tspi_data_tx ← { WREN_COMM, 56'hFF_FF_FF_FF_FF_FF_FF }

Tras completar esta transacción en CPY_WREN_WAIT, se entra en CPY_PP, donde se programa la palabra en la ruta destino tspi_data_tx ← { PP_COMM, copy_dst, copy_word }

Completada la operación de página en CPY_PP_WAIT, la FSM pasa al estado CPY_NEXT, donde:

- Se calcula el nuevo valor de copy_bytes_left:

$$\texttt{next\_left} = \begin{cases} \texttt{copy\_bytes\_left} - 4, & \text{si } \texttt{copy\_bytes\_left} > 4 \\ 0, & \text{en caso contrario} \end{cases}$$

- Se incrementan `copy_src` y `copy_dst` en 4 bytes y se actualiza `copy_bytes_left` con `next_left`.

Si `next_left = 0`, la FSM transita a `CPY_DONE`; de lo contrario, vuelve a `CPY_READ_CMD` para procesar el siguiente bloque de 32 bits. El estado `CPY_DONE` marca la finalización de la operación de copia entonces se emite un comando `END_BST` hacia la FIFO, notificando que tanto el arranque como la copia Flash–Flash han concluido, se desactiva `copy_active`, permitiendo que el sistema vuelva a aceptar nuevos comandos `START_BST` en `8'd1` y la FSM retorna a su bucle principal de espera.

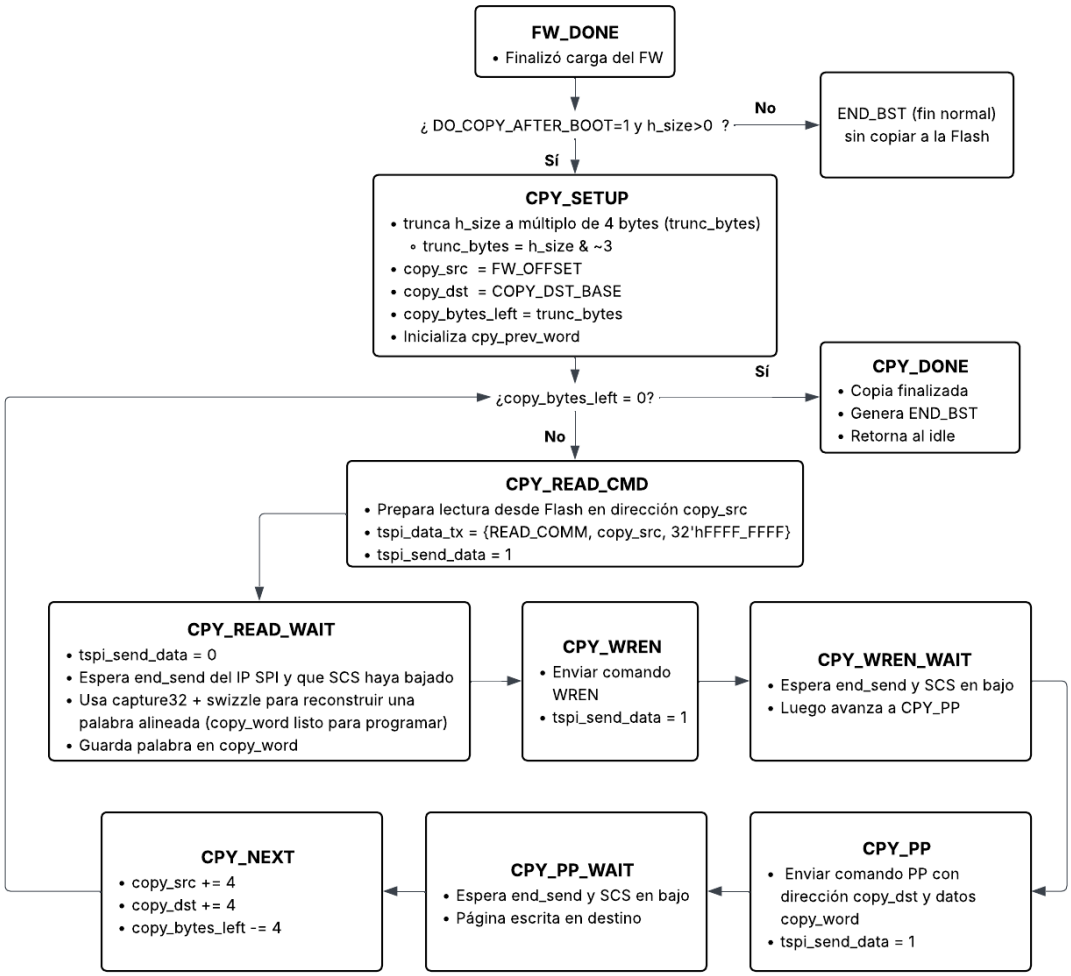

Figura 3.6. Flujo del proceso de copia Flash–Flash.

Flujo de borrado (ERASE) mediante comandos JEDEC .

El segundo pilar de la reutilización de la Flash como memoria secundaria es el soporte explícito de operaciones de borrado. Estas se disparan desde el estado `8'd2`, cuando el módulo decodifica palabras de control provenientes de la FIFO.

Si el campo `DESTINO` indica `SPI` y el campo `INDEX` coincide con alguno de los índices de borrado (`ERASE_SE4K`, `ERASE_BE32K`, `ERASE_BE64K`, `ERASE_CHIP`), la FSM interpreta la palabra como una solicitud de ERASE.

En ese caso, la dirección de borrado `erase_addr` se toma directamente del campo `ADDR` recibido desde la FIFO. A partir del índice asociado al comando, el módulo selecciona el opcode adecuado para la operación: si el índice corresponde a un borrado de 4 KiB se utiliza `SE4K_COMM`; si indica un borrado de 32 KiB, se selecciona `BE32K_COMM`; para 64 KiB se emplea `BE64K_COMM`; y si se solicita un borrado completo del chip, se elige `CE_COMM`. Una vez determinados tanto la dirección como el comando de borrado, la máquina de estados avanza hacia `ERASE_WREN`, desde donde inicia la secuencia de habilitación de escritura necesaria para ejecutar cualquier operación de borrado en la memoria Flash.

A partir de aquí, la secuencia sigue el patrón JEDEC estándar:

1. En `ERASE_WREN` se emite el comando de habilitación de escritura: `tspi_data_tx` ← { `WREN_COMM`, 56'hFF_FF_FF_FF_FF_FF_FF } y se pasa a `ERASE_WREN_WAIT` hasta que la transacción concluya.

2. En `ERASE_CMD` se construye el frame de borrado `tspi_data_tx` ← { `erase_opcode`, `erase_addr`, 32'hFFFF_FFFF } que combina el opcode de borrado con una dirección de 24 bits. Para comandos de chip erase (`CE_COMM`) la dirección es ignorada por el modelo de Flash, pero se envía igualmente para mantener un formato uniforme.

3. En `ERASE_CMD_WAIT` se espera a que la operación se complete en el interfaz SPI.

4. Finalmente, `ERASE_DONE` registra en simulación el borrado y devuelve el control al estado `8'd1`. A diferencia del flujo de arranque o de copia, aquí no se genera un `END_BST`, ya que el testbench monitoriza directamente la memoria del stub para verificar el efecto del borrado.

Estos comandos, combinados con las capacidades de lectura ya presentes en el módulo, permiten gestionar la Flash con las tres operaciones fundamentales de una memoria secundaria: leer, escribir y borrar, sin introducir dependencias adicionales sobre otros bloques del sistema. La FSM del `top_spi` encapsula así, dentro de un único módulo, tanto el arranque desde Flash como las funciones de mantenimiento y reutilización de la misma memoria como almacenamiento persistente posterior.

