

Máster Universitario en Electrónica y Tecnología Digital Aplicada (METDA)

Trabajo Fin de Máster

**Diseño e Implementación de instrucciones personalizadas
para aceleración de criptografía post-cuántica en
arquitecturas RISC-V**

Autor: EMMA REYES RAMOS CHINEA

**Tutores: DR. PEDRO PÉREZ CARBALLO
AGONEY J. MORENO RODRÍGUEZ**

Fecha: 6 de agosto de 2026

Máster Universitario en Electrónica y Tecnología Digital Aplicada (METDA)

Trabajo Fin de Máster

**Diseño e Implementación de instrucciones personalizadas
para aceleración de criptografía post-cuántica en
arquitecturas RISC-V**

HOJA DE FIRMAS

	Nombre	Firma
Estudiante:	Emma Reyes Ramos Chinaea	
Tutor	Dr. Pedro Pérez Carballo	
Académico:		
Cotutor:	Agoney J. Moreno Rodríguez	

Fecha: 6 de agosto de 2026

Máster Universitario en Electrónica y Tecnología Digital Aplicada (METDA)

Trabajo Fin de Máster

**Diseño e Implementación de instrucciones personalizadas
para aceleración de criptografía post-cuántica en
arquitecturas RISC-V**

HOJA DE EVALUACIÓN

Calificación:

.....

Presidente: Antonio Núñez Ordóñez Fdo.:

Secretario: Fernando de la Puente Arrate Fdo.:

Vocal: Francisco Javier del Pino Suárez Fdo.:

Fecha: 6 de agosto de 2026

Resumen

Los avances en la computación cuántica representan una amenaza para los sistemas criptográficos clásicos en los que se basa la seguridad informática actualmente. Para mitigar esta posible vulnerabilidad, el National Institute of Standards and Technology (NIST) ha liderado un proceso de estandarización de nuevos esquemas de Criptografía Post-Cuántica (PQC). Este proceso se desarrolló en varias rondas de evaluación y descarte continuado, que acabó con la estandarización de ML-KEM como algoritmo de establecimiento de clave y ML-DSA y SLH-DSA como algoritmos de firma digital. La integración de estos algoritmos en sistemas embebidos con recursos limitados supone un reto debido al tamaño de las claves de estos estándares y a la capacidad de computación necesaria.

En este contexto, la elección de la arquitectura de computación es importante. Las arquitecturas *open-source* han ganado una especial relevancia debido a la versatilidad y adaptabilidad que ofrecen. Destacan los desarrollos en arquitecturas RISC-V, que cuenta con un gran respaldo liderado por la fundación RISC-V Internacional. Esta arquitectura presenta un diseño abierto y modular, que le permite adaptarse a todo tipo de aplicaciones.

El presente Trabajo de Fin de Máster se centra en el diseño e implementación de una extensión del juego de instrucciones para la aceleración de sistemas criptográficos post-cuánticos en una arquitectura RISC-V. Para ello, se presenta en primer lugar un estudio de conceptos criptográficos clave y los avances que han llevado a los estándares de criptografía post-cuántica actuales. Además, se hace un análisis de la arquitectura RISC-V y algunos desarrollos disponibles. Entre ellos destacan PicoRV32, Ibex y NEORV32. Tras la implementación de los núcleos más destacados, se ha seleccionado el núcleo NEORV32 por su diseño modular, su amplia documentación y la disponibilidad de una *Custom Functions Unit* (CFU), ideal para la integración de aceleradores *hardware*.

Inicialmente se prueba una ejecución puramente *software* del algoritmo de firma ML-DSA, usando la librería `mldsa-native`. Posteriormente, a este diseño se añade el generador de números aleatorios verdadero (TRNG) integrado en el NEORV32 para garantizar la seguridad criptográfica necesaria en la generación de claves. Seguidamente, se identifica que una de las funciones dentro del algoritmo que permite su aceleración mediante instrucciones personalizadas es SHAKE. Por consiguiente, se desarrolla un acelerador dedicado para el núcleo Keccak dentro de la CFU, controlable a través de cinco nuevas instrucciones tipo R añadidas a la ISA del procesador.

La evaluación del sistema se llevó a cabo mediante su síntesis e implementación sobre una FPGA Artix-7, concretamente, la placa de prototipado Nexys Video. Al añadir la CFU en el diseño aumenta el número de recursos utilizados. Por otro lado, al analizar el rendimiento, se obtiene que la aceleración en hardware de las operaciones SHAKE logra un factor de aceleración de aproximadamente 60x frente a su ejecución en *software*. Aplicando la Ley de Amdahl se obtiene una aceleración global de 1.03x, que evidencia la necesidad de aplicar otras aceleraciones a la ejecución del algoritmo.

Abstract

Advances in quantum computing pose a threat to the classical cryptographic systems on which current IT security is based. To mitigate this potential vulnerability, the National Institute of Standards and Technology (NIST) has led a process to standardise new Post-Quantum Cryptography (PQC) schemes. This process involved several rounds of evaluation and continuous elimination, culminating in the standardisation of ML-KEM as a key encapsulation algorithm and ML-DSA and SLH-DSA as digital signature algorithms. Integrating these algorithms into embedded systems with limited resources poses a challenge due to the size of the keys used in these standards and the computing power required.

In this context, the choice of computing architecture is important. Open-source architectures have become particularly significant due to the versatility and adaptability they offer. Developments in RISC-V architectures are particularly noteworthy, with strong support led by the RISC-V International Foundation. This architecture features an open and modular design, enabling it to adapt to all kinds of applications.

This Master's Thesis focuses on the design and implementation of an instruction set extension for the acceleration of post-quantum cryptographic systems on a RISC-V architecture. To this end, it first presents a study of key cryptographic concepts and the advances that have led to current post-quantum cryptography standards. Furthermore, it analyses the RISC-V architecture and some available developments. Notable among these are PicoRV32, Ibex and NEORV32. Following the implementation of the most prominent cores, the NEORV32 core was selected due to its modular design, extensive documentation and the availability of a Custom Functions Unit (CFU), which is ideal for the integration of hardware accelerators.

Initially, a purely software-based execution of the ML-DSA signature algorithm is tested, using the `mldsa-native` library. Subsequently, the true random number generator (TRNG) integrated into the NEORV32 is added to this design to ensure the necessary cryptographic security in key generation. It is then identified that one of the functions within the algorithm that allows for its acceleration via custom instructions is SHAKE. Consequently, a dedicated accelerator for the Keccak core is developed within the CFU, controllable via five new R-type instructions added to the processor's ISA.

The system was evaluated by synthesising and implementing it on an Artix-7 FPGA, using the Nexys Video prototyping board. Adding the CFU to the design increases the number of resources used. Furthermore, an analysis of performance shows that hardware acceleration of SHAKE operations achieves an acceleration factor of approximately 60x compared to their execution in *software*. Applying Amdahl's Law yields an overall acceleration of 1.03x, which highlights the need to apply further optimisations to the algorithm's execution.

Índice

Resumen	i
Abstract	ii
Índice	iii
Índice de figuras	v
Índice de tablas	vii
Índice de códigos	viii
Acrónimos	ix
1. Introducción	1
1.1. Antecedentes	1
1.2. Objetivos	3
1.3. Organización de la memoria	3
2. Criptografía post-cuántica	5
2.1. Introducción a la criptografía	5
2.1.1. Criptografía clásica	5
2.1.2. La amenaza de la computación cuántica	6
2.2. Criptografía post-cuántica	8
2.3. Proceso de estandarización del NIST	8
2.3.1. FIPS 203: ML-KEM	9
2.3.2. FIPS 204: ML-DSA	10
2.3.3. FIPS 205: SLH-DSA	10
2.3.4. NIST SP 800-208: LMS y XMSS	10
2.4. Retos y oportunidades	10
2.5. Conclusión	11
3. Arquitectura RISC-V	13
3.1. Introducción	13
3.2. ISA no privilegiada	13
3.2.1. Formato de las instrucciones	14
3.2.2. Registros e Interfaz Binaria de Aplicación	15
3.3. ISA privilegiada	16
3.3.1. Modos de ejecución	16
3.3.2. Registros CSR	17
3.3.3. Manejo de excepciones e interrupciones (<i>Traps</i>)	18
3.4. Conclusión	18
4. Estudio y análisis de un algoritmo de firma digital post-cuántico	19
4.1. Algoritmos de firma digital	19
4.2. ML-DSA	20

4.3. Análisis del estado del arte	24
4.4. Conclusión	28
5. Implementación de procesadores con arquitectura RISC-V	29
5.1. Estudio previo	29
5.2. PicoRV32	30
5.3. Ibex	33
5.4. NEORV32	35
5.5. Conclusión	39
6. Implementación de ML-DSA en un núcleo RISC-V	43
6.1. Implementación <i>software</i>	43
6.2. Aceleración <i>hardware</i>	48
6.2.1. <i>True Random-Number Generator</i>	49
6.2.2. <i>Custom Functions Unit</i> del NEORV32	50
6.2.3. Extensión del juego de instrucciones	51
6.3. Conclusión	65
7. Evaluación de los resultados	67
7.1. Presentación de los resultados	67
7.2. Comparativa con otros trabajos	76
7.3. Conclusión	77
8. Conclusiones y trabajos futuros	79
8.1. Conclusiones	79
8.2. Trabajos futuros	80
Referencias	81
Objetivos de Desarrollo Sostenible (ODS)	86
Anexos	87

Índice de figuras

Figura 1	Plazos para la migración a criptografía post-cuántica	2
Figura 2	Cifrado César	5
Figura 3	Criptografía simétrica	6
Figura 4	Criptografía asimétrica	6
Figura 5	Formatos de las instrucciones RISC-V.	14
Figura 6	Combinaciones de modos de privilegio.	17
Figura 7	Proceso de firma digital	20
Figura 8	Algoritmo para la generación de claves	22
Figura 9	Algoritmo para el procedimiento de firma	23
Figura 10	Algoritmo para la verificación	24
Figura 11	Arquitectura SIMD del procesador propuesta por Ye et al.	25
Figura 12	Subsistema RISC-V integrado con acelerador de hardware	26
Figura 13	Arquitectura propuesta para la aceleración de ML-DSA y ML-KEM	26
Figura 14	Arquitectura a alto nivel del sistema ML-DSA.. . . .	27
Figura 15	Arquitectura del núcleo PicoRV32	30
Figura 16	Timing Report de la implementación del PicoRV32	32
Figura 17	Arquitectura del procesador Ibex	33
Figura 18	Diagrama de bloques del Ibex Demo System	34
Figura 19	Timing Report de la implementación del Ibex Demo System	35
Figura 20	Diagrama de bloques del procesador NEORV32.	36
Figura 21	Simulación del ejemplo de uso de la CFU para XTEA.	37
Figura 22	Timing Report de la implementación del NEORV32	39
Figura 23	Espacio de diseño representando T_c para $F_{\max}$ del núcleo frente a SRNU	41
Figura 24	Espacio de diseño representando la potencia a total consumida (en mW) frente a SRNU para los casos de estudio.	41

Figura 25	Ejemplo básico de ejecución de ML-DSA-44 nativa.	44
Figura 26	Ejemplo de ejecución de mldsa-native en el núcleo NEORV32.	46
Figura 27	Proceso de depurado con OpenOCD y GDB.	48
Figura 28	Ejemplo de ejecución de mldsa-native en el núcleo NEORV32 con TRNG.	50
Figura 29	Estructura de esponja implementado en Keccak.	52
Figura 30	Diagrama de bloques de la CFU.	54
Figura 31	FSM de control de la CFU	55
Figura 32	Resultado en simulación de la instrucción keccak_hw_reset	58
Figura 33	Resultado en simulación de la instrucción keccak_hw_init	58
Figura 34	Resultado en simulación de las instrucciones keccak_hw_data y keccak_hw_data_last	58
Figura 35	Resultado en simulación de la ejecución del núcleo Keccak	59
Figura 36	Resultado en simulación de la instrucción de lectura de resultados (1)	59
Figura 37	Resultado en simulación de la instrucción de lectura de resultados (2)	59
Figura 38	Ejemplo de ejecución del algoritmo Keccak.	60
Figura 39	Ejecución de ML-DSA en <i>software</i>	64
Figura 40	Ejecución de ML-DSA con aceleración <i>hardware</i>	64
Figura 41	Resultado de la implementación del NEORV32 sobre la placa de prototipado Nexys Video.	67
Figura 42	Timing report de la implementación de NEORV32	68
Figura 43	Resultado de la implementación del NEORV32 con CFU sobre la placa de prototipado Nexys Video.	69
Figura 44	Resultado de la implementación de la CFU y el TRNG en NEORV32.	70
Figura 45	Timing report de la implementación de NEORV32 con CFU para una frecuencia de 100 MHz	71
Figura 46	Informe de la ruta crítica de la implementación de NEORV32 con CFU	71
Figura 47	Resultado de simulación de una ejecución completa	73
Figura 48	Resultados en ciclos de reloj de la ejecución de ML-DSA con aceleración y sin aceleración.	75

Índice de tablas

Tabla 1	Codificación del código de operación principal en RISC-V	15
Tabla 2	Ejemplo de una instrucción de suma	15
Tabla 3	Convención de registros de enteros en RV32I	16
Tabla 4	Niveles de privilegio en RISC-V	16
Tabla 5	Tamaños en bytes de las claves y las firmas de ML-DSA	21
Tabla 6	Núcleos RISC-V analizados	29
Tabla 7	Utilización de recursos lógicos tras la implementación del núcleo PicoRV32 (con extensión M y multiplicador rápido)	32
Tabla 8	Estimación del consumo de potencia de la implementación del sistema PicoRV32 . . .	33
Tabla 9	Utilización de recursos lógicos del núcleo Ibex	34
Tabla 10	Estimación del consumo de potencia de la implementación del Ibex Demo System. . .	35
Tabla 11	Utilización de recursos lógicos post-implementación del núcleo NEORV32.	38
Tabla 12	Estimación del consumo de potencia de la implementación del núcleo NEORV32. . . .	39
Tabla 13	Comparación de valores PPA (Potencia, Rendimiento y Área) para dispositivo AMD Artix 7™ de las implementaciones RISC-V estudiadas.	40
Tabla 14	Interfaz de la CFU	51
Tabla 15	Utilización de recursos de la FPGA (segundo reporte de implementación)	68
Tabla 16	Estimación del consumo de potencia de la implementación del núcleo NEORV32. . . .	68
Tabla 17	Utilización de recursos de NEORV32+CFU tras la síntesis e implementación	69
Tabla 18	Estimación del consumo de potencia de la implementación del núcleo NEORV32. . . .	71
Tabla 19	Caracterización del acelerador de instrucción en hardware	72
Tabla 20	Tiempos de ejecución y factor de aceleración local de las operaciones SHAKE.	74
Tabla 21	Comparativa de implementaciones	76
Tabla 22	Relación del proyecto con los Objetivos de Desarrollo Sostenible (ODS)	86

Índice de códigos

Código 1	<i>Software C</i> para el parpadeo de los leds de la placa en las pruebas del núcleo PicoRV32	31
Código 2	Compilación del firmware del núcleo PicoRV32	31
Código 3	Definición de instrucciones personalizadas para XTEA.	37
Código 4	Programa para la ejecución de mldsa-native en NEORV32	45
Código 5	Configuración de OpenOCD para depuración del NEORV32	47
Código 6	Comandos principales de GDB para la depuración.	47
Código 7	Función randombytes() usando el TRNG del NEORV32	49
Código 8	Configuración de parámetros del TRNG	49
Código 9	Definición de opcodes soportados para instrucciones personalizadas	50
Código 10	Definición de los tipos de instrucciones personalizadas	51
Código 11	Definición de las instrucciones personalizadas para la aceleración de Keccak.	53
Código 12	Decodificador de instrucciones de la CFU	54
Código 13	Proceso de la CFU encargado de guardar los datos de entrada	56
Código 14	Proceso de la CFU encargado de guardar los datos de salida	57
Código 15	Programa para la ejecución del algoritmo Keccak.	60
Código 16	Función mld_shake256 con la directiva #ifdef.	64
Código 17	CFU diseñada para la ejecución de la función Keccak	87

Acrónimos

ABI – Interfaz binaria de aplicación [15](#), [15](#), [15](#), [15](#)

AES – *Advanced Encryption Standard* [7](#), [8](#), [11](#)

ALU – Unidad Aritmético-Lógica [25](#), [51](#)

API – Interfaz de Programación de Aplicaciones [43](#)

ASIC – *Application-Specific Integrated Circuit* [35](#)

BRAM – *Block RAM* [69](#), [80](#)

CFU – *Custom Functions Unit* [36](#), [36](#), [40](#), [50](#), [50](#), [50](#), [51](#), [51](#), [51](#), [51](#), [52](#), [53](#), [53](#), [54](#), [54](#), [54](#), [54](#), [57](#), [57](#), [59](#), [62](#), [63](#), [63](#), [63](#), [65](#), [67](#), [68](#), [68](#), [68](#), [69](#), [71](#), [76](#), [77](#), [79](#), [79](#), [80](#), [80](#), [80](#)

CNSA – *Commercial National Security Algorithm Suite* [10](#)

CPU – Unidad Central de Procesamiento [37](#), [38](#), [39](#), [51](#), [54](#), [63](#), [72](#), [80](#), [80](#)

CRQC – *Cryptographically Relevant Quantum Computers* [7](#), [11](#)

CSR – Registros de Control y Estado [17](#), [17](#), [18](#)

DES – *Data Encryption Standard* [11](#)

DMEM – *Data Memory* [37](#)

DS – *Digital Signature* [8](#)

DSP – *Digital Signal Processing* [32](#), [34](#), [38](#), [40](#)

ECC – *Elliptic Curve Cryptography* [7](#)

FIFO – *First In, First Out* [49](#)

FIPS – *Federal Information Processing Standards* [9](#), [9](#), [9](#), [9](#), [10](#), [10](#), [20](#), [21](#)

FN-DSA – *FFT over NTRU-Lattice-Based Digital Signature Algorithm* [9](#)

FPGA – *Field-Programmable Gate Array* [25](#), [25](#), [29](#), [30](#), [30](#), [32](#), [35](#), [39](#), [49](#), [49](#), [67](#), [68](#), [69](#), [70](#), [77](#), [80](#)

FSM – *Finite State Machine* [54](#), [72](#)

GCC – *GNU Compiler Collection* [43](#)

GDB – *GNU Debugger* [46](#), [46](#), [46](#), [46](#), [46](#)

GPIO – *General-Purpose Input/Output* [37](#)

HDL – Lenguaje de descripción *hardware* [29](#), [29](#), [29](#), [30](#), [39](#)

HQC – *Hamming Quasi-Cyclic* [2](#), [9](#)

IMEM – *Instruction Memory* [37](#), [38](#)

ISA – Instruction Set Architecture 2, 3, 3, 13, 13, 13, 13, 13, 13, 13, 14, 14, 14, 15, 15, 15, 16, 16, 18, 18, 36, 50, 50, 50, 67, 79

JTAG – Joint Test Action Group 38, 46

KEM – Key Encapsulation Mechanism 8, 9, 9, 9

LMS – Leighton-Micali Signature 10, 10

LUT – Look Up Table 31, 34, 38, 40, 67, 68, 68

ML-DSA – Module-Lattice-Based Digital Signature Algorithm 1, 2, 3, 9, 10, 10, 10, 10, 19, 20, 21, 21, 21, 24, 24, 25, 25, 26, 27, 28, 28, 28, 43, 43, 43, 43, 48, 49, 49, 51, 53, 59, 65, 67, 74, 75, 76, 77, 79, 79, 80, 80, 80

ML-KEM – Module-Lattice-Based Key-Encapsulation Mechanism 1, 2, 9, 9, 10, 24, 25, 27

NIST – National Institute of Standards and Technology 1, 1, 2, 5, 8, 8, 9, 9, 9, 9, 10, 10, 10, 11, 11, 20, 28, 52, 79, 79

NSA – National Security Agency 10

NTT – Number Theoretic Transform 25, 25, 26, 26, 27, 27, 27, 27, 76

OpenOCD – Open On-Chip Debugger 46, 46, 46

PKE – Public-Key Encryption 8

PQC – Post-Quantum Cryptography 2, 2, 2, 2, 8, 8, 76, 79

RAM – Random Access Memory 38, 67, 68

RISC – Reduced Instruction Set Computer 15

RNG – Random Number Generator 21, 49

ROM – Read-Only Memory 37

RSA – Rivest-Shamir-Adleman 7

SHA – Secure Hash Algorithm 52

SHA-3 – Secure Hash Algorithm 3 8

SIKE – Supersingular Isogeny Key Encapsulation 9

SIMD – Single-Instruction-Multiple-Data 2, 2, 24, 24, 24

SLH-DSA – Stateless Hash-Based Digital Signature Algorithm 1, 2, 9, 10, 10

SNRU – Suma Normalizada de los Recursos Utilizados 39

SoC – System-on-Chip 25, 33, 35, 40

TFM – Trabajo Fin de Máster 1, 3, 3, 29, 40, 48, 79, 86

TLS – Transport Layer Security 11

TRNG – True Random-Number Generator 49, 49, 49, 49, 49, 65, 67, 69, 79, 79

UART – Universal Asynchronous Receiver-Transmitter 38, 44, 44, 59

VHDL – VHSIC Hardware Description Language [29](#), [55](#), [56](#), [57](#), [76](#)

VPN – Virtual Private Network [11](#)

WNS – Worst Negative Slack [32](#), [34](#), [34](#), [38](#), [67](#)

XMSS – eXtended Merkle Signature Scheme [10](#), [10](#)

XOF – Extendable-Output Function [21](#)

XTEA – Extended Tiny Encryption Algorithm [36](#), [36](#)

Capítulo 1

Introducción

En este capítulo se define el alcance del proyecto, analizando los antecedentes, el contexto de desarrollo y los objetivos definidos.

1.1. Antecedentes

El presente [Trabajo Fin de Máster \(TFM\)](#) se desarrolla como parte de la mención dual, en las prácticas con la empresa ARQUIMEA Research Center (ARC).

La criptografía es la base de la seguridad informática en la actualidad. Cuando se comparte información sensible en la red, es necesario asegurar su integridad y confidencialidad [1]. Existen distintos algoritmos criptográficos capaces de cumplir con esta función y que están altamente integrados en las comunicaciones actuales. Tradicionalmente, estos algoritmos se han considerado seguros, ya que exigen un alto nivel de computación para comprometer su seguridad, lo cual los ordenadores disponibles actualmente no son capaces de realizar. Sin embargo, los avances en el desarrollo de computadores cuánticos, junto con la existencia de algoritmos cuánticos como el algoritmo de Shor, suponen una amenaza potencial para la seguridad de los algoritmos criptográficos ampliamente utilizados en la actualidad, al abrir la posibilidad de resolver de forma eficiente los problemas matemáticos en los que se basa su seguridad [2].

En este contexto, surge la necesidad de desarrollar mecanismos criptográficos resistentes a ataques cuánticos. Existen dos aproximaciones, la criptografía post-cuántica y la criptografía cuántica. La criptografía post-cuántica destaca como una de las alternativas más relevantes y se está trabajando en su adopción [3].

En 2016, el [National Institute of Standards and Technology \(NIST\)](#) inició un proyecto para buscar nuevos algoritmos criptográficos que resultaran seguros tanto para los ordenadores clásicos como para los cuánticos [4]. Tras varias rondas de evaluación, análisis y trabajo que involucraron a la industria, el mundo académico y los gobiernos, en 2024 el NIST publicó los algoritmos candidatos elegidos para su estandarización. Los finalistas y actuales estándares son: [Module-Lattice-Based Key-Encapsulation Mechanism \(ML-KEM\)](#), [Module-Lattice-Based Digital Signature Algorithm \(ML-DSA\)](#) y [Stateless Hash-Based Digital Signature Algorithm \(SLH-DSA\)](#). Estos estándares definen respectivamente un mecanismo de encapsulación de claves y dos esquemas de firma digital.

En los próximos años será necesario llevar a cabo la migración a sistemas resistentes a ataques cuánticos [5], como muestra la [Figura 1](#). Esto plantea un nuevo reto especialmente para los sistemas con recursos limitados, como son los sistemas embebidos. Las implementaciones exclusivamente *software* de esquemas basados en retículos, códigos o funciones *hash* suelen presentar una mayor latencia computacional y un mayor tamaño de memoria o código en comparación con los algoritmos tradicionales. Por tanto, se vuelve necesario optar por soluciones *hardware* especializadas que permitan acelerar y optimizar la ejecución de estos algoritmos [6].

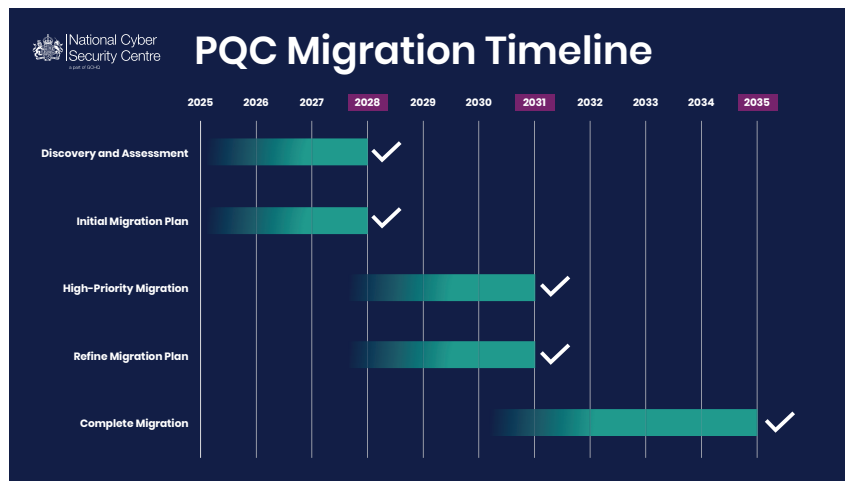


Figura 1: Plazos para la migración a criptografía post-cuántica [7].

En este contexto, la elección de la arquitectura de computación adquiere un papel clave. Las arquitecturas *open-source* están ganando especial relevancia debido a la versatilidad y adaptabilidad que ofrecen, como se refleja en distintos trabajos recientes.

En [6] se presenta una arquitectura de *Post-Quantum Cryptography* (PQC) denominada HORCRUX, que ofrece una extensión unificada del conjunto de instrucciones compatible con todos los algoritmos de PQC aprobados por el NIST. HORCRUX aborda la fragmentación actual en la compatibilidad *hardware*, donde las extensiones existentes suelen centrarse en algoritmos individuales o en subconjuntos limitados de esquemas PQC, y se centra en los núcleos comunes compartidos por ML-KEM, ML-DSA, SLH-DSA y *Hamming Quasi-Cyclic* (HQC). Para abordar los principales cuellos de botella computacionales de todos estos algoritmos (aritmética modular, multiplicación de matrices y transformaciones *hash*), la extensión introduce nuevas instrucciones RISC-V ejecutadas por un coprocesador estrechamente acoplado.

Ye et. al [8] presentan un procesador criptográfico eficiente basado en una estructura reticular con una instrucción *Single-Instruction-Multiple-Data* (SIMD) personalizada. La arquitectura SIMD propuesta permite la ejecución paralela eficiente de diversas operaciones polinómicas en modo de 256 bits y la aceleración de Keccak en modo de 320 bits, ambas reutilizando recursos de manera eficiente.

Fritzmann et al. [9] presentan RISQ-V, una arquitectura RISC-V mejorada que integra un conjunto de aceleradores estrechamente acoplados para acelerar la PQC basada en retículos. RISQ-V permite un uso eficiente de los recursos del procesador y reduce el número de accesos a la memoria, consiguiendo así aumentar el rendimiento global. Las principales contribuciones de este trabajo son: propuesta de un conjunto de aceleradores *hardware* integrados en el *pipeline* de RISC-V y ampliación de la *Instruction Set Architecture* (ISA) de RISC-V con 29 instrucciones nuevas para realizar de forma eficiente operaciones de criptografía basadas en retículos.

RISC-V destaca como una ISA [10] abierta y modular, respaldada a nivel internacional por la Fundación RISC-V [11], que permite su implementación en múltiples tecnologías, incluyendo ASICs (Application-Specific Integrated Circuits) y FPGA (Field-Programmable Gate Array). Su carácter extensible facilita la incorporación de extensiones de instrucciones específicas, lo que permite adaptar la arquitectura a necesidades concretas como la aceleración de algoritmos criptográficos avanzados.

A su vez, permite la introducción de extensiones de instrucciones que ayudan a personalizar las soluciones adaptadas a las nuevas necesidades. Como puede ser el caso de la criptografía post-cuántica, cuyo impacto en a nivel de ISA ya está siendo evaluado en un grupo de trabajo [12].

En este contexto, el presente trabajo se centra en el diseño y desarrollo de extensiones de instrucciones para arquitectura RISC-V orientadas a la aceleración de algoritmos de firma digital post-cuántica. De este modo, se pretende contribuir al desarrollo de soluciones eficientes y seguras para la implementación de criptografía post-cuántica en sistemas digitales.

1.2. Objetivos.

El objetivo principal de este TFM es investigar, diseñar y desarrollar una implementación de un juego de instrucciones personalizado en arquitectura RISC-V para un algoritmo de firma digital post-cuántica. En particular, se busca diseñar un acelerador en *hardware* usando instrucciones personalizadas en un núcleo RISC-V. Se pretende así reducir la carga computacional en *software* que requiere la ejecución de un algoritmo criptográfico. En concreto, se plantea aplicarlo a un algoritmo de firma digital post-cuántico.

Para conseguir este objetivo principal, se hace un estudio de arquitecturas RISC-V disponibles y posteriormente se selecciona una que permita la extensión de su juego de instrucciones. Además, se estudia y selecciona un algoritmo de firma digital post-cuántico para su implementación. En este punto se prueba la implementación de dicho algoritmo y se selecciona la parte que pueda ser acelerada mediante la extensión del juego de instrucciones.

Este objetivo general se puede desglosar en los siguientes objetivos operativos:

- O1 Estudiar la arquitectura RISC-V y su set de instrucciones.
- O2 Estudiar algoritmos de criptografía post-cuántica y seleccionar un algoritmo para su implementación.
- O3 Seleccionar y configurar la plataforma *hardware*.
- O4 Diseñar y desarrollar una extensión del juego de instrucciones para aplicaciones de criptografía post-cuántica.
- O5 Implementar y evaluar el rendimiento obtenido.
- O6 Documentar el trabajo realizado.

1.3. Organización de la memoria

En este capítulo inicial se define el alcance del proyecto, analizando los antecedentes que justifican su puesta en marcha, el contexto en el que desarrolla y los objetivos definidos. En el [Capítulo 2](#) se presenta la criptografía post-cuántica y se hace un repaso de otros conceptos clave para entender el contexto de desarrollo enmarcado en el estado actual de la técnica.

A continuación se profundiza en distintas tecnologías que permiten desarrollar el núcleo de este TFM. En el [Capítulo 3](#) se estudia la ISA de RISC-V, sentando las bases necesarias que permiten establecer la modularidad de su ISA. En el [Capítulo 4](#) se estudian los algoritmos de firma digital, profundizando en el estándar ML-DSA.

Finalmente se pasa a presentar el desarrollo realizado. En el [Capítulo 5](#) se realiza un análisis de posibles implementaciones RISC-V que permitan la extensión del juego de instrucciones. Tras

elegir un núcleo sobre el que trabajar, en el [Capítulo 6](#) se implementa una extensión del juego de instrucciones para la aceleración de un algoritmo de firma. En el [Capítulo 7](#) se presentan los resultados obtenidos. Para cerrar el documento, en el [Capítulo 8](#) se presentan las conclusiones del proyecto y se proponen líneas de trabajo futuras.

Capítulo 2

Criptografía post-cuántica

En este capítulo se da una visión general de la criptografía post-cuántica y para ello, antes se repasan otros conceptos clave de la criptografía clásica. También se analiza el concurso del [NIST](#) que ha sido clave para el desarrollo y estandarización de algoritmos post-cuánticos.

2.1. Introducción a la criptografía

La criptografía es la utilización de protocolos de cifrado para proteger la información que es transmitida o almacenada, asegurando que solo es accesible para aquellos que tienen permiso para descifrarla. En la actualidad, la criptografía se ha convertido en una herramienta crucial de ciberseguridad para proteger la información. Puede parecer que la criptografía es un resultado de la era tecnológica en la que nos encontramos, pero realmente su historia se remonta a la antigüedad [\[13\]](#). Un ejemplo es el cifrado César, que debe su nombre a Julio César, quien lo utilizaba en el siglo I a. C. para proteger el contenido de sus mensajes a los mensajeros que los transportaban. Este cifrado consistía en sustituir una letra del mensaje original por otra letra que se encuentra un número fijo de posiciones más adelante en el alfabeto. En la [Figura 2](#) se muestra un caso con desplazamiento de tres espacios.

