TRABAJO FIN DE MASTER

ACELERADOR HARDWARE PARA EL ESQUEMA DE FIRMA DIGITAL POSTCUÁNTICA FALCON

TRABAJO FIN DE MASTER
PARA LA OBTENCIÓN DEL
TÍTULO DE **MASTER EN ELECTRÓNICA INDUSTRIAL**

FEBRERO **2026**

Antonio Carreño Gómez

DIRECTOR DEL TRABAJO FIN DE MASTER:
Jorge Portilla Berrueco
Jaime Señor Sánchez

INDUSTRIALES
ETSII | UPM

POLITÉCNICA

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES

DEPARTAMENTO DE ELECTRÓNICA INDUSTRIAL

MADRID

TRABAJO DE FIN DE MASTER

Acelerador hardware para el esquema de firma digital postcuántica Falcon

Autor: Antonio Carreño Gómez

Director: Jorge Portilla Berrueco y Jaime Señor Sánchez

Titulación: Master en Electrónica Industrial

FEBRERO 2026

INDUSTRIALES ETSII | UPM

POLITÉCNICA

"The decisions we make about communication security today will determine the kind of society we live in tomorrow"

- W. Diffie

AGRADECIMIENTOS

A mis padres y a mi familia, que siempre me motivaron a hacer lo que más me apasione y me dieron la confianza para conseguir lo que me proponga.

A Andrea, por mostrarme su apoyo incondicional a lo largo de los años e intentar siempre sacar nuestra mejor versión.

A mis amigos de toda la vida, por estar siempre presentes y ofrecerme el respiro necesario para seguir con más fuerza.

A mi compañeros de la escuela, juntos hemos recorrido un largo camino y nos hemos ayudado en cuanto fuera posible. En especial, quiero mencionar a Guille, Luis y María con los que fui muy feliz durante el grado. También a Álvaro, Jorge y Óscar con los que he tenido el mejor final de formación posible.

A todos los investigadores y profesores del CEIMM, con los que siempre se puede contar para compartir ideas y buscar soluciones para avanzar en este mundo de la investigación.

Por último, agradecer a Jorge y a Jaime, mis dos tutores que me ofrecieron la posibilidad de realizar este trabajo y han estado pendientes del proyecto en todo momento. Gracias a ellos estoy viviendo un increíble periodo de investigación y formación en el CEIMM.

Y como suelo decir, piedra a piedra se construyó el Monasterio. Cada uno ha sido una pequeña parte de este camino, gracias a todos.

RESUMEN EJECUTIVO

Los esquemas de criptografía clásicos utilizados actualmente para proteger a la sociedad frente a ataques de terceros pueden verse amenazados por la llegada de los ordenadores cuánticos. Estos ordenadores son capaces, de forma teórica, de ejecutar algoritmos de factorización de números grandes o búsqueda en una lista no estructurada con un rendimiento mucho mayor que los superordenadores actuales. El *National Institute of Standards and Technology* ha realizado un proceso de estandarización para esquemas de criptografía postcuántica con el objetivo de seleccionar esquemas que sean apropiados para distintos entornos de aplicación. Uno de los campos que abarca esta estandarización es el referente a los esquemas de firma digital, de los cuales Falcon fue elegido por su compacidad de claves y firma apto para entornos de bajos recursos.

El objetivo de este trabajo es el diseño e implementación de un acelerador *hardware* diseñado desde cero del esquema de firma digital Falcon en una FPGA. Existen en la literatura una gran variedad de aceleradores parciales *hardware*/*software* que implementan diferentes partes del esquema de firma digital ya sea por su exponencial reducción del número de ciclos o por su factible paralelización. Sin embargo, esta es la primera implementación en *hardware* del esquema al completo de Falcon sin utilizar síntesis de alto nivel y con el código diseñando desde cero. Debido al tamaño del diseño, se ha decidido utilizar la plataforma Nexys Video para la implementación del acelerador. Esta FPGA no dispone de sistema de procesamiento por lo que se deberá reservar una parte de la zona reconfigurable para sintetizar un MicroBlaze que utilice el acelerador a través de señales de control y registros para la transferencia de datos. Los pasos a seguir en este trabajo serán los siguientes:

1. Estudio del esquema de firma digital Falcon y las diferentes implementaciones existentes en la literatura.
2. Estructuración de la implementación y toma de decisiones relativas al diseño *hardware*.
3. Diseño de los módulos *hardware* basado en el código *software* de referencia de Falcon.
4. Simulación y validación en el entorno de desarrollador Vivado de los módulos *hardware* y la relación entre estos.
5. Desarrollo de una IP personalizada para el acelerador *hardware* con interfaz AXI Lite para la comunicación, así como un diseño de bloques que la conecte al MicroBlaze.
6. Prueba y validación del acelerador en la placa Nexys Video en el entorno de desarrollador Vitis.
7. Elaboración de un banco de pruebas y comparación de tiempo y número de ciclos con la referencia *software*.

El diseño tiene una utilización de 85261 LUTs, 41382 FFs, 44 BRAMs y 142 DSPs, lo que supone un porcentaje de 63,34 % LUTs, 15,37 % FFs, 12,05 % BRAMs y 19,19 % DSPs respecto al total disponible en la placa Nexys Video. El banco de pruebas proporciona unos resultados de reducción del número de ciclos de ejecución del esquema de firma digital del 92 % para la generación de claves, 97 % para la generación de firma y 91 % para la verificación de firma en comparación con el código de referencia ejecutado en el MicroBlaze. Con el MicroBlaze operando a 75 MHz esta reducción de ciclos supone una reducción de tiempos de ejecución de 5127,07 ms para la generación de claves, 422 ms para la generación de firma y 7,68 ms para la verificación de

firma. Estos resultados suponen una mejora en cuanto al tiempo de ejecución y, junto al hecho de ser la primera implementación *hardware* al completo de Falcon, lo convierten en una opción viable para una futura implementación en entornos reales.

Palabras clave - acelerador, criptografía postcuántica, Falcon, firma digital, FPGA, hardware.

Códigos UNESCO

1203.26 SIMULACIÓN

1203.61 CIBERSEGURIDAD

1203.63 CRIPTOGRAFÍA

3307.86 SISTEMAS ELECTRÓNICOS Y DE COMUNICACIONES PARA APLICACIONES DE SEGURIDAD

3325.64 SISTEMAS EMBEBIDOS

ABSTRACT

Classic cryptographic schemes which are currently used in order to protect society against third party attacks may be threatened by the arrival of quantum compuers. These quantum computers are theoretically capable of running large integers factoring algorithms or non-structured database search algorithms achieving greater performance than current supercomputers. The National Institute of Standards and Technology made a post-quantum cryptography standardization process with the aim of selecting several schemes which fit in different application environments. One of the areas covered by this standardization is digital signature schemes, among which Falcon was selected for its key and signature compactness suitable for resource constrained environments.

The purpose of this work is to design and implement a hardware accelerator developed from scratch of the Falcon digital signature scheme in a FPGA. There is a wide variety of hardware/software codesigns in the literature either because of its exponential clock cycle reduction or because of its feasible paralelization. Nevertheless, this is the first complete hardware Falcon implementation without using high level synthesis and coded from scratch. Due to the size of the design, it has been decided to implement the accelerator into the Nexys Video board. This FPGA does not have processing system, consequently a part of the programmable logic must be reserved to synthesize a Microblaze able to use the accelerator through control signals and registers for data transfer. Final stages of this project are the following:

1. Study of the Falcon digital signature scheme and the hardware implementations in the literature.
2. Structuring implementation and decision making related to hardware design.
3. Design of the hardware modules based on the software reference code of Falcon.
4. Simulation and validation in the integrated development enviornment Vivado of the hardware module and the communication between them.
5. Development of an accelerator custom IP using AXI Lite inteface for data transfer, as well as a block design to connect the IP to the Microblaze.
6. Test and validation of the accelerator in the Nexys Video board using the integrated development environment Vitis.
7. Elaboration of a testbench and comparison of execution time and number of cycles with the software reference.

The design has a resource utilizacion of 85261 LUTs, 41382 FFs, 44 BRAMs y 142 DSPs, which means a 63,34 % LUTs, 15,37 % FFs, 12,05 % BRAMs and 19,19 % DSPs of the resources avaliables in the Nexys Video board. The testbench shows a reduction of the number of clock cycles of the digital signature scheme of 92 % in the key generation, 97 % in the signature generation and 91 % in the signature verification compared to the software reference running in the MicroBlaze. As the Microblaze is running at 75 MHz, this clock cycle reduction means an execution time reduction of 5127,07 ms in the key generation, 422 ms in the signature generation y 7,68 ms in the signature verification. These results represent an improvement on execution time and, along with the fact of being the first complete hardware implementation of Falcon, make it a viable option for future implementation in real-world environments.

Keywords - accelerator, digital signature, Falcon, FPGA, hardware, post-quantum cryptography.

UNESCO Codes

1203.26 Simulation

1203.61 Cybersecurity

1203.63 Cryptography

3307.86 Electronic and communication systems for security applications

3325.64 Embedded systems

ÍNDICE

1. INTRODUCCIÓN

La criptografía permite a cualquier pareja de individuos transmitir información de manera segura y fiable gracias a métodos como el intercambio de claves o las firmas digitales. La criptografía es una ciencia fundamental en la sociedad, ya que protege desde archivos de gran valor para entidades bancarias o gubernamentales hasta un simple mensaje enviado entre familiares. Se han desarrollado numerosos esquemas criptográficos destinados a proteger la información, algunos ejemplos de esquemas de criptografía clásicos son *Advanced Encryption Standard* (AES) comúnmente utilizado en criptografía simétrica o *Rivest-Shamir-Adleman* (RSA) en criptografía asimétrica.

Estos esquemas clásicos se ven amenazados por la llegada de los ordenadores cuánticos, los cuales serán capaces de utilizar una serie de algoritmos para romper dichos esquemas en un tiempo suficiente para que la información tenga valor para los atacantes. En la sección 1 se estudiará de forma teórica la base sobre la que se fundamentan los ordenadores cuánticos y su capacidad para romper los esquemas clásicos tradicionales, qué medidas de prevención se están tomando para la llegada de los ordenadores cuánticos, y finalmente se presentarán los objetivos que se pretenden cubrir en este trabajo.

1.1. Introducción a la computación cuántica

La teoría sobre la que se fundamentan los ordenadores cuánticos se remonta a finales del siglo XX. Entre los años 1981 y 1982 el físico Richard Feynman [1] en el intento de simular la naturaleza planteaba el uso de ordenadores para simular sistemas físicos. En el caso de la física clásica al ser causal, se puede simular un estado futuro en base a la información que se tiene de ese estado en el pasado. Sin embargo, cuando se pasa a intentar simular la física cuántica entra en juego la probabilidad y por tanto se necesitan ordenadores probabilísticos que calculen esta y después la interpreten para simular el futuro. Esto hace que el tamaño del ordenador que se necesita crezca exponencialmente y hace que sea imposible simular calculando la probabilidad. Pocos años mas tarde, en 1985 fue David Deutsch quien presentó un modelo completo cuántico para computación y describió de forma teórica el primer ordenador cuántico capaz de simular cualquier sistema físico finito [2].

Durante la década de los 90 se realizaron grandes aportaciones al campo de la computación cuántica, definiendo los conceptos teóricos y desarrollando las primeras aplicaciones en computación cuántica. En 1993 Charles Bennet definió el concepto de teleportación cuántica [3], el cual permitiría el movimiento de estados en unidades de información cuántica distantes entre sí sin precisar de un movimiento físico de los mismos. Uno de los primeros autores en evidenciar las ventajas de los ordenadores cuánticos al resolver ciertos problemas fue Simon en 1994, el cual definió un algoritmo [4] que se aprovechaba de la superposición para evaluar diferentes estados en paralelo.

La unidad mínima de información de los ordenadores cuánticos mencionada anteriormente fue denominada por Benjamin Schumacher en 1995 [5] como *qubit*. A diferencia de los *bits* de la computación clásica que sólo disponen de dos estados claramente diferenciados como 0 y 1, los *qubits* se encuentran en una superposición de ambos estados hasta el momento de ser medidos siguiendo la ecuación

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{1.1}$$

en la que $\alpha, \beta \in \mathbb{C}$ cumplen la condición de normalización:

$$|\alpha|^2 + |\beta|^2 = 1 \tag{1.2}$$

Esta condición implica que el *qubit* puede colapsar al estado $|0\rangle$ con una probabilidad $|\alpha|^2$ o al estado $|1\rangle$ con una probabilidad $|\beta|^2$ al ser medidos. Estos son dos de los principios fundamentales de la computación cuántica conocidos como la superposición y la medición. Adicionalmente cabe destacar otros dos principios utilizados en la computación cuántica que permiten ver con mayor claridad las ventajas que ofrece respecto al cálculo de la probabilidad en sistemas físicos. Estos dos principios son el entrelazamiento cuántico y la interferencia cuántica. El entrelazamiento cuántico [6] supone que cuando dos o más partículas quedan relacionadas el estado de una depende del estado de la otra. Esta relación de estados se produce de manera inmediata, independientemente de la distancia entre partículas. Por ejemplo si dos *qubits* se encuentran en el estado de Bell

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|0\rangle|0\rangle + |1\rangle|1\rangle) \tag{1.3},$$

si al medir el primer *qubit* este se encuentra en el estado $|0\rangle$ asignará al segundo *qubit* instantáneamente el estado $|0\rangle$. Por su parte la interferencia cuántica consiste en tomar ciertas mediciones de forma parcial para ir rechazando varios estados posibles y así reducir progresivamente la cantidad de computación.

Al mismo tiempo que se sentaban las bases y los principios fundamentales de la computación cuántica, también se desarrollaban algoritmos de computación capaces de aprovecharse de los recursos de los ordenadores cuánticos. Entre los algoritmos más relevantes se encuentra el descrito por Peter Shor en 1994 [7] de factorización de números grandes. Este algoritmo empieza por encontrar el orden de un elemento x en el grupo multiplicativo $(\bmod n)$, siendo este el menor número entero positivo r que cumpla la condición

$$x^r \equiv 1 \bmod n \tag{1.4}$$

siendo n el número que se pretende factorizar. Una vez se consigue este orden r se calcula $\gcd\left(x^{r/2} - 1, n\right)$, que dará como resultado uno de los valores de la factorización. La factorización del entero N presentada por Shor tenía una complejidad temporal del orden de $O((\log N)^2 \cdot (\log\log N) \cdot (\log\log\log N))$ pero tras optimizaciones posteriores se reduce a $O((\log N)^3)$ [8]. Otro de los algoritmos más importantes es el presentado por Lov Grover en 1995 [9] para búsqueda no estructurada en una lista, el cual consigue reducir el número de consultas desde $O(N)$ de forma clásica hasta un número de consultas de $O(\sqrt{N})$.

Los ordenadores cuánticos podrían utilizar estos algoritmos para resolver problemas basados en probabilidad de una manera mucho más eficiente que los superordenadores actuales, lo cuál derivará en un ahorro de tiempo y recursos para determinadas aplicaciones. Sin embargo, esta ventaja tiene una contraparte respecto a la criptografía clásica ya que su seguridad parcialmente se basa en el tiempo que se tarda en romperlos y la información pierde valor para los atacantes. Los algoritmos de Shor y de Grover mencionados anteriormente comprometen a esquemas de criptografía clásicos como RSA y AES respectivamente. Se teoriza que los ordenadores cuánticos serán capaces de romper estos algoritmos en horas en comparación con la cantidad años que llegan a tardar los superordenadores actuales. En [10] se conseguiría romper RSA-2048 en aproximadamente 7 horas utilizando una gran cantidad de *qubits*, aunque ciertos artículos proponen

una mejor compensación entre el tamaño del ordenador cuántico y el tiempo empleado [11][12]. Es por este motivo que urge la creación de nuevos esquemas de criptografía que sean capaces de proteger la información frente a ataques de ordenadores cuánticos. Se estima como fecha límite que la migración hacia estos esquemas debe suceder antes del año 2030 con el objetivo de evitar el almacenamiento de información relevante por parte de los atacantes ante la llegada de los ordenadores cuánticos.

1.2. Proceso de estandarización del NIST

La elaboración de los denominados esquemas de criptografía postcuántica es esencial para al seguridad de la información en un futuro próximo, no tan solo para proteger la información de estos ordenadores cuánticos sino también de seguir protegiendo frente a las amenazas actuales. Hace prácticamente una década, en el año 2016, el NIST anunció un proceso de estandarización para esquemas de criptografía postcuántica [13] con el objetivo de seleccionar varios esquemas seguros en la próxima era de los ordenadores cuánticos. Este proceso de selección se basa en tres criterios principales: la seguridad que ofrece, el coste computacional y de almacenamiento de memoria, y las características del algoritmo.

En el ámbito de la seguridad, el NIST impuso unos niveles de seguridad equiparándolos con los esquemas clásicos de criptografía, por ejemplo nivel 3 del NIST equivale a la protección ofrecida por AES-192. Era necesario que protegiera tanto frente a ataques cuánticos así como ataques clásicos, aportando pruebas y análisis posterior. En la parte de coste se debía buscar la eficiencia computacional para ahorrar recursos y aportar datos concretos sobre tamaños de clave y memoria requerida. Por último, también se tenía en cuenta la flexibilidad del algoritmo a la hora de implementarlo tanto por parámetros como por plataformas, la facilidad para integrarse en los protocolos actuales y la simplicidad del algoritmo en sí mismo.

Bajo estos requisitos se hizo la primera convocatoria de propuestas con un resultado de 69 candidatos aceptados de 82 presentados totales, elegidos por cumplir los requisitos mínimos para ser considerados viables. Estos candidatos elegidos pasaron a una segunda fase de rondas de selección con la finalidad de eliminar progresivamente a los candidatos menos aptos para la estandarización. En la primera ronda del proceso de estandarización [14] en enero de 2019 fueron seleccionados 26 algoritmos para el avance a la segunda ronda, se eligieron en base a llegar al nivel 3 de seguridad del NIST, las estimaciones de rendimiento en la plataforma ofrecida por el NIST como referencia y la flexibilidad de implementación en diferentes plataformas, entre otros factores. En julio de 2020, se terminó la segunda ronda del proceso de estandarización [15] y en ella se seleccionaron 15 finalistas para la estandarización eliminando a los que tenían debilidades significativas, peor rendimiento y optimización que los demás candidatos en su campo, baja seguridad frente ataques de canal lateral o por su poca adaptación a los sistemas actuales.

Este Trabajo de Fin de Máster centra su atención en la tercera ronda del proceso de estandarización de criptografía postcuántica. En julio de 2022 concluyó la tercera ronda del proceso de estandarización [16], finalmente cuatro esquemas fueron elegidos para su estandarización por el NIST, siendo uno de ellos un mecanismo de encapsulado de claves (KEM), CRYSTALS-Kyber [17], y los tres restantes algoritmos de firma digital (DSA), CRYSTALS-Dilithium [18], Falcon [19] y SPHINCS+ [20]. Posteriormente dio comienzo una cuarta ronda adicional del proceso de estandarización [21], con el objetivo de seleccionar al menos un KEM más con una base diferente al ya seleccionado CRYSTALS-Kyber, los candidatos fueron BIKE, Classic McEliece, HQC y SIKE. Recientemente en marzo de 2025 HQC [22] fue elegido como apto para la estandarización, dando por terminado el proceso que comenzó años atrás. Los motivos de elección de estos esquemas se expondrán en la siguiente sección.

Actualmente ya están publicados los estándares oficiales *Federal Information Processing Standards* (FIPS) de los siguientes esquemas criptográficos: CRYSTALS-Kyber como *Module-Lattice-Based Key-Encapsulation Mechanism Standard* FIPS 203 [23], CRYSTALS-Dilithium como *Module-Lattice-Based Digital Signature Standard* FIPS 204 [24] y SPHINCS+ como *Stateless Hash-Based Digital Signature Standard* FIPS 205 [25]. El estándar de Falcon tenía planeado su borrador para 2024 y se espera que sea publicado lo antes posible, mientras que el estándar de HQC está planeado para ser publicado en 2027.

1.3. Esquemas de criptografía postcuántica elegidos por el NIST

La criptografía tiene diferentes procedimientos dependiendo del contexto en el que está empleado, los recursos de los que dispone y la finalidad con la que se aplica, como la confidencialidad, la integridad, la autenticidad o una combinación de estas. Entre las más relevantes se encuentran la criptografía simétrica y la criptografía asimétrica. La criptografía simétrica se basa en la existencia de una clave privada compartida entre los dos individuos que comparten información, la cual sirve tanto para cifrado del mensaje por parte del emisor como para el descifrado del mensaje por parte del receptor. Este tipo de criptografía es rápida pero tiene el inconveniente de necesitar una distribución segura de la propia clave para no comprometer la seguridad de la información. Por otro lado la criptografía asimétrica utiliza un par de claves pública y privada, la clave pública generada por el receptor es utilizada por el emisor para cifrar la información, que sólo puede ser descifrada por la clave privada del receptor. Este tipo de criptografía es más lenta que la simétrica pero ofrece una mayor seguridad, es por ello que ambos tipos no son excluyentes sino que son utilizados por los sistemas según el contexto.

El NIST en su proceso de estandarización sólo tuvo en cuenta el campo de la criptografía asimétrica ya que es la mas vulnerable al ataque de los ordenadores cuánticos. La criptografía simétrica alcanza seguridad suficiente frente a estos ataques aumentando el tamaño de las claves [26]. En el proceso de estandarización de criptografía postcuántica se consideraron tres tipos de criptografía: *Public Key Encryption* (PKE), *Key Encapsulation Mechanisms* (KEM) y *Digital Signature Algorithms* (DSA).

1.3.1. Public Key Encryption

La finalidad de este algoritmo es la garantizar la confidencialidad de la información transmitida. Es el tipo de criptografía mas cercano a la definición dada anteriormente. El receptor genera un par de claves, una clave pública que es transmitida al emisor y cuyo conocimiento por parte de un tercero no implica una brecha de seguridad, y una clave privada que es almacenada por el receptor para utilizar sobre un mensaje cifrado. El emisor utiliza la clave pública que ha obtenido del receptor para cifrar la información, la cual es indescifrable en su transmisión por terceros, y en recepción se utiliza la clave privada para descifrar esta información. Este método será seguro mientras la clave privada no sea conocida por terceros. Desafortunadamente ningún esquema de este tipo fue elegido por el NIST como apto para la estandarización.

1.3.2. Key Encapsulation Mechanisms

La finalidad de este algoritmo no es proteger la información transmitida sino establecer una vía segura para el uso de la criptografía simétrica. Al igual que en los PKE el receptor generará un par de claves pública y privada, pero en los KEM el emisor generará también una clave secreta.

El emisor utilizará la clave pública del emisor para encapsular la clave secreta que ha generado, este encapsulamiento se transmite al receptor, que utilizará la clave privada para desencapsular la clave secreta. Es así como los dos individuos ya disponen de una clave secreta compartida y pueden transmitir información cifrada de forma rápida. Estos algoritmos son más rápidos que los PKE y más seguro que los de criptografía simétrica. El NIST eligió dos algoritmos de este tipo para su estandarización.

1. CRYSTALS-Kyber [17] : la base de este algoritmo es la dificultad para resolver el problema *Learning With Errors* (LWE) sobre módulos (MLWE). Este problema trata de conseguir una relación lineal entre vectores de polinomios a través de muestras aleatorias a las que se les aplica un ruido. Entre las características que llevaron a su elección destacan la seguridad bajo ataque de texto cifrado adaptativo (IND-CCA2), su rendimiento en comparación con los otros candidatos KEM y por último, la cantidad de investigación que se ha realizado sobre este algoritmo.
2. HQC [22] : este algoritmo está basado en la dificultad del problema *Quasi-Cyclic Syndrome Decoding* (QCSD), el cual consiste en obtener un error con bajo peso que satisfaga la ecuación
$$s = He^T \tag{1.5}$$
dados la matriz de paridad H y el síndrome s. Fue elegido debido por la necesidad de un esquema con fuerte seguridad contra IND-CCA2 y las pocas modificaciones que necesita.

1.3.3. Digital Signature Algorithm

La finalidad de este algoritmo es garantizar la autenticidad y preservar la autoría de la información transmitida, el contenido de la información pasa a un segundo plano. Este tipo de algoritmos permiten al receptor confirmar que la información no ha sido modificada por terceros y asegurar la autoría por parte del emisor, que no podrá negar que la información proviene de este. El emisor genera un par de claves pública y privada, la clave privada será usada para generar una firma del mensaje mientras que la pública será transmitida al receptor. Este último podrá entonces verificar la firma usando la clave pública del emisor comparándolo con el mensaje original. Tres algoritmos de firma digital diferentes fueron elegidos para su estandarización.