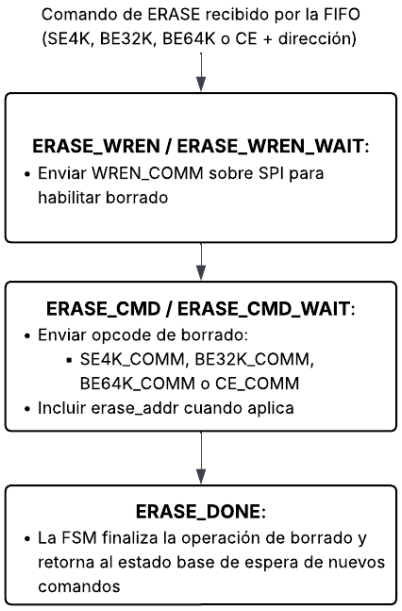

Figura 3.7. Flujo del proceso de borrado.

3.4. Testbench

Con el fin de validar el rediseño del módulo `top_spi`, se desarrolló un testbench que reproduce tanto la secuencia completa de arranque basada en encabezado como las operaciones de programación y borrado disponibles en el modo operativo. Este entorno de verificación está compuesto por dos elementos principales: el testbench como tal y un modelo comportamental de memoria Flash SPI (`spi_flash_stub`), el cual permite reconstruir un entorno de interacción al que enfrentaría el módulo en un sistema real.

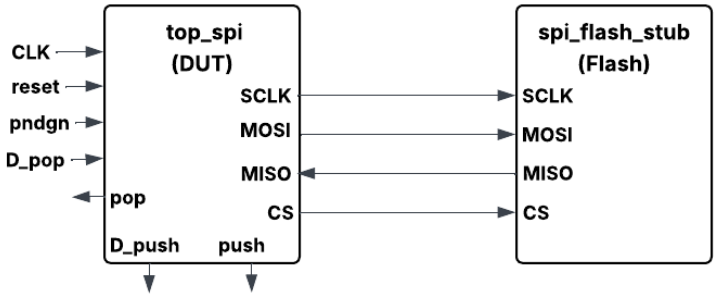

Figura 3.8. Conexión general del Testbench.

La verificación se estructuró en tres ejes fundamentales: flujo de arranque y manejo del encabezado, operaciones de programación mediante Page Program y borrados sectoriales y globales. Cada aspecto se valida mediante la observación de las transacciones SPI, del tráfico a la interfaz FIFO del sistema y del estado interno de la memoria modelada.

3.4.1. Verificación del flujo de arranque

El testbench genera distintos escenarios de encabezado y contenido para la memoria Flash, evaluando el mecanismo de arranque. Las variantes probadas incluyen:

- Valores `MAGIC` válidos e inválidos.
- Tamaños `SIZE` alineados y no alineados a 4 bytes.
- Casos límite: `SIZE = 0`, `SIZE` muy pequeño, `SIZE` mayor que la capacidad esperada.
- Versiones arbitrarias y payloads con patrones progresivos.

Para cada caso, se verificaron de manera las siguientes condiciones funcionales:

- La lógica de detección del `MAGIC` responde correctamente, abortando el arranque si el valor no coincide con el esperado.
- El número exacto de bytes indicado por `SIZE` es transferido al sistema mediante comandos `WRITE_4BYTE`, `WRITE_2BYTE` o `WRITE_BYTE`.
- No se produce ningún efecto colateral en casos con encabezado inválido.
- El comando `END_BST` es generado en el ciclo preciso que corresponde a la última transferencia esperada.
- Los bytes reconstruidos y enviados al sistema mantienen el orden correcto según la alineación detectada y el modo *swizzle*.

La combinación de estas verificaciones garantiza que la lógica de arranque no es ambigua bajo todas las combinaciones de encabezado previstas.

3.4.2. Verificación de programación (Page Program)

Una vez finalizada la etapa de arranque, el módulo `top_spi` habilita el conjunto de comandos de escritura del protocolo JEDEC. El testbench captura y analiza cada palabra enviada mediante el comando `PP` (*Page Program*) para verificar lo siguiente:

- Las direcciones enviadas en cada operación coinciden exactamente con las direcciones esperadas para la copia Flash–Flash.
- El orden de bytes en el bus SPI respeta la convención MSB-first utilizada por las memorias comerciales.
- No se exceden los límites de página de 256 bytes definidos por el estándar.
- La secuencia `WREN + PP` se replica de forma consistente para cada palabra programada.

Además de monitorear las transacciones, el testbench compara la memoria interna del *stub* con el historial de escrituras registradas para confirmar que la operación de programa sea funcionalmente correcta.

3.4.3. Verificación de operaciones de borrado

El diseño rediseñado también soporta los comandos estándar de borrado de Flash. Para comprobar su integridad, el testbench ejecuta los siguientes casos:

- `SE4K`: borrado de un sector de 4 KiB.
- `BE32K`: borrado de un bloque de 32 KiB.
- `BE64K`: borrado de un bloque de 64 KiB.
- `CE`: borrado completo del chip.

Previo a cada prueba, el testbench precarga las regiones afectadas con patrones distintos de `0xFF`, permitiendo validar:

- Que el módulo SPI emite correctamente la secuencia `WREN + ERASE`.
- Que la dirección enviada coincide con la base del sector o bloque a borrar.
- Que el *stub* restaura exactamente las regiones asociadas al borrado a `0xFF`.

Estas pruebas aseguran la funcionalidad completa del módulo en operaciones de mantenimiento y reprogramación de la memoria Flash.

3.4.4. Stub de memoria Flash SPI

Para reproducir adecuadamente el comportamiento de una memoria Flash real, se desarrolló un módulo comportamental denominado `spi_flash_stub`.

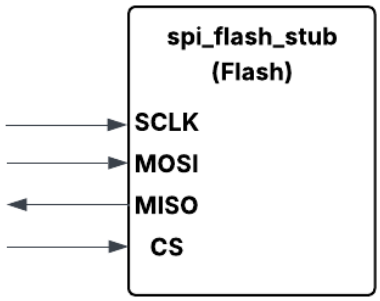

Figura 3.9. Módulo spi_flash_stub.

Este modelo se comporta como un dispositivo JEDEC típico, simulando tanto las transacciones SPI como el efecto real de cada comando sobre una matriz interna de memoria.

A diferencia de un generador simple de datos, el stub implementa:

- Una matriz de memoria interna de tamaño configurable (`MEM_BYTES`), inicializada completamente a `0xFF`.
- Un esclavo SPI compatible con modos 0 y 3, con soporte MSB-first y opción de operación LSB-first mediante parámetro.
- Decodificación completa de comandos JEDEC:
  - Lecturas: `READ`, `FASTREAD`
  - Escrituras: `WREN`, `PP`
  - Borrados: `SE4K`, `BE32K`, `BE64K`, `CE`
  - Lectura del registro de estado: `RDSR`
- Manejo interno de `WEL` y `WIP`, coherente con el estándar JEDEC.
- Reconstrucción de bytes y direccionamiento autoincremental a partir de una dirección de 24 bits.
- Borrado inmediato de regiones para acelerar la simulación.
- Una cola de datos (`queue_bytes`) que permite al testbench cargar secuencias arbitrarias, manteniendo compatibilidad con un stub previo.