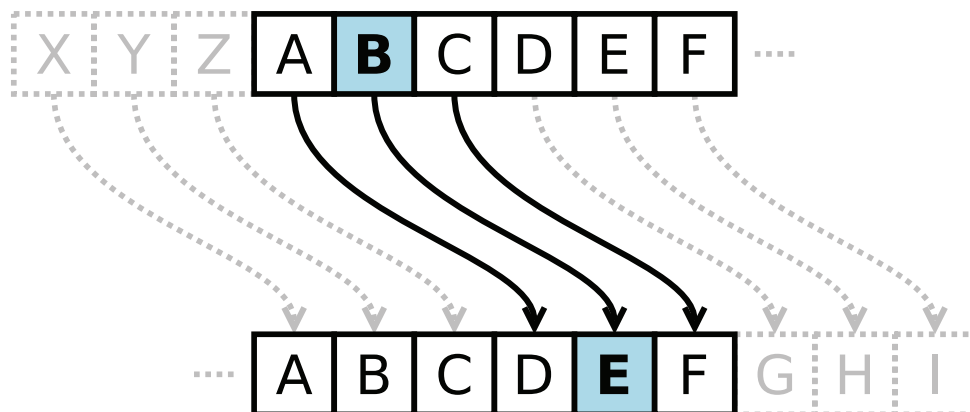


Figura 2: Cifrado César [\[14\]](#).

2.1.1. Criptografía clásica

La criptografía clásica incluye aquellos protocolos y métodos criptográficos usados actualmente que han sido diseñados para ser efectivos y seguros contra los ataques de ordenadores convencionales. Estos algoritmos se pueden diferenciar en dos grandes grupos: la criptografía simétrica y la criptografía asimétrica.

Los sistemas de cifrado simétricos o de clave privada son aquellos que usan una única clave para cifrar y descifrar el mensaje. En este caso, la clave es conocida por el emisor y el receptor. En la [Figura 3](#), se observa el proceso de cifrado y descifrado de un mensaje mediante una clave simétrica. Este tipo de cifrado destaca por su rapidez y eficiencia computacional, lo que lo hace adecuado para proteger grandes bloques de información, aunque requiere de un mecanismo seguro para el intercambio previo de la clave.

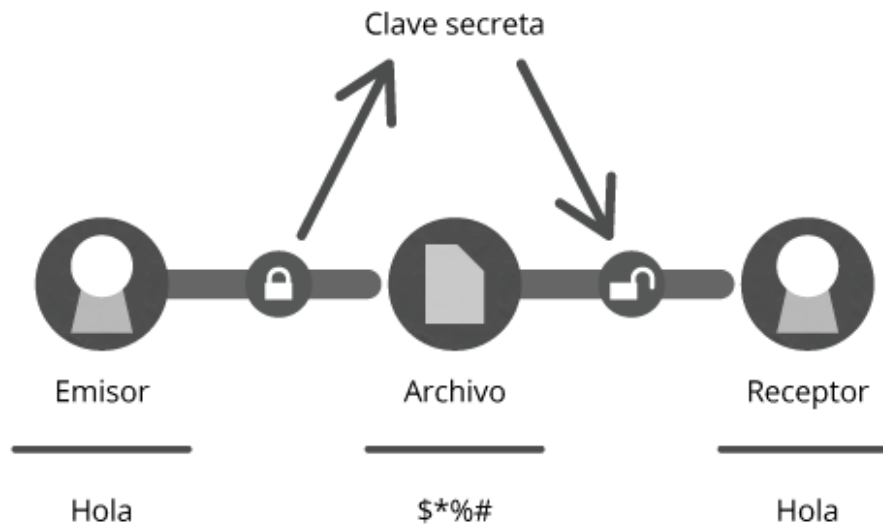


Figura 3: Criptografía simétrica [15].

Los sistemas de cifrado asimétricos o de clave pública son aquellos en los que se usan dos claves: una pública y otra privada. Ambas claves están relacionadas matemáticamente y se generan por el que receptor del mensaje, que distribuye la clave pública y guarda la clave secreta. Para compartir un mensaje cifrado, el emisor usa la clave pública para cifrar el mensaje, como se observa en la Figura 4. De este modo, el mensaje solo podrá ser descifrado por el receptor que posee la clave privada. Este método se utiliza también para la verificación de firmas digitales. En este caso, se utiliza la clave privada para firmar un mensaje o documento. Cualquier receptor que disponga de la clave pública podrá verificar la firma, comprobando la identidad del autor y la integridad del mensaje.

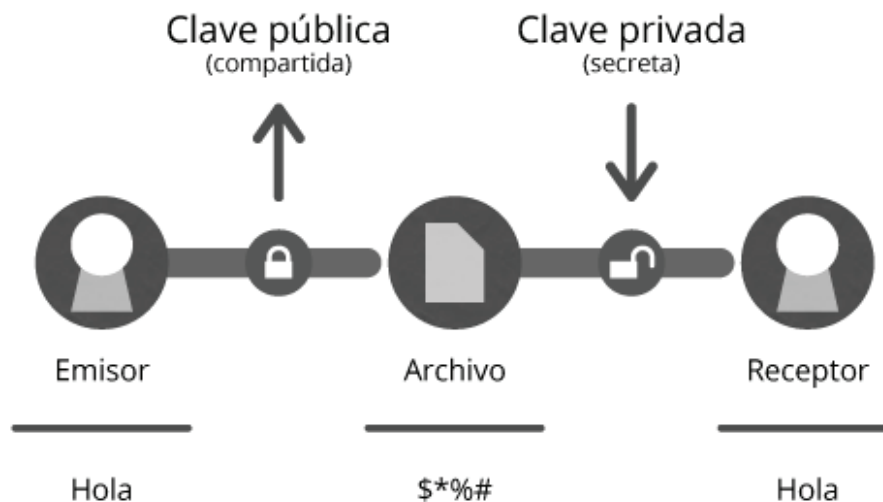


Figura 4: Criptografía asimétrica [15].

2.1.2. La amenaza de la computación cuántica

Un ordenador clásico funciona con bits, que pueden tener un valor de 0 o 1. Sin embargo, los ordenadores cuánticos utilizan bits cuánticos o qubits, cuyo valor puede ser un 0, un 1 o ambos

simultáneamente. De esta forma, un sistema con n qubits puede tener 2^n estados posibles al mismo tiempo. Este paralelismo es la ventaja principal de la computación cuántica, permitiendo evaluar múltiples combinaciones y soluciones a un problema complejo de forma simultánea.

Actualmente, los ordenadores cuánticos siguen bajo estudio, pero su desarrollo y construcción a gran escala están cada vez más cerca [16]. La capacidad computacional de estos ordenadores se estima que puede ser tal que consigan resolver problemas matemáticos complejos. Esto hace que los algoritmos criptográficos basados en sistemas matemáticos considerados hasta ahora seguros podrían dejar de serlo.

El algoritmo de Shor, propuesto por Peter Shor en 1994 [17], es una de las razones por las que los ordenadores cuánticos suponen una amenaza para la criptografía asimétrica. Este algoritmo es capaz de resolver de manera eficiente los problemas de factorización de números enteros grandes, una tarea que es difícil para los ordenadores clásicos.

La descomposición en factores puede tardar meses si se emplean los métodos actuales. En cambio, con el uso de un ordenador cuántico con la capacidad adecuada, la seguridad de algoritmos como *Elliptic Curve Cryptography* (ECC) y *Rivest-Shamir-Adleman* (RSA) se pone en riesgo, posibilitando que las claves se rompan en minutos. Esto tiene un impacto significativo en protocolos fundamentales, como sistemas de autenticación y firmas digitales, que bajo esas circunstancias quedan completamente desprotegidos.

Por otro lado, el algoritmo de Grover, desarrollado por Lov Grover en 1996 [18], supone una amenaza para los sistemas de criptografía simétrica. Este algoritmo obliga a aumentar el tamaño de las claves para mantener un nivel de seguridad frente a la utilización de computadores cuánticos.

Grover emplea una técnica cuántica para disminuir el número de iteraciones requeridas de forma polinómica, en vez de examinar todas las posibles cifras individualmente. Pese a que este algoritmo no es capaz de romper los sistemas de cifrado como el algoritmo de Shor, permite reducir significativamente el tiempo necesario para acceder a la información cifrada. Esto obliga a que métodos de cifrado, como puede ser AES, tengan que aumentar el tamaño de las claves para mantener la seguridad.

Los algoritmos presentados anteriormente pueden ser aplicados una vez que se implanten los *Cryptographically Relevant Quantum Computers* (CRQC) [19]. Este concepto hace referencia a aquellos ordenadores cuánticos que alcancen la madurez tecnológica y la potencia computacional suficientes para ejecutar ataques cuánticos de manera efectiva.

Sin embargo, existen otras amenazas para la criptografía actual, como pueden ser los ataques basados en almacenar información para descifrarla en el futuro. El concepto "*Harvest now, decrypt later*" recoge este tipo de ataques, basados en la adquisición y el almacenamiento a largo plazo de datos cifrados, actualmente no descifrables, a la espera de avances en la computación cuántica que permitan leerlos en el futuro [20], [21]. En situaciones en las que la confidencialidad a largo plazo es clave, como en las comunicaciones de defensa, sanitarias o gubernamentales, el nivel de amenaza es significativamente alto.

Frente a estas amenazas, existen dos alternativas criptográficas: la criptografía cuántica y la post-cuántica. En el presente trabajo, se centra en el estudio en la criptografía post-cuántica, ya que puede ser integrada en la tecnología actual.

2.2. Criptografía post-cuántica

La criptografía post-cuántica, en inglés [PQC](#), es una parte de la criptografía cuyos algoritmos buscan ser resistentes a ataques por ordenadores tradicionales u ordenadores cuánticos. Dichos algoritmos post-cuánticos están diseñados para ser ejecutados en los ordenadores actuales basados en transistores y en la infraestructura correspondiente, y no en ordenadores cuánticos [22]. Para ello, se estudian los problemas matemáticos que presentan resistencia cuántica. A continuación se presentan las familias de algoritmos matemáticos más utilizadas en la criptografía post-cuántica:

- **Basada en retículos** (*Lattice-based*). Se fundamenta en la dificultad geométrica de encontrar el vector más corto o más cercano en una estructura de red multidimensional compleja.
- **Basada en funciones *hash*** (*Hash-based*). Su seguridad no depende de problemas algebraicos complejos, sino de la resistencia a colisiones y preimágenes de funciones *hash* estandarizadas.
- **Basada en códigos de corrección de errores** (*Code-based*). Se basa en la dificultad de decodificar un mensaje lineal al que se le ha introducido ruido aleatorio.
- **Basada en ecuaciones multivariantes** (*Multivariate*). Basa su seguridad en la resolución de sistemas de ecuaciones polinómicas no lineales de múltiples variables sobre cuerpos finitos.
- **Basada en isogenias** (*Isogeny-based*). Emplea mapas algebraicos entre curvas elípticas supersingulares.

2.3. Proceso de estandarización del NIST

El *National Institute of Standards and Technology* creó un grupo de trabajo sobre [PQC](#) en 2012, con el objetivo de analizar los avances de la computación cuántica y las amenazas que esto supone para la tecnología. Durante los años previos, el volumen y la relevancia de las publicaciones académicas en torno a esta problemática habían experimentado un crecimiento, evidenciando la preocupación y necesidad de transicionar a nuevos sistemas criptográficos.

Por ello, en 2016 el [NIST](#) lanzó un concurso abierto para la selección de estándares de criptografía post-cuántica en dos categorías: *Digital Signature (DS)* y *Public-Key Encryption (PKE)/Key Encapsulation Mechanism (KEM)* [23]. Este concurso buscaba la colaboración de la comunidad académica e industrial del sector para desarrollar algoritmos criptográficos que sean seguros tanto frente a ordenadores cuánticos como clásicos. Estos algoritmos sustituirán a los sistemas criptográficos clásicos de clave pública.

En momento ya se habían propuesto varios sistemas criptográficos post-cuánticos, entre los que se incluyen los basados en retículos, los basados en códigos, los multivariantes, las firmas basadas en *hash* y otros. Sin embargo, en el caso de la mayoría de estas propuestas, era necesario seguir investigando para reforzar la confianza en su seguridad y para mejorar su rendimiento.

Este concurso se diferenciaba de otros propuestos anteriormente, como los realizados para la selección de [AES](#) o [SHA-3](#). En este caso, era poco probable que existiera un sustituto directo, siendo necesario el planteamiento de varios estándares de distintas tecnologías. Esto requería un proceso importante, con un esfuerzo significativo y varias rondas para desarrollar, normalizar e implementar nuevos sistemas criptográficos post-cuánticos.

En 2017 se cerró el plazo para entregar las primeras propuestas, dando comienzo a un proceso que se estructuró en varias rondas de evaluación y descarte continuado. La primera ronda comenzó con 69 propuestas aceptadas, aunque 5 de ellas fueron retiradas por la detección de vulnerabilidades por los autores, dejando 64 propuestas. Los algoritmos evaluados por familia fueron:

- 26 basados en retículos
- 19 basados en códigos de corrección de errores
- 9 basados en ecuaciones multivariables
- 10 basados en otros algoritmos (funciones *hash* y otros problemas matemáticos)

A lo largo de la segunda (2019) y la tercera ronda (2020), el número de candidatos se redujo a medida que la comunidad internacional identificaba debilidades estructurales, ataques teóricos o ineficiencias de rendimiento.

El primer grupo de algoritmos seleccionados para su estandarización se anunció en julio de 2022. Para el establecimiento de claves (**KEM**), el elegido fue el esquema basado en retículos CRYSTALS-Kyber. Para las firmas digitales, se seleccionaron CRYSTALS-Dilithium, FALCON y SPHINCS+. Finalmente, en agosto de 2024 se publicaron los primeros estándares *Federal Information Processing Standards* (FIPS) [4]: FIPS 203 (denominado **ML-KEM** y basado en Kyber), FIPS 204 (denominado **ML-DSA** y basado en Dilithium) y FIPS 205 (denominado **SLH-DSA** y basado en SPHINCS+). El estándar correspondiente a FALCON (previsto como *FFT over NTRU-Lattice-Based Digital Signature Algorithm* (FN-DSA)) continúa en proceso de adaptación para entornos con severas restricciones de ancho de banda. Como se puede observar, al publicar los estándares se unificaron los nombres para reflejar la familia criptográfica base para cada estándar.

Paralelamente, el **NIST** extendió el proceso a una cuarta ronda con el objetivo de estandarizar un **KEM** de respaldo fundamentado en una familia matemática alternativa a los retículos de Kyber, garantizando así una estrategia de defensa ante futuros ataques. Durante este escrutinio, algunas propuestas se descartaron de forma definitiva: el esquema *Supersingular Isogeny Key Encapsulation* (SIKE) (basado en isogenias) fue roto en 2022 mediante computación clásica en cuestión de horas, mientras que Classic McEliece (basado en códigos) no se seleccionó debido al tamaño excesivo de las claves públicas. Finalmente, este proceso paralelo concluyó con la selección del algoritmo **HQC** como el estándar de respaldo de la familia basada en códigos.

Es importante destacar que el **NIST** evaluó las implementaciones basándose en cinco niveles o categorías de seguridad. Estos niveles se definieron comparando el esfuerzo computacional necesario para romper el algoritmo con el esfuerzo para romper primitivas clásicas estandarizadas, yendo desde el Nivel 1 (equivalente a la seguridad de AES-128) hasta el Nivel 5 (equivalente a AES-256) [24].

A continuación, se analizan los algoritmos cuyos estándares ya han sido publicados formalmente.

2.3.1. FIPS 203: ML-KEM

ML-KEM es un mecanismo de encapsulación de claves (**KEM**) derivado de la propuesta CRYSTALS-Kyber [25]. Su propósito principal en el ecosistema criptográfico es establecer material de clave simétrica secreta entre dos partes a través de un canal público no seguro. El **NIST** lo ha posicionado como el algoritmo principal y de propósito general para el intercambio de claves gracias a su equilibrio entre seguridad y rendimiento computacional. El estándar especifica tres configuraciones

correspondientes a distintos niveles de seguridad: ML-KEM-512 (Nivel 1), ML-KEM-768 (Nivel 3) y ML-KEM-1024 (Nivel 5).

2.3.2. FIPS 204: ML-DSA

CRYSTALS-Dilithium es un esquema post-cuántico de firma digital estandarizado por el NIST en el FIPS 204, donde se denomina oficialmente *Module-Lattice-Based Digital Signature Algorithm (ML-DSA)* [26]. Está diseñado para proporcionar seguridad frente a ataques provenientes tanto de ordenadores clásicos como cuánticos, basándose, al igual que ML-KEM, en el problema matemático de retículos. El estándar contempla tres variantes: ML-DSA-44 (Nivel 2), ML-DSA-65 (Nivel 3) y ML-DSA-87 (Nivel 5).

2.3.3. FIPS 205: SLH-DSA

SLH-DSA [27], propuesto bajo el nombre SPHINCS+, es un algoritmo post-cuántico de firma digital basado en funciones *hash*. La gran ventaja frente a otros algoritmos basados en funciones *hash* es que no necesita recordar el estado de las firmas emitidas previamente. Esta característica elimina el riesgo de comprometer la clave privada por una reutilización accidental ante posibles fallos en el sistema. Aunque produce firmas de un tamaño significativamente mayor y requiere más ciclos de computación que ML-DSA, su seguridad depende únicamente de la resistencia a colisiones de las funciones *hash* subyacentes, ofreciendo seguridad en caso de que en el futuro se rompan los algoritmos basados en retículos.

2.3.4. NIST SP 800-208: LMS y XMSS

De forma previa a los estándares FIPS para la criptografía post-cuántica, el NIST ya había emitido la recomendación especial SP 800-208 centrada en esquemas de firma digital basados en *hash* con estado. En este documento se especifican el sistema de firmas de *Leighton-Micali Signature (LMS)* y el esquema de firma de *eXtended Merkle Signature Scheme (XMSS)* [28].

A diferencia de los algoritmos como ML-DSA o SLH-DSA, la seguridad de LMS y XMSS depende de que el firmante mantenga un control estricto sobre el estado de la clave privada, ya que la reutilización de un mismo estado comprometería por completo la seguridad de la firma. Debido a este requisito de gestión, no son idóneos para un uso generalizado, pero ofrecen un nivel de seguridad post-cuántica alto en entornos controlados. La *National Security Agency (NSA)* los recomienda para la firma y la verificación de *firmware* dentro de su suite de algoritmos *Commercial National Security Algorithm Suite (CNSA)* 2.0 [29].

2.4. Retos y oportunidades

La implantación de la criptografía post-cuántica se encuentra en pleno crecimiento y desarrollo. Este cambio presenta una serie de desafíos y abre nuevas oportunidades de innovación.

El proceso de estandarización liderado por el NIST ha supuesto un gran avance. Sin embargo, quedan algunos retos en la implementación de los algoritmos post-cuánticos:

- **Madurez del ecosistema.** Las implementaciones de esquemas criptográficos resistentes a ataques cuánticos son pocas. Aunque en los últimos años el número de publicaciones ha crecido significativamente, aún queda un gran trabajo para asegurar que estos sistemas sean resistentes a ataques mediante criptoanálisis o incluso algoritmos cuánticos novedosos que puedan surgir.

- **Tamaño de las claves.** Los algoritmos post-cuánticos, especialmente los basados en retículos y códigos, generalmente requieren de clave y de texto cifrado de gran tamaño. Esto implica una mayor carga computacional y mayor necesidad de recursos de almacenamiento, que pueden ser un problema crítico en algunos sistemas embebidos con recursos limitados.
- **Ataques de canal lateral.** Aunque un algoritmo sea matemáticamente seguro contra ataques cuánticos, su implementación física puede ser vulnerable. Los ataques de canal lateral pueden acceder a información sensible mediante el análisis de los consumos de potencia o las señales electromagnéticas que se producen en la ejecución de un algoritmo. No se ha probado que los estándares propuestos sean resistentes a este tipo de ataques.
- **Complejidad de la transición.** Se estima que para 2030 los sistemas críticos deberían completar la transición a la criptografía post-cuántica. No se trata de un cambio trivial, ya que requiere de modificaciones en los protocolos y el rediseño de algunos sistemas [30]. Como ejemplo histórico, casi veinte años después de la estandarización del algoritmo AES como reemplazo de la familia *Data Encryption Standard (DES)*, este último se sigue utilizando en la industria de pagos, en configuraciones antiguas de *Transport Layer Security (TLS)* y en redes *Virtual Private Network (VPN)* [31]. Esta dificultad para actualizar los sistemas criptográficos es una tendencia que podría repetirse.

Por otro lado, la necesidad de adopción de estos algoritmos está sirviendo para impulsar la investigación y da lugar a ciertas oportunidades:

- **Impulso de la investigación.** El concurso del NIST y la necesidad de encontrar implementaciones que puedan ser llevadas al mundo real provocan que la investigación en torno a los estándares criptográficamente seguros vaya en aumento. Esto ha sido clave, por ejemplo, para detectar fallos de seguridad de algunos algoritmos propuestos. El análisis constante asegura la detección temprana de vulnerabilidades, garantizando la seguridad a largo plazo de los nuevos estándares.
- **Implementación de *hardware* especializado.** La necesidad de implementar operaciones complejas está impulsando el desarrollo de nuevos aceleradores criptográficos en *hardware* y coprocesadores específicos.

2.5. Conclusión

La protección de la información ha sido una necesidad permanente a través de la historia. Es por ello que la criptografía adquiere un papel tan relevante en el desarrollo y la seguridad de los sistemas actuales. Pero ante la amenaza que representan los CRQC, la criptografía post-cuántica ha cobrado una gran relevancia.

En el presente capítulo se ha estudiado la evolución de los sistemas criptográficos, así como la búsqueda continua de algoritmos seguros frente a nuevas capacidades de ataque. Para comprender por qué se han desarrollado las diferentes familias de criptografía post-cuántica y cómo funcionan, es necesario entender estos conceptos clásicos.

El concurso del NIST ha impulsado la colaboración y trabajo de la comunidad científica e industrial, que concluye con la presentación de los primeros estándares de algoritmos post-cuánticos.

Sin embargo, la adopción de estos nuevos estándares es sólo el primer paso. Como se ha comentado, su integración en el mundo real presenta importantes retos, especialmente por el incremento en el tamaño de las claves, la carga computacional y la posible vulnerabilidad a ataques de canal lateral.

Capítulo 3

Arquitectura RISC-V

Este capítulo da una visión general de la arquitectura [ISA](#) de RISC-V, explicando tanto la arquitectura no privilegiada como la privilegiada.

3.1. Introducción

RISC-V es una [ISA](#) diseñada originalmente para apoyar a la investigación y la enseñanza en el ámbito de la arquitectura de computadores, pero que aspira a convertirse en una referencia en aplicaciones industriales [\[10\]](#). Este proyecto nació en la Universidad de Berkeley en el año 2010 y actualmente es gestionado por la Fundación RISC-V International [\[11\]](#) localizada en Suiza.

La [ISA](#) de RISC-V se ha definido evitando, en la medida de lo posible, detalles de implementación. Por ello, más que definir una estructura *hardware* concreta, define un conjunto de instrucciones, registros y tipos de datos que actúa como una interfaz o contrato abstracto entre el software y el *hardware*. Esto permite que cualquier procesador compatible ejecute el mismo programa compilado, independientemente de cómo esté diseñado internamente su circuito [\[32\]](#).

Generalmente se habla de la [ISA](#) de RISC-V, pero realmente RISC-V es una familia formada por cuatro [ISA](#) básicas relacionadas. Cada uno de estos conjuntos de instrucciones se caracteriza por el ancho de los bancos de registros, el tamaño correspondiente del espacio de direcciones y el número de registros. Existen dos variantes principales orientados a enteros, RV32I y RV64I, que proporcionan espacios de direcciones de 32 o 64 bits, respectivamente. RV32E es una variante del conjunto de instrucciones base RV32I que cuenta con la mitad de registros y se ha añadido para dar soporte a microcontroladores pequeños. Finalmente, se plantea una futura variante, RV128I, con un espacio de direcciones de 128 bits.

RISC-V ha sido diseñado para admitir una amplia personalización y especialización. Cada [ISA](#) base puede ampliarse con una o más extensiones opcionales del conjunto de instrucciones, las cuales se dividen en tres categorías distintas:

- Estándar: Definidas por la Fundación RISC-V. Cualquier otra extensión que no esté definida por la Fundación se considera no estándar.
- Reservadas: No están definidas actualmente. Se guardan para futuras extensiones estándar.
- Personalizadas: Las codificaciones personalizadas nunca se utilizarán para extensiones estándar y se ponen a disposición de extensiones no estándar específicas de cada fabricante.

3.2. ISA no privilegiada

En el diseño de la [ISA](#) no privilegiada [\[32\]](#), se eliminó cualquier dependencia de características microarquitecturales particulares, como el tamaño de la línea de caché o de detalles de la arquitectura privilegiada, como la traducción de páginas. Este enfoque aporta tanto simplicidad como la máxima flexibilidad para microarquitecturas alternativas o arquitecturas privilegiadas alternativas.

La **ISA** base se denomina «I» y contiene instrucciones de cómputo de enteros, instrucciones de carga y almacenamiento de enteros e instrucciones de flujo de control.

Para dar soporte a un desarrollo de *software* más general, se define un conjunto de extensiones estándar:

- M: Multiplicación y división de enteros estándar.
- A: Lectura, modificación y escritura atómica en memoria.
- F: Coma flotante de precisión simple.
- D: Coma flotante de precisión doble.
- C: Instrucciones comprimidas.

Por ejemplo, para referirse a una **ISA** específica que admita las extensiones M, A, F y D, las letras se concatenan para definir la arquitectura como RV32IMAFD. Este conjunto en particular también puede representarse mediante la letra «G»: RV32G. La letra G no representa una extensión de la **ISA** en sí, sino que significa General.

3.2.1. Formato de las instrucciones

En RV32I, las instrucciones se codifican utilizando palabras de 32 bits alineadas cada 4 bytes y en formato *little-endian*. Dentro de RV32I, existen seis formatos principales de instrucciones, que incluyen cuatro formatos básicos y dos formatos derivados (B y J), como se observa en la **Figura 5**.

- Tipo R (Register): Utilizado para operaciones aritméticas y lógicas con registros.
- Tipo I (Immediate): Utilizado para operaciones entre un registro y un inmediato de 12 bits.
- Tipo S (Store): Utilizado para instrucciones de almacenamiento.
- Tipo B (Branch): Utilizado para instrucciones de bifurcación condicional. Emplea un inmediato de 13 bits reorganizado para realizar un direccionamiento relativo al PC.
- Tipo U (Upper): Utilizado para instrucciones con inmediatos grandes de 20 bits (como LUI y AUIPC).
- Tipo J (Jump): Utilizado para saltos incondicionales.

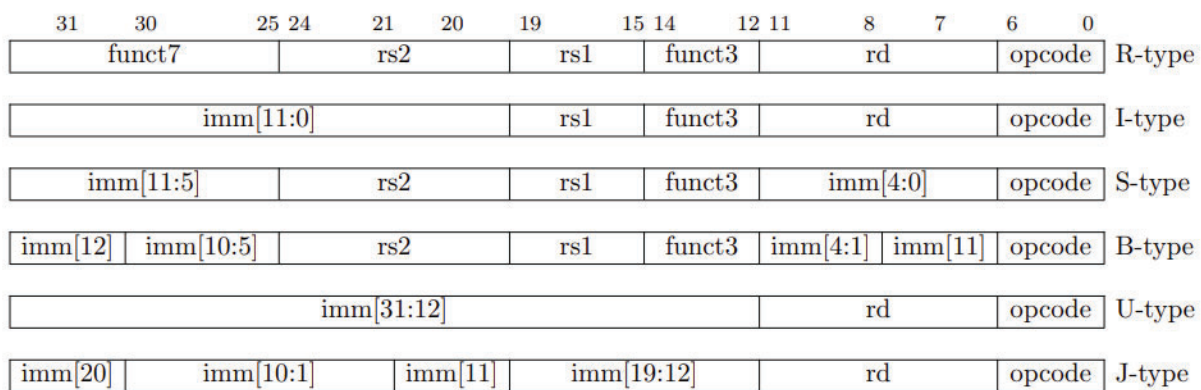


Figura 5: Formatos de las instrucciones RISC-V.

Puede observarse que los campos se mantienen en la misma posición en los distintos formatos. El código de *major opcode* se encuentra siempre al inicio, en las posiciones [0:6], seguido por los distintos registros: rd (registro destino), rs1 (registro fuente 1) y rs2 (registro fuente 2), los cuales, si

se utilizan, se ubican en [7:11], [15:19] y [20:24], respectivamente. Por último, se encuentran funct3 y funct7 que, junto con el *opcode*, codifican de manera jerárquica qué instrucción se va a ejecutar (*opcode* → funct3 → funct7). Los campos de los inmediatos se sitúan siempre hacia el final (lado izquierdo) de la instrucción y se codifican de forma diferente según el tipo de instrucción.

El *opcode* principal en RV32I siempre comenzará con los bits 11 en las posiciones [1:0]. El resto del *opcode* se codifica siguiendo la [Tabla 1](#).

Tabla 1: Codificación del código de operación principal en RISC-V

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	<i>custom-0</i>	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	<i>custom-1</i>	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	<i>reserved</i>	<i>custom-2/rv128</i>	48b
11	BRANCH	JALR	<i>reserved</i>	JAL	SYSTEM	<i>reserved</i>	<i>custom-3/rv128</i>	≥ 80b

Por ejemplo, la instrucción «add x1, x2, x3» puede desglosarse como una instrucción de tipo R como la que se muestra en la [Tabla 2](#).

Tabla 2: Ejemplo de una instrucción de suma

Bits	[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
Campo	funct7	rs2	rs1	funct3	rd	opcode
Binario	0000000	00011	00010	000	00001	0110011
Equivalente	–	x3	x2	–	x1	add

3.2.2. Registros e Interfaz Binaria de Aplicación

La [ISA](#) base también especifica los 32 registros de la CPU, los cuales tienen todos un ancho de 32 bits, además del contador de programa (program counter). El único registro especial es x0, cuyo valor siempre se lee como 0, tal como ya se ha implementado en muchas [ISA RISC](#) anteriores.

Aunque a nivel de *hardware* todos los registros están disponibles para uso general, la interoperabilidad entre diferentes piezas de *software* requiere establecer reglas de uso. Es aquí donde entra en juego la [Interfaz binaria de aplicación \(ABI\)](#) [33].

Mientras que la [ISA](#) define cómo se comunica el *software* con el *hardware*, la [ABI](#) de RISC-V define cómo las herramientas de *software* se comunican entre sí a nivel de código máquina.

De este modo, la [ABI](#) asigna un «nombre simbólico» y un propósito específico a cada uno de los 32 registros genéricos. En la [Tabla 3](#) se muestran los registros, su alias según la [ABI](#) y la descripción de su función.

Tabla 3: Convención de registros de enteros en RV32I

Registro	Nombre ABI	Descripción
x0	zero	Cero
x1	ra	Dirección de retorno
x2	sp	Puntero de pila (<i>stack pointer</i>)
x3	gp	Puntero global
x4	tp	Puntero de hilo (<i>thread pointer</i>)
x5–x7	t0–t2	Registros temporales
x8–x9	s0–s1	Registros guardados
x10–x17	a0–a7	Registros de argumentos
x18–x27	s2–s11	Registros guardados
x28–x31	t3–t6	Registros temporales

3.3. ISA privilegiada

Mientras que la ISA no privilegiada especifica los aspectos visibles para el programador de aplicaciones, la ISA privilegiada [34] define la interfaz necesaria para dar soporte a sistemas operativos, entornos de ejecución protegidos y la gestión directa del *hardware*. Su objetivo fundamental es garantizar la seguridad, el aislamiento entre procesos y el manejo correcto de los recursos críticos del sistema.

3.3.1. Modos de ejecución

RISC-V organiza la ejecución del software en diferentes modos de ejecución, también conocidos como niveles de privilegio. El nivel de privilegio determina a qué recursos *hardware* y zonas de memoria tiene acceso el código que se está ejecutando en cada instante. La especificación define tres modos principales, como muestra la Tabla 4.

Tabla 4: Niveles de privilegio en RISC-V

Nivel	Codificación	Nombre	Abreviatura
0	00	Usuario / Aplicación	U
1	01	Supervisor	S
2	10	<i>Reservado</i>	–
3	11	Máquina	M

- **Modo Máquina (M).** Es el nivel de privilegio más alto y el único obligatorio en cualquier implementación de RISC-V. Tiene acceso total y sin restricciones al *hardware*. Es el modo en el que se ejecuta el *firmware* de arranque y los monitores de ejecución segura.

- **Modo Supervisor (S).** Diseñado específicamente para dar soporte a sistemas operativos de propósito general, como puede ser Linux. Permite gestionar la memoria virtual, la paginación y la protección de memoria necesaria para aislar los procesos.
- **Modo Usuario (U).** Es el nivel de menor privilegio, donde se ejecutan las aplicaciones de usuario. El código en este modo no puede acceder directamente a los recursos del sistema ni modificar la configuración del *hardware*. Para ello, debe solicitar servicios al sistema operativo mediante llamadas al sistema.