- CRYSTALS-Dilithium [18] : este algoritmo está basado en el MLWE explicado en la sección anterior combinado con el paradigma de Fiat-Shamir, que transforma el protocolo de firma interactivo entre individuos a un protocolo no interactivo. Las principales razones que llevaron a su elección fueron su fácil implementación, su elevado rendimiento y su seguridad frente a ataques de mensaje elegido (CPA/CCA/CMA).
- Falcon [19] : este algoritmo se fundamenta en el uso de retículos de clase *N-th degree Truncated polynomial Ring Units* (NTRU), en un nuevo método de muestreo llamado *fast Fourier sampling* y en el marco teórico Gentry-Peikert-Vaikuntanathan (GPV) [27] sobre el paradigma *"hash-and-sign"*, aplicar una función *hash* al mensaje previo a su firma. Fue seleccionado por su compacidad tanto en el tamaño de firma como en el tamaño de claves, adecuado para entornos de bajos recursos.
- SPHINCS+ [20] : la principal característica de este algoritmo es el uso de funciones *hash* de un único uso que se organizan en forma de cadenas o de árbol. La principal ventaja de este algoritmo es su simplicidad en comparación con las complejas estructuras algebraicas de los otros DSA.

1.4. Objetivos del trabajo

En este Trabajo de Fin de Máster se pretende implementar un acelerador puramente *hardware* diseñado desde cero del esquema de firma digital Falcon al completo. Este acelerador debe ser capaz de reproducir el comportamiento del código *software* referencia ejecutando la generación de claves, la generación de firma y la verificación de firma de forma independiente. Con este acelerador se generará una IP personalizada para implementar en la FPGA Nexys Video junto con un MicroBlaze para la ejecución del código *software*, unidad AXI UART Lite para poder transmitir datos con el ordenador y una unidad AXI Timer para realizar pruebas de velocidad de ejecución. Se realizarán diferentes pruebas alterando los valores de entrada del acelerador para sacar conclusiones en cuanto a rendimiento en comparación con la referencia y el estado del arte.

Este trabajo ha sido realizado gracias al proyecto *INARTRANS 4.0* financiado por la convocatoria TransMisiones 2023 (AEI) en las que participa el Centro de Electrónica Industrial y Sistemas Multimodales. *INARTRANS 4.0* tiene como objetivo una transición digital hacia una industria avanzada en soluciones de inteligencia artificial para el sector de las infraestructuras de transporte. Por parte de la Universidad Politécnica de Madrid, se realizan trabajos de implementación y despliegue de redes de sensores inalámbricos en un entorno del *Internet of Things* (IoT). La transmisión de información entre estos nodos en el IoT debe realizarse de forma segura e íntegra por lo que se necesita contar con esquemas de criptografía postcuántica de manera eficiente y en el menor tiempo posible para la aplicación en tiempo real. El uso de un acelerador *hardware* permitiría limitar las tareas del nodo a la medición de sensores y a la transmisión de información liberando la carga que corresponde a la ejecución de algoritmos criptográficos.

La organización de este documento es la siguiente: en la Sección 2 se presentará una revisión del algoritmo de firma digital Falcon en profundidad y una parte del estado del arte sobre implementaciones parciales *hardware*/*software* de Falcon, en la Sección 3 se explicarán las decisiones tomadas en el diseño en relación a los problemas que derivan de la implementación en *hardware*, en la Sección 4 se mostrará una vista general sobre el diagrama de bloques de la implementación del procesador junto al acelerador, en la Sección 5 se estudiarán los resultados de rendimiento del acelerador así como la simulación previa realizada y, finalmente, una conclusión sobre el proyecto en la Sección 6.

2. ESQUEMA DE FIRMA DIGITAL Falcon

En esta sección 2 se expondrá el trasfondo teórico del esquema de firma digital Falcon [19], elegido por el NIST en la tercera ronda del proceso de estandarización de criptografía postcuántica [16]. El nombre completo, *Fast Fourier lattice-based compact signatures over NTRU*, refleja algunas de las características principales de este esquema de firma digital como son la compacidad, el uso de la transformada rápida de Fourier y el uso de retículos de clase NTRU. Este esquema fue elegido por el NIST por la compacidad de sus firmas y claves, que fue el motivo inicial para el desarrollo del esquema. Los esquemas de criptografía postcuántica tienden a requerir una gran cantidad de recursos, las claves y las firmas tienen un tamaño cada vez mayor, lo que dificulta su implementación en ciertos entornos de recursos limitados.

Falcon tiene tres conceptos teóricos principales sobre los cuales se desarrolló el algoritmo. El primero de ellos es el marco teórico GPV [27] que muestra la forma de obtener esquemas de firma digital del tipo *hash-and-sign* aprovechando el uso de *trapdoors* como clave privada. Una vez elegido el marco teórico GPV, se debe decidir la clase de retículos sobre las que se va a aplicar, lo que lleva al segundo concepto, los retículos de clase NTRU [28]. Se eligieron los retículos de clase NTRU debido a que reducen la complejidad computacional en un factor de $O(n/\log n)$ y el tamaño de la clave pública en el orden de $O(n)$, además de haber sido probadas seguras en numerosos análisis criptográficos. Se dispone de cuatro polinomios f,g,F,$G \in \mathbb{Z}[x]/(\phi)$, es decir, un anillo de polinomios con coeficientes enteros módulo $\phi = x^n + 1$ que cumplen la ecuación de NTRU

$$fG - gF = q \bmod \phi \tag{2.1}$$

que serán considerados clave privada y se podrá obtener un polinomio h considerado clave pública. La seguridad que ofrecen viene dada por la dificultad de encontrar dos polinomios f' y g' que cumplan la condición

$$h = g' \cdot (f')^{-1} \bmod q \tag{2.2}$$

pese a ser estos polinomios de tamaño reducido para favorecer la compacidad.

El último concepto teórico radica en el *trapdoor sampler*, el cual ha sido diseñado con una variante del "fast Fourier nearest plane" [29]. Esta variante permite trabajar sobre retículos de NTRU de una forma mucho más eficiente trabajando de una forma recursiva.

Las principales ventajas de este algoritmo están resumidas en [19], de las cuales cabe destacar como ya se ha dicho anteriormente la compacidad, los rápidos procesos de generación y verificación de firmas y la seguridad probada en los modelos ROM y QROM. También destacan otros aspectos relevantes como la modularidad que presenta en el tipo de retículos a utilizar y la poca computación que necesita la verificación de firmas. También se describen los principales inconvenientes como son la dificultad tanto para entender la estructura del algoritmo como para implementarlo, el uso de números en tipo coma flotante siguiendo el estándar del IEEE 754 [30] que limita el uso en sistemas que no dispongan de unidad de cálculo en coma flotante (FPU) y la resistencia a ataques de canal lateral puede ser menor dependiendo del tipo de implementación que se utilice.

Los esquemas de firma digital se dividen en tres partes claramente diferenciadas: la generación de claves, la generación de firma y la verificación de firma. En lo restante de esta sección 2, se

explicarán estas partes con los algoritmos que se necesitan para llevarlas a cabo y, para finalizar, una revisión del estado del arte.

2.1. Generación de claves

La generación de claves es el proceso que más tiempo y recursos gasta del esquema al completo. Esta parte inicia con la generación de dos polinomios f y g de manera aleatoria, los cuales servirán para resolver la ecuación (2.1) de NTRU. Al ser un proceso costoso computacionalmente, los valores de estos polinomios se guardarán con el fin de ser utilizados, pese a poder ser re-computados de manera dinámica. La generación de claves es un proceso del tipo "prueba y error" en el que en un bucle infinito se van sucediendo iteraciones hasta que se encuentran dentro de los valores límite necesarios. En los algoritmos (1)(2) se muestra este proceso al completo.

Algorithm 1 Keygen (ϕ,q)

Require: A monic polynomial $\phi \in \mathbb{Z}[x]$, a modulus q
Ensure: A secret key sk, a public key pk
1: $f, g, F, G \leftarrow$ **NTRUGen**(ϕ, q)
2: $\mathrm{B} \leftarrow \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}$
3: $\hat{\mathrm{B}} \leftarrow FFT(\mathrm{B})$
4: $\mathrm{G} \leftarrow \hat{\mathrm{B}} \times \hat{\mathrm{B}}^*$
5: $\mathrm{T} \leftarrow$ **ffLDL***(G)
6: **for** each leaf $leaf$ of T **do**
7: $\quad leaf.value \leftarrow \sigma/\sqrt{leaf.value}$
8: **end for**
9: $sk \leftarrow (\hat{\mathrm{B}}, \mathrm{T})$
10: $h \leftarrow gf^{-1} \bmod q$
11: $pk \leftarrow h$
12: return sk,pk

Algoritmo de generación de claves de Falcon [19].

Es la parte del esquema que menos veces será ejecutada en promedio, debido a que un par de claves generadas es válida para un número de firmas del orden de cientos de miles o ante una posible brecha de seguridad como la publicación de la clave privada.

2.1.1. Clave privada

Como se ha explicado anteriormente, la clave privada sk es generada al resolver la ecuación de NTRU (2.1) con los polinomios f, g, F y G. Es aconsejable guardar los valores de estos polinomios para evitar pérdida de tiempo en computación innecesaria, y a ser posible en representación FFT. En el algoritmo (2) se puede ver cómo se generan estos polinomios y en el algoritmo (3) cómo se resuelve la ecuación de NTRU (2.1).

Algorithm 2 NTRUGen (ϕ,q)

Require: A monic polynomial $\phi \in \mathbb{Z}[x]$ of degree n, a modulus q
Ensure: Polynomials f,g,F,G

$\sigma_{\{f,g\}} \leftarrow 1{,}17\sqrt{q/2n}$
for i from 0 to $n-1$ **do**
  $f_i \leftarrow D_{\mathbb{Z},\sigma_{\{f,g\}},0}$
  $g_i \leftarrow D_{\mathbb{Z},\sigma_{\{f,g\}},0}$
end for
$f \leftarrow \sum_i f_i x^i$
$g \leftarrow \sum_i g_i x^i$
if NTT(f) contains 0 as coefficient **then**
  restart
end if
$\gamma \leftarrow \max\{||\ (g, f)\ ||, ||\ (\frac{qf^*}{ff^*+gg^*}, \frac{qg^*}{ff^*+gg^*})\ ||\}$
if $\gamma > 1{,}17\sqrt{q}$ **then**
  restart
end if
$F, G \leftarrow \mathbf{NTRUSolve}_{n,q}(f, g)$
if $(F, G) = \perp$ **then**
  restart
end if
return f,g,F,G

Algoritmo de generación de polinomios [19].

Algorithm 3 NTRUSolve$_{n,q}(f, g)$

Require: $f, g \in \mathbb{Z}[x]/(x^n + 1)$ with n a power of two
Ensure: Polynomials F,G such that 2.1 is verified

if $n = 1$ then **then**
  Compute $u, v \in \mathbb{Z}$ such that $uf - vg = \gcd(f, g)$
  if $\gcd(f, g) \neq 1$ **then**
    abort and return $\perp$
  end if
  $(F, G) \leftarrow (vq, uq)$
  return (F,G)
else
  $f' \leftarrow \mathrm{N}(f)$
  $g' \leftarrow \mathrm{N}(g)$
  $(F', G') \leftarrow \mathbf{NTRUSolve}_{n/2,q}(f', g')$
  $F \leftarrow F'(x^2)g(-x)$
  $G \leftarrow G'(x^2)f(-x)$
  Reduce(f, g, F, G)
end if
return (F,G)

Algoritmo de solución de la ecuación de NTRU [19].

2.1.2. Clave pública

La clave pública pk es la información necesaria para poder verificar la firma por parte del receptor del mensaje. Para obtenerla, se necesita haber calculado previamente la clave privada y obtener los polinomios f y g. Siguiendo la ecuación

$$h \leftarrow g \cdot f^{-1} \bmod(\phi, q) \tag{2.3}$$

se obtiene un polinomio h que actuará como clave pública pk.

2.1.3. *Falcon tree*

Dos elementos son generados a partir de la clave privada, el primero es una matriz $\hat{B}$ con las representaciones FFT de los polinomios f, g, F y G. El segundo es el Falcon tree (T), el cual es un árbol binario de polinomios de altura κ, con una estructura que se define por las siguientes características.

$$\hat{\mathrm{B}} = \begin{bmatrix} \mathrm{FFT}(g) & -\mathrm{FFT}(f) \\ \mathrm{FFT}(G) & -\mathrm{FFT}(F) \end{bmatrix}$$

- El valor de un nodo del árbol es un polinomio $l \in \mathbb{Q}[x]/(x^n + 1)$ con $n = 2^\kappa$, es decir, perteneciente al anillo de polinomios de coeficientes racionales módulo $x^n + 1$. Siempre y cuando no sea un nodo hoja, que tiene un valor real $\sigma > 0$.
- Tiene una estructura de padres e hijos en la que cada nodo tiene dos árboles de altura $\kappa - 1$, como se puede observar en 2.1.
- Los polinomios se guardan en representación FFT, salvo los nodos del tipo hoja, que son valores reales.

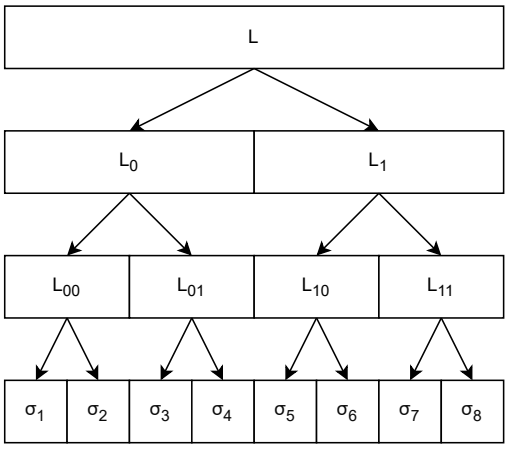

Figura 2.1: Ejemplo de Falcon tree de altura tres.

Los algoritmos que se utilizan para computar este Falcon tree se pueden ver a continuación.

Algorithm 4 ffLDL*(G)

Require: A full-rank Gram matrix G $\in$ FFT$(\mathbb{Q}[x]/(x^n+1))^{2x2}$
Ensure: A binary tree T
Format : All polynomials are in FFT representation.
(L,D) $\leftarrow$ **LDL***(G)
T.value $\leftarrow L_{10}$
if $n = 2$ **then**
  T.leftchild $\leftarrow D_{00}$
  T.rightchild $\leftarrow D_{11}$
  return T
else
  $d_{00}, d_{01} \leftarrow$ **splitfft**(D_{00}) $\{d_{i,j}\}$
  $d_{10}, d_{11} \leftarrow$ **splitfft**(D_{11})
  $\mathrm{G}_0 \leftarrow \begin{bmatrix} d_{00} & d_{01} \\ d_{01}^* & d_{00} \end{bmatrix}, \mathrm{G}_1 \leftarrow \begin{bmatrix} d_{10} & d_{11} \\ d_{11}^* & d_{10} \end{bmatrix}$
  T.leftchild $\leftarrow$ **ffLDL***(G_0)
  T.rightchild $\leftarrow$ **ffLDL***(G_1)
  return T
end if

Algoritmo de la generación del Falcon tree [19].

Algorithm 5 LDL*(G)

Require: A full-rank self-adjoint matrix G $= G_{ij} \in$ FFT$(\mathbb{Q}[x]/(\phi))^{2x2}$
Ensure: The LDL* decomposition G=LDL* over FFT$(\mathbb{Q}[x]/(\phi))$
Format : All polynomials are in FFT representation.
$D_{00} \leftarrow G_{00}$
$L_{10} \leftarrow G_{10}/G_{00}$
$D_{11} \leftarrow G_{11} - L_{10} \odot L_{10}^* \odot G_{00}$
$\mathrm{L} \leftarrow \begin{bmatrix} 1 & 0 \\ L_{10} & 1 \end{bmatrix}, \mathrm{D} \leftarrow \begin{bmatrix} D_{00} & 0 \\ 0 & D_{11} \end{bmatrix}$
return (L,D)

Algoritmo de la descomposición LDL [19].

2.2. Generación de firma

La generación de firma requiere haber computado la clave privada *pk* previamente y se necesita un mensaje m que firmar. El primer paso es hacer una operación *hash* sobre el mensaje a firmar dando como resultado un valor c y, posteriormente, se utiliza la clave privada para encontrar dos valores s_1 y s_2 que cumplan con la ecuación

$$s_1 + s_2 h = c \bmod q \tag{2.4}$$

con $c \in \mathbb{Z}_q[x]/(\phi)$. La generación de firma también necesita el Falcon tree, cuya computación es costosa, es por eso que conviene guardar los valores del árbol para realizar múltiples firmas con la misma clave privada. En el algoritmo (6) se puede ver la generación de firma paso a paso.

Algorithm 6 Sign (m,sk,$\lfloor\beta^2\rfloor$)

Require: A message m, a secret key sk, a bound $\lfloor\beta^2\rfloor$
Ensure: A signature sig of m
1: $r \leftarrow \{0,1\}^{320}$ uniformly
2: $c \leftarrow \textbf{HashToPoint}(r \mid\mid m, q, n)$
3: $t \leftarrow (-\frac{1}{q}FFT(c) \odot FFT(F), \frac{1}{q}FFT(c) \odot FFT(f))$
4: **repeat**
5: **repeat**
6: $z \leftarrow \textbf{ffSampling}_n(t, \mathrm{T})$
7: $s = (t - z)\hat{\mathrm{B}}$
8: **until** $\mid\mid s \mid\mid^2 \leq \lfloor\beta^2\rfloor$
9: $(s_1, s_2) \leftarrow invFFT(s)$
10: $s \leftarrow \textbf{Compress}(s_2, 8 \cdot sbytlen - 328)$
11: **until** $(s \neq \perp)$

Algoritmo de generación de firma de Falcon [19].

2.2.1. ffSampling

Al elegir el marco teórico GPV [27] como base, se necesita elegir un trapdoor sampler. Este sampler debe encontrar un vector s que cumpla la condición

$$s^t A = c \bmod q \tag{2.5}$$

donde A es una matriz y c es un vector objetivo. Como se ha explicado en la introducción de la sección 2, se eligió una variante que utilizaba la transformada rápida de Fourier sobre retículos de manera eficiente. Este algoritmo de muestreo permite dividir los polinomios en polinomios de menor tamaño utilizando la función *"splitfft"*, encontrar el vector resultado de la ecuación (2.5) de ambos polinomios menores y recombinar los resultados utilizando la función *"mergefft"*. El algoritmo de esta función se muestra a continuación.

Algorithm 7 ffSampling (t,T)

Require: t= $(t_o, t_1) \in \text{FFT}(\mathbb{Q}[x]/(x^n+1))^2$, a Falcon tree T
Ensure: z= $(z_o, z_1) \in \text{FFT}(\mathbb{Z}[x]/(x^n+1))^2$
Format : All polynomials are in FFT representation.
1: **if** $n = 1$ **then**
2: $\sigma' \leftarrow$ T.value
3: $z_0 \leftarrow$ **SamplerZ**(t_0, σ')
4: $z_1 \leftarrow$ **SamplerZ**(t_1, σ')
5: return z= (z_o, z_1)
6: **end if**
7: $(l, \text{T}_0, \text{T}_1) \leftarrow$ (T.value, T.leftchild, T.rightchild
8: $\text{t}_1 \leftarrow$ **splitfft**(t_1)
9: $\text{z}_1 \leftarrow$ **ffSampling**$_{n/2}(\text{t}_1, \text{T}_1)$
10: $z_1 \leftarrow$ **mergefft**(z_1)
11: $t'_0 \leftarrow t_0 + (t_1 - z_1) \odot l$
12: $\text{t}_0 \leftarrow$ **splitfft**(t'_0)
13: $\text{z}_0 \leftarrow$ **ffSampling**$_{n/2}(\text{t}_0, \text{T}_0)$
14: $z_0 \leftarrow$ **mergefft**(z_0)
15: return z= (z_o, z_1)

Algoritmo del sampler [19].

2.3. Verificación de firma

La verificación de firma es realizada por el receptor, que necesita conocer la clave pública del emisor pk, la firma digital y el mensaje original para hacer la comprobación. Este mensaje original m pasa por una función *hash* dando lugar a un polinomio $c \in \mathbb{Z}[x]/(\phi)$. A su vez, la firma s se decodifica dando lugar a un polinomio $s_2 \in \mathbb{Z}[x]/(\phi)$. Se busca entonces la solución a la ecuación

$$s_1 = c - s_2 h \bmod q \tag{2.6}.$$

Se verifica que una firma proviene del emisor cuando se cumple la condición

$$|| (s_1, s_2) ||^2 \leq \lfloor \beta^2 \rfloor \tag{2.7}$$

de aceptación, en caso contrario ha sido manipulada pro terceros y queda rechazada. A continuación se muestra el algoritmo correspondiente a la verificación de firma, un proceso menos complejo y más rápido que los anteriores.

Algorithm 8 Verify (m,sig,pk,$\lfloor\beta^2\rfloor$)

Require: A message m, a signature $sig = (r, s)$, a public key $pk = h \in \mathbb{Z}_q[x]/(\phi)$, a bound $\lfloor\beta^2\rfloor$
Ensure: Accept or reject

$c \leftarrow$ **HashToPoint**$(r \mid\mid m, q, n)$
$s_2 \leftarrow$ **Decompress**$(s, 8 \cdot sbytelen - 328)$
if $s_2 = \perp$ **then**
  reject
end if
$s_1 \leftarrow c - s_2 h \bmod q$
if $\mid\mid (s_1, s_2) \mid\mid^2 \leq \lfloor\beta^2\rfloor$ **then**
  accept
else
  reject
end if

Algoritmo de verificación de firma de Falcon [19].

2.4. Estado del arte

Para acabar con la sección 2 se va a realizar una revisión del estado del arte en la que se analizarán distintas implementaciones *hardware* de Falcon en FPGA. Hasta la fecha de publicación de este Trabajo de Fin de Master y en conocimiento del autor todavía no existe ninguna implementación puramente *hardware* al completo de Falcon diseñado desde cero, es por eso que las implementaciones que se revisarán serán diseños parciales *hardware/software*. Falcon tiene dos motivos principales por los que es complicado hacer una implementación *hardware*, la propia estructura del algoritmo y el uso del tipo coma flotante. La estructura del algoritmo es difícil de comprender como ya se explicó anteriormente en las limitaciones que tiene este esquema y, adicionalmente, utiliza funciones recursivas que para ser eficientes necesitan una gran cantidad de recursos. El uso de variables en tipo coma flotante es un problema para sistemas que no disponen de FPU de manera natural como los entornos *hardware*, se necesita emplear una mayor cantidad de recursos o incluso modificar el algoritmo para tratar estos datos.

La única implementación del esquema de firma digital Falcon al completo está desarrollada en *High Level Synthesis* (HLS) [31]. Es una opción cómoda ya que no necesita desarrollo *hardware*, sino que se utiliza código *software* para generar código *Hardware Description Language* (HDL). En esta implementación usan datos del tipo *float* o *double* de manera natural a lo largo de todo el algoritmo. Se modifica el código de las funciones recursivas para que sea sintetizable, utilizando grandes buffers y guardando el estado de la ejecución. Se emplean también ventajas que ofrece HLS para optimizar el código como *array partitioning*, *loop unrolling*, *function inlining*, *pipelining* y *dataflow*.

En cuanto a los diseños parciales *hardware/software* existe una gran variedad debido a que Falcon es un algoritmo extenso en el que existen muchas partes diferentes que se pueden optimizar. Estas optimizaciones pueden ir desde una de las partes principales como la generación de claves hasta el nivel más bajo de operaciones entre variables como la multiplicación. Dentro del primer tipo se encuentra FalconSign [32] que como su nombre indica implementa la parte de firma del algoritmo en *hardware*. El objetivo principal de esta implementación es reducir la latencia y aumentar el número de firmas por segundo. Esto lo consigue gracias a varias decisiones de diseño como una estructura de memoria céntrica accesible desde varios módulos y a su vez dividida para poder acceder a ella de forma concurrente. También diseñan una FPU que permite hacer las siguientes operaciones: suma, resta, multiplicación, FFT/IFFT, *splitfft* y *mergefft*. Se modifican

los diferentes módulos, como el Gaussian sampler, usando *pipelining* para reducir la latencia.

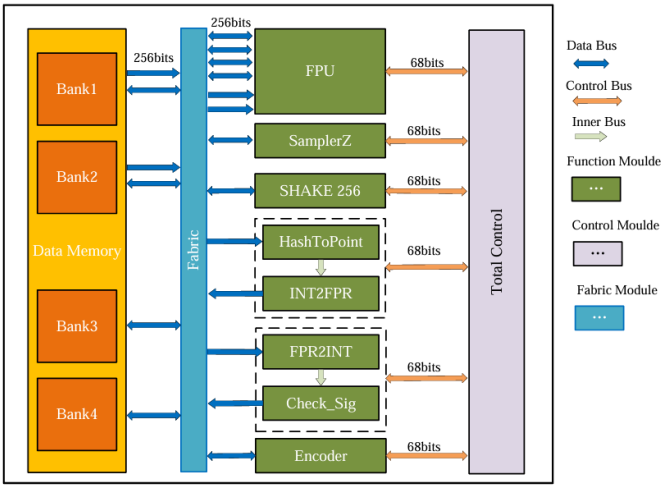

Figura 2.2: Arquitectura de FalconSign [32].