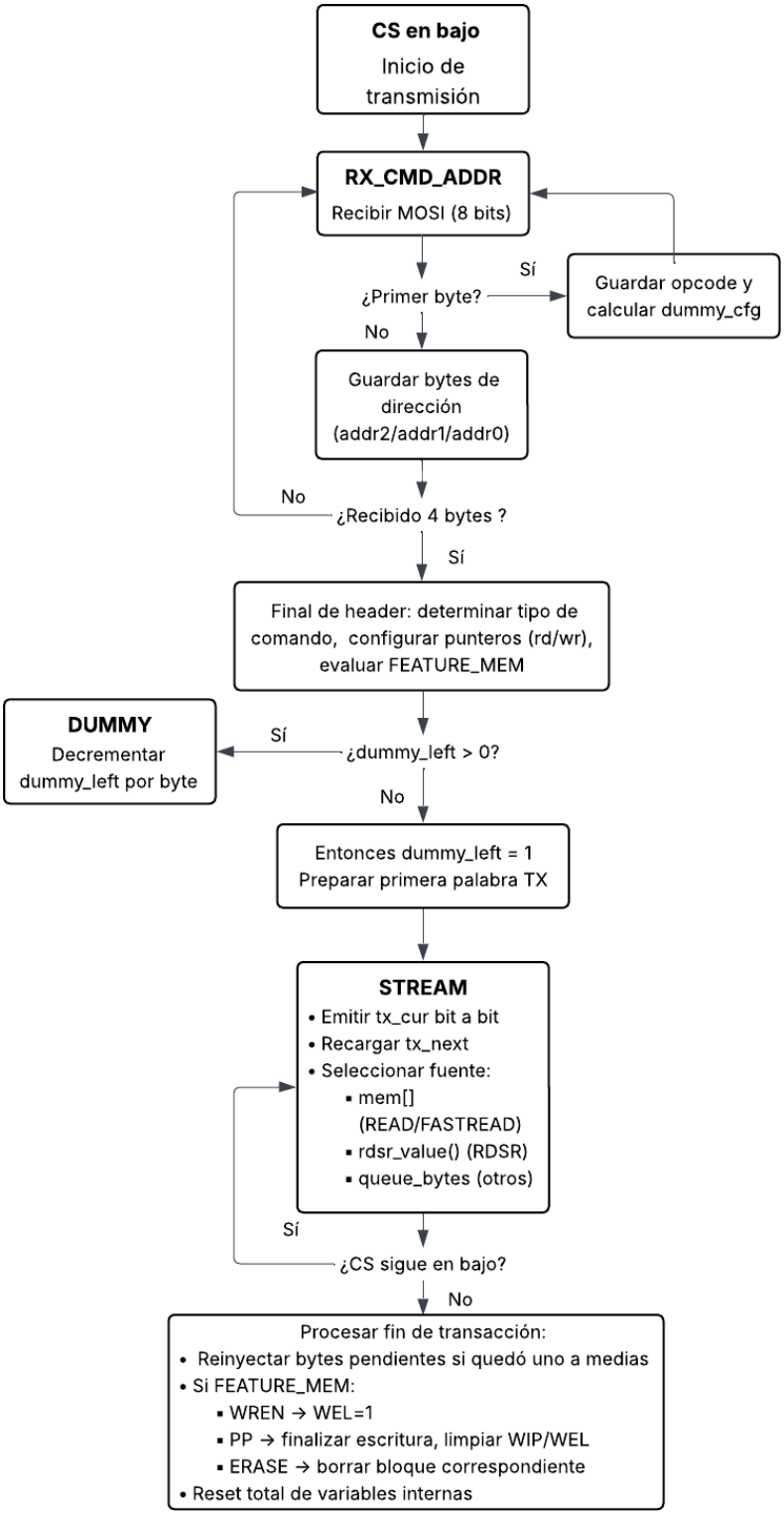

Figura 3.10. Flujo general de funcionamiento del Stub.

El módulo está parametrizado para funcionar en dos modos principales. En el modo de compatibilidad, con FEATURE_MEM desactivado, el stub consume los comandos y direcciones recibidos pero no mantiene un array de memoria asociado. En este caso, los datos que se devuelven por MISO se extraen exclusivamente de la cola queue_bytes, que el testbench puede rellenar mediante tareas auxiliares (clear_stream, push_byte, push_word32). Este enfoque permite reproducir exactamente el comportamiento temporal de un stub heredado, sin introducir efectos de memoria adicionales.

En el modo de memoria, con FEATURE_MEM activado, se habilita una matriz interna mem[0:MEM_BYTES-1] que modela una Flash SPI real. Durante la inicialización, toda la matriz se pone en 0xFF, emulando el estado borrado típico del silicio. A partir de ese momento, los comandos READ y FASTREAD leen datos directamente de dicha matriz, con un puntero de lectura rd_addr_ptr que se inicializa con la dirección de 24 bits recibida por MOSI y se autoincrementa conforme avanza el flujo de datos. Si se solicita una dirección fuera de rango, el modelo devuelve 0xFF, reproduciendo el comportamiento habitual de lectura sobre zonas no programadas.

El manejo de escritura se organiza alrededor de los bits de estado WEL (write enable latch) y WIP (write in progress). El comando WREN no produce efectos inmediatos en el flujo de datos, pero al finalizar la transacción, cuando CS vuelve a nivel alto, el modelo activa WEL. A partir de ahí, si se recibe un comando de programación de página (PP) con dirección de 24 bits, el stub entra en una fase interna de recepción de datos de escritura; cada byte recibido se almacena en la posición correspondiente de mem, empezando en wr_addr y con un contador wr_count que impone el límite de PAGE_SIZE bytes por operación. Durante esta fase se activa WIP para reflejar que hay una programación en curso. Al subir de nuevo CS, se considera que la operación ha concluido de forma instantánea: el modelo limpia WIP y WEL y vuelve al estado inactivo de escritura.

Los comandos de borrado (SE4K, BE32K, BE64K, CE) se implementan de forma análoga. Si al finalizar la transacción el último opcode corresponde a un borrado y WEL está activo, el modelo calcula una dirección base alineada al tamaño de sector o bloque y recorre el rango afectado dentro de mem, escribiendo 0xFF en cada celda. La operación se realiza de manera instantánea cuando CS pasa a nivel alto, lo que simplifica la simulación y evita la introducción de retardos largos. Durante el borrado, el bit WIP se activa temporalmente y se refleja en el valor devuelto por el comando RDSR, cuya respuesta encapsula de forma coherente los bits WEL y WIP en el registro de estado simulado.

La lógica de recepción de MOSI y de control de transacción se organiza en torno a tres fases internas: RX_CMD_ADDR, DUMMY y STREAM. Cuando CS baja, se entra siempre en RX_CMD_ADDR y se reconstruyen bytes a partir de los bits entrantes, respetando el orden MSB-first o LSB-first según el parámetro LSB_FIRST. El primer byte recibido se interpreta como opcode, y los tres siguientes, en modo memoria, se almacenan como dirección de 24 bits. En ese punto se calcula cuántos bytes *dummy* deben descartarse antes de empezar a devolver datos útiles, en función del comando y del parámetro DUMMY_BYTES_DEFAULT. Si el número de bytes dummy es distinto de cero, la máquina pasa a la fase DUMMY, donde simplemente consume ciclos de reloj y mantiene la línea MISO en alta impedancia. Cuando se ha agotado el contador de bytes dummy, o directamente si no se requieren, la lógica entra en la fase STREAM, en la que se emite un flujo continuo de bytes desde la fuente seleccionada.

Para garantizar un comportamiento temporal correcto frente al maestro, la ruta de transmisión por MISO se implementa mediante un pequeño *pipeline* de dos etapas, con los registros tx_cur y tx_next, acompañados de un índice de bit bit_idx y una marca de prelectura. tx_cur contiene siempre el byte actualmente emitido, mientras que tx_next almacena de manera anticipada el siguiente byte, ya sea proveniente de la cola queue_bytes, de la matriz mem o del valor sintético del registro de estado. Cada vez que se completa un byte en la fase STREAM, se avanza el pipeline y se intenta rellenar de nuevo tx_next, de modo que el siguiente byte esté disponible antes de comenzar su emisión. La selección del flanco de SCLK en el que se muestrea MOSI y en el que se actualiza MISO se controla mediante el parámetro SPI_MODE, que determina internamente el valor de CPHA_SLAVE y permite conmutar entre modos SPI 0 y 3 sin modificar el resto de la lógica.