Dependiendo de la complejidad y de las necesidades de seguridad, las implementaciones de RISC-V pueden combinar estos modos en distintas configuraciones de sistema [35]. En la [Figura 6](#) se ilustran las tres jerarquías de ejecución soportadas:

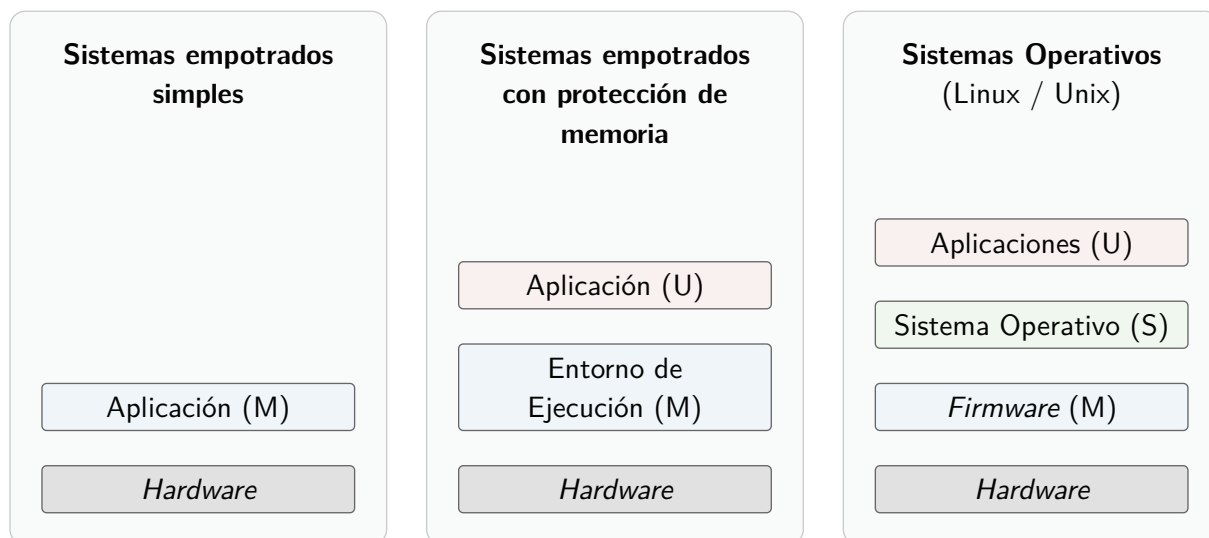


Figura 6: Combinaciones de modos de privilegio.

Para garantizar el aislamiento y la seguridad, el flujo de ejecución no puede cambiar libremente entre estos niveles. Generalmente, se utiliza la siguiente regla:

- **Para elevar privilegios.** El *software* debe desencadenar una excepción controlada utilizando la instrucción `ecall`.
- **Para reducir privilegios.** El procesador utiliza las instrucciones de retorno de excepción correspondientes, como `mret` para salir del modo máquina o `sret` para salir del modo supervisor.

3.3.2. Registros CSR

Para gestionar todas las funciones privilegiadas, RISC-V introduce un espacio de direcciones completamente separado de los 32 registros de propósito general, compuesto por los [Registros de Control y Estado \(CSR\)](#).

Estos registros permiten configurar, supervisar y controlar el comportamiento interno del procesador, como por ejemplo, habilitar o deshabilitar interrupciones, configurar temporizadores, etc. El acceso a estos registros está restringido según el modo de privilegio. Si un programa en modo U intenta leer o escribir en un [CSR](#) exclusivo del modo M, el *hardware* bloqueará la operación.

3.3.3. Manejo de excepciones e interrupciones (*Traps*)

Una parte esencial de la ISA privilegiada es el mecanismo de transferencia de control protegido, denominado de forma genérica como *traps*. Este mecanismo se encarga de gestionar dos tipos de eventos que alteran el flujo normal de ejecución:

- **Excepciones.** Eventos síncronos inusuales causados directamente por la ejecución de una instrucción, como una división por cero, un acceso a una dirección de memoria no válida o una instrucción ilegal.
- **Interrupciones.** Eventos asíncronos externos al núcleo del procesador, como la señal de un periférico, la llegada de datos por una interfaz de red o el desbordamiento de un temporizador *hardware*.

Cuando ocurre un *trap*, el procesador suspende la ejecución del programa actual, almacena el estado y la dirección de la instrucción actual en los CSR correspondientes, y eleva el nivel de privilegio, generalmente al modo M o S, para saltar de forma segura a una rutina de *trap handler*.

3.4. Conclusión

En este capítulo se presenta un análisis de la arquitectura de RISC-V y sus principales variantes de la ISA. Además, se describen los formatos de instrucciones base, modos de ejecución y el manejo de las interrupciones. Este análisis será la base para el desarrollo de los siguientes capítulos.

Capítulo 4

Estudio y análisis de un algoritmo de firma digital post-cuántico

En este capítulo se estudia la firma digital y posteriormente se profundiza en el análisis del estándar [ML-DSA](#).

4.1. Algoritmos de firma digital

Una firma digital es el equivalente electrónico de una firma manuscrita [\[36\]](#). Concretamente, este mecanismo criptográfico se utiliza para garantizar tres propiedades de seguridad de la información [\[37\]](#):

- autenticidad: el supuesto firmante es quien realmente originó la información,
- no repudio: el firmante no puede negar haber firmado la información,
- integridad: asegura que los datos no han sido modificados tras su firma.

Un algoritmo de firma digital incluye necesariamente dos fases, que se ejecutan de forma secuencial:

1. un proceso de generación de la firma, y
2. un proceso de verificación de la firma.

El firmante es el único que posee la clave privada, usada para firmar. Por otro lado, el verificador o receptor utiliza la clave pública durante el proceso de verificación para comprobar la autenticidad de la firma. El propietario del par de claves es la única entidad autorizada a utilizar la clave privada para generar firmas digitales. En la [Figura 7](#) se muestra un diagrama de bloques con el proceso de firma digital. En el procedimiento mostrado en dicho diagrama se hace un *hash* del mensaje antes de su firma. Cabe destacar que este paso no está contemplado en todos los estándares de firmas digitales.

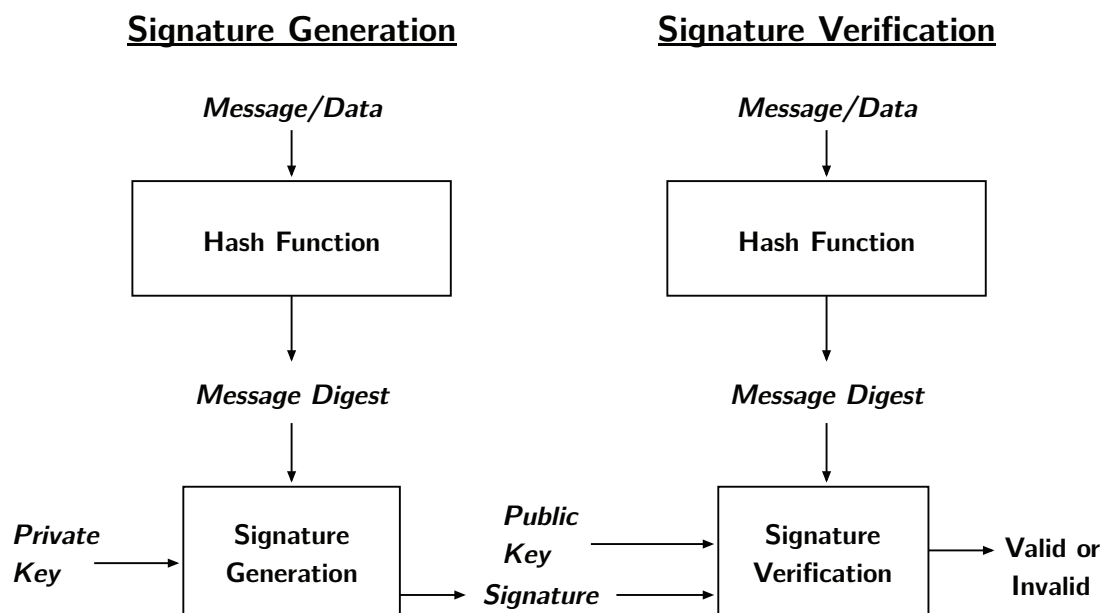


Figura 7: Proceso de firma digital [36].

4.2. ML-DSA

ML-DSA es un sistema de firma digital basado en CRYSTALS-DILITHIUM, que ha sido estandarizado por el **NIST** como **FIPS 204** [26]. Dilithium es un esquema de firma digital que ofrece una gran seguridad frente a los ataques de mensaje elegido, basado en la dificultad de los problemas de retículos.

Un ataque de mensaje elegido es una técnica de criptoanálisis donde un atacante puede obtener firmas digitales de mensajes específicos de su elección, acceder a un oráculo de firma y buscar debilidades en esquemas de autenticación. El funcionamiento del ataque es como sigue:

1. **Acceso al oráculo.** El atacante interactúa con un sistema que firma datos legítimos usando una clave secreta.
2. **Selección de datos.** El atacante elige de forma deliberada qué textos o bloques enviar para que sean firmados.
3. **Observación de resultados.** Analiza los pares de mensajes y firmas resultantes para intentar deducir la clave privada o falsificar firmas futuras.

Este concepto de seguridad implica que un adversario que tenga acceso a un oráculo de firma no puede generar una firma de un mensaje cuya firma aún no haya visto, ni generar una firma diferente de un mensaje que ya haya visto firmado [38], [39]. Este esquema se diseñó teniendo en cuenta los siguientes criterios:

- **Fácil de implementar de forma segura.** Los esquemas de firma basados en retículos más compactos utilizan muestreo gaussiano discreto, que conlleva vulnerabilidades frente a ataques de canal lateral. Por ello, Dilithium únicamente utiliza muestreo uniforme. Además, todas las demás operaciones (como la multiplicación de polinomios y el redondeo) se pueden implementar en tiempo constante.

- **Reducir al mínimo el tamaño de la clave pública y la firma.** Dado que muchas aplicaciones requieren la transmisión tanto de la clave pública como de la firma, para reducir latencias se busca minimizar la suma de estos parámetros.
- **Estructura modular y escalabilidad de seguridad.** Las dos operaciones que constituyen la parte principal de los procedimientos de firma y verificación son la *Extendable-Output Function (XOF)*, para las que utiliza SHAKE-128 y SHAKE-256, y la multiplicación del anillo polinómico.

El estándar *ML-DSA* especifica tres conjuntos de parámetros. Estos parámetros definen algunos valores que son utilizados para la generación de claves, la firma y la verificación (*Tabla 5*). Los nombres de los conjuntos de parámetros tienen la forma «*ML-DSA- $k\ell$* », donde (k, ℓ) son las dimensiones de la matriz A . En este contexto, A no está compuesta por números o escalares simples, sino que representa una matriz de k filas y ℓ columnas cuyos elementos son polinomios definidos sobre un anillo matemático específico. Esta matriz forma la estructura matemática central sobre la que operan las claves del algoritmo.

Con este enfoque, el nivel de seguridad no se describe mediante una única cifra, como por ejemplo «128 bits de seguridad». En su lugar, se afirma que cada conjunto de parámetros de *ML-DSA* es, como mínimo, tan seguro como un cifrado por bloques genérico con un tamaño de clave predeterminado o como una función *hash* genérica con una longitud de salida predeterminada. Concretamente, se afirma que el conjunto de parámetros *ML-DSA-44* está en la categoría de fortaleza de seguridad 2 (comparable a AES-128), *ML-DSA-65* está en la categoría 3 (comparable a AES-192) y *ML-DSA-87* está en la categoría 5 (comparable a AES-256).

Tabla 5: Tamaños en bytes de las claves y las firmas de ML-DSA

	Clave privada	Clave pública	Tamaño de firma
ML-DSA-44	2560	1312	2420
ML-DSA-65	4032	1952	3309
ML-DSA-87	4896	2592	4627

Este estándar utiliza las funciones SHAKE256 y SHAKE128, como se definen en *FIPS 202* [40]. Aunque la norma *FIPS 202* especifica estas funciones como entradas y salidas de bits, la mayoría de las implementaciones tratan las entradas y salidas como bytes [26, Sec. 3.7].

ML-DSA siempre llama a estas funciones con una longitud de salida que sea un múltiplo de ocho bits y trata la salida de estas funciones como una cadena de bytes. Por ello, en el documento se definen las funciones H y G como $H(\text{str}, \ell) = \text{SHAKE256}(\text{str}, 8\ell)$ y $G(\text{str}, \ell) = \text{SHAKE128}(\text{str}, 8\ell)$.

La ejecución de *ML-DSA* se divide principalmente en tres funciones: generación de claves, procedimiento de firma y verificación. A continuación se explica el funcionamiento de cada una de ellas.

Generación de Claves

Se utiliza un *Random Number Generator (RNG)* para generar una semilla aleatoria ξ de 32 bytes. Esta semilla no se puede reutilizar y debe ser totalmente aleatoria. Como se observa en la

Figura 8, a partir de esta semilla, se genera una matriz A de dimensiones $k \times \ell$, donde cada una de sus entradas es un polinomio en el anillo R_q . Posteriormente, el algoritmo muestrea vectores de clave privada aleatorios s_1 y s_2 . Cada coeficiente de estos vectores es un elemento de R_q con coeficientes pequeños de tamaño máximo η . Finalmente, la segunda parte de la clave pública se calcula como $t = As_1 + s_2$. Todas las operaciones algebraicas en este esquema se asumen sobre el anillo de polinomios R_q .

Algorithm 6 ML-DSA.KeyGen_internal(ξ)

Generates a public-private key pair from a seed.

Input: Seed $\xi \in \mathbb{B}^{32}$

Output: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen}(q-1)-d)}$

and private key $sk \in \mathbb{B}^{32+32+64+32 \cdot ((\ell+k) \cdot \text{bitlen}(2\eta)+dk)}$.

- 1: $(\rho, \rho', K) \in \mathbb{B}^{32} \times \mathbb{B}^{64} \times \mathbb{B}^{32} \leftarrow H(\xi || \text{IntegerToBytes}(k, 1) || \text{IntegerToBytes}(\ell, 1), 128)$
 - 2: ▷ expand seed
 - 3: $\hat{A} \leftarrow \text{ExpandA}(\rho)$ ▷ A is generated and stored in NTT representation as $\hat{A}$
 - 4: $(s_1, s_2) \leftarrow \text{ExpandS}(\rho')$
 - 5: $t \leftarrow \text{NTT}^{-1}(\hat{A} \circ \text{NTT}(s_1)) + s_2$ ▷ compute $t = As_1 + s_2$
 - 6: $(t_1, t_0) \leftarrow \text{Power2Round}(t)$ ▷ compress t
 - 7: ▷ PowerTwoRound is applied componentwise (see explanatory text in Section 7.4)
 - 8: $pk \leftarrow \text{pkEncode}(\rho, t_1)$
 - 9: $tr \leftarrow H(pk, 64)$
 - 10: $sk \leftarrow \text{skEncode}(\rho, K, tr, s_1, s_2, t_0)$ ▷ K and tr are for use in signing
 - 11: **return** (pk, sk)
-

Figura 8: Algoritmo para la generación de claves [26].

Procedimiento de Firma

El algoritmo de firma genera un vector de polinomios de enmascaramiento y con coeficientes menores a γ_1 . Luego, el firmante calcula $w = Ay$ y establece w_1 como los bits de «orden superior» de los coeficientes en este vector. En particular, cada coeficiente w en el vector puede escribirse de forma canónica como $w = w_1 \cdot 2\gamma_2 + w_0$; w_1 es entonces el vector que comprende todos los bits de orden superior. El desafío c se crea entonces como el *hash* del mensaje y w_1 . La salida c es un polinomio en R_q con exactamente τ valores ± 1 y el resto 0. La razón de esta distribución es que c tiene una norma pequeña y proviene de un dominio de tamaño $\binom{n}{\tau} 2^\tau$, el cual se busca que esté entre 128 y 256.

La firma potencial se calcula entonces como $z = y + cs_1$. Si z se emitiera directamente en este punto, el esquema de firma sería inseguro debido a que la clave secreta se filtraría. Para evitar la dependencia de z con respecto a la clave secreta, se utiliza el *rejection sampling*. El parámetro β se establece como el máximo coeficiente posible de cs_1 . Dado que c tiene τ valores ± 1 y el coeficiente máximo en s_i es η , es fácil ver que $\beta \leq \tau \cdot \eta$. Si cualquier coeficiente de z es mayor que $\gamma_1 - \beta$, entonces se rechaza y se reinicia el procedimiento de firma. Además, si cualquier coeficiente de los bits de orden inferior de $w - cs_2$ es mayor que $\gamma_2 - \beta$, se reinicia. La primera comprobación es necesaria para la seguridad, mientras que la segunda es necesaria tanto para la seguridad como para la corrección. El bucle *while* en el procedimiento de firma se sigue repitiendo hasta que se satisfagan las dos condiciones anteriores, como se presenta en el algoritmo de la Figura 9.

Algorithm 7 ML-DSA.Sign_internal(sk, M', rnd)

Deterministic algorithm to generate a signature for a formatted message M' .

Input: Private key $sk \in \mathbb{B}^{32+32+64+32 \cdot ((\ell+k) \cdot \text{bitlen}(2\eta)+dk)}$, formatted message $M' \in \{0, 1\}^*$, and per message randomness or dummy variable $rnd \in \mathbb{B}^{32}$.

Output: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell \cdot 32 \cdot (1+\text{bitlen}(\gamma_1-1))+\omega+k}$.

```
1:  $(\rho, K, tr, s_1, s_2, t_0) \leftarrow \text{skDecode}(sk)$ 
2:  $\hat{s}_1 \leftarrow \text{NTT}(s_1)$ 
3:  $\hat{s}_2 \leftarrow \text{NTT}(s_2)$ 
4:  $\hat{t}_0 \leftarrow \text{NTT}(t_0)$ 
5:  $\hat{A} \leftarrow \text{ExpandA}(\rho)$   $\triangleright A$  is generated and stored in NTT representation as  $\hat{A}$ 
6:  $\mu \leftarrow H(\text{BytesToBits}(tr)||M', 64)$   $\triangleright$  message representative that may optionally be
   computed in a different cryptographic module
7:  $\rho'' \leftarrow H(K||rnd||\mu, 64)$   $\triangleright$  compute private random seed
8:  $\kappa \leftarrow 0$   $\triangleright$  initialize counter  $\kappa$ 
9:  $(z, h) \leftarrow \perp$ 
10: while  $(z, h) = \perp$  do  $\triangleright$  rejection sampling loop
11:    $y \in R_q^\ell \leftarrow \text{ExpandMask}(\rho'', \kappa)$ 
12:    $w \leftarrow \text{NTT}^{-1}(\hat{A} \circ \text{NTT}(y))$ 
13:    $w_1 \leftarrow \text{HighBits}(w)$   $\triangleright$  signer's commitment
14:    $\triangleright$  HighBits is applied componentwise (see explanatory text in Section 7.4)
15:    $\tilde{c} \leftarrow H(\mu||w_1\text{Encode}(w_1), \lambda/4)$   $\triangleright$  commitment hash
16:    $c \in R_q \leftarrow \text{SampleInBall}(\tilde{c})$   $\triangleright$  verifier's challenge
17:    $\hat{c} \leftarrow \text{NTT}(c)$ 
18:    $\langle\langle cs_1 \rangle\rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{s}_1)$ 
19:    $\langle\langle cs_2 \rangle\rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{s}_2)$ 
20:    $z \leftarrow y + \langle\langle cs_1 \rangle\rangle$   $\triangleright$  signer's response
21:    $r_0 \leftarrow \text{LowBits}(w - \langle\langle cs_2 \rangle\rangle)$ 
22:    $\triangleright$  LowBits is applied componentwise (see explanatory text in Section 7.4)
23:   if  $\|z\|_\infty \geq \gamma_1 - \beta$  or  $\|r_0\|_\infty \geq \gamma_2 - \beta$  then  $(z, h) \leftarrow \perp$   $\triangleright$  validity checks
24:   else
25:      $\langle\langle ct_0 \rangle\rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{t}_0)$ 
26:      $h \leftarrow \text{MakeHint}(-\langle\langle ct_0 \rangle\rangle, w - \langle\langle cs_2 \rangle\rangle + \langle\langle ct_0 \rangle\rangle)$   $\triangleright$  Signer's hint
27:      $\triangleright$  MakeHint is applied componentwise (see explanatory text in Section 7.4)
28:     if  $\|\langle\langle ct_0 \rangle\rangle\|_\infty \geq \gamma_2$  or the number of 1's in  $h$  is greater than  $\omega$ , then  $(z, h) \leftarrow \perp$ 
29:     end if
30:   end if
31:    $\kappa \leftarrow \kappa + \ell$   $\triangleright$  increment counter
32: end while
33:  $\sigma \leftarrow \text{sigEncode}(\tilde{c}, z \bmod^\pm q, h)$ 
34: return  $\sigma$ 
```

Figura 9: Algoritmo para el procedimiento de firma [26].

Verificación

El verificador recibe la firma z junto con el desafío c . En este punto, calcula w_1' como los bits de orden superior de $Az - ct$, el cual coincidirá con el valor original w_1 , como refleja la Figura 10.

El verificador acepta la firma como válida si se cumplen simultáneamente las siguientes condiciones: que todos los coeficientes del vector de firma z son estrictamente menores que el límite de seguridad $\gamma_1 - \beta$ y que al aplicar la función *hash* sobre el mensaje y el vector reconstruido w'_1 , el resultado coincide exactamente con el desafío c recibido en la firma.

Algorithm 8 *ML-DSA.Verify_internal*(pk, M', σ)

Internal function to verify a signature σ for a formatted message M' .

Input: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen}(q-1)-d)}$ and message $M' \in \{0, 1\}^*$.

Input: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell \cdot 32 \cdot (1+\text{bitlen}(\gamma_1-1))+\omega+k}$.

Output: Boolean

```

1:  $(\rho, t_1) \leftarrow \text{pkDecode}(pk)$ 
2:  $(\tilde{c}, z, h) \leftarrow \text{sigDecode}(\sigma)$   $\triangleright$  signer's commitment hash  $\tilde{c}$ , response  $z$ , and hint  $h$ 
3: if  $h = \perp$  then return false  $\triangleright$  hint was not properly encoded
4: end if
5:  $\hat{A} \leftarrow \text{ExpandA}(\rho)$   $\triangleright$   $A$  is generated and stored in NTT representation as  $\hat{A}$ 
6:  $tr \leftarrow H(pk, 64)$ 
7:  $\mu \leftarrow (H(\text{BytesToBits}(tr) || M', 64))$   $\triangleright$  message representative that may optionally be
   computed in a different cryptographic module
8:  $c \in R_q \leftarrow \text{SampleInBall}(\tilde{c})$   $\triangleright$  compute verifier's challenge from  $\tilde{c}$ 
9:  $w'_{\text{Approx}} \leftarrow \text{NTT}^{-1}(\hat{A} \circ \text{NTT}(z) - \text{NTT}(c) \circ \text{NTT}(t_1 \cdot 2^d))$   $\triangleright w'_{\text{Approx}} = Az - ct_1 \cdot 2^d$ 
10:  $w'_1 \leftarrow \text{UseHint}(h, w'_{\text{Approx}})$   $\triangleright$  reconstruction of signer's commitment
11:  $\triangleright$  UseHint is applied componentwise (see explanatory text in Section 7.4)
12:  $\tilde{c}' \leftarrow H(\mu || w1\text{Encode}(w'_1), \lambda/4)$   $\triangleright$  hash it; this should match  $\tilde{c}$ 
13: return  $[\|z\|_\infty < \gamma_1 - \beta]$  and  $[\tilde{c} = \tilde{c}']$ 

```

Figura 10: Algoritmo para la verificación [26].

4.3. Análisis del estado del arte

En los siguientes apartados, se presentan distintas implementaciones de *ML-DSA* identificadas en la literatura reciente. Estas implementaciones sirven de referencia para una mejor comprensión de las distintas técnicas de optimización empleadas.

A Highly-efficient Lattice-based Post-Quantum Cryptography Processor for IoT Applications (2024)

Ye *et al.* [8] presentan un procesador para criptografía basada en retículos sobre una arquitectura RISC-V con instrucciones personalizadas *SIMD*. El diseño, evaluado sobre una FPGA Xilinx Artix-7, se centra en la optimización de esquemas como *ML-KEM* y *ML-DSA*.

Partiendo del núcleo RISC-V CV32E40P, se adapta su arquitectura para operar como un procesador *Dual-Issue*, permitiendo la ejecución simultánea de instrucciones, como cargas de memoria y operaciones aritméticas. Con ello se persigue mantener el cauce de ejecución ocupado durante las operaciones de firma, intensivas en cómputo. En la Figura 11 se muestra la arquitectura *SIMD* propuesta.

Una parte fundamental del diseño es la ampliación del archivo de registros *SIMD*, que ha sido modificado para poder procesar simultáneamente varios coeficientes de 32 bits. Esto resulta crítico para paralelizar las operaciones matemáticas vectoriales de *ML-DSA*. Para acompañar esta mejora,

se desarrolló una **Unidad Aritmético-Lógica (ALU)** especializada que ejecuta multiplicaciones de Montgomery y operaciones *butterfly* de la *Number Theoretic Transform (NTT)* en un único ciclo de reloj.

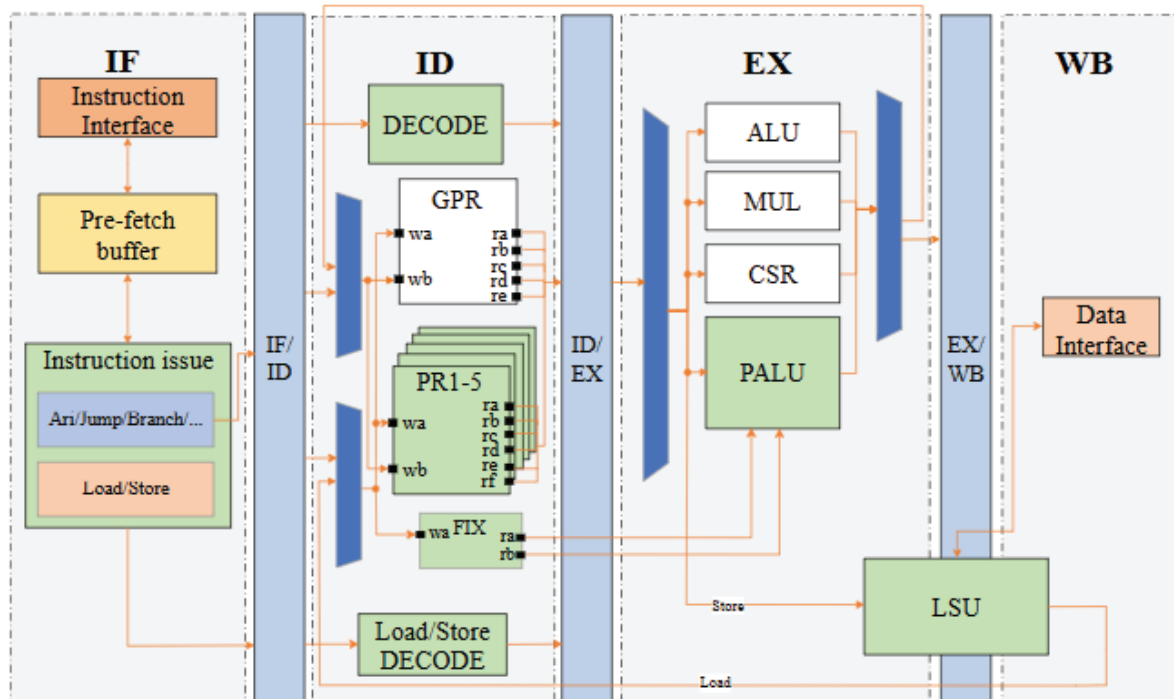


Figura 11: Arquitectura SIMD del procesador propuesta por Ye et al. [8].

Por otra parte, para evitar cuellos de botella en la reordenación de coeficientes y el flujo de datos, se modifican las extensiones de carga y almacenamiento. Estas unidades logran mover los datos polinómicos entre la memoria principal y los registros de forma eficiente, reduciendo notablemente los tiempos de espera del procesador.

Optimized Hardware-Software Co-Design for Kyber and Dilithium on RISC-V SoC FPGA (2024)

En este trabajo, Wang et al. [41] presentan un co-diseño *hardware/software* implementado sobre la plataforma *System-on-Chip (SoC) Field-Programmable Gate Array (FPGA)* Microchip PolarFire SoC, que integra múltiples núcleos RISC-V SiFive. La estrategia de los autores se basa en una aceleración *hardware Tightly-Coupled*, combinando el rendimiento de aceleradores dedicados con optimizaciones de código a bajo nivel. En la Figura 12 se muestra el subsistema utilizado.

Dado que **ML-DSA** consume una enorme cantidad de tiempo en la función *hash* SHAKE para la fase de muestreo, el diseño incorpora un núcleo Keccak *hardware* liberando al procesador RISC-V de esta carga. En paralelo, se diseñó un acelerador **NTT** que soporta diferentes módulos primos, permitiendo alternar entre las operaciones de **ML-KEM** y **ML-DSA** sin necesitar una reconfiguración del *hardware*.

Para disminuir las latencias de comunicación, se implementa una interfaz de memoria entre el procesador y la **FPGA**, garantizando que el núcleo RISC-V no tenga ciclos de parada mientras espera los resultados polinómicos.

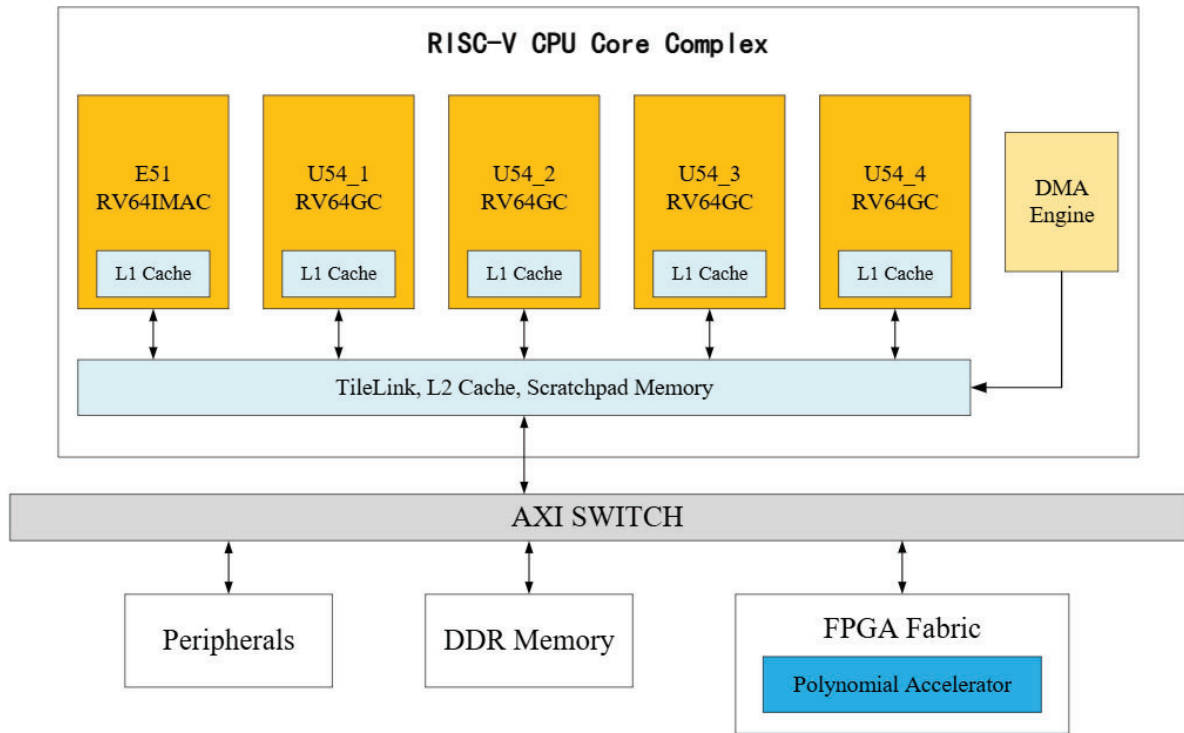


Figura 12: Subsistema RISC-V integrado con acelerador de hardware [41].

A Low-Latency Polynomial Arithmetic Unit for ML-KEM and ML-DSA Standards (2025)

Nguyen et al. [42] introducen una arquitectura basada en una unidad aritmética dedicada que funciona como acelerador *hardware*, que se muestra en la Figura 13. El principio de este diseño es la delegación de las operaciones computacionalmente complejas (NTT, NTT inversa y multiplicación punto a punto) al acelerador, mientras que la lógica de alto nivel y el flujo de control de ML-DSA, como el muestreo de rechazo y la gestión de firmas, se mapean sobre el procesador.

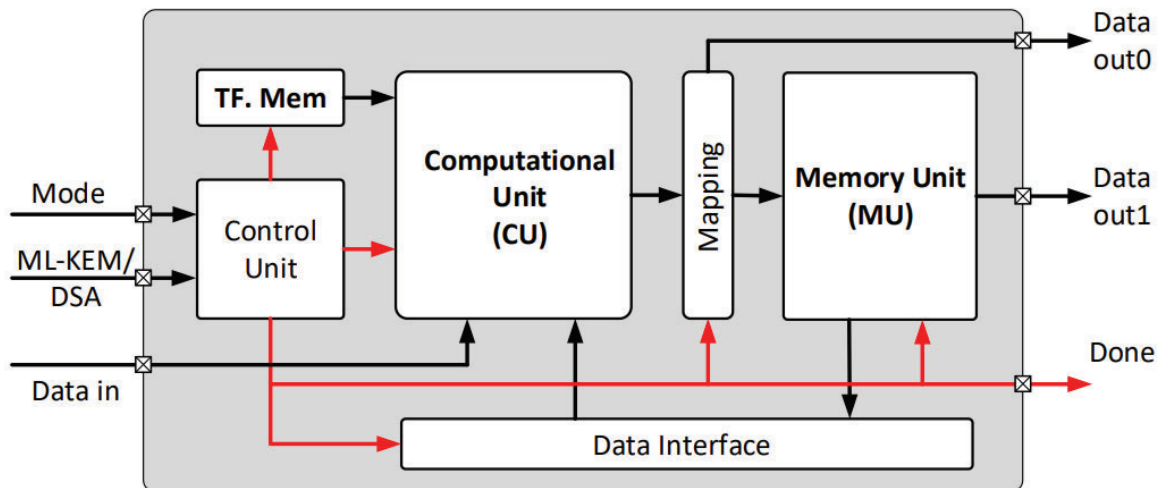


Figura 13: Arquitectura propuesta para la aceleración de ML-DSA y ML-KEM [42].