Beckwith et al. [33] diseñaron un acelerador que permitía mejorar el rendimiento tanto de CRYSTALS-Dilithium como de la verificación de Falcon. Se centraron en la verificación por ser un proceso simple en comparación con la firma o la generación de claves. Implementaron la decodificación y descompresión de polinomios para optimizar al máximo la ejecución de NTT. También se aprovechan de la paralelización de las partes de la verificación para reducir la latencia, como el *hash* junto a la NTT.

Las siguientes implementaciones bajan un nivel en la arquitectura para optimizar operaciones específicas en las que existe una opción real de aumentar el rendimiento. Uno de los cuellos de botella en la ejecución del algoritmo es la función SamplerZ, existen dos implementaciones que proponen optimizaciones diferentes. La primera llamada Bi-SamplerZ [34] realiza las dos llamadas a la función consecutivas en paralelo para ahorrar tiempo de ejecución. Procuran reducir el porcentaje de utilización compartiendo recursos entre ambas llamadas en paralelo sin reducir la velocidad. Emplean *pipelining* para reducir la latencia en caso de rechazo ya que consideran que ambas iteraciones no están siempre en el mismo estado. Otra de las funcionalidades que ofrecen es un mecanismo de asistencia en el que uno de los módulos que ya ha conseguido un valor correcto ayuda al otro que ha sido rechazado, aumentando así el ratio de aceptación y reduciendo el impacto del rechazo en la latencia.

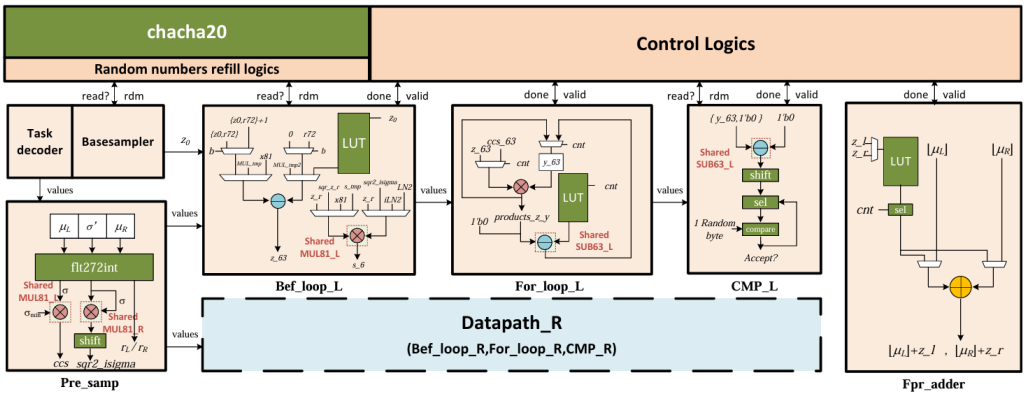

Figura 2.3: Muestra del pipelining de Bi-SamplerZ [34].

La segunda implementación que optimiza la función SamplerZ fue publicada recientemente en [35]. Se realizan modificaciones a nivel de arquitectura como el uso de *pipelining* o unidades FPU de menor latencia en caminos críticos. También se modifica a nivel de algoritmo, sustituyendo el método secuencial de Horner por el esquema de Estrin, que agrupa los coeficientes de los polinomios en pareja y los evalúa en paralelo.

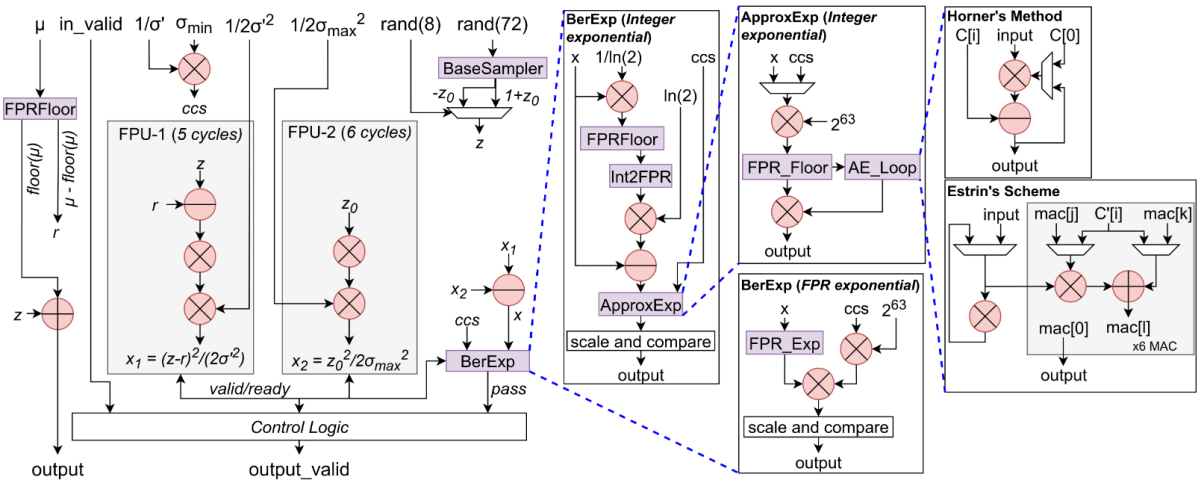

Figura 2.4: Diagrama de bloques de SamplerZ [35].

Cómo se ha visto previamente en esta sección 2, una parte de la generación de claves es resolver la ecuación de NTRU. Esta función NTRUSolve es optimizada en [36], donde implementan la multiplicación de enteros tanto grandes como pequeños, la multiplicación de Montgomery y la conversión entre números en representación *residue numeral system* (RNS) y números enteros. Se utilizan también circular shift registers (CSR) para la transmisión de datos entre módulos.

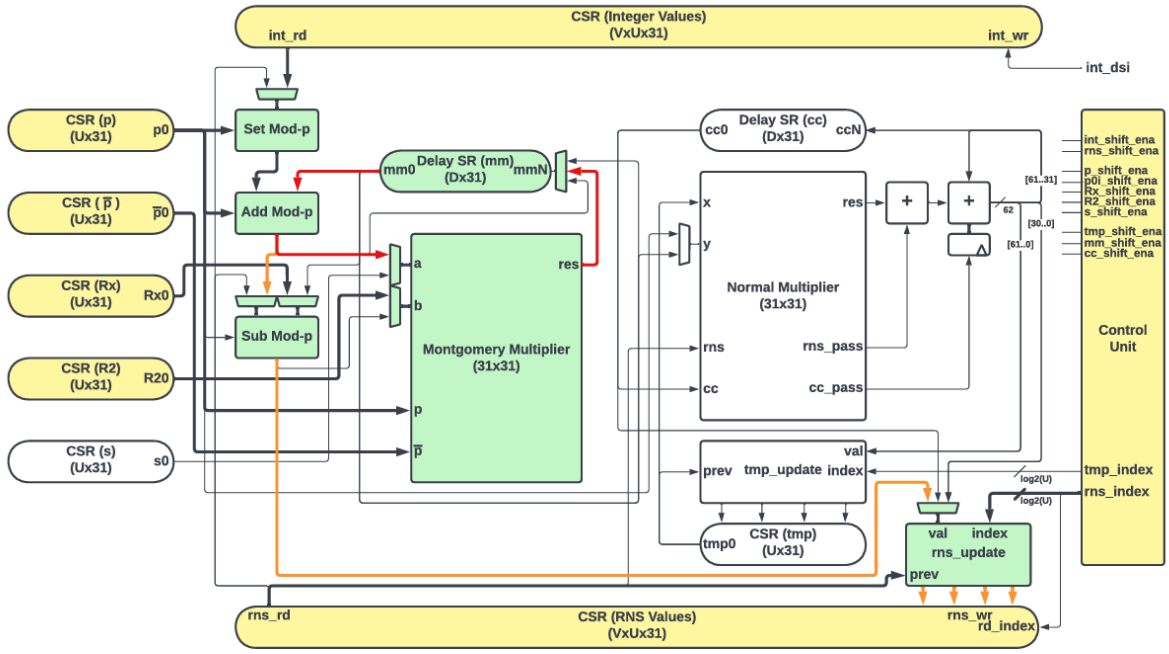

Figura 2.5: Arquitectura para descomposición RNS [36], conversión de entero a RNS.

Falcon depende en gran medida de dos operaciones polinómicas, la transformada rápida de Fou-

rier (FFT) y la transformada numérica teórica (NTT). Este hecho unido a que son utilizadas también por otros algoritmos, son objetivo de muchas implementaciones *hardware*. En [37] agrupan en un mismo acelerador ambas transformadas, reutilizando los recursos que se utilizan en la FFT para la NTT. Implementan una unidad de reorganización para que los valores se mantengan en el orden correcto tras cada etapa, así como una unidad de control para operar sobre los coeficientes correctos.

En [38] se diseña un acelerador únicamente para la FFT cuyo objetivo es reducir la utilización de recursos. El principal cambio en esta implementación es pasar del modelo de Cooley-Tuckey que más simple de implementar pero utiliza un mayor número de multiplicaciones por el modelo de Winograd, que utiliza menos multiplicaciones y reduce la latencia al trabajar con polinomios de grado alto. En esta implementación se diseñan tres módulos de computación eficiente de números complejos para la multiplicación, la suma y la resta.

Por último, recientemente en [39] se diseñó un acelerador para la NTT que puede ser utilizado en los esquemas de firma digital Falcon, CRYSTALS-Dilithium y HAWK. En esta implementación se busca aprovecharse de la velocidad de cálculo de la NTT en Radix-4. Se ha diseñado un módulo que utiliza tanto Radix-2 como Radix-4 para evitar reconfiguración dependiendo del grado del polinomio. Cuando el grado del polinomio no es potencia de cuatro, se aplica la primera ronda de la NTT en Radix-2 y se divide en dos polinomios sobre los que sí se puede aplicar la NTT en Radix-4.

Una comparación del uso de recursos y el rendimiento de estas implementaciones se puede ver a continuación.

Ref	*LUTs*	*FFs*	*BRAMs*	*DSPs*
[31]	100,6K/45,2K/13,6K	91K/41,4K/8619	69/37/14	1215/182/15
[32]	80496	46495	58	220
[33]	13956	6737	2	4
[34]	14327	10841	-	85
[35]	23000	11000	-	542
[36]	15169	26789	-	-
[37]	17395	7950	4	20
[38]	8396	2526	9,5	9
[39]	710	588	3	3

Tabla 2.1: Utilización de recursos del estado del arte.

Ref	*Frecuencia (MHz)*	Ciclos de reloj	Latencia (μs)
[31]	100/214,3/187,5	32,03M/1,86M/269,61K	320,3K/8,7K/1,3K
[32]	185	320K	1730
[33]	142	4687	32,8
[34]	150	59	0,39
[35]	128	28	0,219
[36]	371,5	1392	3,744
[37]	134	4096	31
[38]	200	19800	99
[39]	215	1392	6,47

Tabla 2.2: Rendimiento del estado del arte.

3. DECISIONES DE DISEÑO DEL ACELERADOR

El objetivo de este Trabajo de Fin de Master es diseñar e implementar un acelerador puramente *hardware* del esquema de firma digital Falcon. Las implementaciones *hardware* son utilizadas en la actualidad ya que permiten desviar la carga de trabajo del procesador hacia un elemento externo como la FPGA. El procesador utilizará la FPGA como un periférico a través de registros que permitan configurar o actuar sobre la implementación, pasando a trabajar en paralelo y consultando periódicamente o por interrupción si la FPGA ha finalizado. Una implementación bien optimizada es capaz de conseguir los mismos resultados que el *software* de manera más eficiente y reduciendo el tiempo de ejecución, lo que se denominaría acelerador.

En esta sección 3 se explicarán las decisiones que se han debido tomar a lo largo del proceso de diseño del acelerador. Al ser la primera implementación puramente *hardware* diseñada desde cero de Falcon al completo, se ha desarrollado el código HDL haciendo módulos equivalentes a las funciones de la referencia base [40]. El hecho de estar diseñada desde cero es esencial para explotar el paralelismo que ofrece el diseño *hardware* y reducir la cantidad de ciclos de reloj necesarios por función. Sin embargo, hay ciertas características del lenguaje de programación secuencial C de la referencia que no son extrapolables a lenguaje de programación concurrente como VHDL y que necesitan ser modificados para la implementación. Estas características se mencionan a continuación.

1. Las funciones *software* representan un espacio en memoria que es reutilizado en repetidas ocasiones mientras que los módulos *hardware* ocupan un espacio real dentro de la zona programable de la FPGA.
2. El lenguaje de programación C permite reservar espacios de memoria de tamaño variable que son accesibles desde cualquier función del sistema a partir de punteros mientras que en *hardware* las memorias deben estar definidas previamente.
3. Los datos almacenados en memoria dinámica pueden ser tratados con tipos de datos distintos dependiendo del puntero que acceda a ellos mientras que en *hardware* los tamaños de datos de las memorias quedan definidos previamente.
4. Es posible utilizar tipos de datos en coma flotante utilizando una FPU nativa mientras que en *hardware* la FPU no está implementada de forma natural.
5. Se computan multiplicaciones de números grandes de hasta 64 *bits*, que se traduce en una sucesión de multiplicaciones en *hardware* con alto retardo lógico.
6. El lenguaje de programación secuencial tiene la posibilidad de usar recursividad en sus funciones mientras que en *harwdare* esto implicaría la duplicación de recursos.

En las secciones posteriores se explicarán las medidas tomadas para resolver estos problemas con el fin de adaptar el esquema de firma digital al entorno *hardware*. Estas medidas son de nivel conceptual al tener que tomar decisiones sobre la forma de implementar las funciones y las memorias. También son medidas de nivel estructural del algoritmo ya que se necesitan decidir que ramas del código de referencia se van a implementar. Por último también se toman medidas a nivel del algoritmo, se necesita modificar ciertas partes del algoritmo para replicar el correcto funcionamiento del código de referencia.

3.1. Diseño de tipo modular

El código *software* permite la definición global de las funciones que se van a utilizar, solo es necesario hacer la definición una vez. Estas funciones representan un espacio en memoria en el cual ciertas instrucciones se ejecutan de manera consecutiva cada vez que se hace una llamada a la función, es decir, el uso repetido de las funciones no genera recursos adicionales. En el ejemplo a continuación, se puede ver la comparación entre un bucle generado en *software* y un bucle generado en *hardware*, en el segundo sintetizarán tantos sumadores como iteraciones tiene el bucle mientras que en el software se usa siempre la misma función. Como ventaja, este *hardware* tardará un ciclo en ejecutar todo el bucle.

```
entity suma_id is
    Port (
        A     : in  std_logic_vector(7 downto 0);
        Sum   : out std_logic_vector(7 downto 0)
    );
end suma_id;

architecture Behavioral of suma_id is
begin
    Sum <= std_logic_vector(signed(A) + signed(A));
end Behavioral;

entity bucle_suma is
    Port (
        Input : in  std_logic_vector(7 downto 0);
        Sum_Final : out std_logic_vector(7 downto 0)
    );
end bucle_suma;

architecture Structural of bucle_suma is

    type sum_array is array (0 to 5) of std_logic_vector(7 downto 0);
    signal sums : sum_array;

    component suma_id
        Port (
            A     : in  std_logic_vector(7 downto 0);
            Sum   : out std_logic_vector(7 downto 0)
        );
    end component;

begin

    sums(0) <= Input;

    gen_suma_id: for i in 0 to 4 generate
        DUP: suma_id
            port map(
                A     => sums(i),
                Sum   => sums(i+1)
            );
    end generate gen_duplicadores;

    Sum_Final <= sums(5);

end Structural;
```

Código 3.1: Descripción en lenguaje VHDL de un bucle de suma.

```
int suma_id(int a){
    return a + a;
}
int main(){
    int res = 1;
    for(int = 0; i < 5; i++){
        res = suma_id(res);
    }
    return 0;
}
```

Código 3.2: Código *software* de un bucle de suma.

Para resolver este problema de duplicación de recursos se ha optado por una estructura de tipo modular. Estos módulos son de uso compartido entre todos los que conforman la arquitectura, no están instanciados dentro de cada componente que los necesite sino que son independientes y se comunican entre ellos cuando lo necesiten. En la imagen 3.1 se puede ver la estructura necesaria para la parte de verificación de firma. A la entrada de cada módulo que actuará como esclavo de varios módulos superiores se encuentra un multiplexor que maneja la entrada de datos a través de una señal de control. Dos módulos maestros podrán llamar al mismo módulo esclavo sin generar recursos adicionales siempre y cuando no lo hagan de forma simultánea.

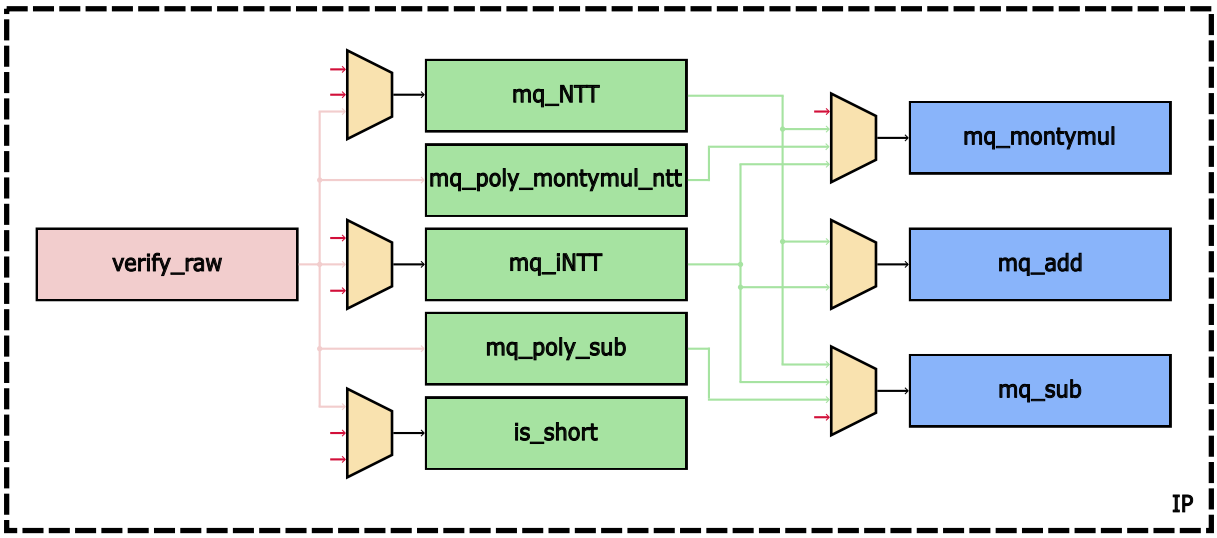

Figura 3.1: Diseño modular de tipo árbol.

La estructura de tipo modular tiene también la gran ventaja de poder sustituir los módulos con facilidad con un cambio de interfaz. El desarrollador podrá cambiar un módulo específico que ofrezca un resultado equivalente haciendo un simple cambio de interfaz. Esta interfaz se muestra en la imagen 3.2 con el caso específico del módulo de resta de números en coma flotante emulada *fpr_sub*. Los módulos dispondrán de señales de reloj para asegurar un diseño síncrono y de reinicio conectado al respectivo botón de la FPGA. También tienen señales de control, las cuales permiten a los módulos maestros de este habilitarlo y saber cuando ha finalizado la ejecución, y permiten al propio módulo hacer lo mismo con sus módulos esclavos. Por último, tienen señales de transmisión de datos bidireccional con los módulos maestros, los módulos esclavos, las memorias ROM de datos constante y la unidad de manejo de memoria (MMU).

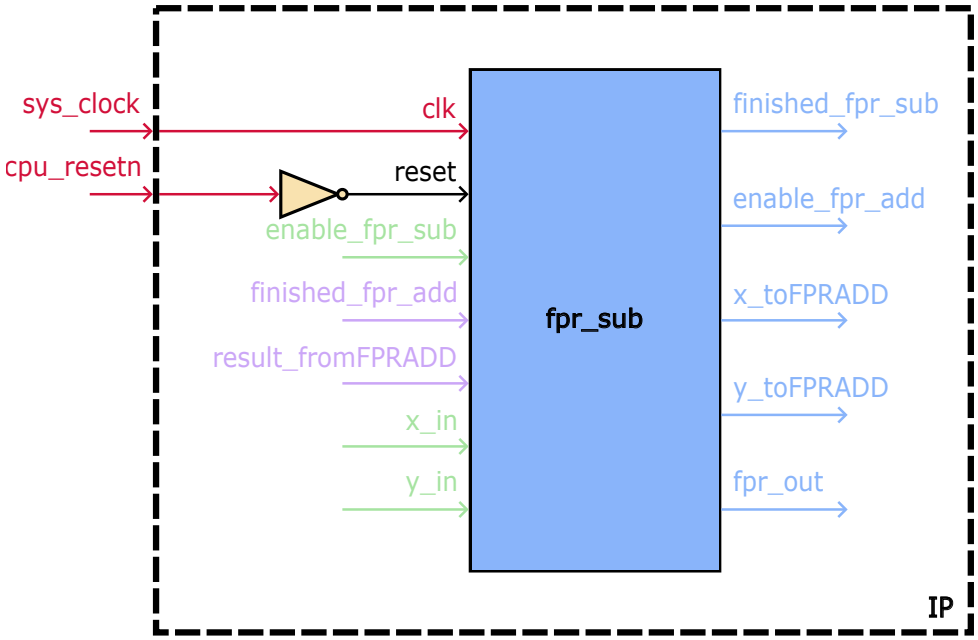

Figura 3.2: Ejemplo de módulo fpr_sub.

3.2. Diseño y comunicación de módulos

En esta sección se mostrará un ejemplo práctico de cómo es la comunicación entre estos módulos maestros y esclavos gracias a las señales de habilitación y finalización. Cómo se ha explicado en la sección previa se disponen de señales de control para gestionar la ejecución de los módulos, estas son las siguientes:

- *enable_module* : esta señal de entrada del módulo estará conectada directamente a la salida del módulo maestro. Mientras que esta señal está activa, el módulo continuará su ejecución hasta finalizar, pasando por una serie de estados intermedios hasta volver al estado de *S_RESET*. Cuando un módulo puede ser habilitado por varios módulos maestros, las señales se combinarán a través de puertas lógicas para reducirlas a una entrada única, como se puede ver en la imagen 3.3
- *finished_module* : esta señal de salida del módulo estará conectada directamente a una entrada de del módulo maestro. Esta señal estará activa un ciclo de reloj mientras que el módulo se encuentre en el estado *S_FINISH*, en el que se da valor a todos los resultados finales. Esta señal actuará en el módulo maestro como una bandera para que no continúe la sucesión de estados mientras que no se tenga en resultado proveniente del módulo esclavo.

Si el módulo debe a su vez habilitar un módulo esclavo también dispondrá de señales de control para actuar sobre este, es decir, actuar como esclavo y como maestro a la vez.

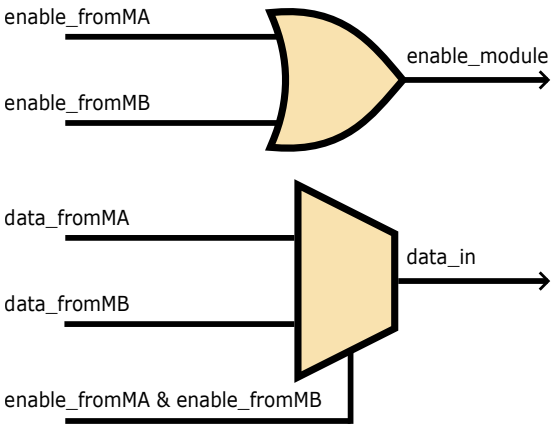

Figura 3.3: Entrada múltiple a un módulo.

Las señales de transmisión de datos siguen una estructura similar a las señales de control, estas permiten obtener los datos necesarios para la ejecución y devolver el resultado correspondiente. Los tipos de señales son los siguientes:

- *data_in* : explicado de forma genérica, son los datos necesarios para la ejecución del módulo, se corresponden a los datos de entrada de las funciones *software*. Es trabajo del módulo maestro transmitir los datos correctos antes de habilitar el módulo esclavo, que iniciará la ejecución tras este suceso. Cuando existan dos buses de datos que puedan ser transmitidos a un mismo módulo esclavo, se dispondrá de un multiplexor que seleccione el bus correcto en cada ocasión. Se puede ver esta selección en la imagen 3.3.
- *data_out* : estas señales permiten transmitir el resultado al finalizar la ejecución del módulo. Es trabajo del módulo esclavo de calcular el resultado previo a llegar al estado *S_FINISH*, devolver el resultado correcto al módulo maestro.