El inicio y fin de cada transacción se determinan exclusivamente por los flancos de CS. En el flanco de bajada se limpian los contadores e índices asociados a la recepción y transmisión de bytes, se resetea la fase a RX_CMD_ADDR y se fuerza la salida MISO a alta impedancia, garantizando que no se conduce el bus antes de tiempo. En el flanco de subida se generan todos los efectos colaterales asociados al opcode recibido (latch de WEL, finalización de programación de página, borrado de sectores o bloques) y se restablece el estado interno para dejar el stub preparado para la siguiente transacción. Adicionalmente, cuando la transacción termina en plena fase de datos, el modelo puede reinyectar en la cola queue_bytes los bytes parcialmente consumidos, de forma que se mantenga la compatibilidad exacta con la versión de referencia del stub.

Este modelo comportamental actúa como referencia durante la verificación: el testbench no solo inspecciona la salida del módulo SPI, sino también el efecto final en la memoria simulada, permitiendo detectar cualquier desviación respecto del comportamiento esperado en un sistema real. En conjunto, el testbench y el stub proporcionan un entorno robusto, reproducible y verificable para evaluar con precisión el rediseño del módulo SPI, garantizando que cumple plenamente las especificaciones funcionales y de arquitectura establecidas.

Capítulo 4

Análisis de resultados

En esta sección se presentan los resultados obtenidos a partir del testbench desarrollado para validar el comportamiento del módulo `top_spi`. Los ensayos abarcan tanto la etapa de arranque (detección y lectura del encabezado y copia del firmware), como las operaciones de borrado y escrituras. La verificación se basa en los logs en tiempo de simulación y en la observación del contenido de la memoria flash simulada tras cada prueba.

4.1. Detección y validación del encabezado

El módulo reconstruye el encabezado a partir del flujo SPI y detecta correctamente el `MAGIC`, la `VERSION` y el `SIZE`, incluso cuando estos valores no se encuentran alineados a palabras de 32 bits. La FSM identifica el desplazamiento (`hdr_off_locked`) y el modo de reordenamiento (`swz_hdr`) necesarios para interpretar el encabezado.

Evidencia de detección del MAGIC

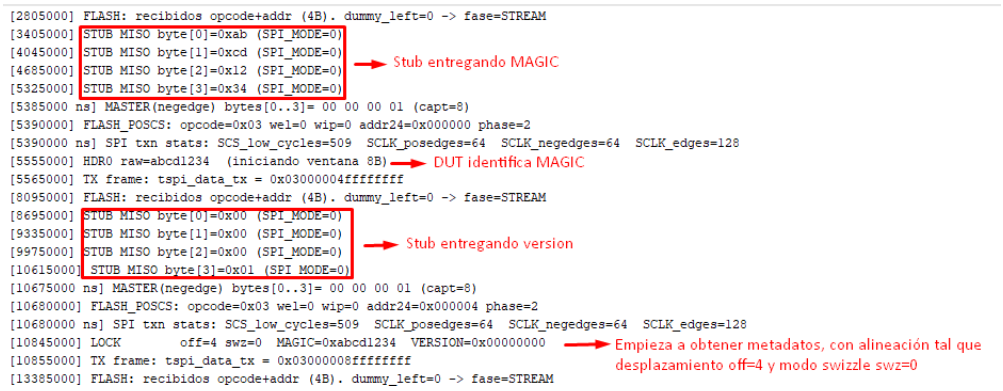

Figura 4.1. Fragmento de log donde se observa la detección del `MAGIC` y el cálculo de alineación.

Extracción del SIZE y VERSION

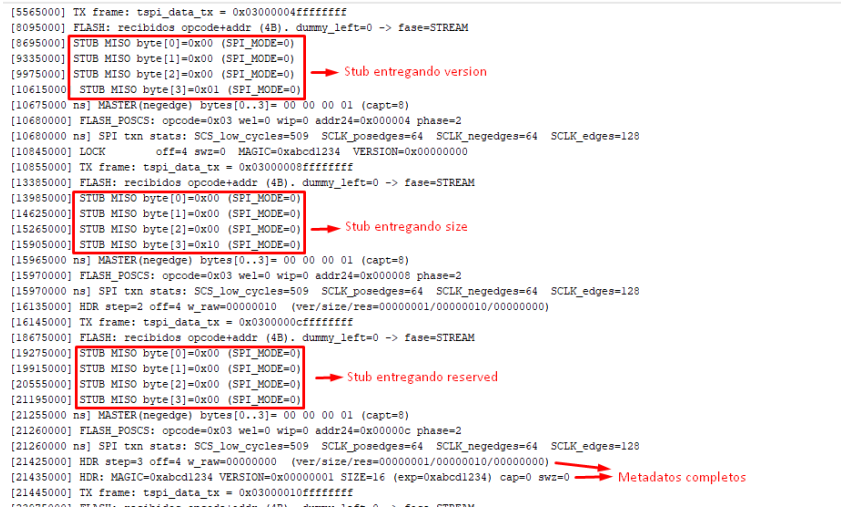

Figura 4.2. Log ilustrando la extracción de `VERSION` y `SIZE`.

4.2. Transferencia del firmware

Una vez validado el encabezado, se inicia la transferencia del payload desde la flash al destino (`COPY_DST_BASE`). El testbench registra cada transacción `WRITE_BYTE`, `WRITE_2BYTE` y `WRITE_4BYTE`, comparando los datos enviados con los almacenados en la flash simulada.

Evidencia de la copia del firmware

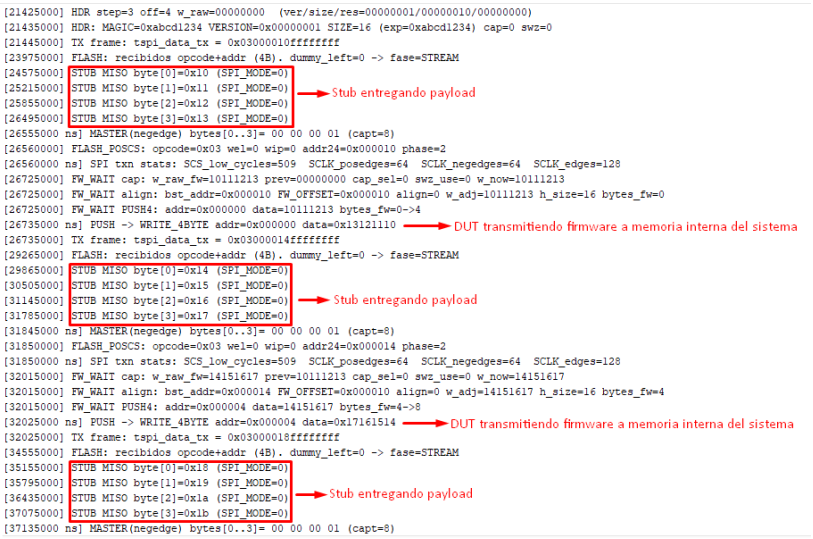

Figura 4.3. Secuencia de `PUSH` generados por el módulo durante la transferencia del firmware.