Debido a que ML-DSA requiere operaciones NTT considerablemente más exigentes que ML-KEM, el núcleo computacional fue dotado de un flujo de datos multi-NTT configurable. En concreto, la unidad soporta una configuración 4x1 que permite procesar cuatro operaciones *butterfly* en paralelo, logrando una reducción drástica en la latencia durante el proceso de firma. Asimismo, para mitigar el cuello de botella que supone el acceso a memoria para obtener los factores de rotación, el trabajo introduce un método de reutilización de dichos factores entre las etapas de NTT y las multiplicaciones punto a punto, reduciendo la sobrecarga de accesos.

Por otra parte, los autores abordan la necesidad de reordenar los coeficientes al moverse entre el dominio NTT y el dominio normal. Para evitar los ciclos de parada, se diseñó una unidad de memoria configurable capaz de realizar este reordenamiento *on-the-fly*, optimizando el uso de los recursos.

An Area-Efficient ML-DSA Accelerator With Interleaved and Dynamic Execution (2025)

Pu et al. [43] presentan un acelerador *hardware* para ML-DSA enfocado en lograr una alta eficiencia en área, que se observa en la Figura 14. Evaluado sobre una FPGA Xilinx Artix-7, el coprocesador emplea una estrategia de múltiples dominios de reloj que maximiza el rendimiento individual de cada módulo. Su aportación principal radica en la introducción de técnicas de ejecución dinámica y entrelazada para elevar el nivel de paralelismo y ocultar las latencias.

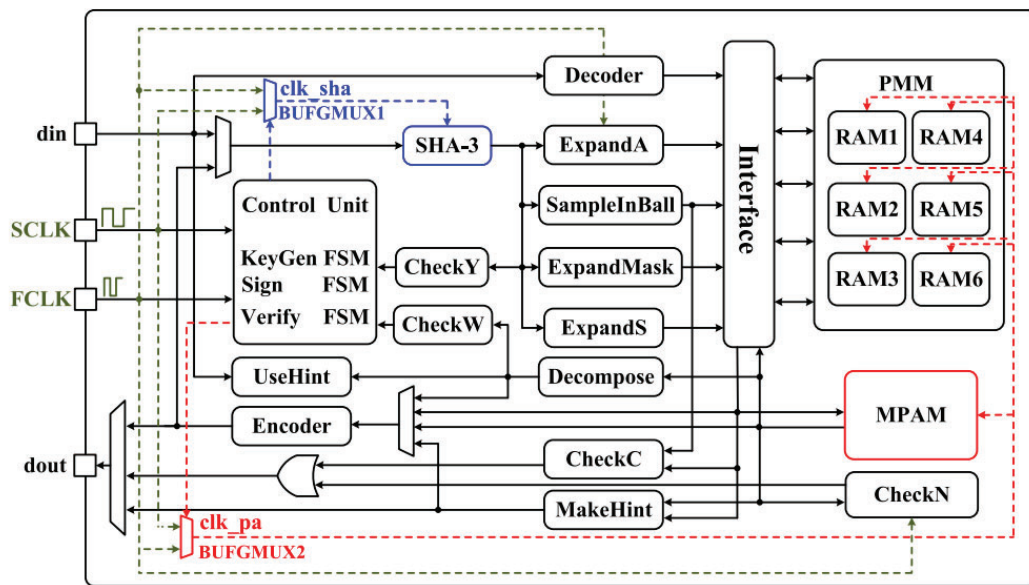


Figura 14: Arquitectura a alto nivel del sistema ML-DSA.[43].

A diferencia de las implementaciones secuenciales clásicas donde el procesador aguarda a que el módulo Keccak finalice el muestreo antes de iniciar la NTT, esta arquitectura propone una ejecución entrelazada. El sistema es capaz de comenzar las transformaciones matemáticas del primer conjunto de coeficientes simultáneamente a la generación del siguiente conjunto por parte del muestreador. Para evitar colisiones en este acceso concurrente, se diseñó un patrón de memoria de doble puerto optimizado que se aprovecha del ancho de banda de las memorias de bloque (BRAM) de la FPGA.

Además, el diseño cuenta con una asignación dinámica de recursos. La memoria interna y las unidades de cálculo se reasignan en función de si el sistema está ejecutando una operación de generación de firma o de verificación. Esta reutilización evita la duplicación de rutas estáticas en el *hardware*, incrementando significativamente la eficiencia de área al maximizar el ciclo de trabajo de la lógica combinacional.

4.4. Conclusión

En este capítulo se han revisado las propiedades de seguridad garantizadas por los algoritmos de firma digital (no repudio, autenticidad e integridad), así como su funcionamiento a alto nivel.

Tras este análisis, se realizó un estudio del esquema [ML-DSA](#), uno de los algoritmos de firma post-cuánticos estandarizados por el [NIST](#). La ejecución de [ML-DSA](#) se divide en tres funciones: generación de claves, procedimiento de firma y verificación.

Por último, se han analizado varias implementaciones de [ML-DSA](#) en el estado del arte, destacando las optimizaciones realizadas siguiendo distintas alternativas de implementación. En todos los casos se ha visto la necesidad de disponer de bloques con un amplio ancho de banda de acceso a memoria.

Capítulo 5

Implementación de procesadores con arquitectura RISC-V

En este capítulo se presenta el análisis de diversas implementaciones de procesadores RISC-V para posteriormente elegir una de estas implementaciones.

5.1. Estudio previo

Con el objetivo de seleccionar un núcleo RISC-V para el desarrollo del presente TFM, se ha realizado una evaluación de distintas implementaciones existentes. Para garantizar la viabilidad del proyecto, se han establecido los siguientes criterios para la selección de candidatos:

1. **Naturaleza de código abierto.** El diseño del procesador debe tener una licencia libre que permita su uso académico.
2. **Documentación.** Es importante que la implementación seleccionada cuente con una documentación técnica clara y abundante que detalle la arquitectura, el mapa de memoria y otros aspectos de diseño claves para su uso.
3. **Lenguaje de descripción hardware (HDL) estandarizado.** El código fuente debe estar desarrollado en lenguajes conocidos y utilizados en la industria, como es el caso de *VHSIC Hardware Description Language (VHDL)* o Verilog/SystemVerilog.
4. **Extensibilidad del juego de instrucciones.** La arquitectura interna debe poseer un diseño modular que permita la extensión del juego de instrucciones, objetivo de este trabajo.
5. **Diseño orientado a FPGA.** El diseño debe ser fácilmente sintetizable e integrable en dispositivos FPGA.

En la Tabla 6 se presentan los núcleos RISC-V evaluados. Como se puede observar, todos cuentan con licencia *open source* y opción de extender el juego de instrucciones. Teniendo en cuenta el criterio del HDL, se pueden descartar algunos de los *cores*.

Tabla 6: Núcleos RISC-V analizados

Núcleo	Licencia	Lenguaje HDL	Target	Extensibilidad	Documentación
PicoRV32[44]	ISC	Verilog	FPGA	Alta	Buena
Rocket[45]	BSD-3	Chisel	ASIC/FPGA	Alta	Buena
Ibex[46]	Apache 2.0	SystemVerilog	ASIC/FPGA	Alta	Excelente
NEORV32[47]	BSD-3	VHDL	FPGA	Alta	Excelente
VexRiscv[48]	MIT	SpinalHDL	FPGA	Alta	Buena


```

void putc(char c)
{
    *(volatile char*)0x10000000 = c;
}

void puts(const char *s)
{
    while (*s) putc(*s++);
}

void *memcpy(void *dest, const void *src, int n)
{
    while (n) {
        n--;
        ((char*)dest)[n] = ((char*)src)[n];
    }
    return dest;
}

volatile int i;
void main() {
    while(1){
        putc (0xFF);
        for(i=0; i<2000000; i++ );
        putc (0x00);
        for(i=0; i<2000000; i++ );
    }
}

```

Código 1: Software C para el parpadeo de los leds de la placa en las pruebas del núcleo PicoRV32

Este programa de prueba se compila mediante la orden `make firmware.hex` (Código 2), y al probar su funcionamiento se detecta que únicamente se enciende el led que indica un *trap*. Para solucionar este problema, se modifica el archivo Makefile del *firmware* para incluir las opciones del compilador `-march=rv32im` y `-mabi=ilp32`. Estas opciones obligan al compilador a generar de forma exclusiva instrucciones RV32IM, lo que garantiza la total compatibilidad con el *hardware* y evita que el procesador se bloquee.

```

make firmware.hex
riscv32-unknown-elf-gcc -Os -ffreestanding -nostdlib -o firmware.elf firmware.S
firmware.c \
    --std=gnu99 -Wl,-Bstatic,-T,firmware.lds,-Map,firmware.map,--strip-debug
-lgcc
/opt/riscv/riscv32-unknown-elf/bin/ld: warning: firmware.elf has a LOAD segment
with RWX permissions
riscv32-unknown-elf-objcopy -O binary firmware.elf firmware.bin
python3 ../../firmware/makehex.py firmware.bin 4096 > firmware.hex

```

Código 2: Compilación del firmware del núcleo PicoRV32

Como se puede observar en la [Tabla 7](#), los resultados de la implementación destacan por el bajo uso de recursos lógicos. El núcleo requiere 1379 *Look Up Table* (LUT), lo que representa un

1.03% de la capacidad total de la [FPGA](#) utilizada, y 770 registros. Además, al tener activada la extensión de multiplicación se utilizan 4 bloques [Digital Signal Processing \(DSP\)](#).

Tabla 7: Utilización de recursos lógicos tras la implementación del núcleo PicoRV32 (con extensión M y multiplicador rápido)

Recurso Lógico	Utilizados	Disponibles	Utilización (%)
Slice LUTs	1379	133800	1.03
LUT as Logic	1335	133800	1.00
LUT as Memory	44	46200	0.10
Slice Registers	770	267600	0.29
Register as Flip Flop	770	267600	0.29
DSP Blocks	4	740	0.54

En cuanto a los resultados temporales del núcleo PicoRV32, en la [Figura 16](#) se muestra el *timing report* correspondiente al camino crítico del diseño tras la implementación con la herramienta Vivado en modo *batch*. Como puede observarse, el diseño se implementó para una frecuencia de 100 MHz, que corresponde a un período de reloj de 10 ns. A esta frecuencia, el camino crítico presenta un [Worst Negative Slack \(WNS\)](#) de 2.438 ns, lo que indica que no existen violaciones temporales y que el diseño dispone de margen adicional antes de incumplir las restricciones de temporización. A partir de este dato, la frecuencia máxima teórica puede estimarse como $f_{\max} = \frac{1}{(T_{\text{period}} - \text{WNS})}$, donde T_{period} es el período de reloj objetivo. Sustituyendo los valores, se obtiene una $f_{\max}$ aproximada de 132 MHz.

```
Timing Report
Slack (MET) : 2.438ns (required time - arrival time)
Source:      picorv32_core/latched_stalu_reg/C
              (rising edge-triggered cell FDRE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
Destination: picorv32_core/reg_next_pc_reg[30]/D
              (rising edge-triggered cell FDRE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
Path Group:  sys_clk_pin
Path Type:   Setup (Max at Slow Process Corner)
Requirement: 10.000ns (sys_clk_pin rise@10.000ns - sys_clk_pin rise@0.000ns)
Data Path Delay: 7.641ns (logic 2.220ns (29.053%) route 5.421ns (70.947%))
Logic Levels: 9 (CARRY4=7 LUT4=1 LUT6=1)
Clock Path Skew: 0.053ns (DCD - SCD + CPR)
Destination Clock Delay (DCD): 5.397ns = ( 15.397 - 10.000 )
Source Clock Delay (SCD): 5.537ns
Clock Pessimism Removal (CPR): 0.193ns
Clock Uncertainty: 0.035ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
Total System Jitter (TSJ): 0.071ns
Total Input Jitter (TIJ): 0.000ns
Discrete Jitter (DJ): 0.000ns
Phase Error (PE): 0.000ns
```

Figura 16: Timing Report de la implementación del PicoRV32

Finalmente, para completar el análisis de la implementación, en la [Tabla 8](#) se resume el consumo de potencia estimado por la herramienta Vivado. El diseño presenta un consumo total estimado de 180 mW, donde la mayor parte corresponde a la potencia estática o de fuga del dispositivo. Dentro del consumo dinámico, el núcleo picorv32_core tiene un consumo muy reducido de 19 mW.

Tabla 8: Estimación del consumo de potencia de la implementación del sistema PicoRV32

Parámetro de Potencia	Consumo Estimado (mW)
Potencia Estática del Dispositivo	151
Potencia Dinámica Total	29
Núcleo PicoRV32	19
Potencia Total Estimada	180

Teniendo en cuenta los datos obtenidos en la implementación, se puede afirmar que el PicoRV32 destaca por usar pocos recursos, tener una $f_{\max}$ alta y un consumo de potencia muy reducido.

Sin embargo, se establece en la documentación que tanto la interfaz de memoria como el manejo de las interrupciones y otros aspectos de diseño son particulares de la implementación realizada y no cumplen con las especificaciones de RISC-V.

5.3. Ibex

Ibex [46] es un núcleo de procesador RISC-V de 32 bits de código abierto desarrollado en SystemVerilog. Ibex fue diseñado originalmente por el equipo PULP de la ETH de Zúrich y la Universidad de Bolonia bajo el nombre de Zero-riscy. En diciembre de 2018, Ibex se cedió a lowRISC, organización que desde entonces se encarga de su mantenimiento y desarrollo.

Este núcleo es altamente parametrizable y resulta idóneo para aplicaciones de control en sistemas embebidos. A nivel de arquitectura, Ibex soporta las extensiones del conjunto de instrucciones I, E, M, C y B. Su arquitectura está detallada en el diagrama de bloques de la Figura 17.

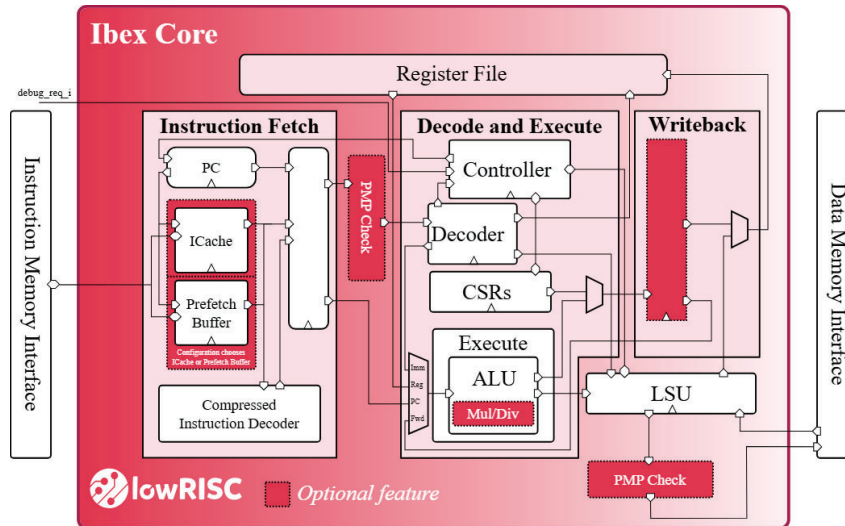


Figura 17: Arquitectura del procesador Ibex [46].

Para probar este procesador se ha usado el SoC Ibex Demo System [51], un sistema de ejemplo que integra el núcleo Ibex junto a interfaces de comunicación y control (UART, GPIO, PWM, Timers y SPI) y un soporte para depuración (Figura 18).

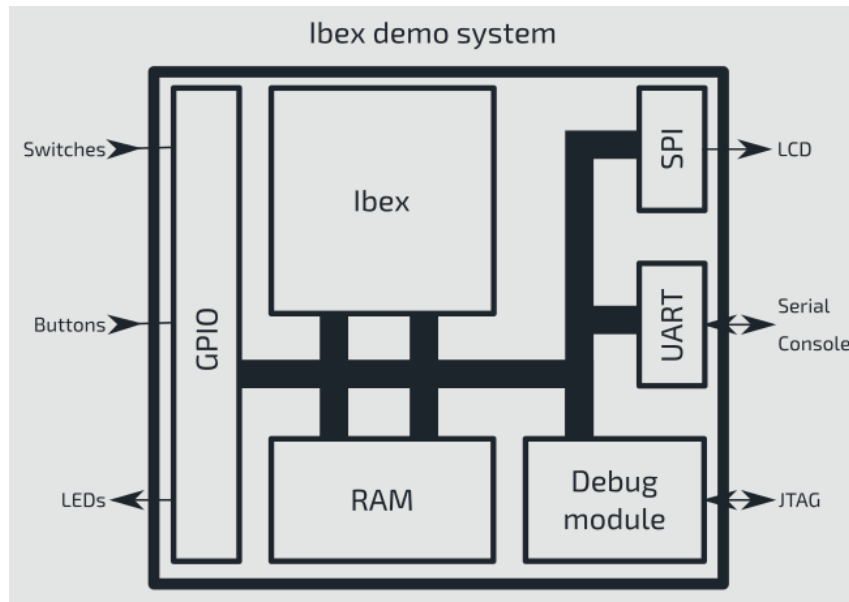


Figura 18: Diagrama de bloques del Ibex Demo System [51].

Igual que para el núcleo anterior, es necesario realizar distintas modificaciones para realizar la implementación de este sistema sobre la placa de prototipado Nexys Video a la versión original distribuida:

1. Añadir un nuevo archivo de restricciones al directorio /data: pins_nexys_video.xdc.
2. Añadir un nuevo *top-level wrapper* a /rtl/fpga: top_nexys_video.sv .
3. Actualizar el archivo ibex_demo_system.core para incluir los archivos anteriores y definir el *synthesis target* para la Nexys Video.

Como se refleja en la Tabla 9, el núcleo Ibex requiere 3236 LUT (2.40%) y 1367 registros (0.51%). Además, esta configuración de Ibex hace uso de 13 bloques DSP, de los cuales 4 están asociados a la unidad de multiplicación y división.

Tabla 9: Utilización de recursos lógicos del núcleo Ibex .

Recurso Lógico	Utilizados	Disponibles	Utilización (%)
Slice LUT	3236	134600	2.40
LUT as Logic	3236	134600	2.40
LUT as Memory	0	46200	0.00
Slice Registers	1367	269200	0.51
Register as Flip Flop	1367	269200	0.51
DSP Blocks	13	740	1.76

En la Figura 19, se observa el *timing report* donde se cumple con los tiempos de *setup* y *hold* para una frecuencia de 50 MHz. En este caso, el WNS obtenido (1.004 ns) refleja un margen positivo. No obstante, al incrementar la frecuencia objetivo a 100 MHz, el WNS decae drásticamente a -8.007

ns, incumpliendo las restricciones temporales. Esto se debe, principalmente, a que la arquitectura está pensada para diseños *Application-Specific Integrated Circuit (ASIC)* y no contempla, a priori, optimizaciones orientadas a alcanzar altas frecuencias en arquitecturas *FPGA* en concreto.

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 1.004 ns	Worst Hold Slack (WHS): 0.106 ns	Worst Pulse Width Slack (WPWS): 3.000 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 11750	Total Number of Endpoints: 11750	Total Number of Endpoints: 4791

All user specified timing constraints are met.

Figura 19: Timing Report de la implementación del Ibex Demo System .

Finalmente, para completar el análisis, en la Tabla 10 se muestra el consumo de potencia estimado por la herramienta. El diseño del *Ibex Demo System* presenta un consumo total de 366 mW, donde 153 mW se atribuyen a la potencia estática del dispositivo. Respecto a la potencia dinámica (213 mW), el núcleo Ibex tiene un consumo de 17 mW.

Tabla 10: Estimación del consumo de potencia de la implementación del Ibex Demo System.

Parámetro de Potencia	Consumo Estimado (mW)
Potencia Estática del Dispositivo	153
Potencia Dinámica Total	213
Gestor de reloj (PLL)	122
Memoria RAM	64
Núcleo Ibex	17
Potencia Total Estimada	366

5.4. NEORV32

NEORV32 [47] es un procesador de código abierto compatible con la arquitectura RISC-V. El procesador está pensado para funcionar como controlador auxiliar en diseños de *SoC* de mayor tamaño o como microcontrolador compacto y personalizado. Tiene una estructura modular que permite incluir las unidades necesarias según la aplicación, como muestra la Figura 20.

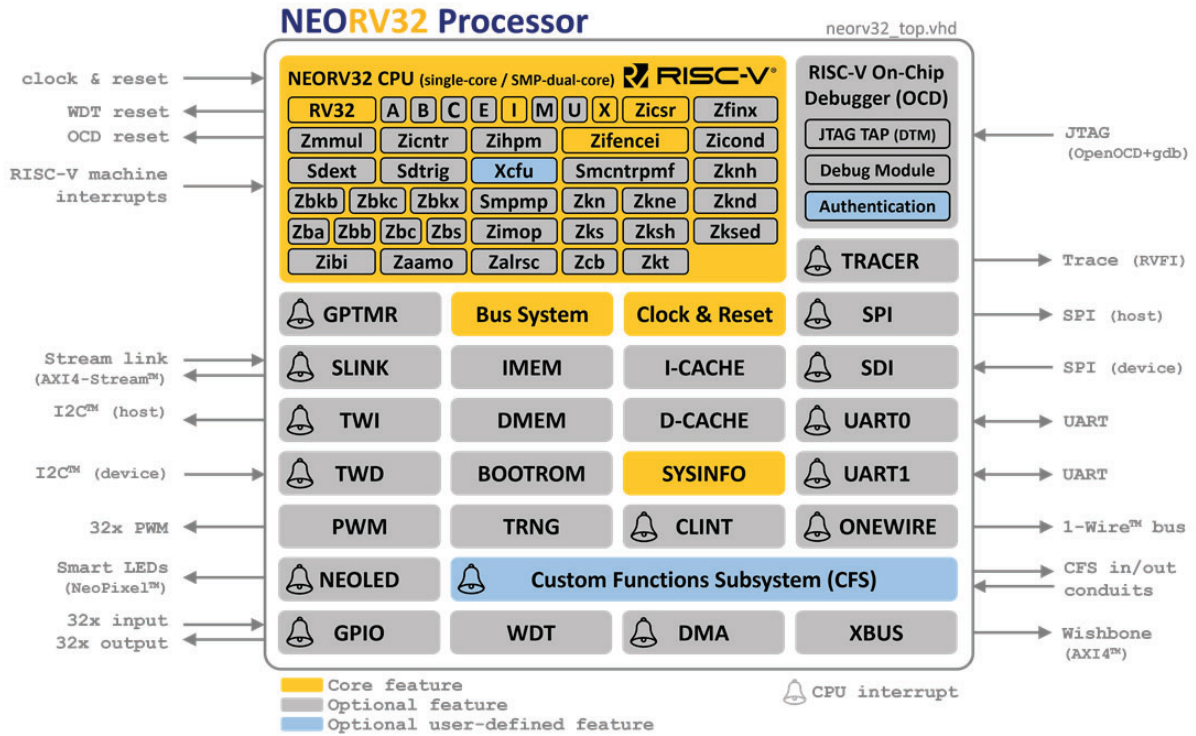


Figura 20: Diagrama de bloques del procesador NEORV32.

Este núcleo destaca por la alta calidad de su documentación y su diseño modular, facilitando su configuración con aquellas extensiones que sean necesarias. Además, cuenta con una *Custom Functions Unit* (CFU) que es una extensión de la ISA específica para NEORV32 (Xcfu) que proporciona un módulo *hardware* genérico que puede utilizarse para implementar instrucciones RISC-V personalizadas [47].

El repositorio de NEORV32 proporciona múltiples ejemplos de aplicaciones *software* listos para ser utilizados tanto en simulación como sobre el *hardware* real. Se han ejecutado varias de ellas, como por ejemplo `hello_world` o `demo_blink_led`, que son programas simples que usan el módulo UART y GPIO.

Es de especial interés la aplicación `demo_cfu`. El módulo CFU predeterminado (`rtl/core/neorv32_cpu_alu_cfu.vhd`) implementa el algoritmo de cifrado ligero *Extended Tiny Encryption Algorithm* (XTEA) [52] como aplicación de ejemplo. El programa de demostración correspondiente se encuentra en `sw/example/demo_cfu`. En este ejemplo se definen las instrucciones personalizadas mostradas en el Código 3. El resultado de la simulación se muestra en la Figura 21. En ella se puede observar la ejecución del algoritmo XTEA y la prueba de una instrucción no implementada que genera una excepción.

```

#define xtea_key_write(i, k)      RISC_V_INSTR_I_TYPE(RISC_V_OPCODE_CUSTOM1,
0b001, k, i)
#define xtea_key_read(i)         RISC_V_INSTR_I_TYPE(RISC_V_OPCODE_CUSTOM1,
0b000, 0, i)
#define xtea_hw_init(s)          RISC_V_INSTR_R_TYPE(RISC_V_OPCODE_CUSTOM0,
0b100, 0b0000000, s, 0)
#define xtea_hw_enc_v0_step(v0, v1) RISC_V_INSTR_R_TYPE(RISC_V_OPCODE_CUSTOM0,
0b000, 0b0000000, v0, v1)
#define xtea_hw_enc_v1_step(v0, v1) RISC_V_INSTR_R_TYPE(RISC_V_OPCODE_CUSTOM0,
0b001, 0b0000000, v0, v1)
#define xtea_hw_dec_v0_step(v0, v1) RISC_V_INSTR_R_TYPE(RISC_V_OPCODE_CUSTOM0,
0b010, 0b0000000, v0, v1)
#define xtea_hw_dec_v1_step(v0, v1) RISC_V_INSTR_R_TYPE(RISC_V_OPCODE_CUSTOM0,
0b011, 0b0000000, v0, v1)
#define xtea_hw_illegal_inst()    RISC_V_INSTR_R_TYPE(RISC_V_OPCODE_CUSTOM0,
0b111, 0b0000000, 0, 0)

```

Código 3: Definición de instrucciones personalizadas para XTEA.

```

<<< NEORV32 Custom Functions Unit (CFU) - Custom Instructions Example >>>

[NOTE] This program assumes the default CFU hardware in
rtl/core/neorv32_cpu_cp_cfu.vhd that implements
the Extended Tiny Encryption Algorithm (XTEA).

XTEA key: 0x207230ba1ffba710c45271efdd01768a

XTEA SW encryption (40 rounds, 64 words)...
XTEA HW encryption (40 rounds, 64 words)...
Comparing results... OK

XTEA SW decryption (40 rounds, 64 words)...
XTEA HW decryption (40 rounds, 64 words)...
Comparing results... OK

Execution timing:
ENC SW = 50780 cycles
ENC HW = 12865 cycles
DEC SW = 72108 cycles
DEC HW = 12946 cycles

Executing not-implemented CFU instruction (raise ILLEGAL INSTRUCTION exception)...
<NEORV32-RTE-PANIC> [cpu0] M Illegal instruction MEPC=0x00000768 MTVAL=0x00F7F78B </NEORV32-RTE-PANIC>

CFU demo program completed.

```

Figura 21: Simulación del ejemplo de uso de la CFU para XTEA.

El proyecto NEORV32 incluye tres configuraciones de prueba minimalistas y preconfiguradas en `rtl/test_setups`. Estas configuraciones de prueba solamente implementan funciones básicas del procesador y de la CPU, y se diferencian principalmente en la configuración de arranque real.

- **Configuración `neorv32_test_setup_approm.vhd`.** Esta configuración establece una **Unidad Central de Procesamiento (CPU)** `rv32imc_Zicsr_Zicntr` con 16 kB de **Instruction Memory (IMEM)** (como **Read-Only Memory (ROM)** preinicializada) y 8 kB de **Data Memory (DMEM)**, e incluye el módulo **General-Purpose Input/Output (GPIO)** para

controlar 8 señales externas (gpio_o) y el módulo para generar interrupciones. Esta configuración utiliza la configuración de *direct boot*, por lo que las aplicaciones de *software* se instalan directamente en la **IMEM** interna del procesador (a través del bitstream) durante la síntesis.

- **Configuración `neorv32_test_setup_bootloader.vhd`.** Mantiene la misma configuración de **CPU** que la configuración anterior y añade una *Universal Asynchronous Receiver-Transmitter (UART)* para interactuar con el *bootloader* o la aplicación mediante un terminal serie. El *setup* utiliza la configuración de *indirect boot*, por lo que las aplicaciones de *software* se pueden cargar y ejecutar en cualquier momento a través del gestor de arranque y un terminal serie.
- **Configuración `neorv32_test_setup_on_chip_debugger.vhd`.** Esta configuración añade el depurador integrado en el chip compatible con RISC-V, al cual se puede acceder a través de una interfaz *Joint Test Action Group (JTAG)* estándar. Esta configuración utiliza el *arranque indirecto*, por lo que las aplicaciones *software* se pueden cargar y ejecutar en cualquier momento a través del gestor de arranque y un terminal serie, y también a través del depurador.

En este caso se ha usado el *setup bootloader* con la configuración por defecto, que incluye la extensión.

Para evaluar esta arquitectura, se han extraído los datos de utilización de recursos mostrados en la **Tabla 11**. En este caso, el núcleo utiliza 2074 **LUT** y 1559 registros, representando un 1.54% y 0.58%, respectivamente. A diferencia de algunas implementaciones anteriores, esta configuración cuenta con 8 bloques *Random Access Memory (RAM)* y 4 bloques **DSP**. El análisis temporal de la **Figura 22** arroja un margen positivo con un **WNS** de 0.837 ns, lo que da como resultado una frecuencia máxima de operación teórica de aproximadamente 109 MHz.

Tabla 11: Utilización de recursos lógicos post-implementación del núcleo NEORV32.

Recurso Lógico	Utilizados	Disponibles	Utilización (%)
Slice LUT	2074	134600	1.54
LUT as Logic	2050	134600	1.52
LUT as Memory	24	46200	0.05
Slice Registers	1559	269200	0.58
Block RAM (BRAM)	8	365	2.19
DSP Blocks	4	740	0.54

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.837 ns	Worst Hold Slack (WHS): 0.120 ns	Worst Pulse Width Slack (WPWS): 3.750 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 4449	Total Number of Endpoints: 4449	Total Number of Endpoints: 1621

All user specified timing constraints are met.

Figura 22: Timing Report de la implementación del NEORV32 .

Para completar la evaluación, la [Tabla 12](#) recoge los resultados del análisis de potencia. El consumo total estimado es de 179 mW, de los cuales 151 mW son de potencia estática de la [FPGA](#). El consumo dinámico refleja que requiere un total de 28 mW, de los cuales 23 mW pertenecen al núcleo central de la [CPU](#).

Tabla 12: Estimación del consumo de potencia de la implementación del núcleo NEORV32.

Parámetro de Potencia	Consumo Estimado (mW)
Potencia Estática del Dispositivo	151
Potencia Dinámica Total	28
Núcleo CPU (neorv32_cpu)	23
Periféricos e Interconexión	5
Potencia Total Estimada	179

Pese a no tener los mejores resultados de consumo de área y rendimiento, es la arquitectura más prometedora por su carácter modular. Presenta facilidades notables y permite adaptar la arquitectura base, soportando un número significativo de extensiones del juego de instrucciones. Por esta razón es una de las implementaciones candidatas a utilizar para el desarrollo de este trabajo.

5.5. Conclusión

Inicialmente, se hizo un estudio de núcleos RISC-V de código abierto, con un lenguaje [HDL](#) conocido, documentación apropiada y que permitan la extensión del juego de instrucciones. Tras este estudio se seleccionan tres opciones para su implementación y obtención de resultados de rendimiento, potencia y área. Los núcleos elegidos inicialmente son PicoRV32, Ibex y NEORV32.

En la [Tabla 6](#) se presenta una comparación de los resultados de implementación de los tres núcleos. PicoRV32 destaca por su reducido uso de recursos, a la vez que presenta la frecuencia máxima más alta. Ibex destaca por ser un núcleo con gran respaldo académico e industrial con múltiples implementaciones. Sin embargo, es el que más recursos consume y opera a una frecuencia menor. Por último, la implementación NEORV32 es el que cuenta con una mejor documentación, recursos y ejemplos. En términos de potencia todos los núcleos tienen resultados similares.

Para representar los datos que conforma el espacio de diseño de las tres soluciones se introduce el concepto de la [Suma Normalizada de los Recursos Utilizados \(SNRU\)](#) [53]. Dado que se trata de recursos heterogéneos, para obtener una métrica coherente se normaliza cada el uso de cada tipo

de recurso (LUT, Registros, DSP) y se calcula la suma de esos recursos normalizados. Se puede expresar como

$$SNRU(x) = \sum_{\text{resource} \in R} \frac{U_{\text{resource}}(X)}{A_{\text{resource}}}$$

donde:

- x es el circuito a evaluar,
- R es el conjunto de recursos disponibles, en este caso $R = \{\text{LUTS}, \text{Registros}, \text{DSP}\}$,
- $U_{\text{resource}}(X)$ son los recursos utilizados en X de cada tipo.
- A_{resource} es el conjunto de recursos disponibles de ese tipo.