Para el caso de transmisión de datos con la MMU, se explicará en las siguientes secciones, ya que estas no necesitan ser habilitadas sino únicamente gestionar sus valores de entrada y de salida. En la siguiente descripción se muestra un ejemplo de cómo están diseñados los módulos que actúan como maestro y como esclavo a la vez.

```
entity module is
    Port (
        clk                     : in std_logic;
        reset                   : in std_logic;
        enable_module           : in std_logic;
        finished_module         : out std_logic;
        enable_slave            : out std_logic;
        finished_slave          : in std_logic;
        x_to_slave              : out std_logic_vector(63 downto 0);
        y_to_slave              : out std_logic_vector(63 downto 0);
        result_from_slave       : in std_logic_vector(63 downto 0);
        x_from_master           : in std_logic_vector(63 downto 0);
        y_from_master           : in std_logic_vector(63 downto 0);
        result_to_master        : out std_logic_vector(63 downto 0)
    );
end module;
```

Código 3.3: Descripción de ejemplo de la entidad de un módulo.

```
architecture Behavioral of module is
    signal state        : state_module;
    signal aux      : std_logic_vector(63 downto 0);
    signal x_aux, y_aux : unsigned(63 downto 0);
begin
    FSM : process(clk, reset)
    begin
        if reset = '1' then
            state <= S_RESET;
        elsif clk'event and clk = '1' then
            if enable_module = '1' then
                case state is
                    when S_RESET =>
                        state <= S_EN_SLAVE;
                    when S_EN_SLAVE =>
                        if finished_slave = '1' then
                            state <= S_FINISH;
                        else
                            state <= S_EN_SLAVE;
                        end if;
                    when S_FINISH =>
                        state <= S_RESET;
                    when others =>
                end case;
            else
                state <= S_RESET;
            end if;
        end if;
    end process;
    module_proc : process(clk, reset)
    begin
        if reset = '1' then
            aux   <= (others => '0');
            x_aux <= (others => '0');
            y_aux <= (others => '0');
        elsif clk'event and clk = '1' then
            if enable_fpr_sub = '1' then
                case state is
                    when S_RESET =>
                        x_aux <= unsigned(x_from_master) + 2;
                        y_aux <= unsigned(y_from_master) + 2;
                        aux   <= (others => '0');
                    when S_EN_SLAVE =>
                        if finished_slave = '1' then
                            aux <= result_from_slave;
                        end if;
                    when others =>
                end case;
            end if;
        end if;
    end process;
    finished_module <= '1' when state = S_FINISH else '0';
    result_to_master <= aux when state = S_FINISH else (others => '0');
    enable_slave <= ('1' xor finished_slave) when state = S_EN_SLAVE else '0';
    x_to_slave <= std_logic_vector(x_aux) when state = S_EN_SLAVE else (others
    => '0');
    y_to_slave <= std_logic_vector(y_aux) when state = S_EN_SLAVE else (others
    => '0');
end Behavioral;
```

Código 3.4: Descripción de ejemplo del comportamiento de un módulo.

Para finalizar la sección, un ejemplo práctico según ciclos de reloj de cómo actuaría el módulo dela descripción 3.4.

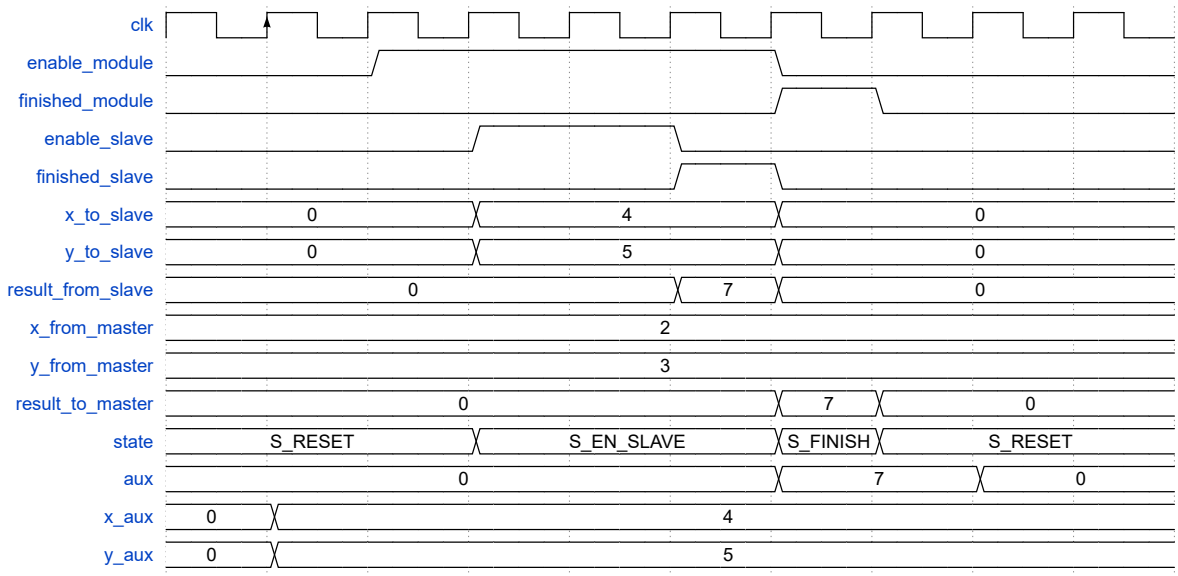

Figura 3.4: Ejemplo de ejecución de un módulo.

3.3. Arquitectura de memoria céntrica

La capacidad de la programación *software* de reservar memoria de manera dinámica permite acceder a una gran cantidad de datos de forma sencilla a través del uso de punteros. Los punteros guardan el valor de una dirección de memoria en la que se alberga un dato, pudiendo acceder a estos datos y posteriores en la misma región de memoria desde diferentes funciones. Esto no pasa en la programación *hardware* en el que la cantidad de memoria esta definida de forma previa a la síntesis. Las memorias instanciadas en *hardware* (BRAM) suelen ser específicas para cada módulo o compartidas entre varios para acceder a la información de manera rápida, sin embargo en el código de referencia de Falcon se accede a las mismas posiciones de memoria desde una gran cantidad de módulos y seguir con el enfoque tradicional resultaría en un entramado de memorias y señales intermedias poco prácticas.

La solución que se ha tomado es implementar una arquitectura de memoria céntrica, en la que se dispone de una memoria con el espacio suficiente para guardar todos los valores temporales utilizados a lo largo del algoritmo. La memoria tiene dos puertos para acelerar la carga y lectura de datos y cuenta con un sistema de multiplexores para el adecuado flujo de datos. Como se verá en la sección 4, esta memoria también es accesible por parte del microprocesador. Un diagrama sobre lo que implica esta arquitectura se puede observar en la imagen 3.5.

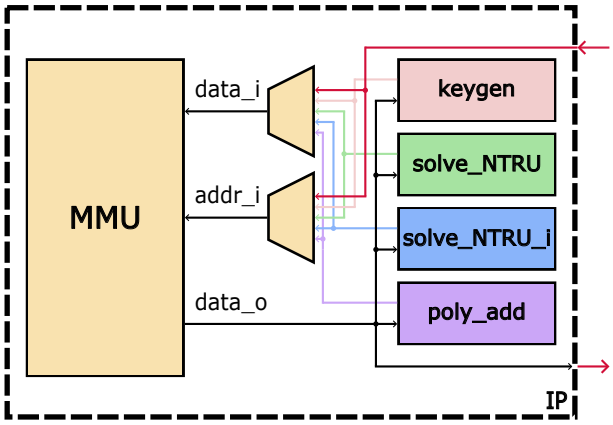

Figura 3.5: Arquitectura de memoria céntrica.

3.4. Unidad de manejo de memoria

El uso de memoria dinámica en programación *software* no sólo facilita el almacenamiento de datos y la accesibilidad de los mismo, también ofrece la posibilidad de utilizar estos datos con tipos diferentes. En el caso de que dos punteros tengan como valor la misma dirección de memoria pero sean de tipos diferentes, por ejemplo el primero de 8 *bits* y el segundo de 16 *bits*, al acceder a esa posición de memoria se leerán dos datos diferentes. Un ejemplo claro se puede ver en la imagen 3.6.

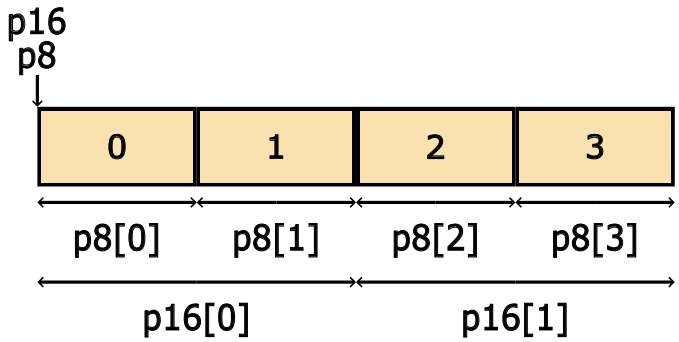

Figura 3.6: Acceso a memoria de dos punteros de tipo diferente.

En el caso del código referencia de Falcon se utilizan tipos de datos de tamaño 8 *bits*, 16 *bits*, 32 *bits* y 64 *bits*. La medida a tomar es cambiar la memoria central que se explicó en el apartado anterior por una MMU que fuera capaz de tratar con todos estos tipos de datos. Se plantearon dos opciones para implementar la MMU:

- Ajustar el tamaño de datos de la memoria al tipo de dato más grande. Una vez leída la zona de memoria que se necesite, se separa en el tipo de dato que se necesita. Este diseño permitiría tener una entrada y salida de la MMU más uniforme, con tamaños de datos fijos, sin embargo el cableado ocuparía demasiado y podría suponer un problema al sintetizar.
- Ajustar el tamaño de datos de la memoria al tipo de dato más pequeño. El tamaño del cableado será lo más reducido posible, ajustando cada acceso a la MMU al tipo necesario de dato, sin embargo se necesitarán recursos adicionales para el tratamiento de estos datos.

Se ha elegido la segunda opción, ya que los recursos adicionales necesarios se pueden compartir entre todos los módulos y no ser específicos de cada bloque. La MMU actuará como un módulo más en el que los otros módulos solicitarán hacer una acción como leer, escribir o mover un bloque de memoria, aportando los valores necesarios, y esperarán a que llegue la señal de finalizado. Este módulo de la MMU constará por tanto de cinco componentes:

1. *RAM* : componente en el que se almacena la información. Como ya se ha comentado anteriormente el tamaño de dato que maneja es de 8 *bits*, el tipo más pequeño entre los usados por el algoritmo. El tamaño de esta memoria es de aproximadamente 90 kB, en ella se pueden almacenar los datos de todo el algoritmo al completo, junta los *buffers* temporales de generación de claves, generación de firma y verificación de firma así como el espacio necesario para almacenar los valores de las claves y la firma.

2. *RW_FSM* : componente que para leer y escribir utiliza una máquina de estados (FSM) que envía las direcciones de memoria a la RAM. Esto es debido a que los tipos de datos de mayor tamaño necesitan acceder a varias direcciones de la memoria. Por ejemplo para almacenar un dato de 32 *bits* en la MMU, se necesita conocer la dirección base que manda el módulo y además las tres siguientes ya que se necesitan escribir cuatro datos de 8 *bits*.

3. *MEM_FSM* : componente que permite leer y escribir de forma consecutiva múltiples direcciones de memoria con el fin de hacer un trasvase de información. Este componente permite ahorrar recursos en los módulos que solicitan un movimiento de memoria, ya que solo deben mandar una dirección de origen, una de destino y una cantidad de datos para después esperar a que haya finalizado. Para evitar solapamientos de memoria, esta máquina de estados tiene en cuenta si el movimiento de memoria se hace hacia una dirección superior a una inferior o viceversa para empezar a escribir desde el final del bloque de memoria o desde el principio.

4. *DECONCATENATE* : componente que divide los datos de entrada en ocho registros de 8 *bits* que serán mandados consecutivamente a la RAM gracias a la máquina de estados. La entrada de este componente y por tanto de la MMU es de 64 *bits* para unificar la manera en la que los datos llegan desde el resto de módulos. Por ejemplo, al intentar escribir un dato de 32 *bits* en los primeros cuatro registros se encontrará el valor de este dato y en los cuatro restantes habrá ceros que no serán transferidos a la RAM.

5. *CONCATENATE* : componente que agrupa los datos de varios registros en función del tipo de dato que se desee leer. Cuenta con siete registros en los cuales el valor se transfiriere de uno a otro hasta que tras el número de ciclos necesarios se adquiere el dato que se desea leer, concatenando el valor de los registros en las salidas. Por ejemplo, para leer un dato de 32 *bits*, se debe esperar tres ciclos de reloj y posteriormente concatenar el dato de la salida de la RAM con los tres primeros registros.

Un diagrama de la MMU con sus cinco componentes y la conexión que se realiza entre ellos se puede ver en la imagen 3.7.

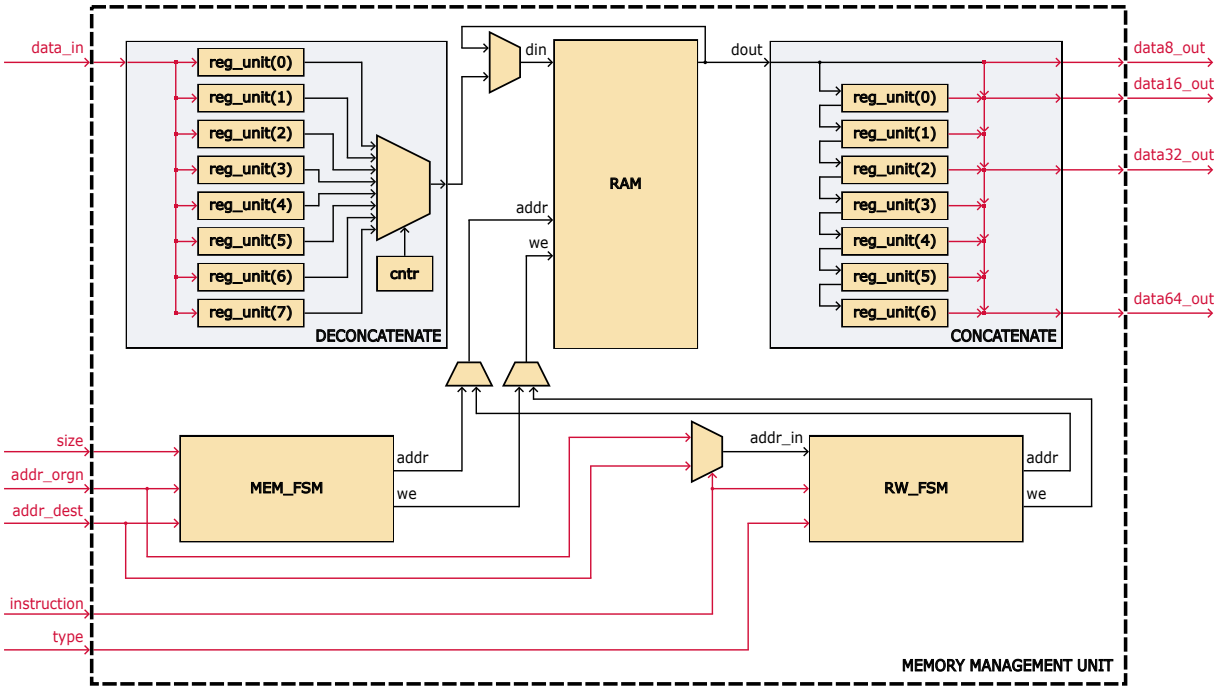

Figura 3.7: Unidad de manejo de memoria.

3.5. Aritmética en coma flotante

La FPGA en la que se va a implementar este diseño no dispone de microprocesador ni de FPU, lo que implica que no se puede trabajar de manera natural con tipos de datos en coma flotante. Los autores de Falcon definieron diferentes ramas del algoritmo en función de las características de la plataforma sobre la que se iba a implementar el algoritmo, algunas sobre la propia arquitectura del procesador o incluso introduciendo código ensamblador para algunos procesadores. Otra de las opciones que daban era seleccionar si el procesador sobre el que se iba a implementar tenía una FPU nativa o necesitaba trabajar sobre tipo de coma flotante emulado. Este trabajo se encuentra en el segundo caso, los módulos han sido diseñados a partir de las funciones que se encontraban bajo la rama del uso de coma flotante emulado.

El tipo de coma flotante emulado sigue el estándar IEEE-754 [30], que permite representar un número en coma flotante a través de un número de 64 *bits*. Este número de coma flotante estará dividido en signo, mantisa y exponente de la siguiente forma:

- El bit más significativo toma el valor del signo. Si el número es positivo el valor de este bit será 0 y en caso de que sea negativo el valor será 0.
- Los 52 bits menos significativos se corresponden a la mantisa. La mantisa alberga la magnitud del número.
- Los 11 bits restantes que quedan entre el signo y la mantisa toman el valor del exponente. El exponente almacena el valor de la potencia de dos que debe ser multiplicada la mantisa.

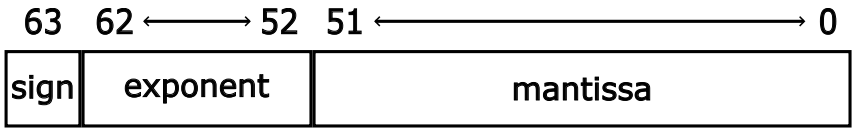

Figura 3.8: Estructura del tipo coma flotante emulado.

Hay ciertos valores especiales que se deben tener en cuenta cuando se usa el tipo de coma flotante emulado. Los relativos al algoritmo de Falcon vienen explicados en el código de referencia [40] y son los siguientes:

- Si el exponente es igual a 2047, el valor es infinito si la mantisa es igual a 0. Cuando la mantisa es diferente de cero el valor es indefinido (NaN).
- Si el exponente es igual a 0, el valor es cero si la mantisa es igual a 0. Cuando la mantisa es diferente de cero el valor sale del rango normalizado.

Es labor de la persona que ejecute el código que los valores utilizados no den lugar a valores especiales como infinitos o no normalizados. En el caso de que las entradas al algoritmo den lugar a estos valores, se producirán resultados indeterminados, sobre los cuales no se puede asegurar la seguridad.

3.6. Algoritmo de Karatsuba

Como ya se ha comentado anteriormente en esta sección 4, se utilizan señales de gran tamaño en diversas partes del algoritmo, como los números en coma flotante emulado. Las operaciones básicas sobre este tipo de señales como el desplazamiento de bits, la suma o la concatenación se pueden realizar sin problema de manera individual o en serie con otras mientras no se genere un elevado retardo lógico. En el caso de las multiplicaciones, se requiere una gran cantidad de recursos y se genera mucho retardo lógico si se implementan sobre puertas lógicas al uso. Es por esto que se tiende a implementar las multiplicaciones en unidades de procesamiento de señales digitales (DSPs) llegando al punto de colocar atributos que fuercen a utilizar estas unidades en la síntesis. En la FPGA que se va a utilizar en este trabajo, hay un total de 740 DSPs que cuentan con un multiplicador 25x18 en complemento a dos y un acumulador de 48 bits. Lo que tiene dos grandes inconvenientes: para hacer una multiplicación de números grandes se necesitan dos o tres DSPs en serie que producen un gran retardo lógico y estas DSPs están en un lugar fijo por lo que al conectarlas en serie se generará un retardo de red adicional.

Para aumentar la frecuencia a la que funciona el reloj y reducir el *worst negative slack* (WNS) se necesita reducir el número de multiplicaciones entre números grandes. La solución a este problema se basa en aplicar el algoritmo de Karatsuba [41] con el fin de dividir la multiplicación en otras más pequeñas. El algoritmo de Karatsuba se pensó para reducir el número de multiplicaciones que se hacen por operación y sustituirlas por sumas. La idea principal es que un número se puede expresar como una suma entre dos números, uno de ellos multiplicado por un factor. Por ejemplo el número 7432 se puede dividir en una suma de 32 y 74 multiplicado por 100, por lo que podríamos tener la siguiente ecuación

$$mul = 7432 \cdot 527 = (74 \cdot 10^2 + 32) \cdot (5 \cdot 10^2 + 27)$$

Esta descomposición se puede extrapolar de la división en base 10 a hacerlo en base 2 para señales binarias, cambiando multiplicaciones por desplazamientos de bits. En el algoritmo 9 se pueden observar los pasos necesarios para hacer la multiplicación de Karatsuba en binario.

Algorithm 9 Binary Karatsuba Algorithm (x,y)

Require: Two operands x, and y
Ensure: A product z
1: $x_h \leftarrow x(size - 1\ dt\ size/2)$
2: $x_l \leftarrow x(size/2\ dt\ 0)$
3: $y_h \leftarrow y(size - 1\ dt\ size/2)$
4: $y_l \leftarrow y(size/2\ dt\ 0)$
5: $p_h \leftarrow x_h \cdot y_h$
6: $p_l \leftarrow x_l \cdot y_l$
7: $s_x \leftarrow x_h + x_l$
8: $s_y \leftarrow y_h + y_l$
9: $p_m \leftarrow s_x \cdot s_y$
10: $s_m \leftarrow p_m - p_h - p_l$
11: $z \leftarrow p_h \ll 64 + s_m \ll 32 + p_l$

Esta alternativa implementada en *hardware* provoca un aumento de la frecuencia pero a raíz de consumir más ciclos de reloj, una multiplicación pasa de ser una operación sencilla de un único ciclo a una de varios ciclos de reloj. Esto no supone un problema en las multiplicaciones de 64 *bits* ya que no se ejecutan tan a menudo y es un coste asumible. Sin embargo hay algunas multiplicaciones de 32 *bits* que también necesitan utilizar el algoritmo de Karatsuba y estas son mas recurrentes. Es por eso que se han diseñado dos aceleradores, uno de ellos que aplica Karatsuba a las multiplicaciones de 64 *bits* y otro que también las aplica a ciertas multiplicaciones de 32 *bits* problemáticas. Se compararán ambos resultados para ver qué implementación resulta más eficiente.

3.7. Recursividad de la función *ffSampling*

La recursividad es un concepto que no existe en programación *hardware*, si necesitas utilizar un módulo en repetidas ocasiones debes instanciar tantos módulos como sea necesario hasta llegar a la última iteración de la recursividad. Esta opción no es viable ya que se generarían una gran cantidad de recursos y habría que reducir la flexibilidad del acelerador eligiendo un grado de polinomios sobre los que operar. Se necesita modificar el algoritmo para poder usar un único módulo en cada llamada a la función recursiva. Se ha optado por implementar dos módulos dependientes, uno es del tipo máquina de estados que habilita el módulo de operaciones y que guarda el valor intermedio del estado previo a volver a habilitar el módulo en una pila. El otro módulo se encarga de la ejecución del equivalente a la función *software* de referencia, y cuando llega a la llamada recursiva informa al módulo de la máquina de estados para que configure la siguiente llamada. Su funcionamiento se puede asemejar a una estructura de árbol en la que se va ejecutando el algoritmo por partes y moviéndose entre nodos padre y nodos hijo.

En la imagen 3.9 se puede ver el diagrama de estados que sigue este módulo de control. Este módulo puede habilitar el módulo de ejecución, recuperar el estado anterior de la pila o guardar el estado actual y descender a uno de los nodos hijo. En el estado *S_FFSAMPLING* se habilita al otro módulo para la ejecución del algoritmo según el estado en el que se encuentre el nodo correspondiente: previo a la primera llamada recursiva, entre llamadas o después de la segunda

llamada recursiva. En los estados *S_RIGHT* y *S_LEFT* se desciende al nodo hijo, por lo que se debe guardar el valor del estado actual, es decir, las direcciones de memoria que se operan, el grado del polinomio sobre el que se opera y el punto de la ejecución en el que se encuentra el nodo. Finalmente, en el estado *S_PREVIOUS* se toman los valores de la pila del nodo padre.

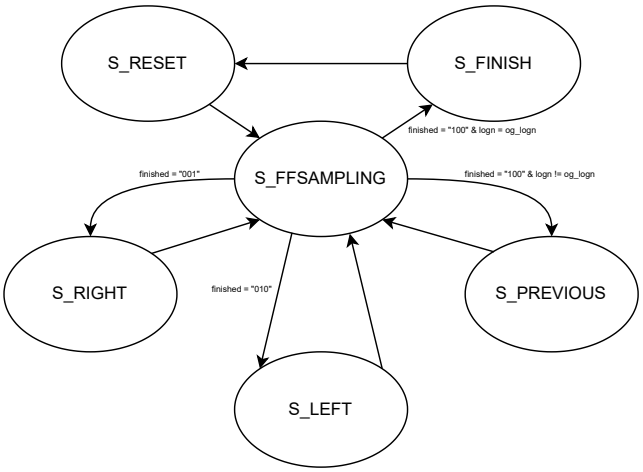

Figura 3.9: Máquina de estados de FSM_Sampling.