Comparación entre datos transmitidos y memoria destino

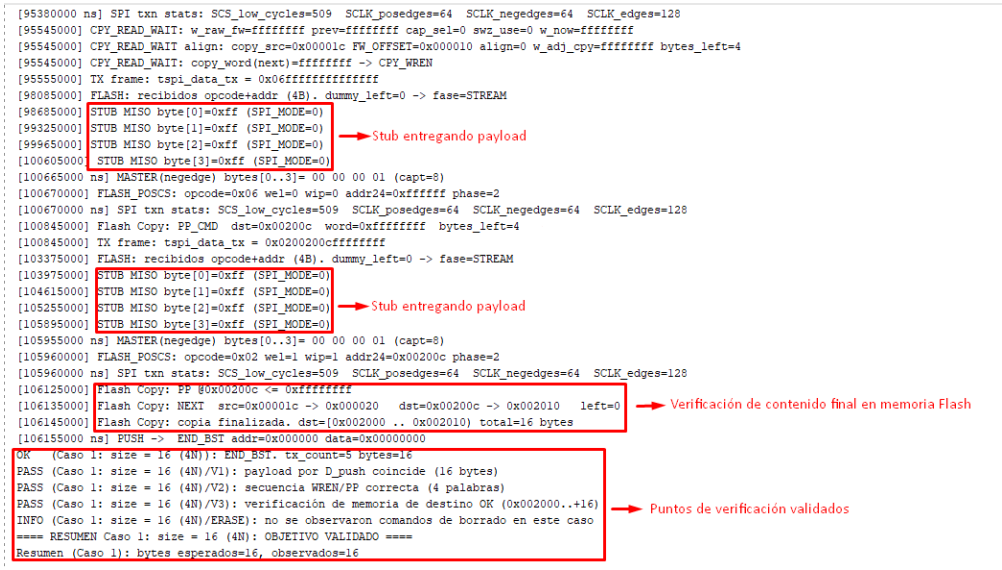

Figura 4.4. Verificación del contenido final en la memoria flash tras las operaciones `PP`.

Se observa la verificación del contenido final en la memoria Flash tras las operaciones Page Program (PP). Se muestra el rango programado `[0x002000..0x002010]` y el resultado `PASS` de la comprobación V3.

En todos los casos válidos, la transferencia se completa exactamente hasta el byte indicado por `SIZE`, sin sobrelecturas ni escrituras fuera de rango.

4.3. Casos límite

El testbench incluye pruebas que fuerzan condiciones extremas o inválidas, tales como:

- MAGIC inválido: el módulo aborta correctamente el arranque.
- SIZE = 0: se genera inmediatamente `END_BST`.
- SIZE mayor que la capacidad: se detecta como error y se aborta.
- SIZE no múltiplo de 4: el módulo transfiere todos los bytes pero solo copia múltiplos de cuatro en memoria destino.

Abortar por MAGIC inválido

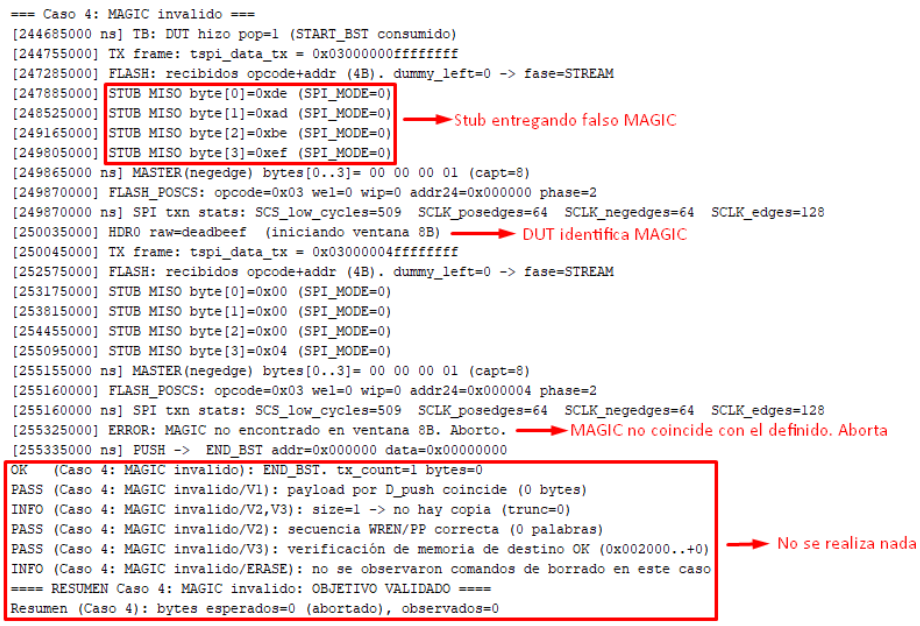

Figura 4.5. Caso en el que el `MAGIC` es inválido y el módulo genera `END_BST`.

SIZE = 0

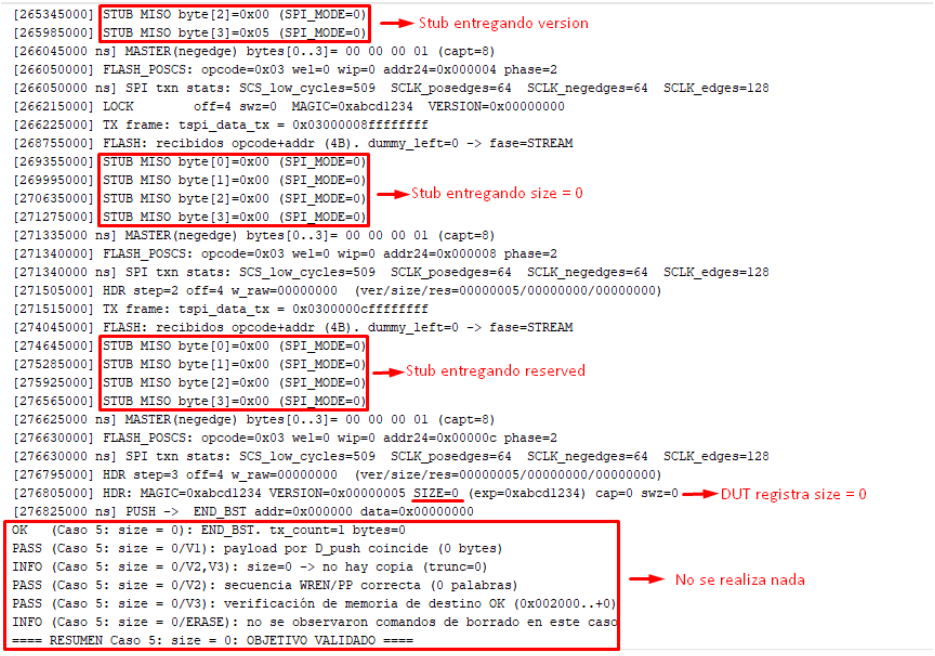

Figura 4.6. Caso donde el `SIZE` equivale a cero y el módulo aborta de forma segura.