Tabla 13: Comparación de valores PPA (Potencia, Rendimiento y Área) para dispositivo AMD Artix 7™ de las implementaciones RISC-V estudiadas.

Núcleo	Recursos				Rendi- miento	Potencia	
	LUT	Registros	DSP	SNRU	$F_{\{\max\}}$ (MHz)	Total (mW)	Núcleo RISC-V (mW)
PicoRV32	1382	796	4	0.015	132	180	19
Ibex	3255	1367	13	0.038	53	366	17
NEORV32	2077	1559	4	0.021	109	179	23

Como se aprecia en la Tabla 13 los valores de SNRU son muy bajos, lo que indica que el impacto del diseño en el dispositivo elegido es poco significativo en cuanto a recursos y no habrá problemas de implementación ni de ruteado en el dispositivo.

Si se representa el espacio de diseño obtenido tanto para el tiempo de ciclo T_c (Figura 23) como para la potencia (Figura 24) se aprecia que la solución basada en Ibex siempre se ubica en la zona de mayor uso relativo de recursos y con peor consumo de potencia.

Tanto el core PicoRV32 como NEORV32 presentan un comportamiento similar, siendo PicoRV32 ligeramente más rápido, por lo que ambos cores pueden ser candidatos a ser utilizados. Sin embargo, como el objetivo final de este TFM es la extensión del juego de instrucciones, se ha decidido utilizar como núcleo RISC-V el NEORV32, con su soporte para implementar un SoC completo. Uno de los factores decisivos es que cuenta con una Custom Functions Unit, con un ejemplo documentado de extensión del juego de instrucciones. Se considera que para cumplir el objetivo final esta implementación es la más indicada.

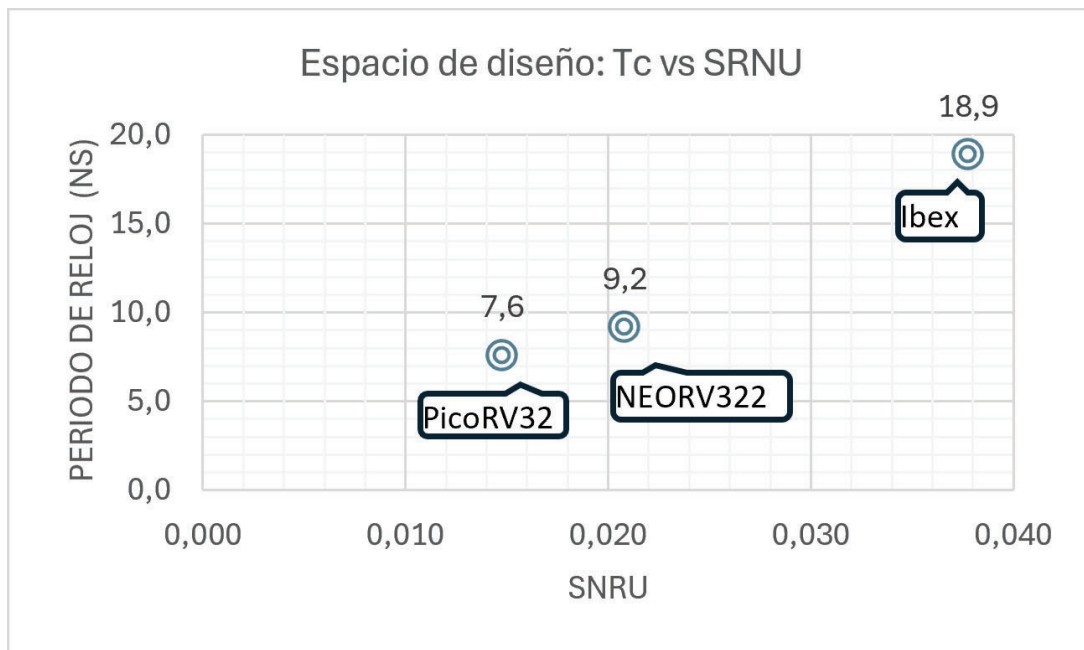


Figura 23: Espacio de diseño representando T_c para $F_{\max}$ del núcleo frente a SRNU .

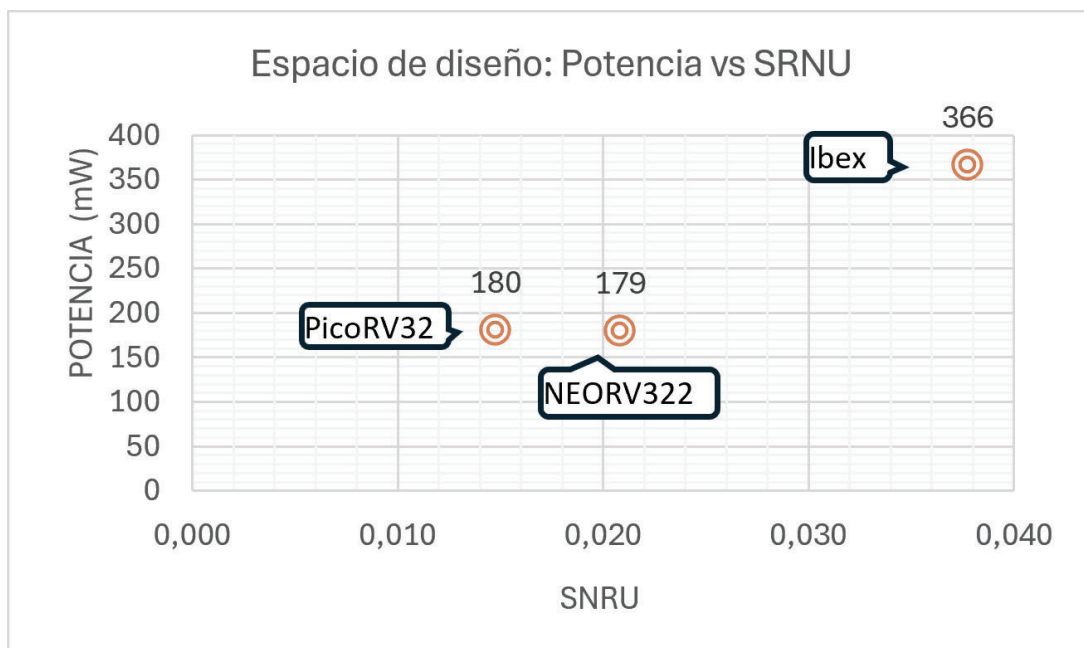


Figura 24: Espacio de diseño representando la potencia a total consumida (en mW) frente a SRNU para los casos de estudio.

Capítulo 6

Implementación de ML-DSA en un núcleo RISC-V

En el presente capítulo se implementa el algoritmo de firma [ML-DSA](#) en el núcleo seleccionado, el NEORV32.

6.1. Implementación *software*

Para la implementación del algoritmo [ML-DSA](#), se ha usado la librería `mldsa-native` [54]. El proyecto `mldsa-native` es una implementación en código C del estándar de firma post-cuántica [ML-DSA](#), que hace hincapié en la seguridad, el rendimiento y la portabilidad [55]. `mldsa-native` cuenta con el respaldo de la Post-Quantum Cryptography Alliance, que forma parte de la Fundación Linux [56].

La [Interfaz de Programación de Aplicaciones \(API\)](#) de esta librería expone las operaciones fundamentales para la firma digital, entre las que se pueden destacar para esta aplicación:

- **`crypto_sign_keypair`**. Es la función encargada de generar el par de claves. Tiene como argumentos dos punteros que actúan como buffers de salida para la clave pública `pk` y para la clave privada `sk`.
- **`crypto_sign_signature`**. Esta función firma el mensaje a partir de una clave privada previamente generada. Toma como parámetros de entrada el mensaje `m` y su longitud `milen`, la clave privada `sk` y otros parámetros opcionales de contexto. Como salidas, utiliza dos punteros para almacenar la firma digital resultante `sig` y su longitud `siglen`.
- **`crypto_sign_verify`**. Esta función implementa el algoritmo de verificación de firma digital. Recibe como parámetros de entrada `sig`, `siglen`, `m`, `milen`, la clave pública `pk` y los parámetros de contexto si se han utilizado. La función devuelve un 0 si la verificación se completa correctamente.

La librería incluye varios ejemplos de implementación. En la [Figura 25](#) se observa un ejemplo básico, en el que se realiza la generación de claves, el proceso de firma y la verificación para los distintos niveles de seguridad.

Para validar el correcto funcionamiento de esta librería sobre la arquitectura seleccionada, se realiza un test inicial en el entorno de ejecución *software* del núcleo RISC-V NEORV32. `mldsa-native` está desarrollada en C estándar y es preciso realizar su compilación cruzada para generar código máquina ejecutable en RISC-V NEORV32.

Este proceso se realiza mediante la herramienta [GNU Compiler Collection \(GCC\)](#) para RISC-V, `riscv32-unknown-elf-gcc`. Es necesario adaptar el código fuente de la aplicación original, así como de la librería, añadiéndose como un nuevo proyecto de aplicación dentro de la estructura de directorios del NEORV32.

En el [Código 4](#) se muestra el programa `main.c` generado para encapsular la aplicación. En el código se observa la ejecución de las tres funciones estudiadas de [ML-DSA](#): (1) generación de claves, (2) procedimiento de firma, y (3) verificación.

```

% make run
./build/test_binary44
ML-DSA-44 Basic Example
=====

Message: This is a test message for ML-DSA digital signature algorithm!
Context: test_context_123

Generating keypair... DONE
Signing message... DONE
Verifying signature... DONE
Sign and open message... DONE

All tests passed! ML-DSA signature verification successful.
./build/test_binary65
ML-DSA-65 Basic Example
=====

Message: This is a test message for ML-DSA digital signature algorithm!
Context: test_context_123

Generating keypair... DONE
Signing message... DONE
Verifying signature... DONE
Sign and open message... DONE

All tests passed! ML-DSA signature verification successful.
./build/test_binary87
ML-DSA-87 Basic Example
=====

Message: This is a test message for ML-DSA digital signature algorithm!
Context: test_context_123

Generating keypair... DONE
Signing message... DONE
Verifying signature... DONE
Sign and open message... DONE

All tests passed! ML-DSA signature verification successful.

```

Figura 25: Ejemplo básico de ejecución de ML-DSA-44 nativa.

En este caso, se hace uso de las funciones de la [UART](#) del NEORV32 para mostrar los resultados por un puerto serie. Dado el tamaño del código generado, se ha modificado el mapa de memoria para almacenar el nuevo código binario compilado. Haciendo uso de las herramientas de compilación (*toolchain*) del procesador, se genera el archivo binario ejecutable enlazado para la arquitectura NEORV32.

Para ejecutar la aplicación, se procede a la carga del programa compilado a través de la [UART](#) utilizando el bootloader de NEORV32. El resultado se muestra en la [Figura 26](#), que incluye el arranque del bootloader, la carga del programa y su posterior ejecución.

```

#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <neorv32.h>

#define MLD_CONFIG_API_PARAMETER_SET 44
#define MLD_CONFIG_API_NAMESPACE_PREFIX PQCP_MLDSA_NATIVE_MLDSA44
#include <mldsa_native.h>

int main(void) {
    // NEORV32 UART Init
    neorv32_rte_setup();
    neorv32_uart0_setup(19200, 0);

    neorv32_uart0_printf("ML-DSA-44 Iniciando en NEORV32...\n");

    uint8_t pk[CRYPTO_PUBLICKEYBYTES];
    uint8_t sk[CRYPTO_SECRETKEYBYTES];
    uint8_t sig[CRYPTO_BYTES];
    size_t siglen;

    uint8_t msg[] = "Mensaje de prueba";
    size_t mlen = strlen((char*)msg);

    // Generación de claves
    neorv32_uart0_printf("Generating keypair... ");
    if (crypto_sign_keypair(pk, sk) != 0) {
        neorv32_uart0_printf("ERROR al generar llaves\n");
        return 1;
    }
    neorv32_uart0_printf("DONE\n");

    // Procedimiento de firma
    neorv32_uart0_printf("Signing message... ");
    if (crypto_sign_signature(sig, &siglen, msg, mlen, NULL, 0, sk) != 0) {
        neorv32_uart0_printf("ERROR al firmar\n");
        return 1;
    }
    neorv32_uart0_printf("DONE\n");

    // Verificación
    neorv32_uart0_printf("Verifying signature... ");
    if (crypto_sign_verify(sig, siglen, msg, mlen, NULL, 0, pk) != 0) {
        neorv32_uart0_printf("ERROR: Firma INVALIDA\n");
        return 1;
    }
    neorv32_uart0_printf("DONE. ¡Firma verificado con éxito!\n");

    return 0;
}

```

Código 4: Programa para la ejecución de mldsa-native en NEORV32

```

11:06:17:870
11:06:17:870
11:06:17:870 NEORV32 Bootloader
11:06:17:870 build: May 1 2026
11:06:17:886
11:06:17:886 Auto-boot in 8s. Press any key to abort.
11:06:25:914
11:06:25:914 Loading from SPI flash @0x00400000... <0x07>ERROR_DEVICE
11:06:25:947 Type 'h' for help.
11:06:25:963 CMD:> h
11:06:37:749 Available CMDs:
11:06:37:749 h: Help
11:06:37:765 i: System info
11:06:37:765 r: Restart
11:06:37:781 u: Upload via UART
11:06:37:781 l: SPI flash - load
11:06:37:797 s: SPI flash - program
11:06:37:813 e: Start executable
11:06:37:829 x: Exit
11:06:37:829 CMD:> u
11:06:43:106 Awaiting neorv32_exe.bin... OK
11:07:15:192 CMD:> e
11:07:24:708 Booting from 0x00000000...
11:07:24:724
11:07:24:724 ML-DSA-44 Iniciando en NEORV32...
11:07:24:740 Generating keypair... DONE
11:07:25:219 Signing message... DONE
11:07:26:563 Verifying signature... DONE. <0xc2><0xa1>Firma verificado con éxito!

```

Figura 26: Ejemplo de ejecución de mlds-native en el núcleo NEORV32.

La verificación del correcto funcionamiento del algoritmo sobre el procesador ha sido realizada utilizando la herramienta de depuración [GNU Debugger \(GDB\)](#). Este proceso requiere realizar ciertas modificaciones en el código. Por ello, se ha diseñado un módulo de adaptación que expone las señales internas del puerto [JTAG](#) del procesador y las hace accesibles mediante un puerto PMOD de la placa Nexys Video.

Utilizando las herramientas [Open On-Chip Debugger \(OpenOCD\)](#) y [GDB](#), se ha depurado el programa. Para ello, se ha creado una nueva configuración `neorv32_test.cfg`, dentro del repositorio de NEORV32 en `/neorv32/sw/openocd`, como se muestra en el [Código 5](#). Este archivo contiene los parámetros de configuración necesarios para acceder internamente al procesador NEORV32.

[OpenOCD](#) abre un puerto con un servidor [GDB](#) al que se puede acceder con un cliente [GDB](#). `riscv32-unknown-elf-gdb` actúa como cliente y carga el programa a depurar. En la [Figura 27](#) se muestra, arriba, la ejecución de [OpenOCD](#) y, abajo, la inicialización del cliente [GDB](#). Esta forma de depuración permite acceder al estado de la memoria y de los registros del NEORV32, además de la inserción de *breakpoints* o la ejecución paso a paso, usando los comandos mostrados en el [Código 6](#).

```

adapter driver ftdi
adapter speed 1000

ftdi vid_pid 0x0403 0x6014
ftdi channel 0
ftdi layout_init 0x00e8 0x60eb

reset_config none

transport select jtag
jtag newtap neorv32 cpu -irlen 5

target create neorv32.cpu riscv -chain-position neorv32.cpu

init
halt

```

Código 5: Configuración de OpenOCD para depuración del NEORV32

```

target extended-remote localhost:3333 # Conexión a OpenOCD

monitor reset halt # Reset del núcleo NEORV32 y parar la ejecución
monitor targets    # Lista de objetivos conectados y estado actual

info registers     # Mostrar el estado de los registros RISC-V

load               # Cargar el archivo ELF en la memoria del NEORV32

break main         # Añadir breakpoint

continue           # Reanudar la ejecución

stepi              # Ejecutar una instrucción a nivel de ensamblador
next               # Ejecutar la siguiente línea de código en C

```

Código 6: Comandos principales de GDB para la depuración.

```
○ * Executing task: openocd -f /home/ereyes/riscv/neorv32/sw/openocd/neorv32_test.cfg

xPack Open On-Chip Debugger 0.12.0+dev-02228-ge5888bda3-dirty (2025-10-04-22:42)
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
Info : clock speed 1000 kHz
Info : JTAG tap: neorv32.cpu tap/device found: 0x00000009 (mfg: 0x004 (RAMKEED Limited), part: 0x0000, ver: 0x0)
Info : datacount=1 progbufsize=2
Info : Disabling abstract command reads from CSRs.
Info : Examined RISC-V core; found 1 harts
Info : hart 0: XLEN=32, misa=0x40901104
Info : [neorv32.cpu] Examination succeed
Info : [neorv32.cpu] starting gdb server on 3333
Info : Listening on port 3333 for gdb connections
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : accepting 'gdb' connection on tcp/3333
Info : JTAG tap: neorv32.cpu tap/device found: 0x00000009 (mfg: 0x004 (RAMKEED Limited), part: 0x0000, ver: 0x0)
Info : Disabling abstract command writes to CSRs.
[]

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS 1
○ ereyes@ARQG-711:~/riscv/neorv32/sw/example/mlds$ riscv32-unknown-elf-gdb main.elf
GNU gdb (GDB) 17.1
Copyright (C) 2025 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=riscv32-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main.elf...
(gdb) target extended-remote localhost:3333
Remote debugging using localhost:3333
main () at main.c:22
22      neorv32_rte_setup();
(gdb) monitor reset halt
JTAG tap: neorv32.cpu tap/device found: 0x00000009 (mfg: 0x004 (RAMKEED Limited), part: 0x0000, ver: 0x0)
(gdb) monitor targets
      TargetName      Type      Endian TapName      State
-----
0* neorv32.cpu      riscv      little neorv32.cpu      halted
(gdb) load
Loading section .text, size 0xa150 lma 0x0
Loading section .rodata, size 0xba0 lma 0xa150
```

Figura 27: Proceso de depurado con OpenOCD y GDB.

6.2. Aceleración *hardware*

Tras completar las primeras pruebas, se ha verificado que se puede ejecutar la aplicación *software* capaz de generar el par de claves, firmar un mensaje y verificar la firma siguiendo el estándar de [ML-DSA](#). Siguiendo con los objetivos marcados en el presente [TFM](#), en este apartado se selecciona el bloque computacional que puede ser acelerado mediante la extensión del juego de instrucciones RISC-V. El objetivo de esta transformación es alcanzar una reducción del 50% en su tiempo de ejecución con respecto a la versión *software*. Además, se realiza la inclusión del generador de números

aleatorios disponible en el núcleo NEORV32 para cumplir con la seguridad requerida en el estándar ML-DSA.

6.2.1. True Random-Number Generator

Como se estudió anteriormente, ML-DSA necesita un generador de números aleatorios para la generación de un par de claves seguro. En la implementación de mldsa-native, se detalla que la implementación de la función randombytes() es solo para pruebas. Esta función debe ser sustituida por un RNG criptográficamente seguro para su uso.

El NEORV32 cuenta con un *True Random-Number Generator (TRNG)* que proporciona números verdaderamente aleatorios desde el punto de vista físico [47]. Se basa en osciladores en anillo de funcionamiento libre que generan ruido de fase cuando son muestreados por un reloj constante. Este ruido de fase se utiliza como fuente de entropía física. El TRNG cuenta con una arquitectura independiente de la plataforma, sin primitivas, macros ni atributos específicos de FPGA, por lo que puede sintetizarse para cualquier FPGA.

Se ha añadido el Código 7 para sustituir la función randombytes() y usar el TRNG del NEORV32. Además, es necesario añadir la directiva MLD_CONFIG_RNG_EXTERNAL para forzar a la librería a utilizar esta nueva implementación.

```
int randombytes(uint8_t *out, size_t outlen) {
    for (size_t i = 0; i < outlen; i++) {
        while (neorv32_trng_data_avail() == 0) {
        }
        out[i] = neorv32_trng_data_get();
    }
    return 0;
}
```

Código 7: Función randombytes() usando el TRNG del NEORV32

Para incluir el TRNG en la arquitectura del procesador, es necesario modificar los parámetros de configuración del núcleo NEORV32. Como se muestra en el Código 8, se debe habilitar el parámetro IO_TRNG_EN con el valor true y definir la profundidad de la memoria *First In, First Out (FIFO)* interna mediante IO_TRNG_FIFO.

```
-- True-Random Number Generator (TRNG) --
IO_TRNG_EN : boolean                := true ; -- implement TRNG
IO_TRNG_FIFO: natural range 1 to 2 ** 15 := 64; -- data FIFO depth
```

Código 8: Configuración de parámetros del TRNG

En la Figura 28 se muestra de nuevo la ejecución del algoritmo, pero esta vez comprobando que está disponible el TRNG del NEORV32 antes de comenzar con la generación de claves. Esta modificación asegura que las claves generadas tengan un componente realmente aleatorio.

```

12:16:16:956 NEORV32 Bootloader
12:16:16:972 build: May 1 2026
12:16:16:972
12:16:25:001 Auto-boot in 8s. Press any key to abort.
12:16:25:001
12:16:25:032 Loading from SPI flash @0x00400000... <0x07>ERROR_DEVICE
12:16:25:032 Type 'h' for help.
12:16:51:676 CMD:> u
12:17:19:715 Awaiting neorv32_exe.bin... OK
12:17:28:879 CMD:> e
12:17:28:895 Booting from 0x00000000...
12:17:28:895
12:17:28:895 TRNG...OK
12:17:28:911 ML-DSA-44 Iniciando en NEORV32...
12:17:29:407 Generating keypair... DONE
12:17:30:734 Signing message... DONE
Verifying signature... DONE. <0xc2><0xa1>Firma verificado con éxito!

```

Figura 28: Ejemplo de ejecución de `mlds-native` en el núcleo NEORV32 con TRNG.

6.2.2. Custom Functions Unit del NEORV32

Para la extensión del juego de instrucciones se ha usado la CFU del NEORV32. La CFU es una extensión de la ISA específica para el NEORV32 que proporciona un módulo *hardware* genérico que puede utilizarse para implementar instrucciones RISC-V personalizadas.

Las instrucciones de la CFU pueden utilizar un conjunto limitado de *opcodes* de RISC-V en el espacio de codificación de instrucciones de 32 bits RV32 para propósitos personalizados. Es posible decidir qué *opcodes* y qué tipo o formatos de instrucciones se utilizan entre los definidos. Hasta el momento, se soportan cuatro *opcodes*, mostrados en el Código 9. Dos de ellos, `custom-0` y `custom-1`, han sido reservados explícitamente por la especificación de la ISA RISC-V para extensiones definidas por el usuario.

```

#define RISC_V_OPCODE_CUSTOM0 0b0001011 // RISC-V "CUSTOM-0"
#define RISC_V_OPCODE_CUSTOM1 0b0101011 // RISC-V "CUSTOM-1"
#define RISC_V_OPCODE_OP32     0b0111011 // RISC-V "OP-32"
#define RISC_V_OPCODE_OPIMM32 0b0011011 // RISC-V "OP-IMM-32"

```

Código 9: Definición de *opcodes* soportados para instrucciones personalizadas [47].

Los otros dos *opcodes*, `op-32` y `op-imm-32`, están reservados por la especificación RISC-V para operaciones extendidas y futuras extensiones de la ISA. Por tanto, su disponibilidad podría cambiar en el futuro.

Las instrucciones personalizadas proporcionadas por la CFU pueden utilizarse en código C puro mediante funciones intrínsecas. Las funciones intrínsecas se comportan como funciones estándar de C, pero sustituyen la llamada original por una secuencia de instrucciones de ensamblador. Actualmente, el marco de software NEORV32 ofrece dos prototipos predefinidos para instrucciones personalizadas, tipo R y tipo I (Código 10).

Las instrucciones de tipo R operan sobre dos registros de origen, `rs1` y `rs2`, y devuelven el resultado del procesamiento a través del registro de destino `rd`. La operación concreta se puede definir utilizando los campos de bits `funct7` y `funct3`.

```
uint32_t RISCV_INSTR_R_TYPE(int opcode, int funct3, int funct7, uint32_t rs1,
uint32_t rs2);

uint32_t RISCV_INSTR_I_TYPE(int opcode, int funct3, uint32_t rs1, int imm12);
```

Código 10: Definición de los tipos de instrucciones personalizadas [47].

Las instrucciones de tipo I operan sobre un registro de origen rs1 y un operando inmediato de 12 bits imm12, y devuelven el resultado del procesamiento a través del registro de destino rd. La operación concreta se puede definir utilizando el campo de bits funct3.

Estas funciones intrínsecas siempre devuelven un valor de 32 bits de tipo uint32_t como resultado del procesamiento, que puede descartarse si no es necesario. Los campos de bits funct3, funct7 e imm12 se utilizan para enviar a la CFU literales estáticos resueltos en tiempo de compilación. Por otro lado, los argumentos rs1 y rs2 transmiten datos en tiempo de ejecución hacia la CFU mediante direcciones de registros. Estos argumentos de registro pueden albergar variables y el compilador se encargará de añadir el código necesario para cargar los datos en un registro antes de transferirlos a la CFU.

El módulo *hardware* de la CFU está estrechamente integrado en la CPU y acoplado como coprocesador de la ALU. Se utiliza una interfaz sencilla para transmitir los operandos, la definición de la operación y el resultado del procesamiento (Tabla 14).

Tabla 14: Interfaz de la CFU

Señal	Dirección	Ancho	Descripción
Globales			
clk_i	entrada	1	Reloj de la CPU, los registros se activan en el flanco de subida
rstn_i	entrada	1	Reset de la CPU, activo a nivel bajo, asíncrono
Solicitud			
start_i	entrada	1	Inicia la operación de la CFU, a nivel alto durante exactamente un ciclo
inst_i	entrada	32	Palabra de instrucción de la CFU
rs1_i	entrada	32	Operando de registro 1, leído del banco de registros de la CPU
rs2_i	entrada	32	Operando de registro 2, leído del banco de registros de la CPU
Respuesta			
result_o	salida	32	Resultado del procesamiento, se escribe en el registro de destino (rd)
valid_o	salida	1	A nivel alto durante exactamente un ciclo cuando la operación de la CFU ha terminado y el resultado es válido

6.2.3. Extensión del juego de instrucciones

Uno de los cuellos de botella en ML-DSA lo representa la generación de funciones *hash*. Para aumentar la eficiencia global del sistema se diseña una extensión del juego de instrucciones que da soporte a la ejecución de los algoritmos SHAKE128 y SHAKE256.

Keccak

Keccak es una función criptográfica versátil, estandarizada por el [NIST](#) como la tercera generación de la familia *Secure Hash Algorithm* (SHA) [26], [57]. Aunque es más conocida como función *hash*, también puede utilizarse para la autenticación, el cifrado y la generación de números pseudoaleatorios. Su estructura se basa en la construcción esponja y utiliza la permutación criptográfica Keccak — f .

La construcción de esponja es un marco de trabajo diseñado para especificar funciones sobre datos binarios con una longitud de salida arbitraria. Esta construcción emplea los tres componentes siguientes:

- Una función subyacente que opera sobre cadenas de longitud fija, denotada por f .
- Un parámetro denominado tasa, en inglés *rate*, denotado por r .
- Una regla de relleno, en inglés *padding*, denotada por pad .

La función resultante que esta construcción genera a partir de dichos componentes se denomina función de esponja y se denota por $\text{SPONGE}[f, pad, r]$. Toda función de esponja recibe dos parámetros de entrada: una cadena de bits, denotada por N , y la longitud en bits deseada para la cadena de salida, denotada por d , representándose como $\text{SPONGE}[f, pad, r](N, d)$. La analogía conceptual con una esponja radica en que un número arbitrario de bits de entrada es «absorbido» en el estado interno de la función, tras lo cual un número arbitrario de bits de salida es «exprimido» desde dicho estado.

La construcción de esponja se ilustra de forma esquemática en la [Figura 29](#). El equipo que está detrás del algoritmo Keccak cuenta con una implementación *hardware* del algoritmo, que se puede encontrar en su repositorio [57].

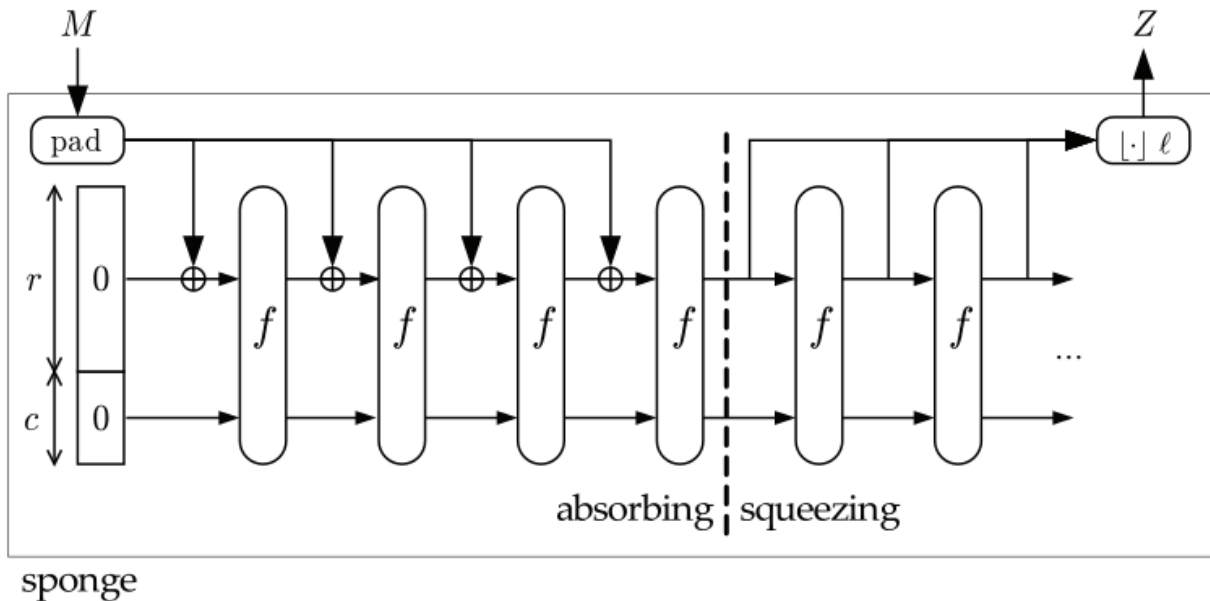


Figura 29: Estructura de esponja implementado en Keccak.

Haciendo uso de la unidad CFU, se han propuesto las instrucciones que se definen en el [Código 11](#).

```

#define keccak_hw_init(func7, rs1, rs2)
RISCV_INSTR_R_TYPE(RISCV_OPCODE_CUSTOM0, 0b000, func7, rs1, rs2)

#define keccak_hw_data(rs1, rs2)
RISCV_INSTR_R_TYPE(RISCV_OPCODE_CUSTOM0, 0b001, 0b0000000, rs1, rs2)

#define keccak_hw_data_last(rs1, rs2)
RISCV_INSTR_R_TYPE(RISCV_OPCODE_CUSTOM0, 0b010, 0b0000000, rs1, rs2)

#define keccak_hw_read()
RISCV_INSTR_R_TYPE(RISCV_OPCODE_CUSTOM0, 0b011, 0b0000000, 0, 0)

#define keccak_hw_reset()
RISCV_INSTR_R_TYPE(RISCV_OPCODE_CUSTOM0, 0b100, 0b0000000, 0, 0)

```

Código 11: Definición de las instrucciones personalizadas para la aceleración de Keccak.

Todas las instrucciones diseñadas se ajustan al formato de tipo R de la arquitectura RISC-V y operan bajo el código de instrucción CUSTOM-0. La unidad de control distingue la operación a ser ejecutada por la [CFU](#) según el valor del campo funct3.

El funcionamiento de cada instrucción se describe a continuación:

- **keccak_hw_init.** Se utiliza para configurar la operación que se va a ejecutar. El campo funct7 selecciona la variante del algoritmo (SHAKE256 o SHAKE128)¹. El registro fuente rs1 configura el número de bytes útiles de la última palabra. El registro fuente rs2 determina la cantidad de bytes de salida para aquellas funciones cuya longitud de salida es variable.
- **keccak_hw_data** y **keccak_hw_data_last.** Son responsables de inyectar los datos de entrada al módulo *hardware* empleando los registros rs1 y rs2. La instrucción keccak_hw_data_last se utiliza para indicar que se va a enviar el último dato.
- **keccak_hw_read.** Extrae el hash resultante, leyendo los datos de salida en palabras de 32 bits.
- **keccak_hw_reset.** Emite una señal de reinicio para limpiar el estado interno del acelerador y prepararlo para nuevas operaciones.

En la [Figura 30](#), se presenta un diagrama de bloques que ilustra la integración de la [CFU](#) y el módulo Keccak para la ejecución de las instrucciones personalizadas.