El módulo que se encarga de la ejecución del algoritmo está dividido en tres partes como se ha comentado antes. Será el módulo de control el que indique la situación en la que se encuentra el módulo de ejecución a través de una señal de tipo bandera. Según esta bandera y el grado del polinomio, el módulo de ejecución pasará del estado *S_RESET* a tres subrutinas diferentes.

4. ESTRUCTURA DE LA IMPLEMENTACIÓN

En esta sección 4 se explicarán las partes principales que conforman la implementación. En la sección 3 se han explicado los condiciones que son susceptibles a generar problemas y como estas se han solventado, pero queda por definir cómo es un módulo básico en sí. Todos los módulos de este acelerador están definidos al igual que el módulo de ejemplo, este se divide en tres partes diferenciadas: dos procesos y una sección de descripción *hardware* combinacional.

1. Máquina de estados : el primer proceso de cada módulo corresponde a la máquina de estados del sistema. En cada módulo siempre habrá un estado de *S_RESET* en el que queda bloqueado cada vez que no está habilitado y también habrá un estado de *S_FINISH* en el se levanta la bandera de finalización del módulo. Los estados intermedios tardarán un número determinado de ciclos dependiendo de si son operacionales o sirven de estado bloqueante hasta la finalización de la ejecución de un módulo esclavo. Un ejemplo gráfico se puede ver en la imagen 4.1.

2. Ejecución del algoritmo : el segundo proceso de cada módulo corresponde a las operaciones necesarias para la ejecución del algoritmo. Este proceso se rige por los estados que actualiza en cada ciclo de reloj el proceso anterior. En el estado de *S_RESET* el proceso inicializa las señales internas del módulo con el valor de las entradas correspondientes. En el estado de *S_FINISH* usualmente no se realiza ninguna operación por lo que las señales no cambian de valor. Los estados intermedios corresponden a la propia actuación del módulo para tener un funcionamiento equivalente a la función *software* de referencia. En estos estados intermedios se podrán realizar operaciones o esperar a que un módulo esclavo finalice su ejecución para asignar los valores resultantes. Un ejemplo gráfico se puede ver en la imagen 4.2.

3. Operaciones combinacionales : estas operaciones asignan el valor de las salidas del módulo. A pesar de ser combinacionales, estas salidas dependen del estado del módulo y los valores de las señales por lo que su actualización también será dependiendo de los ciclos de reloj. Estas salidas son tanto para dar información a los módulos esclavos como devolver los resultados a los módulos maestros. La información hacia los módulos esclavos estará activa mientras el módulo se encuentre en el estado bloqueante correspondiente. Los resultados enviados a los módulos maestros estará activa mientras que el módulo se encuentre en el estado de *S_FINISH*, es decir mientras que la bandera de finalización esté levantada.

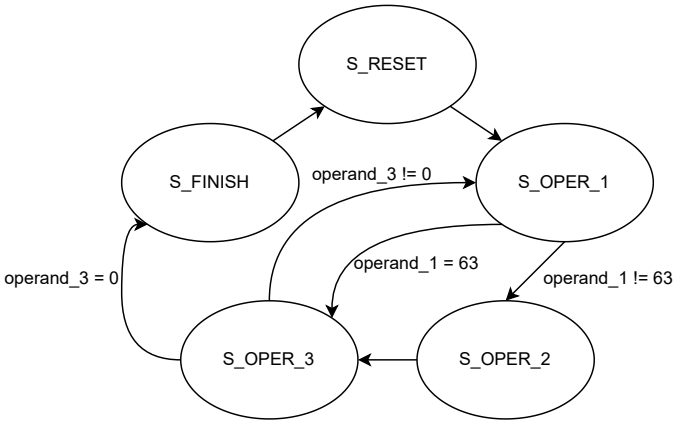

Figura 4.1: Diagrama de estados del módulo *FPR_NORM64*.

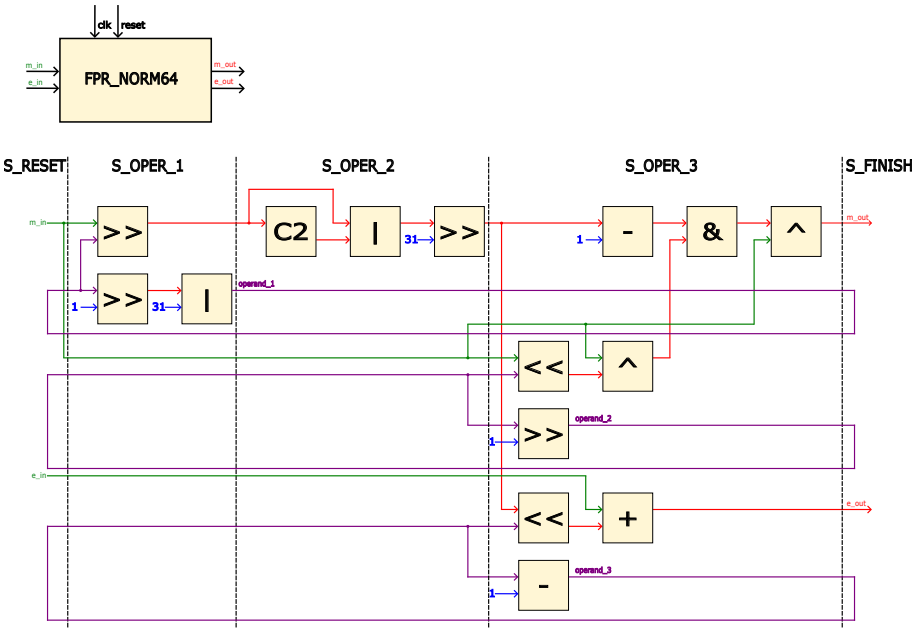

Figura 4.2: Diagrama del módulo *FPR_NORM64*.

Estos módulos quedan agrupados en un *wrapper* siguiendo el diseño modular explicado en la sección 3. No hay agrupaciones intermedias ya que todos los módulos son accesibles por parte de cualquier módulo del diseño, sin embargo se genera una estructura de tipo árbol con varios niveles según si los módulos son maestros, esclavos o ambos a la vez como en la imagen 3.1.

En la parte restante de la sección 4 se presentarán todas las partes que forman parte de la comunicación entre el *software* y el acelerador *hardware*, desde el *wrapper* de los módulos hasta el propio procesador MicroBlaze dentro de la FPGA.

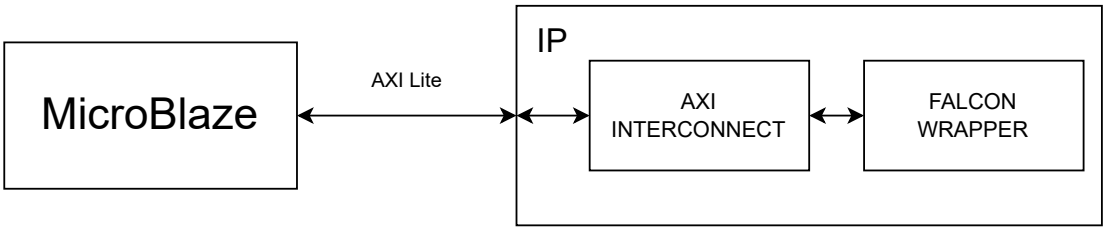

Figura 4.3: Diagrama de la implementación *hardware*.

4.1. Módulo superior de la IP

Todos los módulos del acelerador están recogidos en un *wrapper* que controla el flujo de los tres modos de funcionamiento del acelerador: la generación de claves, la firma y la verificación. En este *wrapper* están instanciados todos los módulos de la implementación, así como sus conexiones y la lógica necesaria para el correcto funcionamiento. Como se ha comentado anteriormente en la sección 3, se instanciarán los multiplexores necesarios para controlar el flujo de datos. Estos multiplexores se utilizarán para las señales de entrada de los módulos esclavos que tengan mas de un maestro, controlados por la señal de habilitación. También se utilizarán para entradas en los módulos maestros que tengan más de un módulo esclavo que devuelva el mismo tipo de dato, controlados por la señal de finalización.

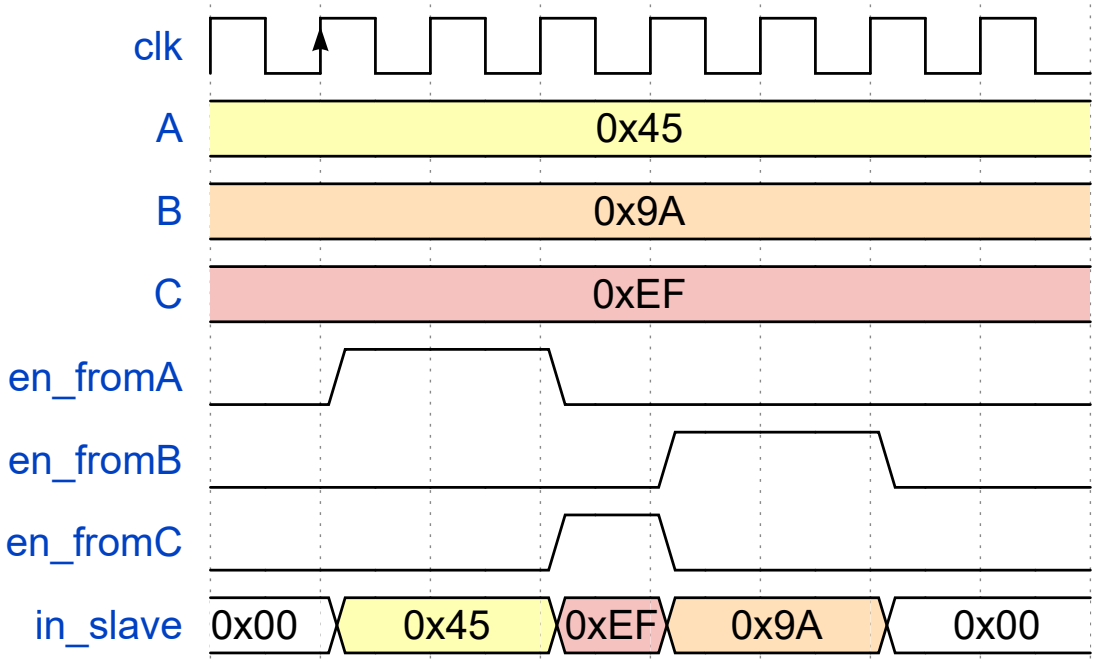

Figura 4.4: Flujo combinacional de un multiplexor 3 a 1.

Este *wrapper* es instanciado en un componente *Falcon_IP* por encima que se encarga de hacer de puente intermedio entre *hardware* y *software*. Este componente procesa las entradas provenientes del MicroBlaze y se encarga de las siguientes funciones:

1. Cargar en una memoria RAM con valores pseudoaleatorios provenientes de una semilla para la generación del contexto *SHAKE256*.
2. Recuperar valores de un contexto *SHAKE256* para completar la codificación de la firma digital.
3. Cargar en una memoria RAM el mensaje sobre el que se genera la firma o cargar el mensaje para poder realizar la verificación de firma.
4. Transferir datos entre el software y la MMU, como la clave pública, la clave privada o la firma digital.
5. Ejecutar la generación de claves, la generación de firma y la verificación de firma.

Los estados en los que se puede encontrar este componente para hacer las funciones anteriores son los siguientes:

- *S_RESET* : estado de espera del acelerador, según lo solicite el procesador pasa a uno de los estados posteriores.
- *S_READ* : estado bloqueante para leer datos de la MMU como claves o firmas. El estado cambiará a *S_RESET* cuando la bandera *CPU_finished_MMU* sea igual a 1.
- *S_WRITE* : estado bloqueante para cargar datos en la MMU como claves o firmas. El estado cambiará a *S_RESET* cuando la bandera *CPU_finished_MMU* sea igual a 1.

- *S_RAM* : en este estado se cargan datos en las RAM, como semillas pseudoaleatorias o mensajes para firmar. El estado cambiará a *S_RESET* en el próximo ciclo de reloj.
- *S_RNG* : en este estado se proporciona al acelerador la dirección de memoria que se necesita leer de la memoria RAM en la que se encuentra el contexto *SHAKE256*. El próximo ciclo de reloj el estado será *S_WAIT*.
- *S_WAIT* : en este estado estado se recibe del acelerador el dato solicitado en *S_RNG*. El estado cambiará a *S_RESET* en el próximo ciclo de reloj.
- *S_ACT* : en este estado se espera a que el acelerador termine de ejecutar una de las tres partes del esquema de firma digital. El estado cambiará a *S_RESET* cuando la bandera *finished_Falcon* sea igual a 1.

4.2. Interfaz *hardware/software*

El intercambio de datos entre *hardware* y *software* se hace a través de entradas y salidas del *Falcon_IP*. Estas son señales de control son las siguientes:

- *clk* : señal de reloj de la FPGA Nexys Video, será la misma tanto para el acelerador como para el procesador.
- *cpu_resetn* : reset físico de la FPGA Nexys Video, el acelerador se reiniciará cuando se pulse el botón correspondiente. Esta señal es activa paso bajo
- *sw_reset* : reset proveniente desde el código *software* ejecutado en el procesador. Esta señal es activa paso alto.
- *IP_start* : habilita al acelerador a empezar la ejecución de una de las tres partes de la firma digital. Esta señal sólo durará un ciclo de reloj y el acelerador pasará al estado *S_ACT*.
- *IP_ready* : el acelerador se encuentra en estado de espera tras finalizar la ejecución. Mientras no está en ejecución esta señal se mantendrá a 1.
- *CPU_read* : solicita a la MMU una lectura de datos. Esta señal sólo durará un ciclo de reloj y el acelerador pasará al estado *S_READ*.
- *CPU_write* : solicita a la MMU una carga de datos. Esta señal sólo durará un ciclo de reloj y el acelerador pasará al estado *S_WRITE*.
- *CPU_ready* : el acelerador se encuentra en estado de espera tras finalizar una carga o lectura de datos desde la MMU. Mientras no está en ejecución esta señal se mantendrá a 1.
- *CPU_read_RNG* : solicita a la memoria RAM que contiene el contexto SHAKE256 una lectura de datos. Esta señal sólo durará un ciclo de reloj y el acelerador pasará al estado *S_RNG*.
- *CPU_write_RAM* : solicita a una una de las memorias RAM que contienen la semilla o el mensaje una carga de datos. Esta señal sólo durará un ciclo de reloj y el acelerador pasará al estado *S_RAM*.
- *CPU_ready_RAM* : el acelerador se encuentra en estado de espera tras finalizar una carga de datos en una de las memorias RAM. Mientras no está en ejecución esta señal se mantendrá a 1.

- *CPU_data_in_a* : datos de entrada provenientes del *software* para escribir en la MMU o las memorias RAM. Estos datos se dirigen al puerto A.
- *CPU_addr_in_a* : direcciones de memoria provenientes del *software* para leer o escribir en la MMU o las memorias RAM. Estas direcciones de memoria se dirigen al puerto A.
- *CPU_data_out_a* : datos de salida provenientes de la MMU o de las memorias RAM. Estos datos provienen del puerto A.
- *CPU_data_in_b* : datos de entrada provenientes del *software* para escribir en la MMU o las memorias RAM. Estos datos se dirigen al puerto B.
- *CPU_addr_in_b* : direcciones de memoria provenientes del *software* para leer o escribir en la MMU o las memorias RAM. Estas direcciones de memoria se dirigen al puerto B.
- *CPU_data_out_b* : datos de salida provenientes de la MMU o de las memorias RAM. Estos datos provienen del puerto B.
- *ram_select* : permite elegir entre escribir en la memoria RAM que almacena la semilla o la memoria RAM que almacena el mensaje. Se utiliza el valor *"01"* para la semilla y el valor *"10"* para el mensaje.
- *acc_select* : permite elegir entre las tres partes de la firma digital. El valor "001" indica generación de claves, el valor "010" indica generación de firma y el valor "100" indica verificación de firma.
- *error_bit* : señal de salida que permite comprobar si la verificación ha finalizado con éxito o por el contrario el mensaje ha sido rechazado. El valor 1 indica rechazo del mensaje.
- *logn_in* : indica el grado del polinomio que se utilizará en el algoritmo. Este valor debe estar en el rango (1,10) sin embargo sólo los valores 9 y 10 ofrecen seguridad para criptografía pos-cuántica, los inferiores sirven para estudio académico.

Las señales de intercambio de datos con las memorias RAM y con la MMU se han unificado con el objetivo de usar un menor número de registros. El tamaño de los datos y las direcciones de memoria son tratadas por la IP para asegurar que no existen errores en síntesis.

Las transferencias de datos con el procesador se realizarán a través de una interfaz AXI Lite. Por ello se ha implementado en la IP un componente AXI Interconnect que permite mapear las entradas y salidas descritas anteriormente en registros de 32 *bits*. En la imagen 4.5 se puede observar como se han distribuido las señales en los registros, empezando por el *slv_reg0* hasta el *slv_reg10*.

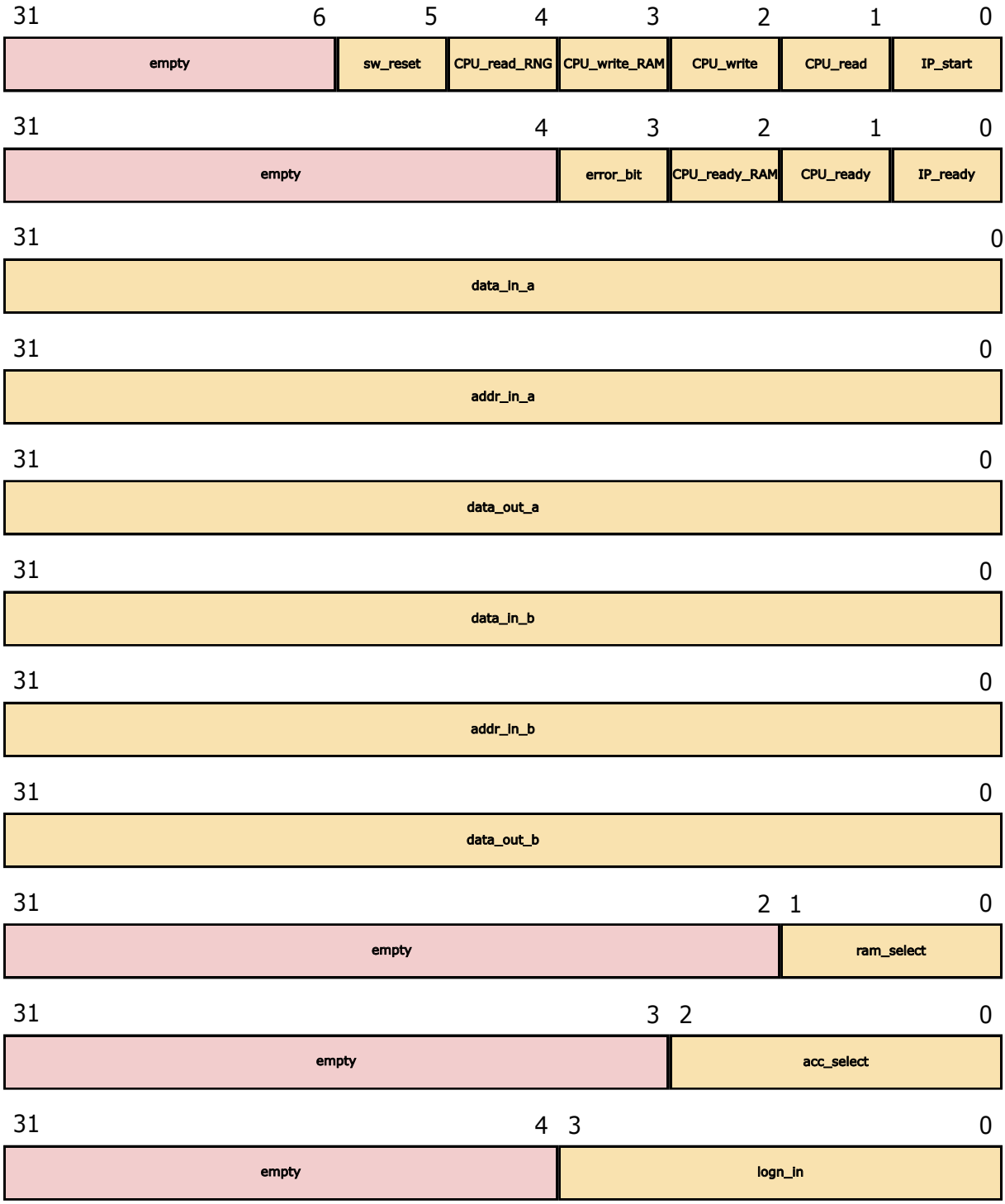

Figura 4.5: Mapa de registros en el componente AXI Interconnect.

4.3. Diseño de bloques en Vivado

La estructura ha sido diseñada en el entorno de desarrollador Vivado, de *Advanced Micro Devices* (AMD). Este programa permite crear una IP personalizada con una interfaz AXI Lite en la que instanciar los módulos descritos en lenguaje VHDL. Con esta IP personalizada, se puede hacer un diagrama de bloques en el que conectar el procesador y los bloques auxiliares. Este diagrama necesitará los siguientes bloques:

- *Custom IP* : IP personalizada del acelerador, su funcionamiento está descrito previamente en esta sección 4.
- *MicroBlaze* : la placa Nexys Video no dispone de microprocesador, es por eso que se utiliza este bloque para incluir uno en la parte programable de la FPGA. Se le aplicará una configuración de microprocesador real con la finalidad de poder medir ciclos de reloj y comunicarse con el ordenador principal via UART.
- *AXI UART Lite* : este bloque habilita la comunicación via UART por un puerto microUSB entre el MicroBlaze y el ordenador principal. Esta comunicación permitiría también la trasferencia de datos entre la FPGA y el sistema que la utilice como periférico.
- *AXI Timer* : este bloque permite medir el número de ciclos que tarda la implementación en realizar las pruebas. Se debe habilitar el modo de contador de 64 *bits* para evitar *overflow* en algunas pruebas.
- *Clocking Wizard* : este bloque permite al desarrollador elegir la frecuencia a la que opera el diseño.

El diagrama de bloques resultante se puede ver en la imagen 4.6.

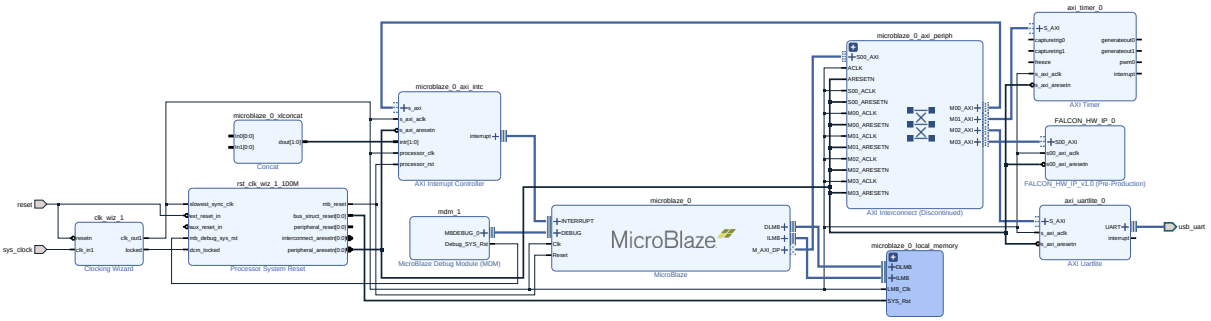

Figura 4.6: Diagrama de bloques completo de la implementación.

Los bloques restantes *Processor System Reset*, *Concat*, *AXI Interrupt Controller*, *MicroBlaze Debug Module*, *AXI Interconnet* y *Local Memory* que aparecen en el diseño son generados automáticamente tras ejecutar los comandos *Run Block Automation* y *Run Connection Automation*. Son necesarios para producir un equivalente a un sistema de procesamiento (PS). El tamaño de la memoria local será modificado manualmente para que pueda albergar el código *software* de la referencia al completo.

Este diagrama de bloques pasará por un proceso de síntesis e implementación. En este proceso hay que tener especial cuidado con no superar la cantidad de recursos disponibles en la FPGA y con no generar un retardo lógico que produzca un *Negative Slack* (NS) que reduzca la frecuencia

de operación. Tras esto se genera un *bitstream* que contiene toda la información del diseño y será exportado como un *Xilinx Support Archive* (.xsa) para crear una plataforma en el entorno de desarrollador Vitis.