SIZE mayor que la capacidad

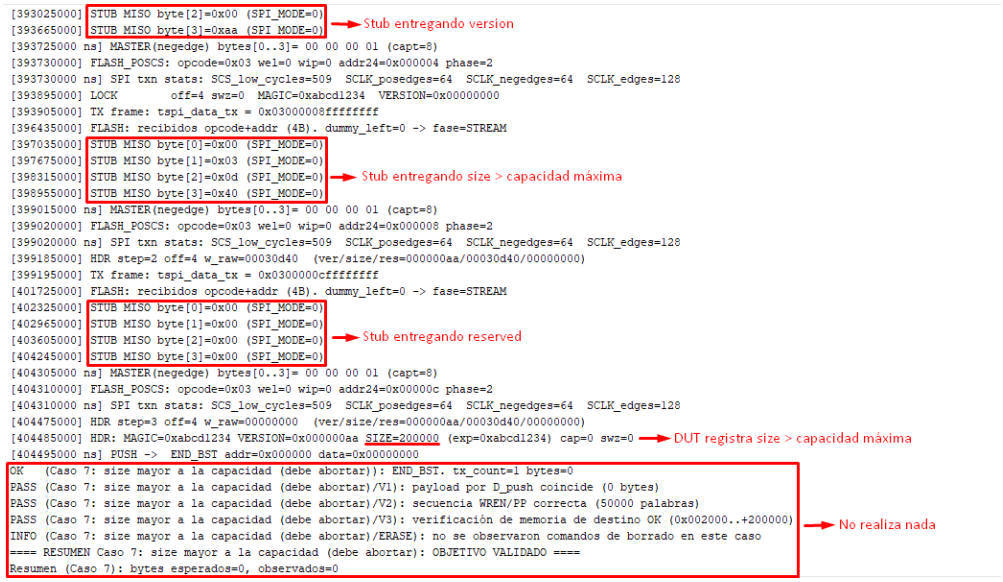

Figura 4.7. Caso donde el `SIZE` excede la capacidad y el módulo aborta de forma segura.

SIZE no múltiplo de 4 (4N+3)

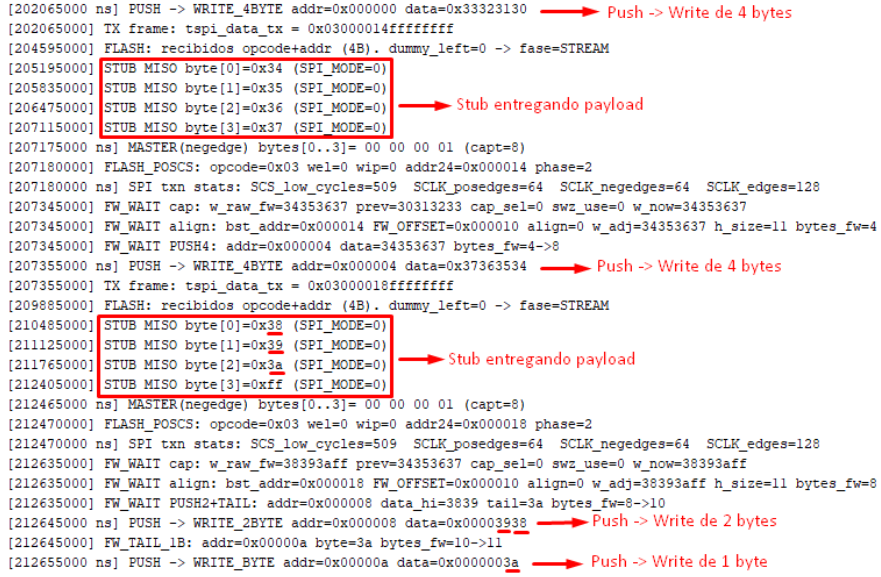

Figura 4.8. Caso donde el `SIZE` no es múltiplo de 4 (4N+3) y el módulo desempeña satisfactoriamente.

4.4. Operaciones de borrado (Erase)

El testbench implementa pruebas independientes para los comandos SE4K (4 KiB), BE32K (32 KiB), BE64K (64 KiB) y CE (Chip Erase).

Cada operación se valida prellenando la memoria con patrones específicos y verificando que la región correspondiente queda completamente en `0xFF`.

Borrado SE4K

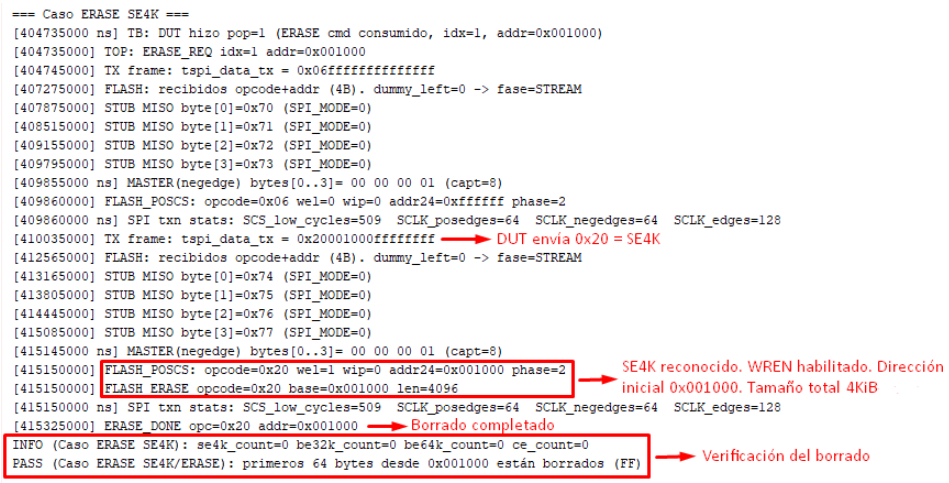

Figura 4.9. Secuencia de borrado de sector y validación posterior.

Borrado Chip Erase

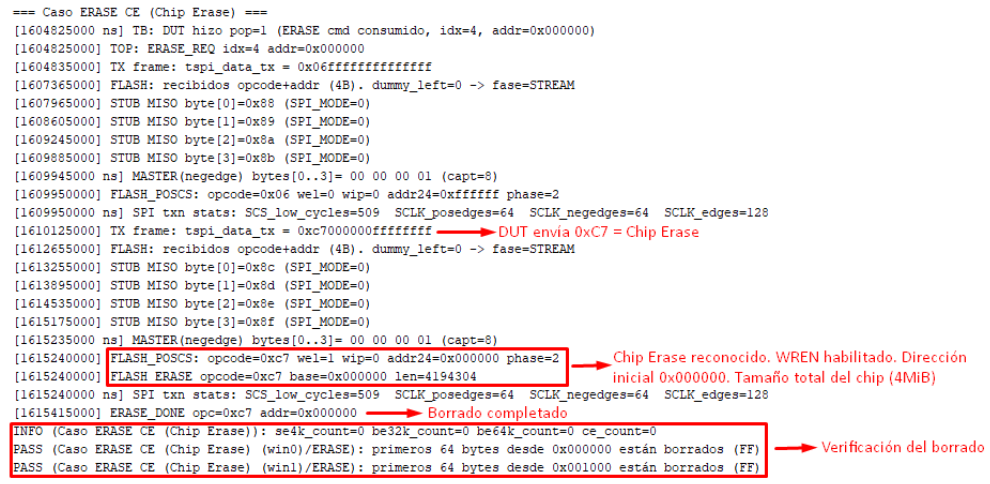

Figura 4.10. Ejecución y comprobación de Chip Erase.

4.5. Reportes de síntesis y comparación con la versión anterior

Con el propósito de cuantificar el impacto del rediseño del módulo SPI, en esta sección se muestran los reportes de temporización, potencia y área, contrastando directamente los resultados obtenidos con los de la versión previa del sistema.