¹Se han probado las funciones SHAKE256 y SHAKE128, por ser las utilizadas en el estándar [ML-DSA](#). Es posible implementar otras funciones del estándar SHA-3, pero requieren modificaciones en el núcleo Keccak que no han sido probadas en este trabajo

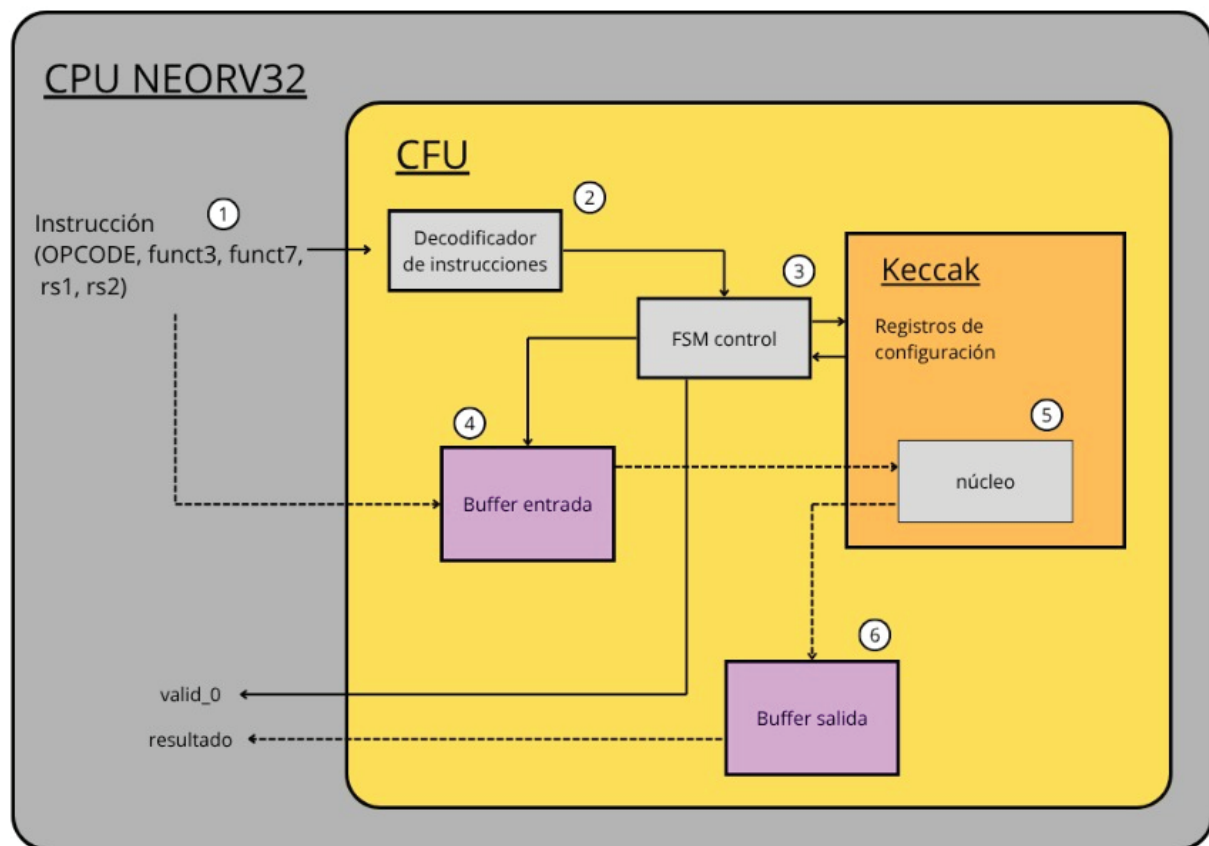


Figura 30: Diagrama de bloques de la CFU.

La secuencia de operación de la CFU se detalla a continuación, siguiendo la distribución mostrada en la Figura 30:

1. El proceso comienza cuando la CPU del NEORV32 recibe una instrucción que identifica que pertenece a la CFU por su *opcode*. En este momento, se establece la señal de activación de la CFU y se pasa la instrucción mediante la interfaz.
2. La CFU incluye un decodificador de instrucciones, que extrae la información de *funct3* y *funct7* para determinar la operación a ejecutar (Código 12). Se comprueba que el *opcode* recibido es correcto y, entonces, se activa la señal *start*.

```
start <= start_i when (inst_i(6 downto 0) = opcode_custom0_c) else '0';
funct3 <= inst_i(14 downto 12);
funct7 <= inst_i(31 downto 25);
```

Código 12: Decodificador de instrucciones de la CFU

3. La unidad *Finite State Machine (FSM)* de control tiene los estados *S_IDLE*, *S_SEND*, *S_WAIT_READ*, *S_READ* y *S_DONE*, como muestra la Figura 31. A continuación, se detalla el comportamiento de cada uno:
 - **S_IDLE.** Es el estado inicial y de reposo. Cuando la señal *start* se activa, se evalúa *funct3* para determinar la instrucción recibida:

- Si se recibe la instrucción `keccak_hw_init`: se configuran los registros del Keccak,
 - Si se recibe la instrucción `keccak_hw_reset`: se activa la señal de reset del Keccak.
 - Para los casos anteriores, y cuando el campo `funct3` no representa ninguna función definida, se pasa al estado `S_DONE`.
 - Si llegan funciones de datos de entrada, se pasa al estado `S_SEND`.
 - Por último, si llega una instrucción de lectura de datos de salida, se pasa al estado `S_WAIT_READ`.
- **S_SEND.** Se entra en este estado cuando el campo `funct3` corresponde a las funciones `keccak_hw_data` y `keccak_hw_data_last`. Seguidamente, pasa al estado `S_DONE`.
 - **S_WAIT_READ.** Se entra en este estado cuando llega una instrucción `keccak_hw_read`. Aquí se comprueba que hay un dato disponible para leer en el *buffer* de salida (FIFO), es decir, que `wr_index` sea mayor que `rd_index`. En caso afirmativo, se pasa al estado `S_READ`.
 - **S_READ.** En este estado se realiza la lectura de un dato en el *buffer* de salida. Posteriormente, se pasa al estado `S_DONE`.
 - **S_DONE.** Desde este estado, en el siguiente ciclo de reloj, se pasa siempre de vuelta al estado de reposo `S_IDLE`.

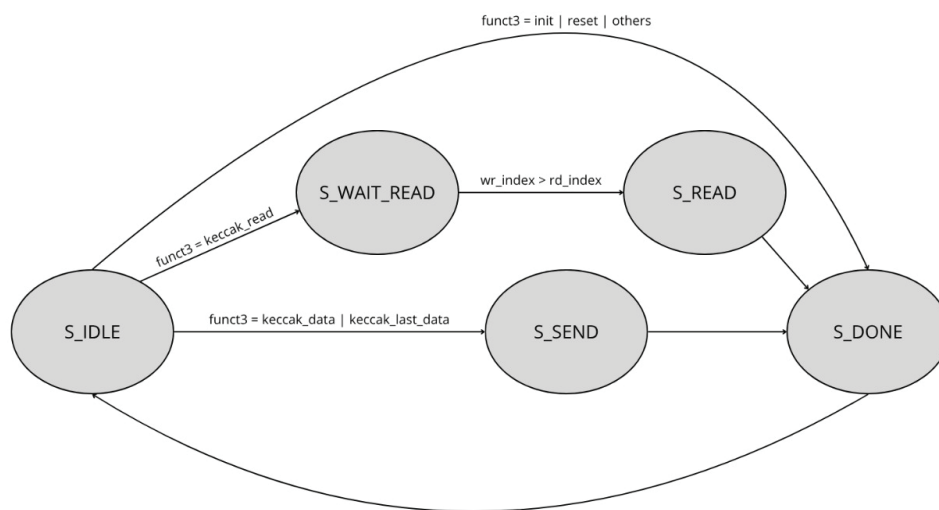


Figura 31: FSM de control de la CFU

4. Cuando se ha procesado una instrucción de lectura (`S_SEND`), los datos se almacenan en un *buffer* hasta que llega el último dato. En este momento, se envían los datos al núcleo de procesamiento del Keccak. En caso de que el módulo Keccak tenga activada la señal `busy`, se espera para enviar los datos. En el [Código 13](#) se muestra el código del proceso VHDL encargado de enviar los datos al Keccak cuando el *buffer* tiene todos los datos necesarios.

```

-- Save Msg (in) --
p_send_data: process(clk_i, rstn_i)
begin
    if rstn_i = '0' then
        flag(0) <= '0';
        flag(1) <= '0';
        k_data_valid_in(0) <= '0';
        k_last_data_in(0) <= '0';
        dr_index <= 0;
    elsif rising_edge(clk_i) then
        flag(0) <= s_data_ready;
        k_data_valid_in(0) <= '0';
        k_last_data_in(0) <= '0';

        if (flag(0) = '1' and s_data_ready = '0') then
            flag(1) <= '1';
            dr_index <= 0;
        elsif (flag(1) = '1' and k_busy = '0') then
            k_data_in(0) <= mydata(dr_index);
            k_data_valid_in(0) <= '1';
            dr_index <= dr_index + 1;

            if (dr_index = dw_index - 1) then
                k_last_data_in(0) <= '1';
                flag(1) <= '0';
            end if;
        end if;
    end if;
end process p_send_data;

```

Código 13: Proceso de la CFU encargado de guardar los datos de entrada

5. El módulo Keccak ha sido configurado previamente y ha recibido los datos de entrada. Se ejecuta el algoritmo seleccionado (SHAKE 256 o SHAKE128) y devuelve la salida con el tamaño solicitado.
6. Cuando el módulo Keccak ha terminado su procesamiento, los datos disponibles a la salida se almacenan en un *buffer* de salida. El proceso VHDL encargado de almacenar los datos en el *buffer* se muestra en el [Código 14](#). Cuando llegue una instrucción de lectura de datos de salida (S_WAIT_READ) se comprueba que hay datos disponibles para leer.

```

-- Save Result (out) --
p_data: process(clk_i, rstn_i)
begin
    if rstn_i = '0' then
        wr_index <= 0;
        myarray <= (others => (others => '0'));
    elsif rising_edge(clk_i) then
        if (start = '1' and funct3 = keccak_init) then
            wr_index <= 0;
            end if;

            if (k_data_valid_out(0) = '1') then
                myarray(wr_index) <= std_logic_vector(k_data_out(0)(31 downto 0));
                myarray(wr_index+1) <= std_logic_vector(k_data_out(0)(63 downto 32));

                if (wr_index < array_size - 2) then
                    wr_index <= wr_index + 2;
                end if;
            end if;
        end if;
    end process p_data;

```

Código 14: Proceso de la CFU encargado de guardar los datos de salida

Para comprobar que estas nuevas instrucciones funcionen correctamente desde el punto de vista lógico antes de ser implementadas físicamente, se utilizó el simulador VHDL de código abierto GHDL [58]. Esta herramienta permite verificar el comportamiento de las señales internas ciclo a ciclo, para validar de forma exhaustiva la máquina de estados de la CFU, la correcta sincronización en la inyección de datos a través de los registros y la extracción precisa de los resultados antes de proceder a la prueba real sobre el procesador. Adicionalmente, se ha usado la herramienta GTKWave, un visualizador gráfico de formas de onda.

En la Figura 32 se muestra el caso de una instrucción de keccak_hw_reset (funct3 = “100”). Se activa el reset del Keccak y se pasa al estado S_DONE para activar valid_o a 1 e indicar que se ha terminado la ejecución de la instrucción. Se puede observar que el valor de entrada inst_i va cambiando, pero solamente cuando la instrucción corresponde a la CFU se activa la señal start_i.

En la Figura 33 se muestra la llegada de una instrucción keccak_hw_init (funct3 = “000”). Se configuran los registros de entrada del Keccak con los valores correspondientes de funct7, rs1 y rs2.

En la Figura 34 se registran cuatro instrucciones de entrada de datos (funct3 = “001”) y una con el último dato (funct3 = “010”). En este caso, se puede comprobar que el buffer de entrada se va llenando y cuando llega el último dato, se envían los datos al módulo Keccak.

El módulo Keccak procesa los datos y genera el hash de salida. En la Figura 35 se observa como estos datos se almacenan en el buffer de salida (wr_index++).

En la Figura 36 se procesa la primera instrucción de lectura de los datos de salida (funct3 = “011”). Los datos se leen del buffer de salida (rd_index++) en result_o. En la Figura 37, se muestra la lectura del último dato disponible en el buffer.

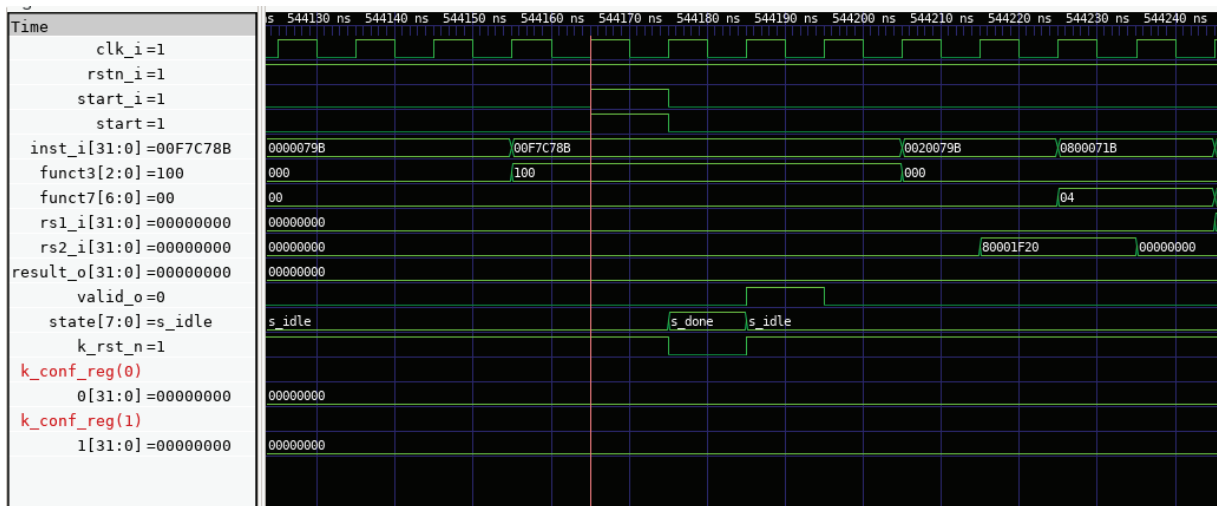


Figura 32: Resultado en simulación de la instrucción keccak_hw_reset

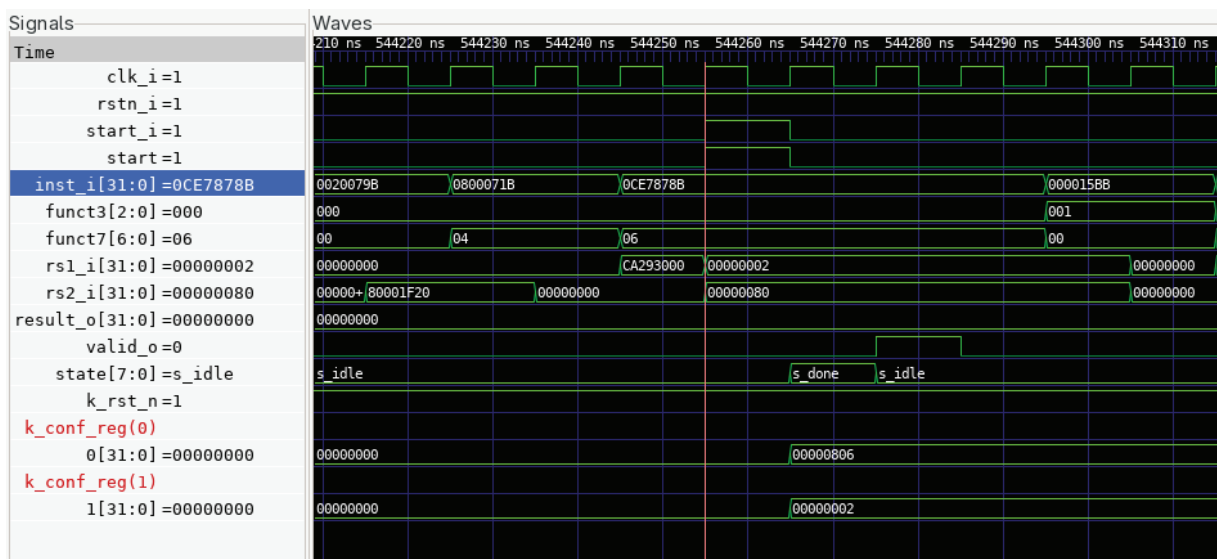


Figura 33: Resultado en simulación de la instrucción keccak_hw_init

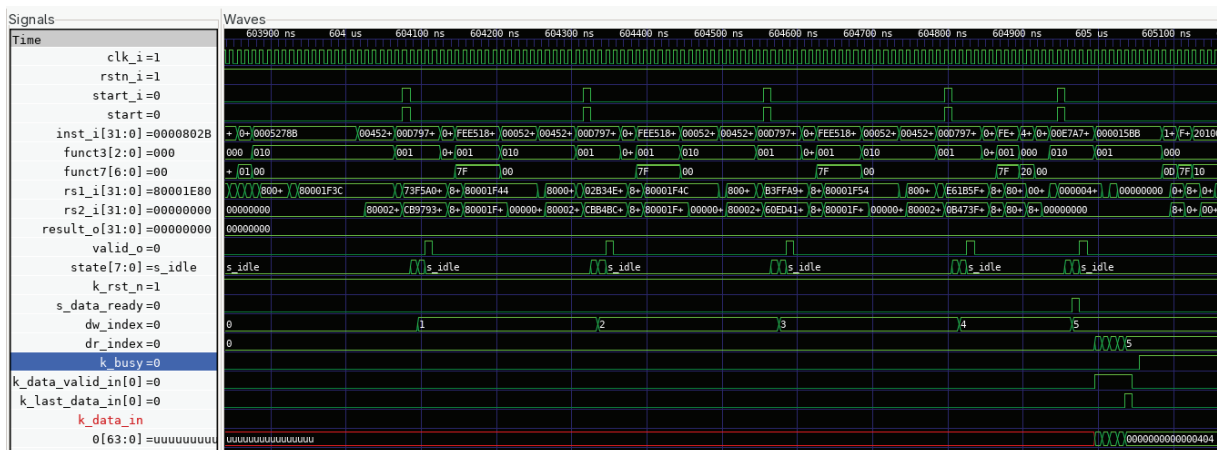


Figura 34: Resultado en simulación de las instrucciones keccak_hw_data y keccak_hw_data_last

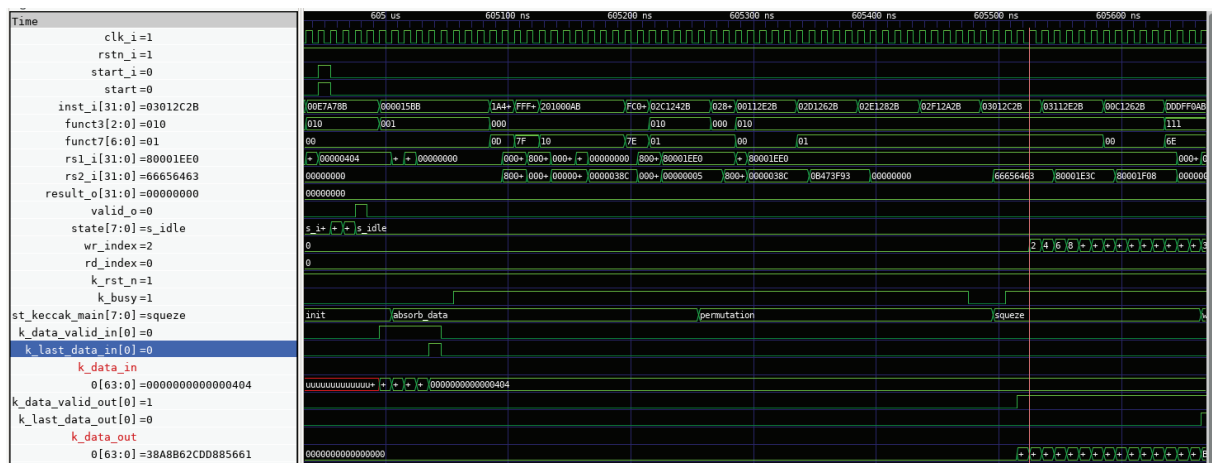


Figura 35: Resultado en simulación de la ejecución del núcleo Keccak

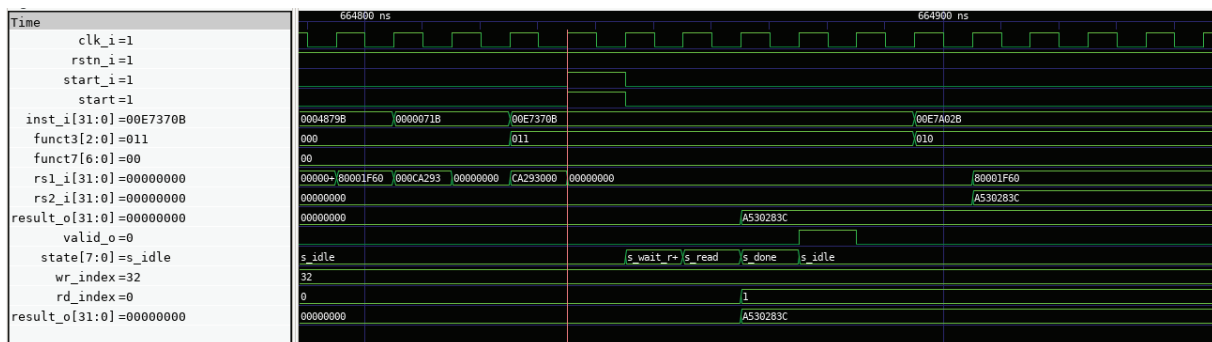


Figura 36: Resultado en simulación de la instrucción de lectura de resultados (1)

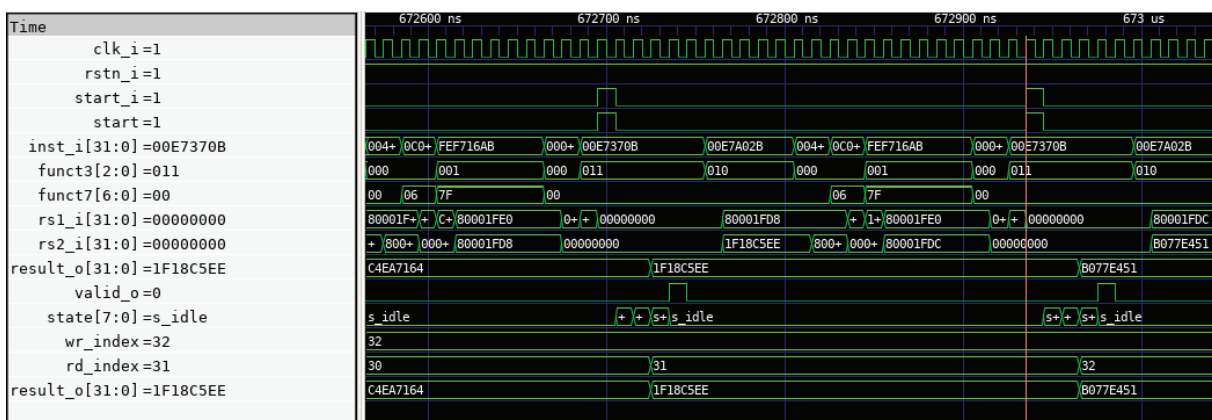


Figura 37: Resultado en simulación de la instrucción de lectura de resultados (2)

A continuación, se muestra en la [Figura 38](#) la salida de la [UART](#) en un ejemplo de ejecución de un SHAKE256 con 34 bytes de entrada y 128 bytes de salida. Este ejemplo se ha usado como referencia, ya que es uno de los tamaños de entrada y salida que se ejecutan dentro de la generación de claves de [ML-DSA](#). En el [Código 15](#) se presenta el programa creado para esta prueba de la [CFU](#). Para verificar que el resultado es correcto, se ha contrastado con datos precalculados con herramientas externas.

```
-- Prueba HW Keccak --
```

Message (34 bytes):

```
ea a0 f5 73 02 93 97 cb
5d 4e b3 02 6d bc b4 cb
92 a9 ff b3 82 41 ed 60
25 5f 1b e6 93 3f 47 0b
04 04
```

Reset hardware...

Hardware init

Sending 34 bytes...

Msg send

Reading 128 bytes...

Result (SHAKE256):

```
a530283c 6e249d46 dd885661 38a8b62c
1cff4357 fcca1eb2 d5a32886 883bcae9
fc56e73b d467502f bc56639d 324e7a0e
e9f2716b 69625c70 8063b42a 55c36001
83c3b8c0 b3c35fae eb2198b2 f1d24b9e
566db49a 892cc85d 9fb6f560 f77270bb
3e4a1043 87a86c6e 0b9316a1 570f7465
6852c7ea c4ea7164 1f18c5ee b077e451
```

Figura 38: Ejemplo de ejecución del algoritmo Keccak.

```
#include <neorv32.h>
#include <string.h>

#define BAUD_RATE 19200
#define MSG_LEN 34 // Tamaño del mensaje en bytes
#define HASH_LEN 128 // Tamaño de salida de SHAKE256 en bytes

// -----
// Define Keccak CFU instructions
// RISCV_INSTR_R_TYPE(opcode, funct3, funct7, rs1, rs2)
// -----
#define keccak_hw_init(funct7, rs1, rs2) RISCV_INSTR_R_TYPE(RISCV_OPCODE_CUSTOM0,
0b000, funct7, rs1, rs2)
#define keccak_hw_data(rs1, rs2) RISCV_INSTR_R_TYPE(RISCV_OPCODE_CUSTOM0, 0b001,
0b0000000, rs1, rs2)
#define keccak_hw_data_last(rs1, rs2) RISCV_INSTR_R_TYPE(RISCV_OPCODE_CUSTOM0,
0b010, 0b0000000, rs1, rs2)
```

```

#define keccak_hw_read() RISC_V_INSTR_R_TYPE(RISC_V_OPCODE_CUSTOM0, 0b011,
0b00000000, 0, 0)
#define keccak_hw_reset() RISC_V_INSTR_R_TYPE(RISC_V_OPCODE_CUSTOM0, 0b100,
0b00000000, 0, 0)

int main() {
    //UART
    neorv32_rte_setup();
    if (neorv32_uart0_available() == 0) return -1;
    neorv32_uart0_setup(BAUD_RATE, 0);

    //CFU Check
    if ((neorv32_cpu_csr_read(CSR_MXISA) & (1 << CSR_MXISA_XCFU)) == 0) {
        neorv32_uart0_printf("ERROR: CFU ('Xcfu') no implementada.\n");
        return -1;
    }

    neorv32_uart0_printf("\n-- Prueba HW Keccak --\n\n");

    // Msg
    uint32_t hash_salida[HASH_LEN / 4]; // 32 bytes
    uint8_t mensaje[MSG_LEN] = {
        0xea, 0xa0, 0xf5, 0x73, 0x02, 0x93, 0x97, 0xcb,
        0x5d, 0x4e, 0xb3, 0x02, 0x6d, 0xbc, 0xb4, 0xcb,
        0x92, 0xa9, 0xff, 0xb3, 0x82, 0x41, 0xed, 0x60,
        0x25, 0x5f, 0x1b, 0xe6, 0x93, 0x3f, 0x47, 0x0b,
        0x04, 0x04
    };

    // Reset Keccak
    neorv32_uart0_printf("Reset hardware...\n");
    keccak_hw_reset();

    // ],
    // caption: [Programa para la ejecución del algoritmo Keccak (1).],
    // supplement: [Código],
    // kind: raw
    //) <code:shake-code>

    //figure(
    // align(left)[
    // c

    // Init
    uint32_t funct7 = 6; // 6:SHAKE256, 5:SHAKE128
    uint32_t outlen = HASH_LEN;
    uint32_t cfg_rs1 = 0;
    if (MSG_LEN % 8 != 0) {
        cfg_rs1 = MSG_LEN % 8;
    }
    keccak_hw_init(funct7, cfg_rs1, outlen);

```

```

neorv32_uart0_printf("Hardware init\n");

//Send DATA
neorv32_uart0_printf("Sending %d bytes...\n", MSG_LEN);

int bytes_procesados = 0;
while ((MSG_LEN - bytes_procesados) > 8) {
    uint32_t low_32 = 0, high_32 = 0;
    memcpy(&low_32, &mensaje[bytes_procesados], 4);
    memcpy(&high_32, &mensaje[bytes_procesados + 4], 4);
    keccak_hw_data(low_32, high_32);
    bytes_procesados += 8;
}
// Send DATA_LAST
int bytes_restantes = MSG_LEN - bytes_procesados;
if (bytes_restantes > 0) {
    uint32_t last_low_32 = 0, last_high_32 = 0;
    if (bytes_restantes >= 4) {
        memcpy(&last_low_32, &mensaje[bytes_procesados], 4);
        if (bytes_restantes > 4) {
            memcpy(&last_high_32, &mensaje[bytes_procesados + 4],
bytes_restantes - 4);
        }
    } else {
        memcpy(&last_low_32, &mensaje[bytes_procesados], bytes_restantes);
    }
    keccak_hw_data_last(last_low_32, last_high_32);
}
neorv32_uart0_printf("Msg send\n");
// Read data
neorv32_uart0_printf("Reading %d bytes...\n", HASH_LEN);

for (int i = 0; i < (HASH_LEN / 4); i += 1) {
    hash_salida[i] = keccak_hw_read(); // Leer registro de salida
}

// Result
neorv32_uart0_printf("\nResult (SHAKE256):\n");
for (int i = 0; i < (HASH_LEN / 4); i++) {
    neorv32_uart0_printf("%x ", hash_salida[i]);
    if ((i + 1) % 4 == 0) neorv32_uart0_printf("\n");
}
return 0;
}

```

Código 15: Programa para la ejecución del algoritmo Keccak.

Problemas encontrados

Durante el diseño de la CFU se encontraron algunas dificultades que ha sido necesario abordar. En este apartado se detallan estos problemas y las soluciones implementadas.

- La implementación *hardware* del módulo Keccak utiliza entradas y salidas de 64 bits, mientras que el microprocesador NEORV32, al tratarse de una arquitectura de base RV32 utiliza registros de 32 bits. Por ello, la entrada se envía en dos registros rs1 y rs2. Para la salida, los datos se guardan en un *array* y se leen como palabras de 32 bits.
- Inicialmente, para comprobar la funcionalidad de la CFU se utilizó el simulador GHDL. El proyecto NEORV32 define la librería neorv32 como librería de trabajo. Durante la integración del módulo Keccak el simulador no detectaba la biblioteca a la que pertenecía el código. Para solucionarlo, es preciso incluir el módulo en la librería neorv32. Además, es necesario activar el modo de compatibilidad con la biblioteca de Synopsys, que no está disponible por defecto en la mayoría de los simuladores. Se resolvió añadiendo la opción `-fsynopsys`²).
- En la simulación se ha detectado que algunas instrucciones provocan una excepción de instrucción ilegal. Esto se debe a que la CPU tiene un tiempo máximo de espera para la ejecución de las instrucciones y, si la instrucción nunca devuelve un resultado válido, activa un *trap*. El problema se resuelve forzando que `valid_o <= "1"` al finalizar cada instrucción.
- El módulo Keccak está configurado de forma que necesita recibir todos los datos en ciclos consecutivos de reloj o empezará a absorber los datos aunque no se haya indicado que ha llegado el último dato. Para solucionarlo, cuando llegan las instrucciones de lectura, se guardan los datos en un *array* y se espera a tener el último dato antes de enviarlos al Keccak.

Mediante la simulación con GHDL, tras solucionar todos los problemas mencionados, se comprueba la funcionalidad de la CFU y la ejecución completa de los algoritmos de generación del *hash*, obteniendo resultados satisfactorios para la funcionalidad de la CFU.

El siguiente paso consisten en la integración de la invocación del módulo SHAKE256 dentro de la ejecución de la generación de claves. Para ello, se añade la directiva `#ifdef` que se muestra en el Código 16. De esta forma, se puede configurar mediante el Makefile si se ejecuta la función en *hardware* o en *software*, añadiendo la directiva `SHAKE_HW`.

Concretamente, la llamada a la función se ejecuta dos veces. En la Figura 39 se muestra una ejecución del código en *software*, mientras que en la Figura 40 se muestra la ejecución con la aceleración en *hardware*.