5. SIMULACIÓN, IMPLEMENTACIÓN Y RESULTADOS

En esta sección se explicará el procedimiento que se ha seguido para validar la implementación desde los módulos más básicos, pasando por una simulación del acelerador al completo y finalmente una implementación real en la Nexys Video. Los programas que se han utilizado para este proceso son los siguientes:

- *Vivado* : los módulos *hardware* son desarrollados en este programa utilizando lenguaje de programación VHDL, a su vez se pueden hacer bancos de prueba con los que simular su comportamiento. Con este programa, como se ha explicado en la sección 4, se obtienen todos los archivos necesarios para la implementación en la FPGA.
- *Visual Studio Code* : el código de referencia *software* es ejecutado en este programa para comprobar los resultados esperados con los vectores de prueba. También permite obtener las parejas de datos semilla-mensaje de forma aleatoria dependiendo del sistema operativo (OS) con los que tomar valores de tiempo de ejecución.
- *Vitis* : las pruebas tras implementación en la FPGA se hacen en este programa. Vitis generará unas librerías para gestionar los registros de la IP personalizada, así como controlar el *AXI Timer*.

5.1. Simulación del hardware en Vivado

La simulación de los módulos es un proceso que se debe realizar en paralelo entre Visual Studio Code y Vivado. En el primero se escogerá la función que se desea validar, asignándole unas entradas y se recogerá la salida. Este es el resultado que se debe conseguir en el módulo *hardware* para que tengan un funcionamiento equivalente. En Vivado se escribirá un archivo del tipo simulación, un banco de prueba que permita asignar las mismas entradas y comprobar que la salida es la misma. En la simulación se pueden comprobar cuatro cosas fundamentalmente:

- El resultado que proporciona el módulo *hardware* es el mismo que se espera de la función de referencia.
- El módulo *hardware* sigue un flujo de estados correcto.
- En el caso de que no siga un flujo de estados correcto, analizar qué estados está saltando o si está bloqueado en un bucle infinito.
- En el caso de que el resultado sea erróneo y el módulo *hardware* siga un flujo de estados correcto, analizar las señales internas para comprobar qué operación da un resultado incorrecto.

El proceso sigue los pasos que se muestran en la imagen 5.1.

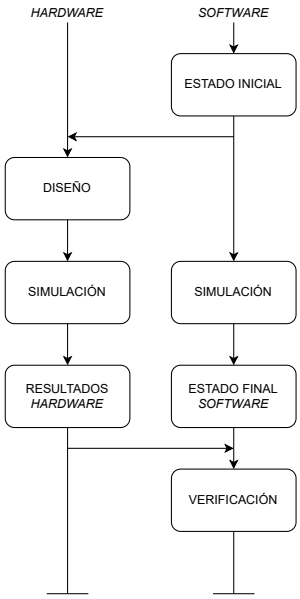

Figura 5.1: Diagrama de flujo de la simulación.

Adicionalmente en la imagen 5.2 se puede ver la interfaz de Vivado al realizar una simulación del módulo *hash_to_point_vartime* en el que se muestra cómo cambian los estados según los ciclos de reloj, cómo las señales internas van actualizándose cuando se ejecuta una operación y cómo se queda el estado bloqueado cuando está ejecutándose un módulo esclavo, en este caso *inner_shake256_extract*.

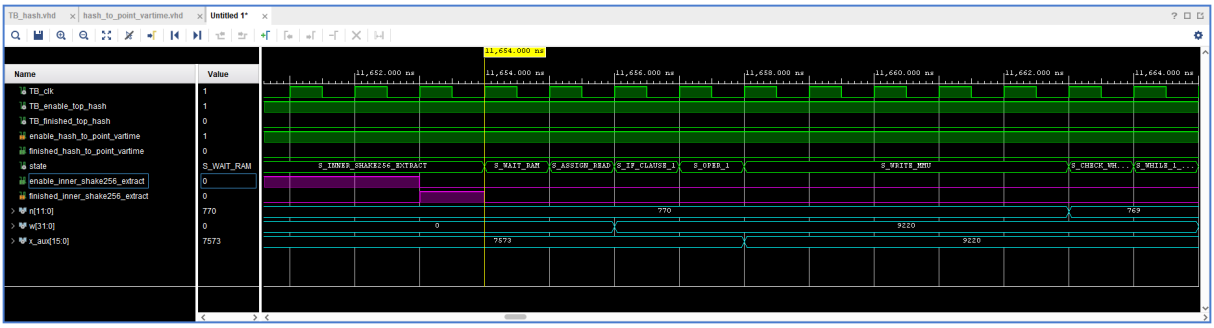

Figura 5.2: Interfaz de Vivado para simulación.

El mismo proceso se aplica para simular el comportamiento de las diferentes partes del algoritmo, se instancian en un *wrapper* los módulos necesarios y se genera un archivo de simulación de tipo banco de pruebas. En algunas ocasiones, el resultado esperado no es una señal sino los valores en una sección de memoria. Para comprobar estos valores existen dos opciones:

- Utilizar la librería *std.textio* para escribir en un archivo externo los valores de la sección de memoria correspondiente.
- Utilizar la *Tcl console* habiendo seleccionado la memoria RAM de la MMU como se muestra

en la imagen 5.3 utilizando el comando `report_values -radix unsigned mem[init:end]` con el rango de memoria que se necesite leer.

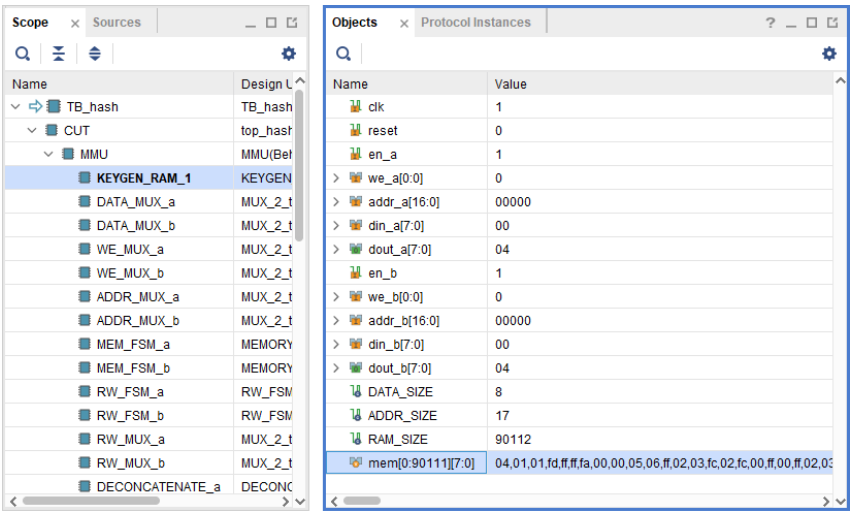

Figura 5.3: Lectura de memoria por línea de comandos.

La primera opción será seleccionada cuando se necesiten comprobar los valores de una sección de gran tamaño de la memoria ya que por *Tcl console* el rango máximo de valores por cada comando es 512. Una vez se tengan los valores, se compararán en Visual Studio Code para ver si son equivalentes o si ha habido algún fallo.

5.2. Implementación en la FPGA Nexys Video

Una vez se ha comprobado el correcto funcionamiento tanto en simulación de conducta como en simulación después de síntesis y la implementación es compatible con la Nexys Video, se generará el *bitstream*. Este archivo contiene toda la información sobre la implementación en la zona programable de la FPGA y, como ya se ha comentado en la sección 4 generar un archivo XSA.

En el entorno de desarrollador Vitis se creará una plataforma a raíz del *hardware* exportado en el archivo XSA. En esta plataforma se ha creado una librería para poder comunicarse con el acelerador a través de los registros, tanto funciones como direcciones de memoria. También se debe crear una aplicación en la que incluir el proyecto de Falcon, sustituyendo las funciones principales por llamadas al acelerador así como carga y lectura de datos.

La medida de ciclos de reloj se realiza con el *AXI Timer* en modo consulta, es decir, se reinicia el contador cada vez que se desea medir y cuando haya finalizado la rutina que se desea medir, se consulta el valor del contador. Para evitar problemas con el *overflow* al llegar al valor máximo del contador, se configura para que opere en modo cascada, con dos contadores de 32 *bits* en serie.

```
u64 ReadTimer64()
{
    u32 higher_previous, higher_current, lower;
    u64 total;

    do {
        higher_previous = XTmrCtr_GetValue(&TimerInst, TIMER_1);
        lower  = XTmrCtr_GetValue(&TimerInst, TIMER_0);
        higher_current = XTmrCtr_GetValue(&TimerInst, TIMER_1);
    }while(higher_previous != higher_current);

    total = (((u64)higher_previous) << 32) | (u64)lower;

    return total;
}

int main(){
    init_platform();
    int Status;

    Status = XTmrCtr_Initialize(&TimerInst, XPAR_XTMRCTR_0_BASEADDR);

    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    XTmrCtr_SetOptions(&TimerInst, TIMER_0, XTC_CASCADE_MODE_OPTION |
    XTC_AUTO_RELOAD_OPTION);
    XTmrCtr_SetResetValue(&TimerInst, TIMER_0, RESET_VALUE);
    XTmrCtr_SetResetValue(&TimerInst, TIMER_1, RESET_VALUE);

    XTmrCtr_Reset(&TimerInst, TIMER_0);
    XTmrCtr_Reset(&TimerInst, TIMER_1);
    XTmrCtr_Start(&TimerInst, TIMER_0);

    /* Measured routine */

    XTmrCtr_Stop(&TimerInst, TIMER_0);
    u64 cycles = ReadTimer64();

    cleanup_platform();
    return 0;
}
```

Código 5.1: Código ejemplo del uso del AXI Timer.

La carga y lectura de datos entre el procesador y el acelerador se ejecuta con bucles del tamaño necesario para la sección de memoria sobre la que se quiere operar. En el caso de las RAMs se envía un dato por puerto en cada iteración, sin embargo cuando se opera con la MMU se aprovecha al máximo el tamaño de 32 *bits* de los registros.

```
// Write h in MMU (4*n to 6*n memory positions)
for(int xx = VVWRITE_INIT_1; xx < VVWRITE_LIM_1; xx++){
    // Write CPU_data_in_a register with index value (Register 2)
    IP_mWriteReg(XPAR_IP_0_BASEADDR, IP_S00_AXI_SLV_REG2_OFFSET, (aux[2 * xx]) &
     REG2_MMU_MASK);
    // Write CPU_data_in_b register with index value (Register 5)
    IP_mWriteReg(XPAR_IP_0_BASEADDR, IP_S00_AXI_SLV_REG5_OFFSET, (aux[2 * xx + 1
    ]) & REG5_MMU_MASK);
    // Write CPU_addr_in_a register with index value (Register 3)
    IP_mWriteReg(XPAR_IP_0_BASEADDR, IP_S00_AXI_SLV_REG3_OFFSET, (8 * xx) & REG3
    _MMU_MASK);
    // Write CPU_addr_in_b register with index value (Register 6)
    IP_mWriteReg(XPAR_IP_0_BASEADDR, IP_S00_AXI_SLV_REG6_OFFSET, (8 * xx + 4) &
    REG6_MMU_MASK);

    // Set CPU_write bit to 1 (Register 0, bit 2)
    IP_mWriteReg(XPAR_IP_0_BASEADDR, IP_S00_AXI_SLV_REG0_OFFSET, (SET(
    CPU_WRITE_BIT)) & REG0_MASK);

    // Read CPU_ready bit until it gets to 1 (Register 1, bit 1)
    while((IP_mReadReg(XPAR_IP_0_BASEADDR, IP_S00_AXI_SLV_REG1_OFFSET) & REG1
    _CPU_READY_MASK) == 0);
}
```

Código 5.2: Código ejemplo de carga de datos en la MMU.

La configuración del acelerador se debe hacer previo a la ejecución asignando el grado del polinomio y la parte que se va a ejecutar. El procesador quedará bloqueado en una sentencia *while* comprobando que la bandera de finalización vuelva a quedar activa.

```
// Select Key Generation as operating mode (Register 9, bit 0)
IP_mWriteReg(XPAR_IP_0_BASEADDR, IP_S00_AXI_SLV_REG9_OFFSET, (SET(KEYGEN_BIT)) &
     REG9_MASK);

// Write polynomial degree (Register 10)
IP_mWriteReg(XPAR_IP_0_BASEADDR, IP_S00_AXI_SLV_REG10_OFFSET, (logn) & REG10
    _MASK);

// Set IP_start bit to 1 (Register 0, bit 0)
IP_mWriteReg(XPAR_IP_0_BASEADDR, IP_S00_AXI_SLV_REG0_OFFSET, (SET(IP_START_BIT))
     & REG0_MASK);

// Read IP_ready bit until it gets to 1 (Register 1, bit 0)
while((IP_mReadReg(XPAR_IP_0_BASEADDR, IP_S00_AXI_SLV_REG1_OFFSET) & REG1
    _IP_READY_MASK) == 0);
```

Código 5.3: Código ejemplo de ejecución de la generación de claves.

Por último, como el algoritmo depende fuertemente de las semillas que se utilizan, se debe realizar un amplio banco de pruebas con mil parejas de semilla-mensaje. Estas parejas de datos se han generado de manera aleatoria en Visual Studio Code con funciones de la librería *bcrypt.h*, la cual ofrece mayor seguridad en la aleatoriedad para entornos de criptografía.

```
char* generate_random_string(int len) {
    if (len <= 0) {
        return NULL;
    }

    char* str = malloc(len + 1);
    if (str == NULL) {
        return NULL;
    }

    const char char_set[] =
    ''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'';
    const int char_set_size = (int)strlen(char_set);  // 62

    const unsigned char threshold = 248; // 62 * 4

    for (int i = 0; i < len; i++) {
        unsigned char byte;
        NTSTATUS status;
        do {
            status = BCryptGenRandom(
                NULL,                    // System default suplier
                &byte,                   // One byte buffer
                1,                       // Size
                BCRYPT_USE_SYSTEM_PREFERRED_RNG
            );
            if (!BCRYPT_SUCCESS(status)) {
                free(str);
                return NULL;
            }
            if (byte < threshold) {
                break;  // Byte accepted
            }
            // Byte rejected
        } while (1);

        int index = byte % char_set_size;
        str[i] = char_set[index];
    }

    str[len] = '\0';
    return string;
}
```

Código 5.4: Código de generación de semillas aleatorias.

5.3. Utilización de recursos del acelerador

Uno de los datos más relevantes en los diseños *hardware* es la utilización de recursos. Un balance óptimo entre rendimiento y utilización de recursos es muy importante, el desarrollador debe diseñar el acelerador siendo consciente del entorno en el que se va a implementar el proyecto y la plataforma de la que dispone. Al encontrarse este trabajo enmarcado en un entorno de recursos limitados, se utiliza la FPGA Nexys Video. Tras los procesos de síntesis e implementación se pueden obtener la utilización de recursos del diseño, en este caso se tienen dos ramas como ya se explicó en la sección 3, una utilizando el algoritmo de Karatsuba en multiplicaciones de 64 *bits* y otra utilizando el algoritmo también en multiplicaciones de 32 *bits* con caminos críticos altos. En las siguientes imágenes se puede ver el diseño que utiliza el algoritmo en ambas multiplicaciones implementado en la FPGA, marcando en magenta el *AXI Timer*, en amarillo el *AXI UART Lite*,

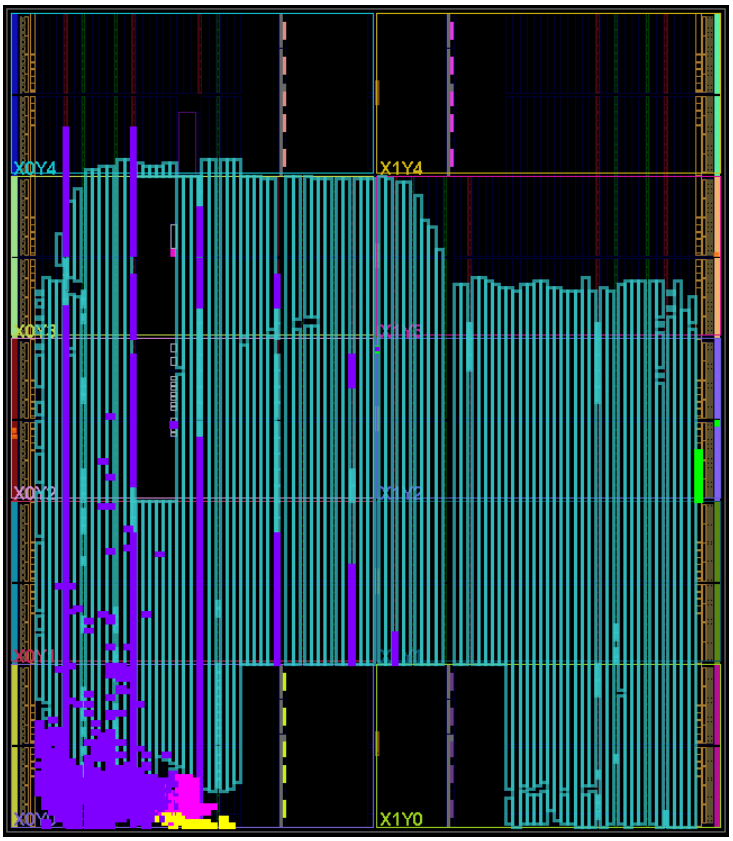

Figura 5.4: Implementación del acelerador.

en verde el *Clocking Wizard*, en morado la implementación de todos los módulos relacionados con el *MicroBlaze* y el restante de color celeste para el acelerador.

Se puede ver cómo ocupa una gran parte de la FPGA, pero todavía queda margen para no estar al límite. Los resultados numéricos de estas dos implementaciones se pueden ver comparados en la siguiente tabla. Se ha añadido a la comparación el único diseño *hardware* de Falcon al completo en la literatura, implementado desde HLS [31].

Diseño	LUTs	FFs	BRAMs	DSPs	Utilización
IP 32b	85750	41834	44	65	1.01×/1.01×/1.0×/0.46×
IP 64b	85261	41382	44	142	1.0×/1.0×/1.0×/1.0×
FalconTakesOff [31]	159174	141018	120	1412	1.87×/3.41×/2.73×/9.94×

Tabla 5.1: Comparativa de utilización de recursos.

Ambas implementaciones diseñadas en este trabajo utilizan aproximadamente la misma cantidad de recursos, es decir, incluir el algoritmo de Karatsuba en multiplicaciones de 32 *bits* no supone un porcentaje sustancial de aumento de recursos, de hecho al dividir en multiplicaciones menores se reduce considerablemente el uso de DSPs.

En comparación con el diseño en [31], las implementaciones en este trabajo consumen muchos menos recursos. En el propio artículo, los autores admiten que el no tener un control total sobre las descripciones HDL hace que se generen bloques redundantes y paralelizaciones que no está claro si son realmente necesarias. Esta diferencia de recursos se verá reflejada en los tiempos de ejecución.

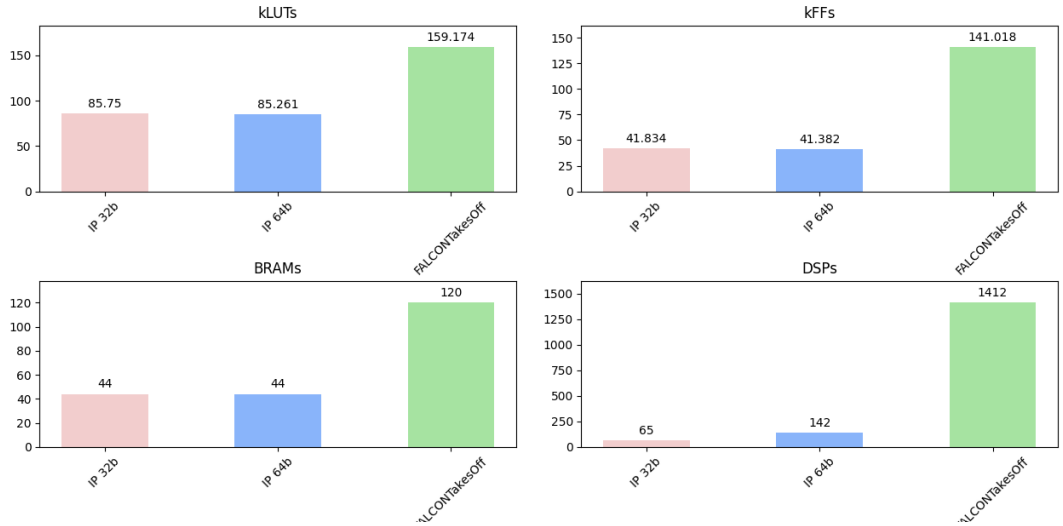

Figura 5.5: Gráfica comparativa utilización de recursos.

5.4. Comparación de resultados con el estado del arte

Las pruebas realizadas han consistido en 1000 parejas semilla-mensaje generadas de manera aleatoria ejecutadas de forma secuencial en la implementación utilizando la placa Nexys Video (*xc7a200tsbg484-1*) con un MicroBlaze como procesador en la zona programable de la FPGA y dos versiones del acelerador que se diferencian en emplear el algoritmo de Karatsuba en multiplicaciones de 64 *bits* o adicionalmente en multiplicaciones de 32 *bits*. Estas pruebas se han realizado bajo la premisa de que el código de referencia sigue la rama definida por la directiva *Falcon_FPEMU* pensada para procesadores que no disponen de una FPU. Se compararán en primera instancia los resultados obtenidos por el MicroBlaze sin utilizar el acelerador ejecutando el código de referencia con el procesador utilizando el acelerador a través de los registros explicados en la sección 4. Se han realizado pruebas adicionalmente sobre un MicroBlaze con FPU y una Pynq-Z2 (ARM-Cortex A9) siguiendo la rama definida por la directiva *Falcon_FPNATIVE* que permite el uso de tipo coma flotante en el código *software*.

En la tabla 5.2, en la tabla 5.3 y en la tabla 5.4 se puede observar comparación en cuanto al número de ciclos y la velocidad de ejecución en los procesos de generación de claves, generación de firma y verificación de firma respectivamente. En la gráfica 5.6 se pueden observar rangos de error ya que ciertas partes del algoritmo son dependientes de la semilla que tienen como entrada, como la generación de claves o las funciones *hash*.

Diseño	*Frecuencia (MHz)*	CCs $\times 10^6$	Latencia (ms)	Rendimiento
ARM Cortex-A9 (emu)	650	378,9	582,92	1.01×
ARM Cortex-A9 (nat)	650	197,27	303,49	1.95×
MicroBlaze (emu)	80	4890,91	61136,37	0.08×
MicroBlaze (nat)	100	3513,81	35138,15	0.11×
IP 32b	80	442,09	5526,1	0.87×
IP 32b (w/o load)	80	435,9	5448,75	0.88×
IP 64b	75	390,72	5209,54	0.98×
IP 64b (w/o load)	75	384,53	5127,07	1.0×

Tabla 5.2: Comparativa de rendimiento en generación de claves.

Diseño	*Frecuencia (MHz)*	CCs $\times 10^6$	Latencia (ms)	Rendimiento
ARM Cortex-A9 (emu)	650	79,2	121,85	0.4×
ARM Cortex-A9 (nat)	650	8,82	13,58	3.59×
MicroBlaze (emu)	80	1014,75	12684,44	0.03×
MicroBlaze (nat)	100	520,36	5203,62	0.06×
IP 32b	80	50,04	625,5	0.63×
IP 32b (w/o load)	80	42,15	526,9	0.75×
IP 64b	75	39,54	527,22	0.8×
IP 64b (w/o load)	75	31,65	422	1.0×

Tabla 5.3: Comparativa de rendimiento en generación de firma.

Diseño	*Frecuencia (MHz)*	CCs $\times 10^3$	Latencia (ms)	Rendimiento
ARM Cortex-A9 (emu)	650	636,49	0,98	0.9×
ARM Cortex-A9 (nat)	650	636,74	0,98	0.9×
MicroBlaze (emu)	80	6252,92	78,16	0.09×
MicroBlaze (nat)	100	6252,92	62,53	0.09×
IP 32b	80	947,19	11,8	0.61×
IP 32b (w/o load)	80	576,01	7,2	1.0×
IP 64b	75	947,19	12,62	0.61×
IP 64b (w/o load)	75	576,01	7,68	1.0×

Tabla 5.4: Comparativa de rendimiento en verificación de firma.

Se puede observar cómo entre las implementaciones del acelerador, las que utilizan el algoritmo de Karatusba en multiplicaciones de 32 *bits* aumentan de manera considerable el número de ciclos, al ser cifras del orden de millones. Es por esto que a pesar de operar a una frecuencia de reloj mayor, acaba ejecutándose más lento que el acelerador con Karatsuba aplicado únicamente a multiplicaciones de 64 *bits*. Se ha diferenciado también entre el número de ciclos que tarda el acelerador en ejecutar la operación en sí y el número de ciclos que tarda en hacer la operación junto con la transferencia de datos desde el procesador al acelerador. Esta distinción es debido a que la transferencia de datos depende en parte del procesador que se está utilizando y cómo maneja este las funciones de lectura y carga en los registros.