4.5.1. Reportes de temporización

```
top_riscv_soc/TOP/central_control/U698/ZN (AOI211D0HPBWP)
                                                  0.22     44.62 r
top_riscv_soc/TOP/central_control/U699/ZN (OAI22D0HPBWP)
                                                  0.13     44.75 f
top_riscv_soc/TOP/central_control/U700/ZN (AOI211D0HPBWP)
                                                  0.28     45.02 r
top_riscv_soc/TOP/central_control/U701/ZN (AOI32D0HPBWP)
                                                  0.21     45.24 f
top_riscv_soc/TOP/central_control/U715/ZN (AOI221D0HPBWP)
                                                  0.22     45.46 r
top_riscv_soc/TOP/central_control/U716/ZN (ND2D0HPBWP)
                                                  0.22     45.68 f
top_riscv_soc/TOP/central_control/st4/q_reg/D (DFCNQD1HPBWP)
                                                  0.00     45.68 f
data arrival time                                          45.68

clock clk (rise edge)                            50.00     50.00
clock network delay (propagated)                  7.28     57.28
top_riscv_soc/TOP/central_control/st4/q_reg/CP (DFCNQD1HPBWP)
                                                  0.00     57.28 r
clock uncertainty                                -0.40     56.88
library setup time                                0.00     56.88
data required time                                         56.88
-----------------------------------------------------------------
data required time                                         56.88
data arrival time                                         -45.68
-----------------------------------------------------------------
slack (MET)                                                11.20
```

Figura 4.11. Reporte de temporización de versión previa.

```
top_riscv_soc/TOP/central_control/U698/ZN (AOI211D0HPBWP)
                                                  0.22     44.62 r
top_riscv_soc/TOP/central_control/U699/ZN (OAI22D0HPBWP)
                                                  0.13     44.75 f
top_riscv_soc/TOP/central_control/U700/ZN (AOI211D0HPBWP)
                                                  0.28     45.02 r
top_riscv_soc/TOP/central_control/U701/ZN (AOI32D0HPBWP)
                                                  0.21     45.24 f
top_riscv_soc/TOP/central_control/U715/ZN (AOI221D0HPBWP)
                                                  0.22     45.46 r
top_riscv_soc/TOP/central_control/U716/ZN (ND2D0HPBWP)
                                                  0.22     45.68 f
top_riscv_soc/TOP/central_control/st4/q_reg/D (DFCNQD1HPBWP)
                                                  0.00     45.68 f
data arrival time                                          45.68

clock clk (rise edge)                            50.00     50.00
clock network delay (propagated)                  7.28     57.28
top_riscv_soc/TOP/central_control/st4/q_reg/CP (DFCNQD1HPBWP)
                                                  0.00     57.28 r
clock uncertainty                                -0.40     56.88
library setup time                                0.00     56.88
data required time                                         56.88
-----------------------------------------------------------------
data required time                                         56.88
data arrival time                                         -45.68
-----------------------------------------------------------------
slack (MET)                                                11.20
```

Figura 4.12. Reporte de temporización de versión modificada.

4.5.2. Reportes de potencia

```
  Cell Internal Power    = 1.55e+05 nW ( 70.6%)
  Net Switching Power    = 6.47e+04 nW ( 29.4%)
Total Dynamic Power      = 2.20e+05 nW (100.0%)

Cell Leakage Power       = 4.25e+04 nW


  Attributes
  ----------
     u  -  User defined power group

Power Group         Internal Power        Switching Power         Leakage Power           Total Power    (   %  )    Attrs
--------------------------------------------------------------------------------------------------------------------------
io_pad                  0.00e+00              2.40e+03              2.84e+04              3.08e+04    ( 11.8%)
memory                  0.00e+00              0.00e+00              0.00e+00              0.00e+00    (  0.0%)
black_box               0.00e+00              0.00e+00              3.55e+02              3.55e+02    (  0.1%)
clock_network           1.43e+05              4.03e+04              4.58e+02              1.84e+05    ( 70.0%)
register                7.10e+03              6.31e+02              7.37e+03              1.51e+04    (  5.8%)
sequential              9.80e+02              1.26e+03              5.88e+02              2.83e+03    (  1.1%)
combinational           4.31e+03              2.01e+04              5.28e+03              2.97e+04    ( 11.3%)
--------------------------------------------------------------------------------------------------------------------------
Total                   1.55e+05 nW           6.47e+04 nW           4.25e+04 nW           2.62e+05 nW
```

Figura 4.13. Reporte de temporización de versión previa.

```
  Cell Internal Power    = 1.53e+05 nW ( 70.2%)
  Net Switching Power    = 6.50e+04 nW ( 29.8%)
Total Dynamic Power      = 2.18e+05 nW (100.0%)

Cell Leakage Power       = 4.24e+04 nW


  Attributes
  ----------
     u  -  User defined power group

Power Group         Internal Power        Switching Power         Leakage Power           Total Power    (   %  )    Attrs
--------------------------------------------------------------------------------------------------------------------------
io_pad                  0.00e+00              2.41e+03              2.84e+04              3.09e+04    ( 11.8%)
memory                  0.00e+00              0.00e+00              0.00e+00              0.00e+00    (  0.0%)
black_box               0.00e+00              0.00e+00              3.35e+02              3.35e+02    (  0.1%)
clock_network           1.41e+05              4.07e+04              3.92e+02              1.82e+05    ( 69.8%)
register                7.06e+03              6.45e+02              7.37e+03              1.51e+04    (  5.8%)
sequential              9.54e+02              1.22e+03              5.85e+02              2.76e+03    (  1.1%)
combinational           4.32e+03              2.00e+04              5.25e+03              2.96e+04    ( 11.4%)
--------------------------------------------------------------------------------------------------------------------------
Total                   1.53e+05 nW           6.50e+04 nW           4.24e+04 nW           2.61e+05 nW
1
```

Figura 4.14. Reporte de temporización de versión modificada.

4.5.3. Reportes de área

```
Area
---------------------------------------
Combinational Area:           151793.20
Noncombinational Area:         23174.40
Buf/Inv Area:                   3922.40
Total Buffer Area:              2776.00
Total Inverter Area:            1146.40
Macro/Black Box Area:          56309.97
Net Area:                             0
Net XLength:                  155242.96
Net YLength:                  158775.80
---------------------------------------
Cell Area (netlist):                          231277.57
Cell Area (netlist and physical only):        761451.97
Net Length:                   314018.76


Design Rules
---------------------------------------
Total Number of Nets:              9842
Nets with Violations:               566
Max Trans Violations:               331
Max Cap Violations:                 355
---------------------------------------

1
icc2_shell>
```

Figura 4.15. Reporte de temporización de versión previa.

```
Area
---------------------------------------
Combinational Area:           151716.40
Noncombinational Area:         23173.20
Buf/Inv Area:                   3825.20
Total Buffer Area:              2688.00
Total Inverter Area:            1137.20
Macro/Black Box Area:          56309.97
Net Area:                             0
Net XLength:                  155079.73
Net YLength:                  159855.62
---------------------------------------
Cell Area (netlist):                          231199.57
Cell Area (netlist and physical only):        764989.57
Net Length:                   314935.35


Design Rules
---------------------------------------
Total Number of Nets:              9828
Nets with Violations:               556
Max Trans Violations:               338
Max Cap Violations:                 342
---------------------------------------

1
```

Figura 4.16. Reporte de temporización de versión modificada.

4.5.4. Comparación con la versión anterior

En esta sección se comparan las diferencias observadas entre los reportes de temporización, potencia y área correspondientes a la versión previa del diseño y la versión actualizada que incorpora el rediseño del módulo SPI. Dado que los reportes corresponden al SoC completo, las variaciones reflejan únicamente el impacto indirecto que el nuevo SPI introduce sobre la implementación del sistema.

Comparación de temporización

En cuanto al margen temporal, el setup slack mínimo mejora ligeramente, pasando de 11.20 ns en la versión previa a 11.35 ns en la versión modificada. La variación absoluta de 0.15 ns representa un incremento cercano al 1.34 %.

Además de la ligera mejora del setup slack, es importante destacar que la ruta crítica permanece esencialmente inalterada. En ambas versiones, el camino temporal atraviesa el mismo bloque de lógica secuencial en central_control.

Tanto los retardos de celda como los parámetros globales del análisis estático de temporización (incertidumbre, tiempos de librería, required time) permanecen prácticamente constantes entre versiones.