²-fsynopsys permite el uso de paquetes no estándares tales como `std_logic_arith`, `std_logic_signed`, `std_logic_unsigned`, `std_logic_textio`. Estos paquetes pueden estar presentes en código heredado pero no se recomienda en nuevos diseños. GHDL no soporta de forma nativa estos paquetes.

```

[12:39:46:334] TRNG...OK  $c_R$   $t_F$ 
[12:39:46:334] ML-DSA-44 on NEORV32...  $c_R$   $t_F$ 
[12:39:46:350] Generating keypair... DONE  $c_R$   $t_F$ 
[12:39:46:766] Signing message... DONE  $c_R$   $t_F$ 
[12:39:47:837] Verifying signature... DONE. Signature verified!  $c_R$   $t_F$ 
[12:39:48:380]  $c_R$   $t_F$ 
[12:39:48:380] SW Execution timing:  $c_R$   $t_F$ 
[12:39:48:396] Keypair SW = 39949535 cycles  $c_R$   $t_F$ 
[12:39:48:412] Sign SW = 105394254 cycles  $c_R$   $t_F$ 
[12:39:48:428] Verify SW = 51433564 cycles  $c_R$   $t_F$ 
[12:39:48:444] TOTAL SW = 196777353 cycles  $c_R$   $t_F$ 

```

Figura 39: Ejecución de ML-DSA en software .

```

[13:30:47:642] TRNG...OK  $c_R$   $t_F$ 
[13:30:47:642] CFU...OK  $c_R$   $t_F$ 
[13:30:47:658] ML-DSA-44 on NEORV32...  $c_R$   $t_F$ 
[13:30:47:674] Generating keypair... DONE  $c_R$   $t_F$ 
[13:30:48:074] Signing message... DONE  $c_R$   $t_F$ 
[13:30:51:814] Verifying signature... DONE. Signature verified!  $c_R$   $t_F$ 
[13:30:52:358]  $c_R$   $t_F$ 
[13:30:52:358] HW Execution timing:  $c_R$   $t_F$ 
[13:30:52:374] Keypair = 39121978 cycles  $c_R$   $t_F$ 
[13:30:52:390] Sign = 372321483 cycles  $c_R$   $t_F$ 
[13:30:52:390] Verify = 51416313 cycles  $c_R$   $t_F$ 
[13:30:52:406] TOTAL = 462859774 cycles  $c_R$   $t_F$ 

```

Figura 40: Ejecución de ML-DSA con aceleración hardware.

```

void mld_shake256(uint8_t *out, size_t outlen, const uint8_t *in, size_t inlen){
    #ifdef SHAKE_HW
        uint32_t hash_salida[outlen / 4];
        uint32_t funct7 = 6;
        uint32_t cfg_rs1 = 0;
        if (inlen % 8 != 0) {
            cfg_rs1 = inlen % 8;
        }
        keccak_hw_init(funct7, cfg_rs1, outlen); // INIT
        int bytes_procesados = 0;
        while ((inlen - bytes_procesados) > 8) {
            uint32_t low_32 = 0, high_32 = 0;
            memcpy(&low_32, &in[bytes_procesados], 4);
            memcpy(&high_32, &in[bytes_procesados + 4], 4);
            keccak_hw_data(low_32, high_32); // SEND DATA
            bytes_procesados += 8;
        }
    }
}

```

```

int bytes_restantes = inlen - bytes_procesados;
if (bytes_restantes > 0) {
    uint32_t last_low_32 = 0, last_high_32 = 0;
    if (bytes_restantes >= 4) {
        memcpy(&last_low_32, &in[bytes_procesados], 4);
        if (bytes_restantes > 4) {
            memcpy(&last_high_32, &in[bytes_procesados + 4], bytes_restantes - 4);
        }
    } else {
        memcpy(&last_low_32, &in[bytes_procesados], bytes_restantes);
    }
    keccak_hw_data_last(last_low_32, last_high_32); // SEND LAST DATA
}

for (int i = 0; i < (outlen / 4); i += 1) {
    hash_salida[i] = keccak_hw_read(); // READ OUTPUT
}
memcpy(out, hash_salida, outlen);
keccak_hw_reset(); // RESET

#else
mld_shake256ctx state;
mld_shake256_init(&state);
mld_shake256_absorb(&state, in, inlen);
mld_shake256_finalize(&state);
mld_shake256_squeeze(out, outlen, &state);
mld_shake256_release(&state);
#endif
}

```

Código 16: Función `mld_shake256` con la directiva `#ifdef`.

6.3. Conclusión

En este capítulo, se ha realizado la implementación de **ML-DSA** sobre el núcleo NEORV32. Inicialmente, se realizó una implementación completamente en *software* con la librería *mldsa-native*. Posteriormente, se incluyó el **TRNG** para asegurar la aleatoriedad de las claves generadas. Por último, se realizó la extensión del juego de instrucciones para la ejecución en *hardware* del algoritmo Keccak. Se implementa en la **CFU**, verificando su correcto funcionamiento tanto a nivel de simulación como sobre el prototipo *hardware*. En el siguiente capítulo se analizan y exponen los resultados obtenidos con ambas implementaciones.

Capítulo 7

Evaluación de los resultados

En este capítulo se analizan los resultados obtenidos con la implementación de [ML-DSA](#) en el microprocesador RISC-V NEORV32, comparando los resultados de la implementación *software* y la implementación con aceleración *hardware*.

7.1. Presentación de los resultados

En primer lugar, se muestran los resultados de la implementación en la [FPGA](#) del procesador RISC-V NEORV32. En la [Figura 41](#), se observa el resultado de la implementación antes del diseño de la [CFU](#) para la extensión del juego de instrucciones. Esta versión incluye el *wrapper* para permitir la depuración, algunas extensiones de la [ISA](#) y el [TRNG](#). Por ello, el uso de recursos mostrado en la [Tabla 15](#) es mayor que el presentado en el [Capítulo 5](#), que solo incluía la configuración mínima y la extensión M de la ISA. En este caso, el núcleo utiliza 3232 [LUT](#), 2248 registros y 34 bloques [RAM](#). El análisis temporal, mostrado en la [Figura 42](#), arroja un margen positivo con un [WNS](#) de 1.178 ns, lo que da como resultado una frecuencia máxima de operación teórica de aproximadamente 113 MHz.

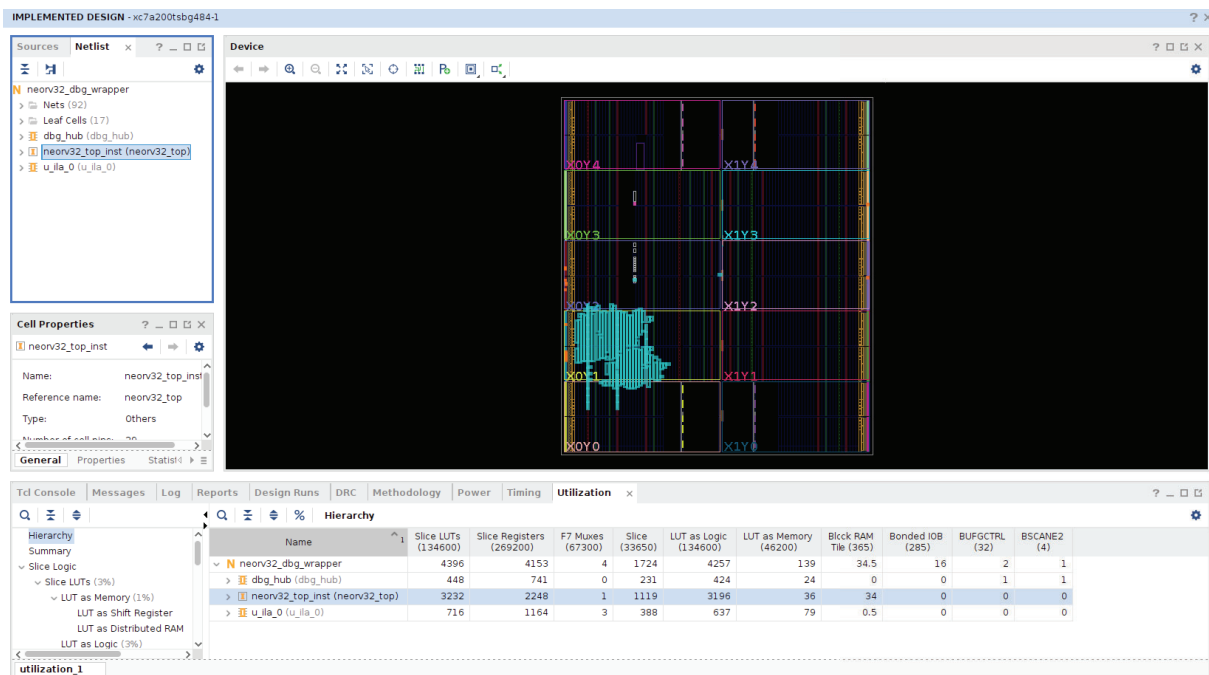


Figura 41: Resultado de la implementación del NEORV32 sobre la placa de prototipado Nexys Video.

Tabla 15: Utilización de recursos de la FPGA (segundo reporte de implementación)

Recurso Lógico	Utilizados	Disponibles	Utilización (%)
Slice LUTs	3232	134600	2.40
LUT as Logic	3196	134600	2.37
LUT as Memory	36	46200	0.08
Slice Registers	2248	269200	0.84
Block RAM (BRAM)	34	365	9.32

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 1.178 ns	Worst Hold Slack (WHS): 0.084 ns	Worst Pulse Width Slack (WPWS): 3.750 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 10317	Total Number of Endpoints: 10301	Total Number of Endpoints: 4403

All user specified timing constraints are met.

Figura 42: Timing report de la implementación de NEORV32

La Tabla 16 recoge los resultados del análisis de potencia proporcionados por la herramienta utilizada. El consumo total estimado es de 191 mW, de los cuales 153 mW son de potencia estática de la FPGA. El consumo dinámico refleja un total de 38 mW.

Tabla 16: Estimación del consumo de potencia de la implementación del núcleo NEORV32.

Parámetro de Potencia	Consumo Estimado (mW)
Potencia Estática del Dispositivo	153
Potencia Dinámica Total	38
Relojes	19
Señales	8
Lógica	6
Bloques RAM	5
Potencia Total Estimada	191

Por otro lado, en la Figura 43 se muestran los resultados tras la implementación añadiendo la CFU diseñada. El núcleo utiliza 37089 LUT, 39769 registros y 38 bloques RAM. En la Tabla 17 se recogen estos resultados, resaltando el consumo de recursos asociado a la CFU. Se puede observar cómo aumenta significativamente el uso de recursos en este caso. Esto se debe en gran medida a los arrays incluidos en la CFU para almacenar los datos de entrada y salida. Estos arrays se han traducido a LUT en hardware que componen estas estructuras de memoria por las características

de la implementación realizada. Como medida de mejora, está la sustitución de dichas estructuras por elementos de memoria dedicados en la propia **FPGA**, como **Block RAM (BRAM)**.

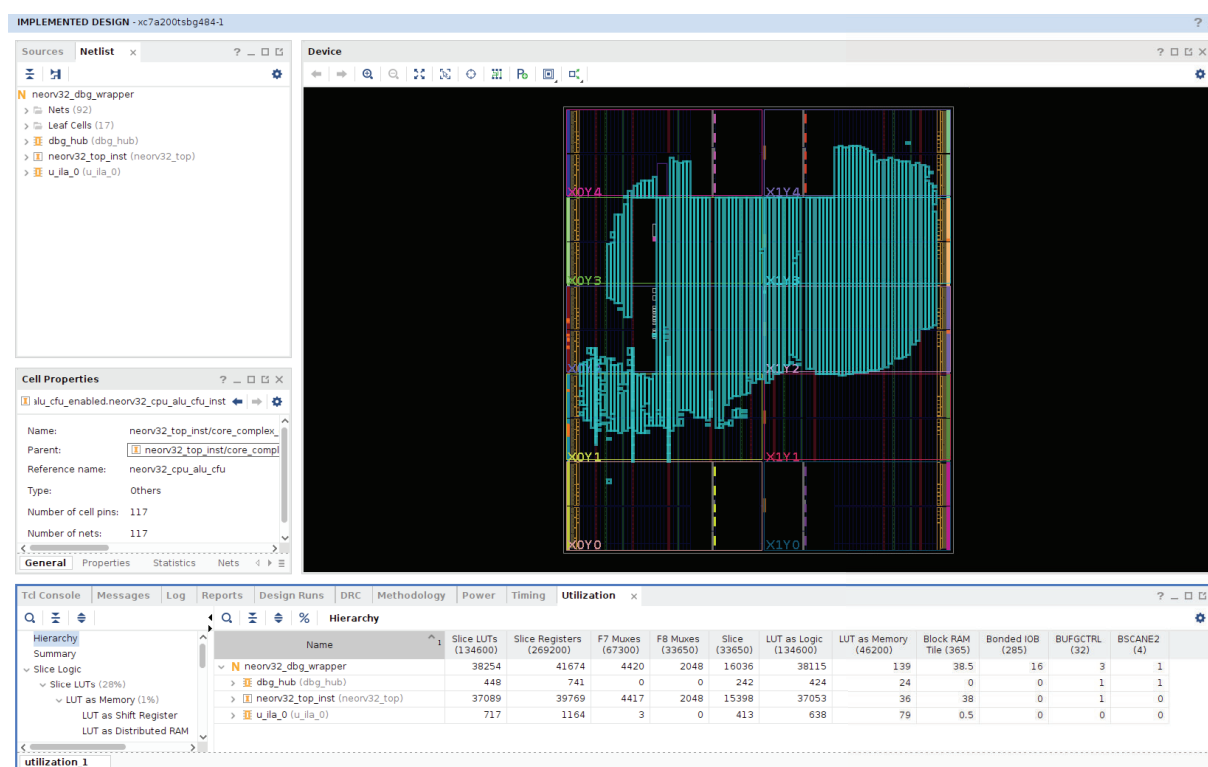


Figura 43: Resultado de la implementación del NEORV32 con CFU sobre la placa de prototipo Nexys Video.

Tabla 17: Utilización de recursos de NEORV32+CFU tras la síntesis e implementación

Recurso Lógico	Utilizados	CFU	Disponibles	Utilización (%)
Slice LUTs	37089	33816	134600	27.55
LUT as Logic	37053	33816	134600	27.53
LUT as Memory	36	0	46200	0.08
Slice Registers	39769	37479	269200	14.77
Block RAM (BRAM)	38	4	365	10.41

En la **Figura 44** se puede ver con más detalle el resultado de esta implementación sobre la placa Nexys Video. Se destaca en colores diferentes las estructuras más relevantes. Como ya han arrojado los datos, la **CFU** (en morado) consume un gran número de recursos. También, se destaca en color rosado el **TRNG**.

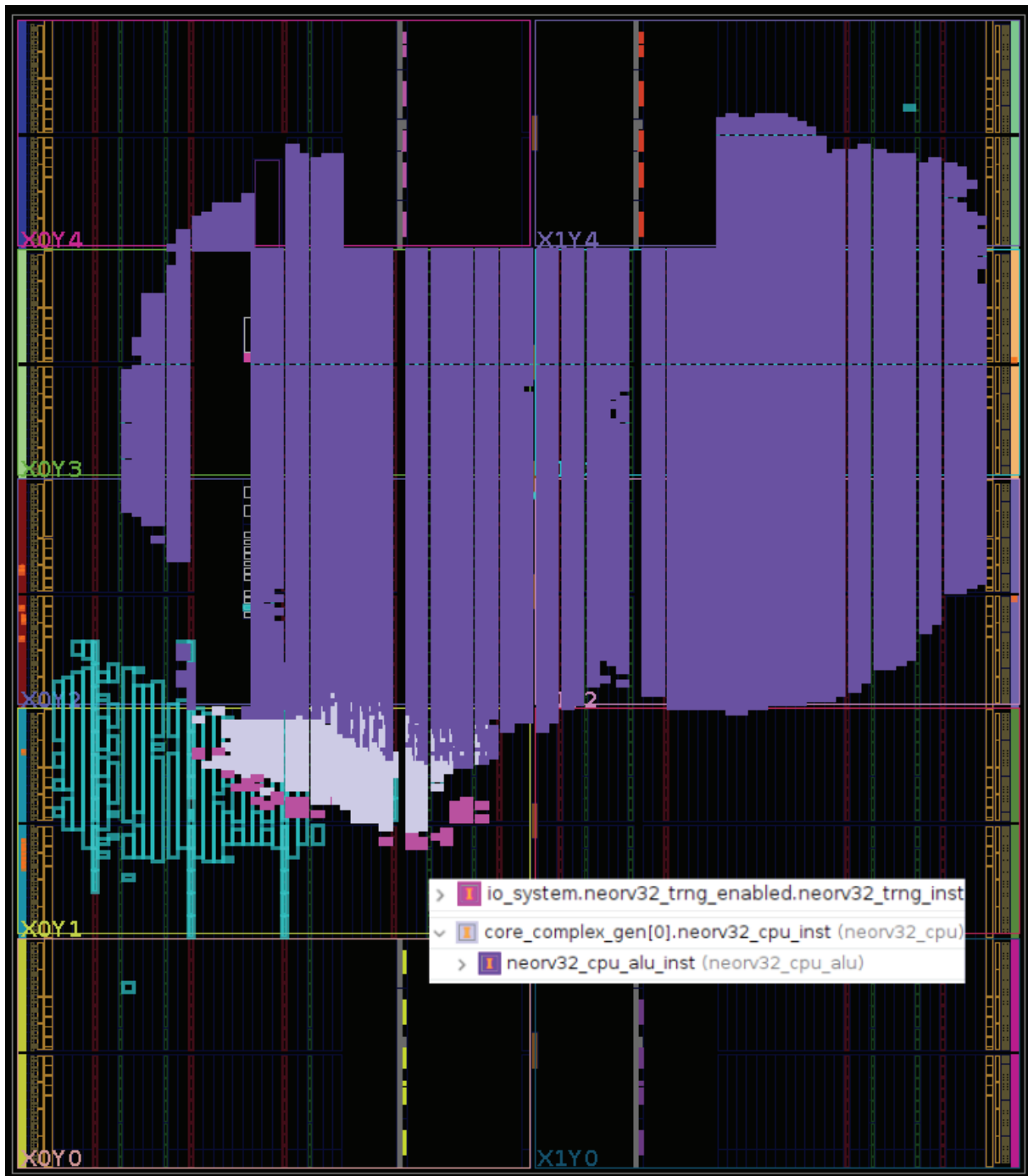


Figura 44: Resultado de la implementación de la CFU y el TRNG en NEORV32.

El análisis temporal se presenta en [Figura 45](#), que muestra que la ruta crítica también ha empeorado, si bien se siguen cumpliendo las restricciones de tiempo de ciclo para el diseño (10 ns). Este incremento en el retardo está directamente relacionado con el aumento de recursos utilizados, ya que la lógica queda mucho más dispersa físicamente a lo largo de la *FPGA*. En la [Figura 46](#) se muestra el *worst path*, que forma parte de la ruta entre el Keccak y el *buffer* de salida.

Design Timing Summary					
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS): 0.214 ns		Worst Hold Slack (WHS): 0.048 ns		Worst Pulse Width Slack (WPWS): 3.750 ns	
Total Negative Slack (TNS): 0.000 ns		Total Hold Slack (THS): 0.000 ns		Total Pulse Width Negative Slack (TPWS): 0.000 ns	
Number of Failing Endpoints: 0		Number of Failing Endpoints: 0		Number of Failing Endpoints: 0	
Total Number of Endpoints: 122365		Total Number of Endpoints: 122349		Total Number of Endpoints: 41734	
All user specified timing constraints are met.					

Figura 45: Timing report de la implementación de NEORV32 con CFU para una frecuencia de 100 MHz

Project Summary		Device	Path 21 - neorv32_dbg_wrapper_timing_summary_routed
Summary			
Name		Path 21	
Slack		0.219ns	
Source		neorv32_top_inst/core_complex_gen[0].neorv32_cpu_inst/neorv32_cpu_alu_inst/neorv32_cpu_alu_cfu_enabled.neorv32_cpu_alu_inst/keccak_inst/data_out_reg[0][61].rep/C	(rising edge-triggered cell FDRE clocked by sys_clk)
Destination		neorv32_top_inst/core_complex_gen[0].neorv32_cpu_inst/neorv32_cpu_alu_inst/neorv32_cpu_alu_cfu_enabled.neorv32_cpu_alu_inst/myarray_reg[623][29]D	(rising edge-triggered cell FDCE clocked by sys_clk)
Path Group		sys_clk_pin	
Path Type		Setup (Max at Slow Process Corner)	
Requirement		10.000ns (sys_clk_pin rise@10.000ns - sys_clk_pin rise@0.000ns)	
Data Path Delay		10.022ns (logic 0.560ns (5.787%) route 9.442ns (94.213%))	
Logic Levels		1 (LUT5=1)	
Clock Path Skew		0.247ns	
Clock Uncertainty		0.035ns	

Figura 46: Informe de la ruta crítica de la implementación de NEORV32 con CFU

Igual que en el caso anterior, la Tabla 18 muestra los resultados del análisis de potencia. El consumo total estimado es de 376 mW, siendo 154 mW de consumo de potencia estática y 222 mW de consumo dinámico.

Tabla 18: Estimación del consumo de potencia de la implementación del núcleo NEORV32.

Parámetro de Potencia	Consumo Estimado (mW)
Potencia Estática del Dispositivo	154
Potencia Dinámica Total	222
Relojes	79
Señales	88
Lógica	47
Bloques RAM	8
Potencia Total Estimada	376

Para analizar en profundidad el tiempo dedicado a cada una de las instrucciones diseñadas para la ejecución del algoritmo, se han extraído sus tiempos de ejecución. En la Tabla 19 se muestra la latencia en milisegundos y en ciclos de reloj de cada una de las fases. Concretamente, se ha medido el tiempo entre el flanco de subida de la señal de entrada de la CFU start_i y el flanco de subida de la señal de salida valid_o.

Se ha analizado la ejecución de una operación SHAKE256 con 34 bytes de entrada y 128 bytes de salida. En primer lugar, observamos que las instrucciones de keccak_hw_reset y keccak_hw_init tardan 2 ciclos en completarse cada una. La ejecución de keccak_hw_data y keccak_hw_data_last tarda 90 ciclos. En este caso, como son 34 bytes de entrada son necesarias cuatro instrucciones keccak_hw_data y una keccak_hw_data_last. Se ha añadido el tiempo que dura el proceso de enviar

los datos al núcleo Keccak una vez que el buffer tiene todos los datos de entrada, que es de 7 ciclos. El núcleo Keccak tarda 67 ciclos desde que recibe el primer dato, primer flanco de subida de `k_data_valid_in`, hasta que devuelve el último dato, flanco de subida de `k_last_data_out`. Por último, la lectura del resultado con `keccak_hw_read` tarda 567 ciclos. En esta simulación, la primera instrucción de `keccak_hw_read` llega antes de tener datos disponibles para leer y la FSM pasa a `S_WAIT_READ` hasta que termina la ejecución del Keccak y hay un dato disponible. También, es importante recordar que la salida de los datos se lee en palabras de 32 bits. En este ejemplo la salida es de 128 bytes por lo que es necesario ejecutar 32 veces `keccak_hw_read`. En la [Figura 47](#) se muestra la ejecución completa, que dura 708 ciclos. Durante este periodo también se registran ciclos en los que la CPU realiza otras ejecuciones, como puede ser la carga de los datos en los registros.

Tabla 19: Caracterización del acelerador de instrucción en hardware

Etapas	Duración (ns)	Ciclos de reloj
<code>keccak_hw_reset</code>	20	2
<code>keccak_hw_init</code>	20	2
<code>keccak_hw_data</code> y <code>keccak_hw_data_last</code>	900	90
<code>p_send_data</code>	70	7
Keccak	670	67
<code>keccak_hw_read</code>	5670	567
Ejecución completa	7080	708

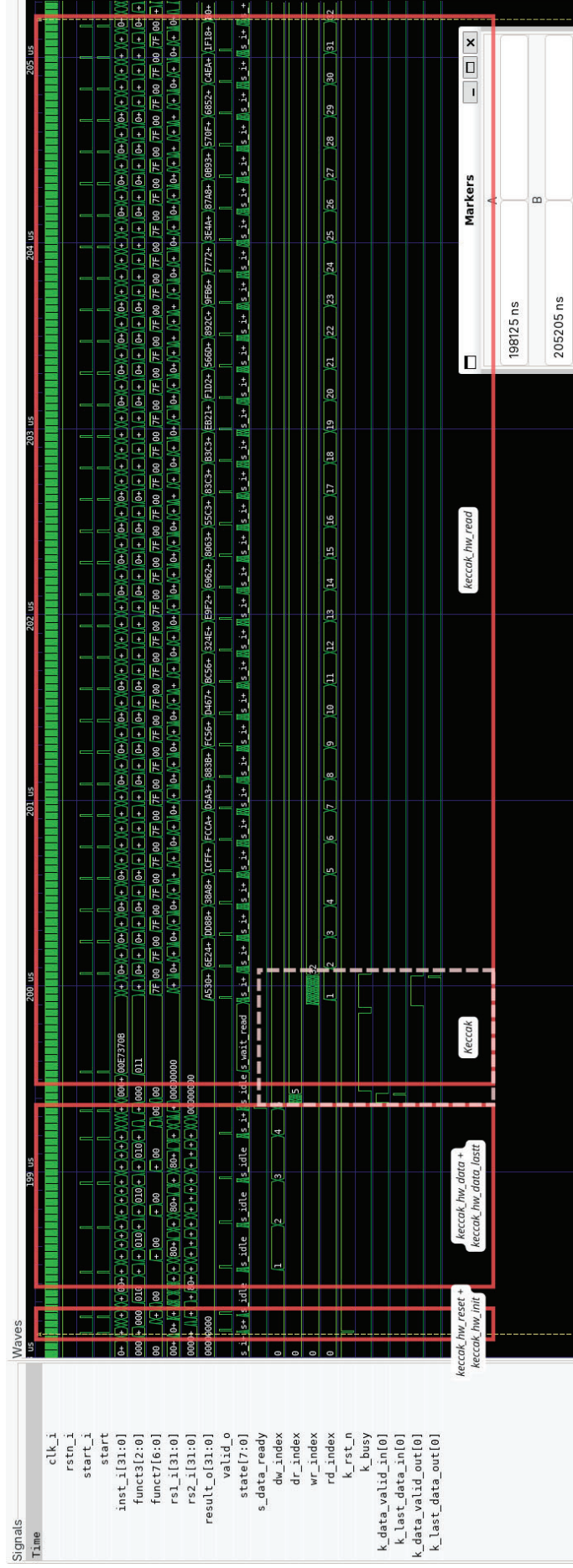


Figura 47: Resultado de simulación de una ejecución completa

La ejecución de SHAKE256 en *hardware* tarda 2.171 ciclos de reloj en ejecutarse para un mensaje de entrada de 34 bytes y un resultado de 128 bytes. La ejecución incluye un `keccak_hw_reset`, la inicialización del sistema mediante `keccak_hw_init`, el envío de datos con `keccak_hw_data` y `keccak_hw_data_last`, la ejecución del Keccak y la recepción de los datos de salida con `keccak_hw_read`. En la [Tabla 20](#), se observan estos datos y su comparación con la ejecución en *software*. En la segunda ejecución, el tiempo es mayor ya que el tamaño de los datos de entrada es superior.

Para cuantificar la mejora de rendimiento a nivel de bloque, se emplea la métrica de aceleración local (A). Este factor se calcula dividiendo el tiempo de ejecución de la implementación original en *software* (T_{software}) entre el tiempo de ejecución de la solución acelerada (T_{hardware}), determinando cuántas veces más rápida es la nueva implementación respecto a la base.

A partir de esta formulación y los datos obtenidos, se puede afirmar que la solución *hardware* logra un rendimiento local 55 veces superior en la primera ejecución y 61 veces mayor en la segunda. De manera conjunta, ambas ejecuciones presentan un factor de aceleración local de x60.

Tabla 20: Tiempos de ejecución y factor de aceleración local de las operaciones SHAKE.

(ML-DSA-44)	SHAKE <i>software</i>	SHAKE <i>hardware</i>	Aceleración local (A)
1ª ejecución	119813	2171	x55
2ª ejecución	1186782	19351	x61
TOTAL	1306595	21522	x60.7

La [Figura 48](#) muestra un gráfico de cajas y bigotes con los resultados de ejecución, en ciclos de reloj, de las distintas fases del algoritmo [ML-DSA](#). Esta representación permite visualizar no solo la tendencia central de los tiempos de ejecución, sino también su dispersión y la presencia de casos atípicos.

En la fase de firma se puede observar que su ejecución presenta un amplio rango de variación, lo cual se debe a que en algunos casos es necesario repetir la firma para asegurarse de que el mensaje cifrado es seguro y no revela información de la clave privada. La gráfica muestra mediante círculos estos casos con valores extremos. Por el contrario, las fases de generación de claves y verificación muestran menor varianza en los resultados.

Dado que la aceleración se incluye en el procedimiento de generación de claves, es en este caso en el que se observa una mayor diferencia entre las ejecuciones *software* y *hardware*.

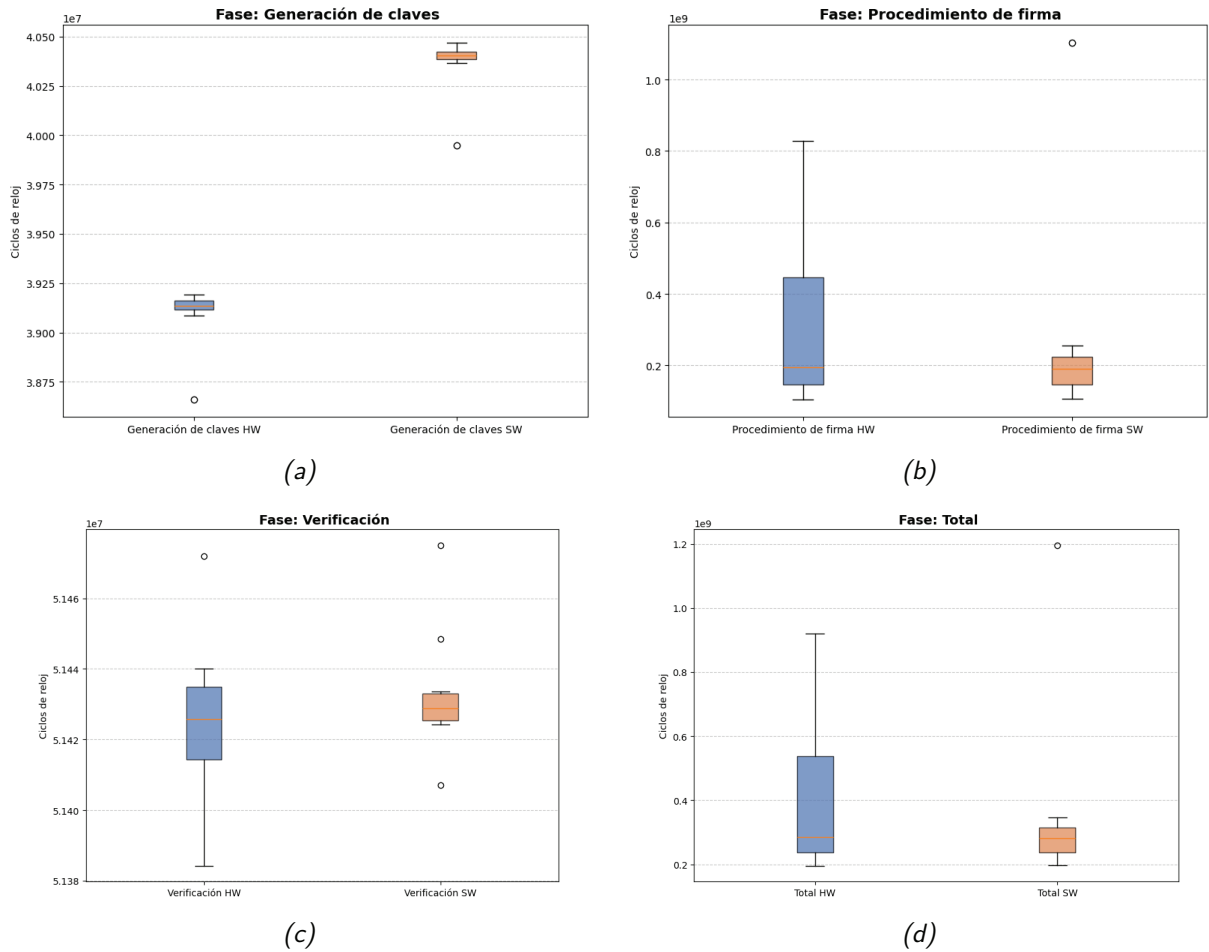


Figura 48: Resultados en ciclos de reloj de la ejecución de ML-DSA con aceleración y sin aceleración.

Para poder interpretar los datos, se ha generado una media para la comparación. En este caso, la diferencia se observa en la fase de generación de claves, única etapa donde se ha integrado la aceleración *hardware*. La ejecución en esta fase pasa de 40.366.001 a 39.092.361 ciclos. Para modelar este comportamiento, se aplica la Ley de Amdahl [59], la cual define la aceleración teórica máxima (S) basándose en la proporción de tiempo que se beneficia de la mejora (P) y la aceleración local aplicada (A):

$$S = \frac{1}{(1 - P) + \frac{P}{A}}$$

Durante la generación de claves en *software*, las llamadas a la función SHAKE (1.306.595 ciclos) suponen una fracción de tiempo mejorable $P = 0.0324$ (3.24%). Sustituyendo este valor y el factor de aceleración local obtenido previamente ($A = 60.7$), se obtiene un *speed-up* de $S = \frac{1}{0.9676 + \frac{0.0324}{60.7}} \approx 1.0329$.

Gran parte de estas operaciones restantes de ML-DSA no están pensadas para una aceleración a nivel de instrucción, ya que manejan un gran volumen de datos y esto supone un cuello de botella. En un sistema final, esto se suele abordar con aceleradores específicos «*loosely-coupled*» conectados directamente a un controlador de memoria que les proporciona los datos de entrada y almacena las

salidas. Por lo tanto, se puede definir como trabajo futuro identificar otras funcionalidades adecuadas para su aceleración a nivel de instrucciones.

7.2. Comparativa con otros trabajos

En este apartado, se compara los resultados obtenidos con otros trabajos recientes en la literatura que implementen aceleración de algoritmos [PQC](#) con arquitecturas RISC-V. Como no se ha realizado un co-diseño completo con aceleración a nivel de instrucciones, más allá de la función SHAKE, se resaltan dos artículos particularmente interesantes y estrechamente relacionados con este desarrollo.

En la [Tabla 21](#) se presenta una comparativa de este trabajo frente a las soluciones propuestas por Li et al. en [\[60\]](#) y Karl et al. en [\[61\]](#).