La vertiente del acelerador que se va a comparar y que es la base en el aparatado de rendimiento será la que implementa Karatsuba únicamente en multiplicaciones de 64 *bits* y tiene una frecuencia de 75 MHz. Esta implementación consigue reducir el número de ciclos empleados por el MicroBlaze en ejecutar el código de referencia tanto utilizando la FPU nativa en 89 %/94 %/91 % como utilizando la versión de coma flotante emulada en 92 %/97 %/91 %, en generación de claves, generación de firma y verificación de firma respectivamente.

El número de ciclos que necesita el ARM Cortex-A9 de la Pynq-Z2 utilizando la coma flotante emulada en la generación de claves es ligeramente menor que el que necesita el acelerador del orden del 1,5 %, mientras que el acelerador reduce en un 60 % en la generación de firma y en un 10 % en la verificación de firma. Sin embargo, al ser un procesador con una frecuencia de 650 MHz, el velocidad de ejecución es menor que en el acelerador. Al probar sobre este procesador pero esta vez utilizando la coma flotante nativa, el número de ciclos disminuye tanto en generación de claves como en generación de firmas, además de seguir ejecutándose a una mayor frecuencia por lo que el uso de una FPU nativa obtiene mejores resultados que el acelerador. La verificación no se ve afectada ya que no utiliza aritmética de coma flotante por lo que los resultados son

similares a los de coma flotante emulada vistos anteriormente.

La mayor parte del estado del arte se trata de implementaciones parciales *hardware/software* que no cubren una parte suficiente del algoritmo como para ser comparadas con esta implementación. Es por ello que se realizará la comparativa del estado del arte con la implementación en [31] completa diseñada desde HLS al igual que se ha hecho con la utilización de recursos y con implementaciones en *software* realizadas por los autores en [19][42][43]. De nuevo como en la comparación anterior, en la tabla 5.5, en la tabla 5.6 y en la tabla 5.7 se tiene la comparación en cuanto al número de ciclos y la velocidad de ejecución en los procesos de generación de claves, generación de firma y verificación de firma respectivamente. En la gráfica 5.6 se han incluido únicamente los márgenes de error que se proporcionaban en el estado del arte, no se han realizado pruebas adicionales con las plataformas objetivo.

Diseño	*Frecuencia (MHz)*	CCs $\times 10^6$	Latencia (ms)	Rendimiento
Intel i5-8259U [19]	2,3K	63,12	27,45	6.09×
Intel i7-6567U [42]	3,6K	78,98	21,94	4.87×
Intel i7-6567U (emu) [42]	3,6K	175,93	48,87	2.19×
ARM Cortex-M4 [42]	168	514,31	3061,38	0.75×
ARM Cortex-M4 [43]	24	284,03	11834,65	1.35×
FalconTakesOff [31]	100	32,03	320,3	12.01×
IP 64b (w/o load)	75	384,53	5127,07	1.0×

Tabla 5.5: Comparativa de rendimiento en generación de claves con el estado del arte.

Diseño	*Frecuencia (MHz)*	CCs $\times 10^6$	Latencia (ms)	Rendimiento
Intel i5-8259U [19]	2,3K	0,782	0,34	40.47×
Intel i7-6567U [42]	3,6K	1,78	0,495	17.78×
Intel i7-6567U (emu) [42]	3,6K	39,94	11,09	0.79×
ARM Cortex-M4 [42]	168	90,37	537,89	0.35×
ARM Cortex-M4 [43]	24	47,78	1990,75	0.66×
FalconTakesOff [31]	214,3	1,91	8,7	16.57×
IP 64b (w/o load)	75	31,65	422	1.0×

Tabla 5.6: Comparativa de rendimiento en generación de firma con el estado del arte.

Diseño	*Frecuencia (MHz)*	CCs $\times 10^3$	Latencia (ms)	Rendimiento
Intel i5-8259U [19]	2,3K	167,9	0,073	3.43×
Intel i7-6567U [42]	3,6K	154,18	0,043	3.74×
Intel i7-6567U (emu) [42]	3,6K	200,74	0,055	2.87×
ARM Cortex-M4 [42]	168	1039,17	6,19	0.55×
ARM Cortex-M4 [43]	24	518,59	30,17	1.11×
FalconTakesOff [31]	187,5	269,61	1,3	2.14×
IP 64b (w/o load)	75	576,01	7,68	1.0×

Tabla 5.7: Comparativa de rendimiento en verificación de firma con el estado del arte.

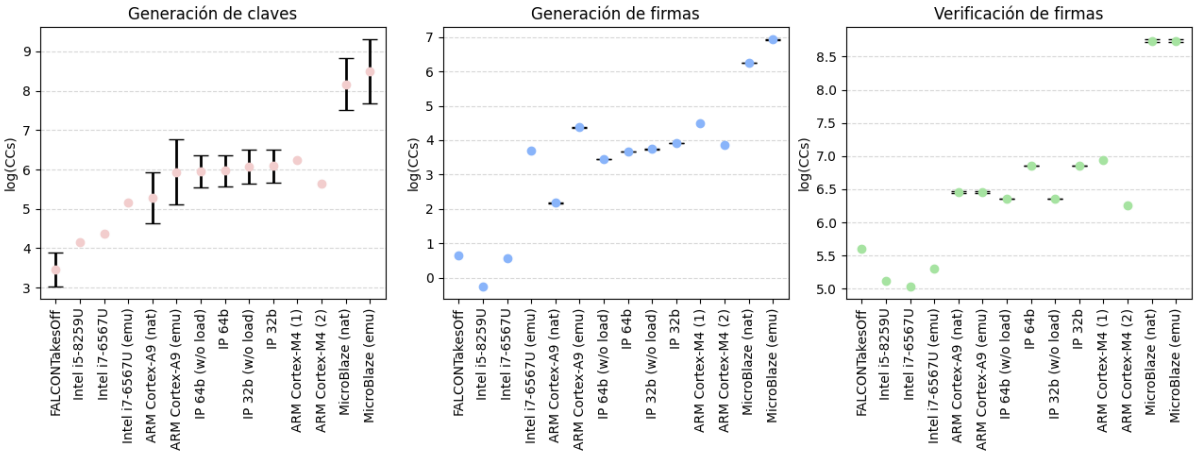

Figura 5.6: Gráficas de valor medio de ejecución de ciclos de reloj con rango de error.

Las implementaciones que se muestran en estas tablas no siguen las mismas directivas, ya que al igual que en las pruebas anteriormente citadas usando el modo nativo o emulado de la coma flotante los autores en [42][43] hacen optimizaciones específicas para el procesador en cuestión. En los procesadores ARM Cortex-M4 tienen optimizaciones modificando el código ensamblador en las operaciones de coma flotante emulado. En el caso de los procesadores Intel, se aplican directivas de AVX2 para optimizar las operaciones con vectores con el uso de instrucciones *Single Instruction Multiple Data* (SIMD) y directivas de FMA para optimizar el uso conjunto de multiplicaciones y sumas.

Los procesadores Intel utilizados por los autores consiguen un gran rendimiento en cuanto a número de ciclos gracias a las optimizaciones comentadas anteriormente, lo que unido a la gran frecuencia a la que trabajan del orden de gigahercios, la velocidad de ejecución es incomparable a la del acelerador. Por otra parte, en [42] toman valores utilizando las directivas de coma flotante emulado en las que se aprecia un aumento del número de ciclos consiguiendo el acelerador reducir el número de ciclos un 21 % en la generación de firma.

Se puede observar cómo este trabajo tiene una reducción de ciclos del 25 %/65 %/45 % respecto a la implementación en el ARM Cortex-M4 realizada en [42] aún con la optimización en código ensamblador de la coma flotante emulada. En [43] consiguen reducir el número de ciclos modificando este código ensamblador, sin embargo esta implementación funciona a 24 MHz por lo que en velocidad de ejecución es más lenta que el acelerador pese a reducir sustancialmente la cantidad de ciclos. Este acelerador sigue teniendo una reducción en el número de ciclos de un 34 % en la generación de firma frente a esta implementación.

Por último el único acelerador *hardware* al completo de Falcon que existe en la literatura desarrollado desde HLS, consigue un rendimiento mayor en cuanto al número de ciclos en las tres partes del algoritmo. Este acelerador está diseñado por separado para cada parte del algoritmo de Falcon y, sumando los recursos que necesitan, llega a consumir prácticamente el triple de recursos en la mayoría de métricas y hasta diez veces la cantidad de DSPs. Esta gran cantidad de recursos se traduce en una mayor paralelización y por tanto menor número de ciclos y mayor aceleración. Sin embargo, el acelerador que se propone en este trabajo se ha diseñado para dedicarse a entornos de recursos limitados, en los que sería imposible disponer de la cantidad necesaria para implementar el acelerador diseñado en HLS.

6. CONCLUSIONES Y LÍNEAS FUTURAS DE INVESTIGACIÓN

El estudio y desarrollo de la criptografía postcuántica ha crecido a lo largo de los últimos años ante la próxima llegada de los ordenadores cuánticos. Estos ordenadores serán capaces de utilizar algoritmos de factorización como el de Shor o el de búsqueda en una lista no estructurada como el de Grover para romper los esquemas clásicos de criptografía en poco tiempo. El *National Institute of Standards and Technology* empezó en 2016 un proceso para la selección de algoritmos de criptografía postcuántica para su estandarización. De entre los candidatos, Falcon fue elegido como apto para la estandarización en el campo de esquemas de firma digital.

El objetivo de este trabajo es diseñar el primer acelerador *hardware* de Falcon al completo desarrollado desde cero. Este acelerador debe ser apto para trabajar en entornos de recursos limitados, en concreto se pretende implementar en la placa Nexys Video (*xc7a200tsbg484-1*). Esta placa no dispone de sistema de procesamiento, por lo que se tendrá que utilizar un MicroBlaze que se sintetice en la zona programable de la FPGA. El acelerador deberá ser capaz de ejecutar el algoritmo al completo de Falcon, pudiendo seleccionar la parte que se desea ejecutar entre generación de claves, generación de firma y verificación de firma, así como la transferencia de datos entre el procesador y la unidad de manejo de memoria.

En la sección 5 se han realizado dos comparativas. La primera consta de las pruebas que se han realizado con 1000 parejas semilla-mensaje en la implementación. De esta comparación se ve como la implementación con mayor rendimiento es la que utiliza el algoritmo de Karatsuba únicamente en las operaciones de 64 *bits*. A su vez, el acelerador consigue reducir el número de ciclos en comparación con el código de referencia *software* ejecutado en el MicroBlaze en un 89 %/94 %/91 % utilizando FPU nativa y en un 92 %/97 %/91 % utilizando coma flotante emulado, en generación de claves, generación de firma y verificación de firma. Las pruebas realizadas en ARM Cortex-A9 indican que se puede conseguir un rendimiento mayor con el acelerador cuando el procesador sigue la directiva del coma flotante emulado reduciendo el número de ciclos en un 60 % en generación de firma y en un 10 % en verificación de firma, sin embargo cuando entra la FPU el número de ciclos se reduce drásticamente salvo en la verificación de firma.

En comparación con el estado del arte, las optimizaciones que se realizan al utilizar las directivas de coma flotante nativo, lenguaje ensamblador, AVX2 o FMA, hacen que las implementaciones *software* en procesadores Intel y ARM Cortex-M4 tengan un gran rendimiento. Pero incluso con estas optimizaciones, se llega a alcanzar un mayor rendimiento que con el ARM Cortex-M4 en cuanto a número de ciclos con una reducción del 25 %/65 %/45 %. La implementación realizada al completo en HLS presenta un mayor rendimiento que este trabajo, sin embargo utiliza una cantidad de recursos que no son compatibles con el entorno de recursos limitados que es una de las bases de este trabajo, como el uso diez veces más DSPs.

En este trabajo se ha conseguido una implementación *hardware* de Falcon al completo desde cero para un entorno de bajos recursos. Los resultados dejan ver una mejora respecto a los procesadores más limitados, incluso cuando disponen de una FPU nativa. Estos resultados pueden ser mejorados siguiendo diferentes líneas de investigación:

- Optimizar los módulos diseñados para obtener un mejor balance entre recursos empleados y velocidad de ejecución. Los módulos pueden ser optimizados para reducir el número de ciclos y a su vez tratar de reutilizar recursos para reducir el porcentaje de utilización. Tratar de aprovechar en mayor medida las ventajas de concurrencia que ofrece el lenguaje

hardware y aplicarlas al algoritmo.

- Utilizar otras FPGA con más margen de recursos disponibles. Este cambio permitiría aumentar la paralelización en numerosas partes del algoritmo y así aumentar el rendimiento.
- Buscar otras alternativas al uso de aritmética en coma flotante. Actualmente hay vertientes de esquemas de firma digital en las que se aconseja el uso de coma fija en vez de coma flotante, lo que podría ser un punto de partida interesante para otra implementación o modificación de esta.

Además de mejorar esta implementación para hacerla más competitiva frente a las implementaciones mostradas en el estado del arte, también se pueden seguir las siguientes vertientes:

- Llevar la implementación a un entorno real, en el que dos sistemas mantengan una comunicación segura gracias a la implementación. Dos placas Nexys Video que se encuentren conectadas a dos ordenadores y que estos utilicen la implementación para establecer un envío de mensajes con autentificación.
- Realizar pruebas de consumo y tratar de modificar la implementación para reducir este lo máximo posible. En un entorno de bajos recursos es esencial que el consumo de la implementación sea reducido al mínimo posible.
- Comprobar la seguridad en un sistema real practicando ataques de canal lateral, frente a los que los esquemas de criptografía son vulnerables. Estos tipo de ataques recogen consumo de energía, señales acústicas, radiación electromagnética o muestras de tiempo, asociados a las entradas que tenga el acelerador..

En resumen, en este trabajo se ha conseguido diseñar una implementación *hardware* adecuada a un entorno de recursos limitados que consigue acelerar el proceso de Falcon, un esquema de criptografía postcuántica. Adicionalmente, se han presentado algunas de las numerosas vertientes de investigación que se podrán llevar a cabo para hacer de esta implementación una opción más robusta, segura y aplicable a un entorno real antes de la llegada de los ordenadores cuánticos.

BIBLIOGRAFÍA

[1] R. P. Feynman. «Simulating physics with computers». En: *Feynman and computation*. cRc Press, 2018, págs. 133-153.

[2] D. Deutsch y R. Penrose. «Quantum theory, the Church–Turing principle and the universal quantum computer». En: *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 400.1818 (ene. de 1997). Publisher: Royal Society, págs. 97-117. DOI: 10.1098/rspa.1985.0070. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1985.0070 (visitado 08-07-2025).

[3] C. H. Bennett et al. «Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels». En: *Physical Review Letters* 70.13 (mar. de 1993), págs. 1895-1899. DOI: 10.1103/PhysRevLett.70.1895. URL: https://link.aps.org/doi/10.1103/PhysRevLett.70.1895 (visitado 12-01-2026).

[4] D. R. Simon. «On the Power of Quantum Computation». En: *SIAM Journal on Computing* 26.5 (oct. de 1997), págs. 1474-1483. ISSN: 0097-5397. DOI: 10.1137/S0097539796298637. URL: https://epubs.siam.org/doi/abs/10.1137/S0097539796298637 (visitado 12-01-2026).

[5] B. Schumacher. «Quantum coding». En: *Phys. Rev. A* 51 (4 abr. de 1995), págs. 2738-2747. DOI: 10.1103/PhysRevA.51.2738. URL: https://link.aps.org/doi/10.1103/PhysRevA.51.2738.

[6] R. Horodecki, P. Horodecki, M. Horodecki y K. Horodecki. «Quantum entanglement». En: *Rev. Mod. Phys.* 81 (2 jun. de 2009), págs. 865-942. DOI: 10.1103/RevModPhys.81.865. URL: https://link.aps.org/doi/10.1103/RevModPhys.81.865.

[7] P. Shor. «Algorithms for quantum computation: discrete logarithms and factoring». En: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. Nov. de 1994, págs. 124-134. DOI: 10.1109/SFCS.1994.365700. URL: https://ieeexplore.ieee.org/document/365700/ (visitado 08-07-2025).

[8] D. Beckman, A. N. Chari, S. Devabhaktuni y J. Preskill. «Efficient Networks for Quantum Factoring». En: *Physical Review A* 54.2 (ago. de 1996). arXiv:quant-ph/9602016, págs. 1034-1063. ISSN: 1050-2947, 1094-1622. DOI: 10.1103/PhysRevA.54.1034. URL: http://arxiv.org/abs/quant-ph/9602016 (visitado 15-01-2026).

[9] L. K. Grover. «A fast quantum mechanical algorithm for database search». En: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*. Philadelphia, Pennsylvania, United States: ACM Press, 1996, págs. 212-219. DOI: 10.1145/237814.237866. URL: http://portal.acm.org/citation.cfm?doid=237814.237866 (visitado 08-07-2025).

[10] C. Gidney y M. Ekerå. «How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits». en. En: *Quantum* 5 (abr. de 2021). arXiv:1905.09749 [quant-ph], pág. 433. ISSN: 2521-327X. DOI: 10.22331/q-2021-04-15-433. URL: http://arxiv.org/abs/1905.09749 (visitado 25-06-2025).

[11] C. Gidney. *How to factor 2048 bit RSA integers with less than a million noisy qubits*. arXiv:2505.15917 [quant-ph]. Mayo de 2025. DOI: 10.48550/arXiv.2505.15917. URL: http://arxiv.org/abs/2505.15917 (visitado 03-09-2025).

[12] M. Ivezic. *Quantum Breakthrough Slashes Qubit Needs for RSA-2048 Factoring*. en-US. Section: Industry News. Mayo de 2025. URL: https://postquantum.com/industry-news/quantum-breakthrough-rsa-2048/ (visitado 03-09-2025).

[13] D. Moody. *Post-Quantum Cryptography: NIST's Plan for the Future*. EN-US. Dic. de 2017. URL: https://csrc.nist.gov/Presentations/2016/pqc-announcement-and-outline-of-nists-call-for-sub (visitado 26-06-2025).

[14] G. Alagic et al. *Status report on the first round of the NIST post-quantum cryptography standardization process*. en. Inf. téc. NIST IR 8240. Gaithersburg, MD: National Institute of Standards y Technology, ene. de 2019, NIST IR 8240. DOI: 10.6028/NIST.IR.8240. URL: https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8240.pdf (visitado 29-07-2025).

[15] D. Moody et al. *Status report on the second round of the NIST post-quantum cryptography standardization process*. en. Inf. téc. NIST IR 8309. Gaithersburg, MD: National Institute of Standards y Technology, jul. de 2020, NIST IR 8309. DOI: 10.6028/NIST.IR.8309. URL: https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf (visitado 29-07-2025).

[16] G. Alagic et al. *Status report on the third round of the NIST Post-Quantum Cryptography Standardization process*. en. Inf. téc. NIST IR 8413-upd1. Gaithersburg, MD: National Institute of Standards y Technology (U.S.), sep. de 2022, NIST IR 8413-upd1. DOI: 10.6028/NIST.IR.8413-upd1. URL: https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413-upd1.pdf (visitado 26-06-2025).

[17] R. Avanzi et al. «CRYSTALS-Kyber algorithm specifications and supporting documentation». En: *NIST PQC Round* 2.4 (2019), págs. 1-43.

[18] V. Lyubashevsky et al. «Crystals-dilithium». En: *Algorithm Specifications and Supporting Documentation* 2 (2020).

[19] T. Prest et al. «Falcon». En: *Post-Quantum Cryptography Project of NIST* (2020).

[20] D. J. Bernstein et al. «The SPHINCS$^+$ Signature Framework». en. En: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. London United Kingdom: ACM, nov. de 2019, págs. 2129-2146. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3363229. URL: https://dl.acm.org/doi/10.1145/3319535.3363229 (visitado 26-06-2025).

[21] G. Alagic et al. *Status report on the fourth round of the NIST post-quantum cryptography standardization process*. en. Inf. téc. NIST IR 8545. Gaithersburg, MD: National Institute of Standards y Technology (U.S.), mar. de 2025, NIST IR 8545. DOI: 10.6028/NIST.IR.8545. URL: https://nvlpubs.nist.gov/nistpubs/ir/2025/NIST.IR.8545.pdf (visitado 10-12-2025).

[22] C. Aguilar-Melchor, O. Blazy, J.-C. Deneuville, P. Gaborit y G. Zémor. «Efficient Encryption From Random Quasi-Cyclic Codes». En: *IEEE Transactions on Information Theory* 64.5 (2018), págs. 3927-3943. DOI: 10.1109/TIT.2018.2804444.

[23] National Institute of Standards and Technology. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. en. Inf. téc. Federal Information Processing Standard (FIPS) 203. U.S. Department of Commerce, ago. de 2024. DOI: 10.6028/NIST.FIPS.203. URL: https://csrc.nist.gov/pubs/fips/203/final (visitado 07-07-2025).

[24] National Institute of Standards and Technology. *Module-Lattice-Based Digital Signature Standard*. en. Inf. téc. Federal Information Processing Standard (FIPS) 204. U.S. Department of Commerce, ago. de 2024. DOI: 10.6028/NIST.FIPS.204. URL: https://csrc.nist.gov/pubs/fips/204/final (visitado 07-07-2025).

[25] National Institute of Standards and Technology. *Stateless Hash-Based Digital Signature Standard*. en. Inf. téc. Federal Information Processing Standard (FIPS) 205. U.S. Department of Commerce, ago. de 2024. DOI: 10.6028/NIST.FIPS.205. URL: https://csrc.nist.gov/pubs/fips/205/final (visitado 07-07-2025).

[26] A. Regenscheid. *Transition to Post-Quantum Cryptography Standards*. en. Inf. téc. NIST IR 8547 ipd. Gaithersburg, MD: National Institute of Standards y Technology, 2024, NIST IR 8547 ipd. DOI: 10.6028/NIST.IR.8547.ipd. URL: https://nvlpubs.nist.gov/nistpubs/ir/2024/NIST.IR.8547.ipd.pdf (visitado 29-07-2025).

[27] C. Gentry, C. Peikert y V. Vaikuntanathan. «Trapdoors for hard lattices and new cryptographic constructions». En: *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*. STOC '08. Victoria, British Columbia, Canada: Association for Computing Machinery, 2008, págs. 197-206. ISBN: 9781605580470. DOI: 10.1145/1374376.1374407. URL: https://doi.org/10.1145/1374376.1374407.

[28] J. Hoffstein, J. Pipher y J. H. Silverman. «NTRU: A ring-based public key cryptosystem». en. En: *Algorithmic Number Theory*. Ed. por J. P. Buhler. Berlin, Heidelberg: Springer, 1998, págs. 267-288. ISBN: 978-3-540-69113-6. DOI: 10.1007/BFb0054868.

[29] L. Ducas y T. Prest. «Fast Fourier Orthogonalization». En: *Proceedings of the 2016 ACM International Symposium on Symbolic and Algebraic Computation*. ISSAC '16. New York, NY, USA: Association for Computing Machinery, jul. de 2016, págs. 191-198. ISBN: 978-1-4503-4380-0. DOI: 10.1145/2930889.2930923. URL: https://dl.acm.org/doi/10.1145/2930889.2930923 (visitado 10-12-2025).

[30] «IEEE Standard for Floating-Point Arithmetic». En: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (jul. de 2019), págs. 1-84. DOI: 10.1109/IEEESTD.2019.8766229. URL: https://ieeexplore.ieee.org/document/8766229/ (visitado 29-07-2025).

[31] M. Schmid, D. Amiet, J. Wendler, P. Zbinden y T. Wei. *Falcon Takes Off - A Hardware Implementation of the Falcon Signature Scheme*. Publication info: Preprint. 2023. URL: https://eprint.iacr.org/2023/1885 (visitado 26-06-2025).

[32] Y. Ouyang et al. «FalconSign: An Efficient and High-Throughput Hardware Architecture for Falcon Signature Generation». en. En: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025.1 (2025). Number: 1, págs. 203-226. ISSN: 2569-2925. DOI:

10.46586/tches.v2025.i1.203-226. URL: https://philosophymindscience.org/index.php/TCHES/article/view/11927 (visitado 26-06-2025).

[33] L. Beckwith, D. T. Nguyen y K. Gaj. *High-Performance Hardware Implementation of Lattice-Based Digital Signatures*. Publication info: Preprint. MINOR revision. 2022. URL: https://eprint.iacr.org/2022/217 (visitado 10-12-2025).

[34] B. Zhao, G. Alsuhi, H. Saleh y B. Mohammad. *Bi-SamplerZ: A Hardware-Efficient Gaussian Sampler Architecture for Quantum-Resistant Falcon Signatures*. arXiv:2505.24509 [cs]. Mayo de 2025. DOI: 10.48550/arXiv.2505.24509. URL: http://arxiv.org/abs/2505.24509 (visitado 26-06-2025).

[35] S. Pendyala, R. Magesh, E. B. Kavun y A. Aysu. *Outrunning the Millennium FALCON: Speed Records for FALCON on Xilinx FPGAs*. Publication info: Preprint. 2025. URL: https://eprint.iacr.org/2025/1490 (visitado 08-01-2026).