Comparación de potencia

El análisis comparativo de potencia revela que la potencia dinámica total pasa de $2{,}20 \times 10^5$ nW a $2{,}18 \times 10^5$ nW, lo que representa una variación relativa del 0.90 % menos. El desglose por categorías muestra un leve incremento en la potencia de conmutación, la que se produce durante el proceso de cambio de estado entre los niveles lógicos (0 a 1 o 1 a 0), pasando de $6{,}47 \times 10^4$ nW a $6{,}50 \times 10^4$ nW, equivalente a un aumento del 0.46 %. Este incremento es consistente con la adición de estados intermedios, lógica de reordenamiento y control adicional dentro del módulo SPI, que introducen algunas transiciones adicionales sobre señales internas.

En contraste, la potencia de fuga se mantiene estable, pasando de $4{,}25 \times 10^4$ nW a $4{,}24 \times 10^4$ nW, implicando una ligera disminución en las corrientes que fluyen a través de transistores apagados.

Comparación de área

Al comparar el área total sintetizada del diseño, ambas versiones presentan valores relativamente cercanos: la versión previa registra 231 277.57 μm^2 y la versión modificada reporta 231 199.57 μm^2. La diferencia absoluta es de 78 μm^2, lo que corresponde a una variación menor al 0.04 %.

Más allá del área, también se evidencia estabilidad en las métricas físicas asociadas al ruteo. El número total de nets pasa de 9842 a 9828, la cantidad de nets con violaciones cambia de 566 a 556; lo cual representa una pequeña mejora, las violaciones de max transition/slew rate (tiempo que tarda una

señal en cambiar de 0 a 1 o de 1 a 0) varían de 331 a 338, y las de máxima capacitancia de 355 a 342.

Este resultado indica que la incorporación del nuevo SPI, pese a incluir lógica de decodificación del encabezado, reconstrucción de ventanas de datos y un manejo más extenso del protocolo, no incrementa la ocupación de celdas de manera muy considerable.

Resumen cuantitativo

La Tabla 4.1 muestra un resumen cuantitativo de las principales métricas comparadas entre ambas versiones.

Tabla 4.1. Comparación entre la versión previa y la versión actualizada del diseño

Métrica	**Previa**	**Modificada**	**Variación absoluta**	**Variación relativa**
Área total * [μm^2]	231 277.57	231 199.57	−78,00	−0,034 %
Área total ** [μm^2]	761 451.97	764 989.57	+3 537,6	+0,464 %
Potencia dinámica [nW]	$2{,}20 \times 10^5$	$2{,}18 \times 10^5$	−0,02	−0,909 %
Potencia conmutación [nW]	$6{,}47 \times 10^4$	$6{,}50 \times 10^4$	+0,03	+0,46 %
Leakage [nW]	$4{,}25 \times 10^4$	$4{,}24 \times 10^4$	−0,01	−0,235 %
Slack setup [ns]	11.20	11.35	+0,15	+1,34 %

(*) netlist.
(**) netlist y physical únicamente.

Capítulo 5

Conclusiones y Recomendaciones

Conclusiones

El rediseño del módulo `top_spi` permitió superar de manera eficaz las limitaciones de la arquitectura original, particularmente aquellas relacionadas con la falta de un mecanismo de arranque robusto y la imposibilidad de reutilizar la memoria Flash una vez completada la secuencia de bootstrap.

- La incorporación de un encabezado estructurado (MAGIC, VERSION, SIZE y RESERVED) eliminó la dependencia del patrón `0xFFFFFFFF` como marcador de fin de firmware, resolviendo la ambigüedad del esquema anterior.
- El módulo fue extendido para habilitar el acceso operativo a la Flash después del arranque, integrando rutinas de lectura, escritura y borrado mediante un conjunto de comandos explícitos. Esto transforma la Flash en un recurso reutilizable durante la ejecución normal y no únicamente durante el proceso de bootstrap.
- La comparación entre ambas versiones del diseño confirma que las mejoras funcionales no introducen penalizaciones significativas en las métricas físicas del sistema. El área lógica varía menos del 0.04 %, la potencia total presenta diferencias menores al 1 %, y la temporización global mantiene la misma ruta crítica con un ligero aumento en el *setup slack*, evidenciando que la modificación no afecta el rendimiento ni la integridad temporal del chip.

En conjunto, el nuevo `top_spi` se integra correctamente dentro de la arquitectura del procesador, incrementa su versatilidad funcional y mantiene su eficiencia en términos de área, potencia y temporización.

Recomendaciones

Aunque los objetivos del proyecto se cumplieron satisfactoriamente, se identifican oportunidades de mejora que podrían explorarse en trabajos futuros:

- Si bien el formato del encabezado actual es adecuado, podrían evaluarse estrategias de optimización (compactación o reorganización) que reduzcan la lógica y, con ello, mejorar las métricas de área, potencia y temporización.
- La memoria IS25WP032D soporta modos Dual, Quad y DTR, que podrían aumentar significativamente el ancho de banda del enlace SPI. Se recomienda extender el testbench a fin de estudiar estas modalidades, especialmente para aplicaciones que requieran tiempos de arranque o transferencia más reducidos.
- Se sugiere complementar el análisis con validaciones sobre prototipos físicos que permitan observar aspectos no modelados por la síntesis, tales como tolerancias eléctricas y comportamiento frente a ruido.

Bibliografía

[1] C. C. de Iniciativas de Desarrollo (CINDE), *Dispositivos médicos se consolidan como el principal producto de exportación de Costa Rica*, https://www.cinde.org/es/noticias/dispositivos-medicos-se-consolidan-como-el-principal-producto-de-exportacion-de-costa-rica, Enero, 2018.

[2] P. de Comercio Exterior (PROCOMER), *Costa Rica es el exportador 1 de dispositivos médicos, piña y jugo de piña en diferentes mercados*, https://procomer.com/costa-rica-es-el-exportador-1-de-dispositivos-medicos-pina-y-jugo-de-pina-en-diferentes-mercados/, Octubre, 2024.

[3] C. y Mercadeo, *Costa Rica creó el primer microcontrolador para microcircuito de aplicaciones médicas totalmente diseñado y desarrollado en el país*, https://www.tec.ac.cr/hoyeneltec/2019/05/08/costa-rica-creo-primer-microcontrolador-microcircuito-aplicaciones-medicas-totalmente, Mayo, 2019.

[4] N. S. Academy, *nRF Connect SDK Intermediate, Serial Peripheral Interface (SPI), Lesson 5*, https://academy.nordicsemi.com/courses/nrf-connect-sdk-intermediate/lessons/lesson-5-serial-peripheral-interface-spi/topic/serial-peripheral-interface-spi/.

[5] ISSI, *IS25WP032D 32Mb SERIAL FLASH MEMORY WITH 133MHZ MULTI I/O SPI QUAD I/O QPI DTR INTERFACE*, https://datasheet4u.com/datasheets/ISSI/IS25LP032D/1303761, Septiembre, 2018.

[6] T. M. Blog, *VLSI Design: A Complete Overview of the VLSI Design Flow*, https://www.themechatronicsblog.com/2023/04/vlsi-design-%20a-complete-overview-of-the-vlsi-design-flow.html, Abril 2023.

[7] R. G.-R. et al., *Siwa: a RISC-V RV32I based Micro-Controller for Implantable Medical Applications*, https://ieeexplore.ieee.org/document/9068952, 2020.

[8] ——, *Siwa: A custom RISC-V based system on chip (SOC) for low power medical applications*, https://www.sciencedirect.com/science/article/pii/S0026269219303787, 2021.

Capítulo 6

Anexos

Anexo A: Enlace al repositorio Git

El código fuente completo del proyecto se encuentra disponible en el siguiente repositorio:

`https://github.com/Chacjean/TFG_SystemverilogFiles_SPI`