Tabla 21: Comparativa de implementaciones

Métrica	Este trabajo	Li et al. (2022) [60]	Karl et al. (2024) [61]
Arquitectura Base	NEORV32	RISC-V con <i>pipeline</i> de 4 etapas	PULPino
Enfoque de Integración Keccak	Fuertemente acoplado (CFU) ejecutando el algoritmo completo	Módulo reconfigurable (instrucción multi-ciclo)	Fuertemente acoplado ejecutando solo la función Keccak-f1600
Aceleración	Aceleración local de 60.7x	Mejora de rendimiento de 100x (con extensión completa)	No se especifica
Aceleración Global (Generación de claves)	Factor de 1.03x (Acelerando solo SHAKE)	-	Factor de 3.33x (Acelerando SHAKE + NTT)
Aumento de Recursos	+33816 LUTs de la CFU	Aumento del área al triple respecto al diseño original	+4.782 LUTs

Comparando estas implementaciones se puede extraer algunas áreas de mejora del trabajo actual.

1. **Aceleración de funciones adicionales:** Los resultados globales muestran que acelerar únicamente el módulo SHAKE proporciona una mejora limitada. El trabajo de Karl et al. demuestra que incorporar un acelerador dedicado para la transformación [NTT](#) y la aritmética polinómica mejora la aceleración global a un factor de 3.33x para la generación de claves y hasta 6.01x para la firma en [ML-DSA](#).
2. **Optimización del almacenamiento de estado y *buffers*:** Como se evidencia en la evaluación de recursos, el uso de *arrays* descritos en [VHDL](#) dentro de la [CFU](#) genera un consumo excesivo de LUTs. Como trabajo futuro se propone modificar la arquitectura para reutilizar los bancos de registros de propósito general para almacenar el estado del Keccak, tal y como implementan Karl et al.

7.3. Conclusión

En este capítulo se ha presenten y evalúan los resultados obtenidos tras la implementación del algoritmo **ML-DSA** en el núcleo RISC-V NEORV32, comparando la ejecución en *software* frente a la versión con aceleración *hardware*.

El análisis de área y potencia revela que la integración de **CFU** implica un incremento en el consumo de recursos lógicos de la **FPGA**. Pese a este aumento de área y al impacto sobre la ruta crítica, el diseño logra cumplir con éxito las restricciones temporales para funcionar a la frecuencia objetivo de 100 MHz.

Los resultados de la extensión del juego de instrucciones muestran que se reduce en promedio en x60 veces el número de ciclos necesario para la ejecución de SHAKE256.

No obstante, al evaluar el impacto de esta mejora en la ejecución completa mediante la Ley de Amdahl, se observa que la ganancia en la fase de generación de claves es moderada. Esto ocurre porque las llamadas a la función SHAKE constituyen un porcentaje reducido del tiempo total de ejecución. Del análisis realizado se extraen conclusiones que indican la dirección en los siguientes trabajos a realizar para la aceleración completa del proceso de firma.

Capítulo 8

Conclusiones y trabajos futuros

8.1. Conclusiones

Este TFM presenta la investigación, el diseño y el desarrollo de la implementación de instrucciones personalizado en arquitectura RISC-V para el algoritmo de firma digital post-cuántica Keccak cumpliendo así con el objetivo propuesto al inicio del proyecto.

Las fases iniciales de investigación se centran en el estudio de la criptografía post-cuántica y la arquitectura RISC-V con objeto de dar un marco de referencia al trabajo. Para estudiar la criptografía post-cuántica se parte de la base conceptual utilizada en la criptografía, justificando la necesidad de disponer de algoritmos seguros a ataques mediante computación cuántica. El NIST y su papel en la estandarización de nuevos algoritmos es clave para entender el futuro próximo de los sistemas criptográficamente seguros. Por otro lado, se investiga sobre la arquitectura RISC-V, su arquitectura, entendiendo para ello la familia de ISA disponibles.

En los siguientes capítulos, los esfuerzos se centran en estudiar en detalle los algoritmos de firma digital post-cuánticos y revisar distintas implementaciones disponibles en la literatura, todas ellas dentro del contexto de su implementación sobre la arquitectura RISC-V. El algoritmo seleccionado para su implementación, ML-DSA, es un estándar aprobado por el NIST. Con objeto de analizar la idoneidad de implementaciones de referencia de RISC-V para dar soporte a la implementación del algoritmo seleccionado, se realiza estudio de dichas implementaciones obteniendo valores cualitativos para los criterios de selección definidos previamente.

Finalmente se seleccionan tres núcleos y se realiza el proceso de análisis cuantitativo de los parámetros PPA con objeto de establecer un arquitecturas RISC-V de referencia para el desarrollo de las nuevas instrucciones. La conclusión obtenida indican que las facilidades aportadas por el mecanismo CFU de NEORV32 para la integración de aceleradores en el cauce de ejecución facilita el trabajo de desarrollo, mientras que los parámetros PPA obtenido para NEORV32 están dentro de márgenes razonables para permitir la implementación en un dispositivo FPGA de bajo coste. Un valor añadido es la disponibilidad del bloque TRNG que permite generar números aleatorios desde el propio hardware.

La implementación de ML-DSA realizada en NEORV32 utiliza la Custom Functions Unit que incluye la arquitectura del NEORV32 junto con una librería PQC. Con esta unidad es posible extender el juego de instrucciones para ejecutar en hardware la función hash SHAKE256 utilizando el TRNG para asegurar que el algoritmo dispone de una semilla realmente aleatoria. La implementación realizada, una vez realizada la simulación del nuevo sistema en VHDL, permite obtener métricas de interés para el análisis de resultados y la futura evolución del proyecto. Se ha visto la aceleración efectiva en la fase de generación de claves en un factor mayor que x55, con lo que podemos dar el desarrollo como exitoso.

Con objeto de poner en contexto del estado de la técnica el trabajo realizado, se comparan los resultados obtenidos con otras implementaciones del estado del arte. Se concluye que el juego

de instrucciones personalizado es capaz de acelerar algunas funciones de [ML-DSA](#), contribuyendo a acelerar la ejecución total del algoritmo.

8.2. Trabajos futuros

Una vez completado el trabajo propuesto, se han identificado trabajos futuros a ser considerados:

1. **Optimizar el uso de recursos lógicos en la [CFU](#).** Actualmente se cumple con los requisitos funcionales propuestos, pero los resultados muestran que el uso de recursos es mejorable. Uno de los puntos ya comentados es la posibilidad de sustituir los *buffers* creados en la [CFU](#) por elementos de memoria dedicados en la propia [FPGA](#), como [BRAM](#).
2. **Evaluar otras aceleraciones de [ML-DSA](#).** Se plantea avanzar con el estudio de otras funcionalidades dentro del algoritmo que sean candidatas a la aceleración *hardware* con el uso de instrucciones personalizadas.
3. **Optimizar la transferencia de datos.** En la ejecución de un ciclo completo de Keccak se comprueba que gran parte del tiempo se consume en la transferencia de los datos. Actualmente, la comunicación entre la [CPU NEORV32](#) y la [CFU](#) depende de las instrucciones de lectura y escritura a través de los registros del procesador. Se propone investigar otros mecanismos que permitan que el acelerador reciba y envíe bloques de datos de forma independiente a la ejecución de la [CPU](#). Ello permitirá solapar los ciclos de ejecución y transferencias de datos con mecanismos tipo *ping-pong* o disponer de un *buffer* de memoria circular.
4. **Analizar la seguridad física del sistema.** El algoritmo [ML-DSA](#) es seguro matemáticamente, pero su implementación puede ser vulnerable a ataques de canal lateral. Una línea de investigación futura puede estar centrada en probar el diseño frente a inyección de fallos, o evaluar contramedidas que permitan concluir con un diseño totalmente seguro.

Referencias

- [1] D. Boneh y V. Shoup, *A Course in Cryptography*. Online Manuscript, 2023. [En línea]. Disponible en: https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_6.pdf
- [2] D. Cordoba y M. Méndez Garabetti, «Criptografía Post Cuántica», en *XIX Workshop de Investigadores en Ciencias de la Computación (WICC 2017, ITBA, Buenos Aires)*, 2017.
- [3] «What Is Post-Quantum Cryptography?», *NIST*, ago. 2024, Accedido: 20 de abril de 2026. [En línea]. Disponible en: <https://www.nist.gov/cybersecurity-and-privacy/what-post-quantum-cryptography>
- [4] I. T. L. Computer Security Division, «Post-Quantum Cryptography | CSRC | CSRC». Accedido: 20 de abril de 2026. [En línea]. Disponible en: <https://csrc.nist.gov/projects/post-quantum-cryptography>
- [5] TNO, «Post-quantum cryptography: Status, developments and recommendations», The Hague, Netherlands, technical report TNO 2024 P11444, 2024. [En línea]. Disponible en: <https://publications.tno.nl/publication/34643386/fXcPVHsX/TNO-2024-pqc-en.pdf>
- [6] A. Dolmeta, V. Piscopo, G. Masera, M. Martina, y M. Hutter, «HORCRUX - A Lightweight PQC-RISC-V eXtension Architecture». Accedido: 20 de abril de 2026. [En línea]. Disponible en: <https://eprint.iacr.org/2025/1934>
- [7] National Cyber Security Centre, «Next steps for preparing for Post-Quantum Cryptography (PQC)», Technical Guidance, 2023. [En línea]. Disponible en: <https://www.ncsc.gov.uk/sites/default/files/documents/NCSC-PQC-timeline.pdf>
- [8] Z. Ye, R. Song, H. Zhang, D. Chen, R. C.-C. Cheung, y K. Huang, «A Highly-efficient Lattice-based Post-Quantum Cryptography Processor for IoT Applications», *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, n.º 2, pp. 130-153, mar. 2024, doi: [10.46586/tches.v2024.i2.130-153](https://doi.org/10.46586/tches.v2024.i2.130-153).
- [9] T. Fritzmann, G. Sigl, y J. Sepúlveda, «RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography». [En línea]. Disponible en: <https://eprint.iacr.org/2020/446>
- [10] RISC-V International, «Ratified Specifications». Accedido: 20 de abril de 2026. [En línea]. Disponible en: <https://riscv.org/specifications/ratified/>
- [11] RISC-V International, «RISC-V International - Home». Accedido: 16 de julio de 2026. [En línea]. Disponible en: <https://riscv.org/>
- [12] «PQC / Keccak Instructions Ratification Plan - Post-Quantum Cryptography - RISC-V Tech Hub». Accedido: 7 de abril de 2026. [En línea]. Disponible en: <https://riscv.atlassian.net/wiki/spaces/PQC/pages/485425183/PQC+Keccak+Instructions+Ratification+Plan>
- [13] «¿Qué es la criptografía? | IBM». Accedido: 8 de junio de 2026. [En línea]. Disponible en: <https://www.ibm.com/es-es/think/topics/cryptography>
- [14] «Cifrado César». 29 de abril de 2026. Accedido: 14 de julio de 2026. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Cifrado_C%C3%A9sar&oldid=173209768

- [15] P. Gutiérrez, «Tipos de criptografía: simétrica, asimétrica e híbrida». Accedido: 13 de julio de 2026. [En línea]. Disponible en: <https://www.genbeta.com/desarrollo/tipos-de-criptografia-simetrica-asimetrica-e-hibrida>
- [16] «The status of quantum computer development». Accedido: 14 de julio de 2026. [En línea]. Disponible en: <https://www.bsi.bund.de/EN/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Quantentechnologien-und-Post-Quanten-Kryptografie/Entwicklungsstand-Quantencomputer/entwicklungsstand-quantencomputer.html?nn=916616>
- [17] P. Shor, «Algorithms for quantum computation: discrete logarithms and factoring», en *Proceedings 35th Annual Symposium on Foundations of Computer Science*, nov. 1994, pp. 124-134. doi: [10.1109/SFCS.1994.365700](https://doi.org/10.1109/SFCS.1994.365700).
- [18] L. K. Grover, «A fast quantum mechanical algorithm for database search». Accedido: 8 de junio de 2026. [En línea]. Disponible en: <http://arxiv.org/abs/quant-ph/9605043>
- [19] Ashish Kundu, Ramana Kompella, «The Quantum-Cryptographic Co-evolution». Accedido: 13 de julio de 2026. [En línea]. Disponible en: <https://arxiv.org/html/2604.02591v1>
- [20] «Half of organizations worry about quantum 'harvest now, decrypt later' attacks». Accedido: 8 de junio de 2026. [En línea]. Disponible en: <https://siliconangle.com/2022/09/20/half-of-organizations-concerned-quantum-harvest-now-decrypt-later-attacks/>
- [21] Palo Alto Networks, «Harvest Now, Decrypt Later: Quantum Security Risk». Accedido: 8 de junio de 2026. [En línea]. Disponible en: <https://www.paloaltonetworks.com/cyberpedia/harvest-now-decrypt-later-hndl>
- [22] N. Bindel, M. Mosca, y B. Munson, «The Quantum Threat to Cybersecurity and Privacy», en *The Boundaries of Data*, 1.^a ed., London: Routledge, 2025, pp. 35-51. doi: [10.5117/9789463729192_CH03](https://doi.org/10.5117/9789463729192_CH03).
- [23] I. T. L. Computer Security Division, «Call for Proposals - Post-Quantum Cryptography | CSRC | CSRC». Accedido: 25 de mayo de 2026. [En línea]. Disponible en: <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/call-for-proposals>
- [24] I. T. L. Computer Security Division, «Security (Evaluation Criteria) - Post-Quantum Cryptography | CSRC | CSRC». Accedido: 16 de julio de 2026. [En línea]. Disponible en: [https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Evaluation-Criteria/Security-\(Evaluation-Criteria\)](https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Evaluation-Criteria/Security-(Evaluation-Criteria))
- [25] National Institute of Standards and Technology (US), «Module-lattice-based key-encapsulation mechanism standard», Washington, D.C., NIST FIPS 203, ago. 2024. doi: [10.6028/NIST.FIPS.203](https://doi.org/10.6028/NIST.FIPS.203).
- [26] National Institute of Standards and Technology (US), «Module-lattice-based digital signature standard», Washington, D.C., NIST FIPS 204, ago. 2024. doi: [10.6028/NIST.FIPS.204](https://doi.org/10.6028/NIST.FIPS.204).
- [27] National Institute of Standards and Technology (US), «Stateless hash-based digital signature standard», Washington, D.C., NIST FIPS 205, ago. 2024. doi: [10.6028/NIST.FIPS.205](https://doi.org/10.6028/NIST.FIPS.205).

- [28] D. Cooper, D. Apon, Q. Dang, M. Davidson, M. Dworkin, y C. Miller, «Recommendation for Stateful Hash-Based Signature Schemes», NIST Special Publication (SP) 800–208, oct. 2020. doi: [10.6028/NIST.SP.800-208](https://doi.org/10.6028/NIST.SP.800-208).
- [29] National Security Agency y Cybersecurity and Infrastructure Security Agency, «Commercial National Security Algorithm Suite 2.0», technical report, may 2025. [En línea]. Disponible en: https://media.defense.gov/2025/May/30/2003728741/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS.PDF
- [30] B. Schneier, «NIST's Post-Quantum Cryptography Standards». Accedido: 16 de julio de 2026. [En línea]. Disponible en: <https://www.schneier.com/blog/archives/2022/08/nists-post-quantum-cryptography-standards.html>
- [31] J. G. Underhill, S. A. Kovac, y X. Bogomolec, «Towards post-quantum symmetric cryptography». Accedido: 16 de julio de 2026. [En línea]. Disponible en: <https://eprint.iacr.org/2019/553>
- [32] «The RISC-V Instruction Set Manual, Volume I :: RISC-V Ratified Specifications Library». Accedido: 22 de mayo de 2026. [En línea]. Disponible en: <https://docs.riscv.org/reference/isa/unpriv/unpriv-index.html>
- [33] RISC-V International, «RISC-V Application Binary Interface (ABI) - v1.0». Accedido: 27 de julio de 2026. [En línea]. Disponible en: <https://docs.riscv.org/reference/abi/v1.0/index.html>
- [34] «The RISC-V Instruction Set Manual, Volume II :: RISC-V Ratified Specifications Library». Accedido: 22 de mayo de 2026. [En línea]. Disponible en: <https://docs.riscv.org/reference/isa/priv/priv-index.html>
- [35] Wadix Technologies, «RISC-V Supported Privilege Modes (User/Supervisor/Machine) Explained». Accedido: 27 de julio de 2026. [En línea]. Disponible en: <https://levelup.gitconnected.com/understanding-risc-v-privilege-modes-isolation-security-and-software-control-29d8a93453d4>
- [36] National Institute of Standards and Technology (US), «Digital Signature Standard (DSS)», Washington, D.C., NIST FIPS 186–5, feb. 2023. doi: [10.6028/NIST.FIPS.186-5](https://doi.org/10.6028/NIST.FIPS.186-5).
- [37] R. W. Shirey, «Internet Security Glossary, Version 2», Request for Comments RFC 4949, ago. 2007. doi: [10.17487/RFC4949](https://doi.org/10.17487/RFC4949).
- [38] S. Bai *et al.*, «CRYSTALS-Dilithium», feb. 2021.
- [39] P. Schwabe, «Dilithium». Accedido: 14 de julio de 2026. [En línea]. Disponible en: <https://pq-crystals.org/dilithium/>
- [40] National Institute of Standards and Technology (US), «SHA-3 standard : permutation-based hash and extendable-output functions», Washington, D.C., error: 202, 2015. doi: [10.6028/NIST.FIPS.202](https://doi.org/10.6028/NIST.FIPS.202).
- [41] T. Wang, C. Zhang, X. Zhang, D. Gu, y P. Cao, «Optimized Hardware-Software Co-Design for Kyber and Dilithium on RISC-V SoC FPGA», *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, n.º 3, pp. 99-135, jul. 2024, doi: [10.46586/tches.v2024.i3.99-135](https://doi.org/10.46586/tches.v2024.i3.99-135).

- [42] T.-H. Nguyen, D.-T. Dam, P.-P. Duong, T.-T. Hoang, y C.-K. Pham, «A Low-Latency Polynomial Arithmetic Unit for ML-KEM and ML-DSA Standards», en *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2025, pp. 1-5. doi: [10.1109/ISCAS56072.2025.11043200](https://doi.org/10.1109/ISCAS56072.2025.11043200).
- [43] J. Pu, Y. Xu, Y. Zhang, y J. Zhang, «An Area-Efficient ML-DSA Accelerator With Interleaved and Dynamic Execution», *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 73, n.º 5, pp. 3480-3493, may 2025, doi: [10.1109/TCSI.2025.3646585](https://doi.org/10.1109/TCSI.2025.3646585).
- [44] YosysHQ, «PicoRV32 - A Size-Optimized RISC-V CPU». Accedido: 14 de junio de 2026. [En línea]. Disponible en: <https://github.com/YosysHQ/picorv32>
- [45] «chipsalliance/rocket-chip». Accedido: 27 de julio de 2026. [En línea]. Disponible en: <https://github.com/chipsalliance/rocket-chip>
- [46] «lowRISC/ibex». Accedido: 23 de junio de 2026. [En línea]. Disponible en: <https://github.com/lowRISC/ibex>
- [47] stnolting, «stnolting/neorv32». Accedido: 18 de junio de 2026. [En línea]. Disponible en: <https://github.com/stnolting/neorv32>
- [48] «SpinalHDL/VexRiscv». Accedido: 27 de julio de 2026. [En línea]. Disponible en: <https://github.com/SpinalHDL/VexRiscv>
- [49] G. Kanda y K. Ryoo, «Design of an Integrated Cryptographic SoC Architecture for Resource-Constrained Devices», *International Journal of Electrical and Electronics Research*, vol. 10, pp. 230-244, jun. 2022, doi: [10.37391/ijeer.100231](https://doi.org/10.37391/ijeer.100231).
- [50] «Nexys Video - Digilent Reference». Accedido: 17 de julio de 2026. [En línea]. Disponible en: <https://digilent.com/reference/programmable-logic/nexys-video/start>
- [51] lowRISC, «Ibex Demo System». Accedido: 25 de junio de 2026. [En línea]. Disponible en: <https://github.com/lowRISC/ibex-demo-system>
- [52] R. M. Needham y D. J. Wheeler, «Tea extensions», Cambridge, U.K., technical report, oct. 1997.
- [53] N. P. Schuster y G. L. Nazar, «GRASP-based High-Level Synthesis Design Space Exploration for FPGAs», en *2023 XIII Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2023, pp. 1-6. doi: [10.1109/SBESC60926.2023.10324241](https://doi.org/10.1109/SBESC60926.2023.10324241).
- [54] «pq-code-package/mldsa-native». Accedido: 19 de julio de 2026. [En línea]. Disponible en: <https://github.com/pq-code-package/mldsa-native>
- [55] C. Harter, «LFX Mentorship: Contributing to the first release of mldsa-native – Post-Quantum Cryptography Alliance». Accedido: 19 de julio de 2026. [En línea]. Disponible en: <https://pqca.org/blog/2026/lfx-mentorship-contributing-to-the-first-release-of-mldsa-native/>
- [56] «About the Linux Foundation». Accedido: 19 de julio de 2026. [En línea]. Disponible en: <https://www.linuxfoundation.org/about>
- [57] «Keccak Team». Accedido: 20 de julio de 2026. [En línea]. Disponible en: <https://keccak.team/index.html>

- [58] «ghdl/ghdl». Accedido: 19 de julio de 2026. [En línea]. Disponible en: <https://github.com/ghdl/ghdl>
- [59] Isaac, «Ley de Amdahl, regla de Pollack, ley de Koomey... ¿qué son?». Accedido: 30 de julio de 2026. [En línea]. Disponible en: <https://arquitectura.es/ley-amdahl-regla-pollack-ley-koomey-que-son>
- [60] H.-Q. Li, T. Chen, W. Li, y L.-M. Nan, «Reconfigurable KECCAK Hardware Design for Lattice-based Post-Quantum Cryptography on the RISC-V Architecture», en *2022 IEEE 16th International Conference on Solid-State & Integrated Circuit Technology (ICSICT)*, 2022, pp. 1-3. doi: [10.1109/ICSICT55466.2022.9963339](https://doi.org/10.1109/ICSICT55466.2022.9963339).
- [61] P. Karl, J. Schupp, T. Fritzmann, y G. Sigl, «Post-Quantum Signatures on RISC-V with Hardware Acceleration», *ACM Trans. Embed. Comput. Syst.*, vol. 23, n.º 2, mar. 2024, doi: [10.1145/3579092](https://doi.org/10.1145/3579092).

Objetivos de Desarrollo Sostenible (ODS)

Tabla 22: Relación del proyecto con los Objetivos de Desarrollo Sostenible (ODS)

Objetivos de Desarrollo Sostenible	Grado de relación con el ODS			
	0 No procede	1 Bajo	2 Medio	3 Alto
ODS 1 Fin de la Pobreza	☒	☐	☐	☐
ODS 2 Hambre cero	☒	☐	☐	☐
ODS 3 Salud y Bienestar	☒	☐	☐	☐
ODS 4 Educación de calidad	☐	☒	☐	☐
ODS 5 Igualdad de género	☒	☐	☐	☐
ODS 6 Agua limpia y saneamiento	☒	☐	☐	☐
ODS 7 Energía Asequible y no contaminante	☒	☐	☐	☐
ODS 8 Trabajo decente y crecimiento económico	☐	☐	☒	☐
ODS 9 Industria, Innovación e Infraestructuras	☐	☐	☐	☒
ODS 10 Reducción de las desigualdades	☒	☐	☐	☐
ODS 11 Ciudades y comunidades sostenibles	☒	☐	☐	☐
ODS 12 Producción y consumo sostenibles	☒	☐	☐	☐
ODS 13 Acción por el clima	☒	☐	☐	☐
ODS 14 Vida submarina	☒	☐	☐	☐
ODS 15 Vida de ecosistemas terrestres	☒	☐	☐	☐
ODS 16 Paz, justicia e instituciones sólidas	☒	☐	☐	☐
ODS 17 Alianzas para lograr objetivos	☒	☐	☐	☐

La realización de este TFM está alineado con los Objetivos de Desarrollo Sostenible indicados en la Tabla 22 a través de los objetivos mencionados a continuación y justificado mediante las siguientes metas concretas indicadas:

- Objetivo 4, “Educación de calidad” (ODS 4): metas 4.3 y 4.4
- Objetivo 8, “Trabajo decente y crecimiento económico” (ODS 8): meta 8.2
- Objetivo 9, “Industria, innovación e infraestructuras” (ODS 9): metas 9.1, 9.5 y 9.b

Anexos

Anexo A: Código VHDL de la CFU

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library neorv32;
use neorv32.neorv32_package.all;
use neorv32.pkg_params_types.all;

entity neorv32_cpu_alu_cfu is
  port (
    -- global control --
    clk_i    : in  std_ulogic; -- global clock, rising edge
    rstn_i   : in  std_ulogic; -- global reset, low-active, async
    -- request --
    start_i  : in  std_ulogic; -- start trigger, single-shot
    inst_i   : in  std_ulogic_vector(31 downto 0); -- full instruction word
    rs1_i    : in  std_ulogic_vector(31 downto 0); -- register source operand 1
    rs2_i    : in  std_ulogic_vector(31 downto 0); -- register source operand 2
    -- response --
    result_o : out std_ulogic_vector(31 downto 0); -- operation result
    valid_o  : out std_ulogic                    -- operation done; result
  );
end neorv32_cpu_alu_cfu;

architecture neorv32_cpu_alu_cfu_rtl of neorv32_cpu_alu_cfu is
  constant array_size : natural := 1024;
  -- CFU opcodes --
  constant opcode_custom0_c : std_ulogic_vector(6 downto 0) := "0001011"; --
  CUSTOM-0 opcode

  -- instruction types (opcode field) --
  constant keccak_r_type_c : std_ulogic_vector(6 downto 0) := opcode_custom0_c;

  -- instruction identifiers (funct3 bit-field) --
  constant keccak_init      : std_ulogic_vector(2 downto 0) := "000"; -- 0:
  Config
  constant keccak_data      : std_ulogic_vector(2 downto 0) := "001"; -- 1: Send
  data
  constant keccak_data_last : std_ulogic_vector(2 downto 0) := "010"; -- 2: Send
  last data
  constant keccak_read      : std_ulogic_vector(2 downto 0) := "011"; -- 3: Read
  constant keccak_reset     : std_ulogic_vector(2 downto 0) := "100"; -- 4: Reset
  state
```

```

-- instruction decoder --
signal start : std_ulogic; -- start valid CFU instruction
signal funct3 : std_ulogic_vector(2 downto 0);
signal funct7 : std_ulogic_vector(6 downto 0);
signal r_valid_o : std_ulogic;

type t_array is array (0 to array_size) of std_logic_vector (31 downto 0);
signal myarray: t_array;
type t_data is array (0 to array_size) of std_logic_vector (63 downto 0);
signal mydata: t_data;
signal wr_index, rd_index, dw_index, dr_index : natural;

-- Keccak IP Signals --
signal k_rst_n : std_logic;
signal k_conf_reg : t_conf_reg;
signal k_stat_reg : t_stat_reg;
signal k_data_valid_in : std_logic_vector(0 downto 0);
signal k_last_data_in : std_logic_vector(0 downto 0);
signal k_data_in : t_din_data;
signal k_busy : std_logic;
signal k_busy_pre : std_logic;
signal k_last_data_out : std_logic_vector(0 downto 0);
signal k_data_valid_out : std_logic_vector(0 downto 0);
signal k_data_out : t_dout_data;

-- Component Declaration --
component keccARC_top is
  generic (
    g_DATA_IN_WIDTH : natural := 64;
    g_DATA_IN_NUM : natural := 1;
    g_CONF_REG_WIDTH : natural := 32;
    g_CONF_REG_NUM : natural := 4;
    g_DATA_OUT_WIDTH : natural := 64;
    g_DATA_OUT_NUM : natural := 1;
    g_HASH_NUM_ROUNDS : natural := 24
  );
  port (
    clk : in std_logic;
    rst_n : in std_logic;
    conf_reg : in t_conf_reg;
    stat_reg : out t_stat_reg;
    data_valid_in : in std_logic_vector(g_DATA_IN_NUM - 1 downto 0);
    last_data_in : in std_logic_vector(g_DATA_IN_NUM - 1 downto 0);
    data_in : in t_din_data;
    busy : out std_logic;
    busy_pre : out std_logic;
    last_data_out : out std_logic_vector(g_DATA_OUT_NUM - 1 downto 0);
    data_valid_out : out std_logic_vector(g_DATA_OUT_NUM - 1 downto 0);
    data_out : out t_dout_data
  );
end component;

```

```

-- FSM states --
type state_t is (S_IDLE, S_SEND, S_WAIT_READ, S_READ, S_DONE);
signal state : state_t;

signal result_reg : std_ulogic_vector(31 downto 0);
signal s_data_ready: std_ulogic;
signal flag: std_ulogic_vector(1 downto 0);

begin

-- Instruction Decode --
start <= start_i when (inst_i(6 downto 0) = opcode_custom0_c) else '0';
funct3 <= inst_i(14 downto 12);
funct7 <= inst_i(31 downto 25);

-- Keccak IP Instantiation --
keccak_inst : keccARC_top
port map (
    clk          => std_logic(clk_i),
    rst_n        => k_rst_n,
    conf_reg     => k_conf_reg,
    stat_reg     => k_stat_reg,
    data_valid_in => k_data_valid_in,
    last_data_in  => k_last_data_in,
    data_in       => k_data_in,
    busy         => k_busy,
    busy_pre     => k_busy_pre,
    last_data_out => k_last_data_out,
    data_valid_out => k_data_valid_out,
    data_out      => k_data_out
);

-- Processing Core --
cfu_control: process(rstn_i, clk_i)
begin
    if (rstn_i = '0') then
        state <= S_IDLE;
        valid_o <= '0';
        result_reg <= (others => '0');
        k_rst_n <= '0';
        k_conf_reg <= (others => (others => '0'));
        rd_index <= 0;
        dw_index <= 0;
    elsif rising_edge(clk_i) then
        k_rst_n <= '1';
        r_valid_o <= '0';
        valid_o <= r_valid_o;
        s_data_ready <= '0';

        case state is

```

```

when S_IDLE =>
  if (start = '1') then
    case funct3 is

      when keccak_init =>
        k_conf_reg(0) <= (others => '0');
        k_conf_reg(0)(3 downto 0) <= std_logic_vector(funct7(3 downto
0));
        k_conf_reg(0)(23 downto 4) <= std_logic_vector(rs2_i(19 downto
0));
        k_conf_reg(1) <= (others => '0');
        k_conf_reg(1)(2 downto 0) <= std_logic_vector(rs1_i(2 downto 0));
        rd_index <= 0;
        dw_index <= 0;
        state <= S_DONE;

      when keccak_data | keccak_data_last =>
        state <= S_SEND;

      when keccak_read =>
        state <= S_WAIT_READ;
      when keccak_reset =>
        k_rst_n <= '0';
        state <= S_DONE;

      when others =>
        valid_o <= '0';
        state <= S_DONE;
    end case;
  end if;

when S_WAIT_READ =>
  if (wr_index > rd_index) then
    state <= S_READ;
  end if;

when S_READ =>
  result_reg <= myarray(rd_index);
  rd_index <= rd_index + 1;
  state <= S_DONE;

when S_SEND =>
  mydata(dw_index) <= std_logic_vector(rs2_i) & std_logic_vector(rs1_i);
  dw_index <= dw_index + 1;
  if funct3 = keccak_data_last then
    s_data_ready <= '1'; --start read
  end if;
  state <= S_DONE;

when S_DONE =>
  valid_o <= '1';

```

```

        state    <= S_IDLE;

    end case;
end if;
end process cfu_control;

-- Save Result (out) --
p_data: process(clk_i, rstn_i)
begin
    if rstn_i = '0' then
        wr_index <= 0;
        myarray <= (others => (others => '0'));
    elsif rising_edge(clk_i) then
        if (start = '1' and funct3 = keccak_init) then
            wr_index <= 0;
        end if;

        if (k_data_valid_out(0) = '1') then
            myarray(wr_index)    <= std_logic_vector(k_data_out(0)(31 downto 0));
            myarray(wr_index+1)  <= std_logic_vector(k_data_out(0)(63 downto 32));

            if (wr_index < array_size - 2) then
                wr_index <= wr_index + 2;
            end if;
        end if;
    end if;
end process p_data;

-- Save Msg (in) --
p_send_data: process(clk_i, rstn_i)
begin
    if rstn_i = '0' then
        flag(0) <= '0';
        flag(1) <= '0';
        k_data_valid_in(0) <= '0';
        k_last_data_in(0)  <= '0';
        dr_index <= 0;

    elsif rising_edge(clk_i) then
        flag(0) <= s_data_ready;
        k_data_valid_in(0) <= '0';
        k_last_data_in(0) <= '0';

        if (flag(0) = '1' and s_data_ready = '0') then
            flag(1) <= '1';
            dr_index <= 0;
        elsif (flag(1) = '1' and k_busy = '0') then
            k_data_in(0) <= mydata(dr_index);
            k_data_valid_in(0) <= '1';
            dr_index <= dr_index + 1;
        end if;
    end if;
end process p_send_data;

```

```

        if(dr_index = dw_index - 1) then
            k_last_data_in(0) <= '1';
            flag(1) <= '0';
        end if;
    end if;

    end if;
end process p_send_data;

result_o <= result_reg;

end neorv32_cpu_alu_cfu_rtl;

```

Código 17: CFU diseñada para la ejecución de la función Keccak