[36] S. Coulon, P. He, T. Bao y J. Xie. «Efficient Hardware RNS Decomposition for Post-Quantum Signature Scheme Falcon». En: *2023 57th Asilomar Conference on Signals, Systems, and Computers*. ISSN: 2576-2303. Oct. de 2023, págs. 19-26. DOI: 10.1109/IEEECONF59524.2023.10476845. URL: https://ieeexplore.ieee.org/document/10476845/ (visitado 26-06-2025).

[37] D.-T. Dam, T.-H. Tran, T.-H. Nguyen, T.-T. Hoang y C.-K. Pham. «Compact FALCON FFT/NTT Accelerator for Post-Quantum Cryptography». En: *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*. ISSN: 2158-1525. Mayo de 2025, págs. 1-5. DOI: 10.1109/ISCAS56072.2025.11043460. URL: https://ieeexplore.ieee.org/abstract/document/11043460 (visitado 10-12-2025).

[38] S. Mandal y D. Basu Roy. «Design of a Lightweight Fast Fourier Transformation for FALCON using Hardware-Software Co-Design». en. En: *Proceedings of the Great Lakes Symposium on VLSI 2024*. Clearwater FL USA: ACM, jun. de 2024, págs. 228-232. ISBN: 979-8-4007-0605-9. DOI: 10.1145/3649476.3660370. URL: https://dl.acm.org/doi/10.1145/3649476.3660370 (visitado 29-07-2025).

[39] Y. Tu y J. Xie. «EMINEM: Efficient FPGA Implementation of Mixed-RadIx NTT Hardware AccElerators for NIST Post-QuantuM Cryptography Falcon, Dilithium, and HAWK». En: *ACM Trans. Reconfigurable Technol. Syst.* 18.4 (2025), 52:1-52:25. ISSN: 1936-7406. DOI: 10.1145/3771287. URL: https://dl.acm.org/doi/10.1145/3771287 (visitado 08-01-2026).

[40] *Falcon*. URL: https://falcon-sign.info/ (visitado 20-01-2026).

[41] A. A. Karatsuba e Y. P. Ofman. «Multiplication of Multidigit Numbers on Automata». En: *Soviet Physics Doklady* 7 (1963). English translation of the 1962 article, págs. 595-596.

[42] T. Pornin. *New Efficient, Constant-Time Implementations of Falcon*. Publication info: Preprint. MINOR revision. 2019. URL: https://eprint.iacr.org/2019/893 (visitado 20-01-2026).

[43] T. Pornin. *Falcon on ARM Cortex-M4: an Update*. Publication info: Preprint. 2025. URL: https://eprint.iacr.org/2025/123 (visitado 20-01-2026).

PLANIFICACIÓN TEMPORAL Y PRESUPUESTO

En esta sección del documento se explicarán las etapas que han conformado la realización de este Trabajo de Fin de Master, así como una descomposición de las horas dedicadas a cada etapa. También se muestra el diagrama de Gantt del proyecto, con el que se puede ver en mayor detalle cómo se ha gestionado el proyecto a lo largo de la duración del mismo. En la segunda parte de esta sección se calculará el coste asociado al proyecto incluyendo coste de personal, coste material y coste de electricidad.

Planificación temporal del proyecto

Las etapas que se han llevado a cabo a lo largo de este proyecto han sido las siguientes

1. Definición del tema y planificación.
   Esta etapa se corresponde a la elección del objetivo del trabajo y definición del proceso para llevarlo a cabo. En este periodo se elige diseñar un acelerador *hardware* del esquema de firma digital Falcon en una FPGA de recursos limitados, al ser una oportunidad de investigación en el campo de la criptografía postcuántica y la seguridad. Se establece el plan a seguir y las fechas límite para la finalización de etapas, de modo que la carga de trabajo se distribuya de manera uniforme en el tiempo.

2. Revisión del estado del arte.
   Para la realización de este Trabajo Fin de Master se necesitan conocimientos sobre criptografía postcuántica así como diseño de *hardware*. Estos son los dos primeros temas sobre los que se realiza un estudio previo para aumentar la eficiencia al iniciar el desarrollo del proyecto. También se realiza una revisión bibliográfica del estado del arte con el fin de encontrar diseños *hardware* existentes en la literatura y poder contrastar alternativas de diseño con otras implementaciones.

3. Diseño de los módulos y la estructura *hardware* en HDL.
   Implementar todas las funciones del código de referencia *software* aprovechando las características de la programación en HDL. Crear la estructura que permita la comunicación entre módulos y el flujo de trabajo que sigue el esquema de firma digital Falcon para la generación de claves, la generación de firma y la verificación de firma.

4. Simulación y validación de las descripciones *hardware*.
   Al mismo tiempo que se diseñan los módulos, se realizan simulaciones parciales para ver su correcto funcionamiento, tanto a nivel de resultados como a nivel de flujo. Esta etapa queda finalizada cuando se realiza una simulación de la IP completa, recreando una de las pruebas de la implementación.

5. Elaboración del diseño de bloques de la implementación.
   Una vez se ha comprobado el correcto funcionamiento de la IP con diferentes valores, se debe realizar un diseño de bloques que contenga al procesador MicroBlaze, la IP personalizada, la comunicación con el ordenador por AXI UART Lite y el AXI Timer para medir el número de ciclos. Con este diseño de bloques se procede a hacer síntesis e implementación, a partir de las cuales se modifican los módulos hasta llegar a valores de utilización de recursos y retardo lógico adecuados.

6. Implementación en la Nexys Video.
   A partir de la implementación se genera un *bitstream* que se exporta como *hardware* para

poder programar la FPGA. Se utiliza el entorno de desarrollador Vitis para modificar el código *software* de referencia para que el procesador utilice el acelerador como un periférico a través de los registros.

7. Validación de la implementación y banco de pruebas.
   Se comprueba que el acelerador funciona correctamente a través de entradas de prueba y comparando con el código de referencia. Una vez validado se ejecuta el banco de pruebas para tomar valores de rendimiento. Sobre estos resultados se hace un análisis estadístico y se llega a una conclusión en relación a las pruebas y el estado del arte.

8. Redacción de la memoria del proyecto.
   Por último, esta etapa se corresponde a todo lo relacionado con el documento del Trabajo de Fin de Master. Esto incluye tanto la redacción de los conceptos teóricos y el diseño realizado como la elaboración de tablas, diagramas, gráficas e imágenes que contribuyan a la comprensión del documento. También se investiga acerca de la normativa, los aspectos legales y la contribución a los objetivos de desarrollo sostenible.

Este proyecto suma un total de 360 horas, en la tabla 6.1 se puede ver la cantidad de horas asociadas a cada etapa del proyecto.

Etapa del proyecto	Duración (horas)	Inicio	Fin
Definición del tema y planificación	3	05/09/2025	09/09/2025
Revisión del estado del arte	20	07/09/2025	18/09/2025
Diseño del *hardware* HDL	155	19/09/2025	20/11/2025
Simulación y validación del *hardware*	42	19/10/2025	05/12/2025
Elaboración del diseño de bloques	12	06/12/2025	11/12/2025
Implementación en la Nexys Video	20	11/12/2025	18/12/2025
Validación de la implementación	48	19/12/2025	18/01/2026
Redacción de la memoria del proyecto	60	14/01/2026	30/01/2026

Tabla 6.1: Duración y distribución temporal de las etapas del proyecto.

Algunas de las etapas de este proyecto pueden realizarse simultáneamente, por ejemplo las simulaciones parciales del algoritmo es compatible con el diseño de módulos restantes. Por otra parte, hay otras etapas que necesitan la finalización de etapas previas, por ejemplo no se puede implementar el diseño en la FPGA si no se tiene el *bitstream* generado por Vivado. Esto se ve reflejado en el diagrama de Gantt del proyecto.

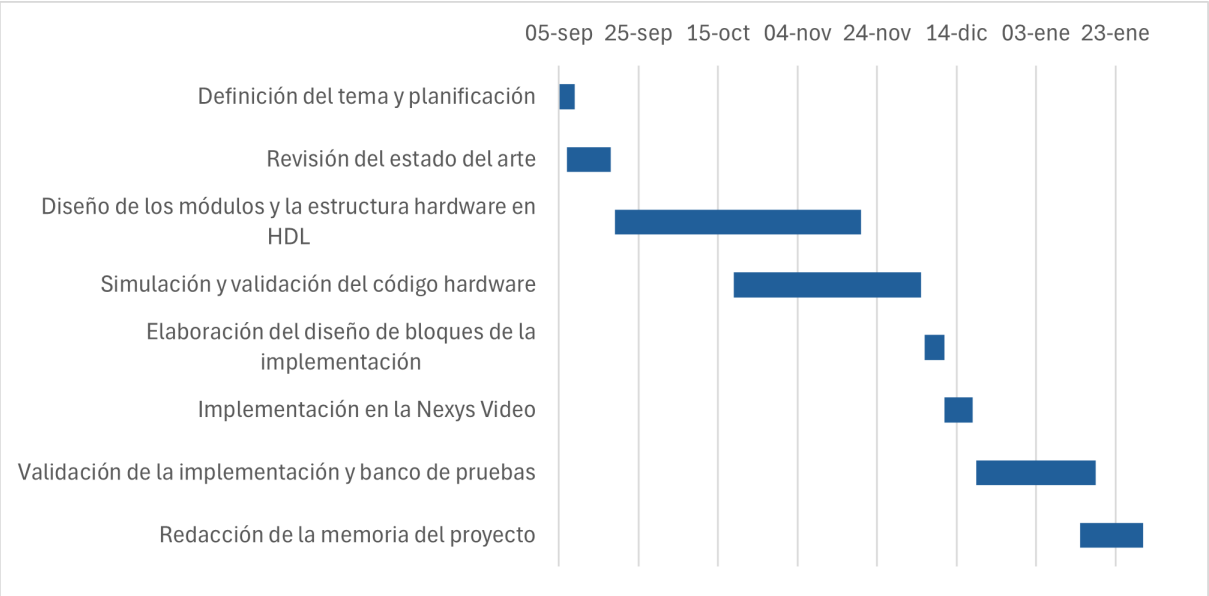

Figura 6.1: Diagrama de Gantt del proyecto.

Presupuesto asociado al proyecto

En cuanto al presupuesto asociado al proyecto se tienen en cuenta tres factores fundamentales: el coste de personal relativo a los participantes en el proyecto, el coste de material utilizado para llevar a cabo el trabajo y el coste de electricidad, ya que al estar utilizando de manera continua equipamiento electrónico e informático es relevante.

- Coste de personal.

El coste de personal se calcula a través de una estimación del salario del alumno y los tutores junto con las horas aplicadas en este proyecto. En el caso del alumno, se estima un salario de 20 €/hora con 360 horas aplicadas. En el caso de los dos tutores, se estima un salario de 45€/hora con un 20 % de las horas aplicadas por el alumno.

	Tiempo (horas)	Salario estimado (€/hora)	Coste total (€)
Alumno	360	20,00	7.200,00
Tutor 1	72	45,00	3.240,00
Tutor 2	72	45,00	3.240,00

Tabla 6.2: Coste de personal.

El coste de personal total viene dado por la siguiente ecuación.

$$Coste\ personal = \sum Coste\ individual = 7{,}200 + 2 \cdot 3{,}240 = 13{,}680\ € \tag{6.1}$$

- Coste de material.

El coste de material queda definido por las diferentes herramientas que se han utilizado para la realización del proyecto. Para la primera parte relativa al desarrollo de código *software*, de descripciones *hardware* y simulación se ha necesitado un ordenador, para el que se estima una vida útil de 8 años con un valor residual del 10 %. La segunda parte relativa a la implementación del *hardware* y la toma de resultados, ademas del ordenador se han necesitado las dos placas Nexys Video y la Pynq-Z2, que han sido considerados como bienes fungibles. Los costes de material se calculan con las siguientes ecuaciones.

$$Horas\ vida\ útil = Vida\ útil \cdot \frac{Días}{Año} \cdot \frac{Horas}{Día} = 8 \cdot 240 \cdot 8 = 15{,}360 \tag{6.2}$$

$$Coste\ material = \frac{V_O - V_r}{Horas\ vida\ útil} \cdot Tiempo\ de\ uso = \frac{1000 - 0{,}1 \cdot 1000}{15{,}360} \cdot 360 = 21{,}10\ € \tag{6.3}$$

$$Coste\ material = \sum Coste\ dispositivos = 21{,}10 + 180 + 550 = 751{,}10\ € \tag{6.4}$$

- Coste de electricidad.

El trabajo realizado con ordenador y con dispositivos electrónicos hace obligatorio el uso de electricidad. Este valor de consumo depende del dispositivo, para el ordenador se estiman 300 W y un tiempo total de 360 horas de uso, para la Nexys Video se estiman 36 W (máximo teórico) con un tiempo total de 60 horas entre implementación y pruebas y para la Pynq-Z2 se estiman 2,5 W (alimentación por USB) con un tiempo total de 8 horas de pruebas.

$$Energía\ total = \sum potencia\ (\text{kW}) \cdot tiempo\ (\text{h}) = 0{,}3 \cdot 360 + 0{,}036 \cdot 60 + 0{,}0025 \cdot 8 = 110{,}18\ kWh \tag{6.5}$$

El coste medio por MWh en el año 2025 se sitúa en 65,28 € en compra por mayoristas, por lo que se estima un coste superior según el portal de compras de la Universidad Politécnica de Madrid en torno a 157 €/MWh.

$$Coste\ total = 110{,}18\ \text{kWh} \cdot \frac{1\ MWh}{1000\ kWh} \cdot 157\ \frac{€}{MWh} = 17{,}30\ € \tag{6.6}$$

- Coste total asociado al proyecto.

A estos costes totales de los apartados personal, material y electricidad se les debe aplicar el impuesto sobre el valor añadido (IVA), quedando un coste total que se puede observar en la tabla a continuación.

Coste personal	13.680,00 €
Coste material	751,10 €
Coste electricidad	17,30 €
Coste total	14.448,40 €
Coste total + IVA	**17.482,56 €**

Tabla 6.3: Coste total del proyecto.

EVALUACIÓN DE IMPACTOS

La investigación en el campo de la criptografía postcuántica es de gran relevancia en la actualidad como se ha visto a lo largo de este trabajo. En concreto, el desarrollo de un acelerador *hardware* de un esquema de firma digital tiene un impacto directo en el ámbito social, en el ámbito económico y en el medioambiental.

Uno de los objetivos principales de la criptografía es garantizar la privacidad de la información y la autenticidad de los mismos. Es por esto que se relaciona directamente con el ámbito social, la implementación *hardware* diseñada en este trabajo permite reforzar la seguridad en la transmisión de información en la industria. Además, la investigación en la criptografía postcuántica en *hardware* genera nuevas oportunidades de trabajo y de estudio, al ser un tema relevante en la actualidad.

Uno de los impactos más relevantes en el ámbito económico se debe a que la mayor parte de las transacciones monetarias se realizan de manera digital y la criptografía es parte fundamental para que estas transacciones se realicen de forma segura. También existe un impacto relativo al uso de aceleradores *hardware* en lugar de plataformas *software*, ya que con este cambio se produce una reducción del consumo energético y por tanto una reducción del coste. Al ser un diseño personalizable y fácil de configurar, se podrán generar en un futuro implementaciones que permitan generar valor económico en el sector de la electrónica.

Para finalizar, el ámbito medioambiental también está relacionado con la reducción del consumo energético por el uso de aceleradores *hardware*. En este caso, el proyecto está enfocado en una plataforma de recursos limitados, por lo que las pruebas no han supuesto un consumo eléctrico elevado y una futura implementación en entornos reales podría suponer una mayor eficiencia energética en la industria.

Posteriormente, en la sección de contribución a los ODS, estos impactos se verán reflejados más en detalle.

ANÁLISIS DE ASPECTOS LEGALES Y ÉTICOS

El diseño del acelerador *hardware* de este trabajo se ha realizado con un enfoque ético, utilizando entornos de desarrollador estándar para la elaboración de código, la simulación y la implementación (Visual Studio Code, Vitis, Vivado). Además se pretende que el trabajo de este acelerador sea publicado como código abierto, impulsando el estudio en la criptografía postcuántica gracias a la modularidad y capacidad de reconfiguración del diseño. Esto permitirá un control de versiones necesario para asegurar en todo momento la fiabilidad del diseño con las nuevas actualizaciones por parte de los autores de Falcon y el NIST.

Para el desarrollo de las descripciones *hardware* se ha utilizado el lenguaje de programación VHDL. El estándar IEEE 1076 define la sintaxis y mecanismos de descripción que permiten producir diseños portables y reutilizables gracias a herramientas como Vivado, que se utiliza en este trabajo. El tipo de coma flotante utilizado en este algoritmo queda definido en el estándar IEEE 754, el cual permite desarrollar un código que emule las operaciones en coma flotante de forma precisa y consistente. Permite la verificación y validación de resultados con las plataformas *software* que disponen de FPU.

En el diseño de bloques de la implementación, el procesador, las IP de AMD y el acelerador se comunican por interfaz AXI Lite. El protocolo AXI queda definido bajo las especificaciones de la arquitectura estandarizada Advanced Microcontroller Bus Architecture (AMBA), desarrollada por ARM. Además, la implementación y las descripciones HDL quedan protegidas por el estándar IEEE 1735. Este estándar garantiza la protección y gestión de la propiedad intelectual a través del cifrado del código.

La criptografía postcuántica requiere una anticipación a la llegada de los ordenadores cuánticos, es por eso que en ETSI TR 103 619 se establece un marco de análisis y actuación para una transición segura hacia la criptografía resistente a ataques cuánticos. Para el caso del esquema de firma digital que se implementa en este trabajo, Falcon todavía no dispone de un estándar oficial FIPS por parte del NIST, es por eso que se deberán realizar las modificaciones necesarias cuando este estándar se haga público para que el acelerador siga en la misma línea que el algoritmo elegido.

Por último, ante una futura implementación en un entorno real, el acelerador tendrá la capacidad de manejar información que se desea transmitir como un punto previo al envío del mensaje. El manejo de estos datos sensibles hace que se deba cumplir con el Reglamento General de Protección de Datos (RGPD UE 2016/679) para garantizar la seguridad y la privacidad de la información.

CONTRIBUCIÓN A LOS OBJETIVOS DE DESARROLLO SOSTENIBLE

A continuación se muestran los Objetivos de Desarrollo Sostenible (ODS) relacionados con este Trabajo de Fin de Master y cómo contribuye este trabajo en el desarrollo de los mismos.

- ODS 4 : Educación de calidad. El trabajo fomenta el estudio en el campo de la criptografía postcuántica, también ofrece como recurso una implementación *hardware* en código abierto a la finalización del mismo. El autor y los tutores con la realización de este trabajo refuerzan la formación en ingeniería y el desarrollo de competencias relativas a la investigación.
- ODS 8 : Trabajo decente y crecimiento económico. La investigación en el campo de la criptografía postcuántica abre la posibilidad de realizar nuevos proyectos, que generarán oportunidades de trabajo para otros jóvenes investigadores. A su vez, este trabajo desarrolla tecnologías de alto valor agregado que favorecen el crecimiento económico a nivel nacional.
- ODS 9 : Industria, innovación e infraestructuras. El desarrollo de aceleradores *hardware* fomenta la innovación tecnológica y refuerza la seguridad en infraestructuras críticas.
- ODS 12 : Producción y consumo responsables. La mejora de eficiencia en los sistemas actuales de criptografía postcuántica reduce el coste computacional y, por tanto el consumo energético.
- ODS 13 : Acción por el clima. El desarrollo e implementación de sistemas alternativos, optimizados y eficientes como los aceleradores *hardware* reducen de forma indirecta la emisión de gases de efecto invernadero, contribuyendo a la reducción de la huella de carbono.

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE CÓDIGOS

ABREVIATURAS, UNIDADES Y ACRÓNIMOS

AES Advanced Encryption Standard
AMBA Advanced Microcontroller Bus Architecture
AMD Advanced Micro Devices
ARM Advanced RISC Machine
AVX2 Advanced Vector eXtensions
AXI Advanced eXtensible Interface
BRAM Block Random Access Memory
CCA Chosen-Ciphertext Attack
clk Clock
CPA Chosen-Plaintext Attack
CMA Chosen-Message Attack
CSR Circular Shift Registers
DSA Digital Signature Algorithms
DSP Digital Signal Processing
emu Emulated floating point
ETSI European Telecommunications Standards Institute
Falcon Fast-Fourier Lattice-based Compact Signatures Over NTRU
FF Flip-Flop
FFT Fast Fourier Transform
FIPS Federal Information Processing Standards
FMA Fused Multiply-Add
FPGA Field-Programmable Gate Array
FPU Floating Point Unit
FSM Finite State Machine
GPV Gentry-Peikert-Vaikuntanathan
HDL Hardware Description Language
HLS High Level Synthesis
IEEE Institute of Electrical and Electronics Engineers
IND-CCA2 Indistinguishability under Adaptive Chosen-Ciphertext Attack
IoT Internet of Things
IP Intellectual Property
KEM Key Encapsulation Mechanism
log Logarithm
LUT Look Up Table
LWE Learning With Errors
MLWE Modular-Learning With Errors
MMU Memory Management Unit
NaN Not a Number
nat Native floating point
NIST National Institute of Standards and Technology
NS Negative Slack
NTRU N-th degree Truncated polynomial Ring Units
NTT Number Theoretic Transform
ODS Objetivos de Desarrollo Sostenible
PKE Public Key Encryption
PL Programmable Logic
PS Processing System
QCSD Quasi-Cyclic Syndrome Decoding
QROM Quantum Random Oracle Model

RAM Random Access Memory
RGPD Reglamento General de Protección de Datos
RNS Residue Numeral System
ROM Read Only Memory
ROM Random Oracle Model
RSA Rivest-Shamir-Adleman
SIMD Single Instruction Multiple Data
UART Universal Asynchronous Receiver-Transmitter
USB Universal Serial Bus
VHDL VHSIC and HDL
VHSIC Very High Speed Integrated Circuit
WNS Worst Negative Slack
XSA Xilinx Support Archive

B byte (unidad de almacenamiento de memoria)
CCs ciclos de reloj (unidad de tiempo)
€ euro (unidad monetaria)
h hora (unidad de tiempo)
Hz Hercio (unidad de frecuencia)
s segundos (unidad de tiempo)
W Vatio (unidad de potencia)
Wh Vatio hora (unidad de energía)

GLOSARIO

- **Acelerador *hardware***

Dispositivo especialmente diseñado para mejorar la eficiencia en la elaboración de determinadas tareas computacionales, generalmente implementado en sistemas embebidos utilizando FPGAs. En este proyecto, un acelerador *hardware* es implementado en la Nexys Video.

- **AXI (*Advanced eXtensible Interface*)**

Protocolo de interfaz de bus estándar siguiendo la arquitectura AMBA, utilizado para la comunicación entre componentes *hardware*, sistemas de procesamiento y demás lógica programable en el ámbito de los sistemas embebidos. En este proyecto, se utiliza AXI Lite como interfaz de comunicación entre el procesador y los componentes del diseño de bloques como la IP personalizada, AXI Timer y AXI UART Lite.

- **Banco de pruebas**

También conocido como *testbench*, es un entorno de simulación utilizado en el diseño de *hardware* para validar la funcionalidad en comparación con el *software*, así como la toma de resultados variando los estímulos de entrada para hacer un análisis probabilístico. En este proyecto, se ha realizado un banco de pruebas de 1000 parejas de semilla-mensaje sobre los que se han realizado las conclusiones.

- **BRAM (*Block Random Access Memory*)**

Bloques de memoria RAM sintetizados en la lógica programable de la FPGA, que proporcionan almacenamiento en formato de buffers. En este proyecto, se generan para almacenar los datos necesarios para el algoritmo como las claves, la firma o el contexto pseudoaleatorio.

- **Criptografía postcuántica**

La criptografía es la ciencia que permite el estudio de diferentes algoritmos y protocolos que permiten proteger la información. Se denomina postcuántica cuando este estudio se realiza para proteger frente a ataques realizados por ordenadores cuánticos. Este proyecto está ligado a la criptografía postcuántica al diseñarse una implementación *hardware* de Falcon.

- **DSP (*Digital Signal Processing*)**

Procesador especializado en señales digitales que se encuentra en una posición fija dentro de la zona programable de la FPGA, optimiza operaciones matemáticas como sumas y multiplicaciones. En este proyecto, se utilizan mayoritariamente para reducir el tiempo de ejecución de las multiplicaciones.

- **Falcon (*Fast-Fourier Lattice-based Compact Signatures Over NTRU*)**

Esquema de firma digital elegido como apto por el NIST en el proceso de estandarización para esquemas de criptografía postcuántica. Es el algoritmo sobre el que se basa esta implementación *hardware*.

- **FPGA (*Field-Programmable Gate Array*)**

Dispositivo que permite la implementación de circuitos *hardware* personalizados mediante lógica reconfigurable tras fabricación. Este proyecto se diseña como *hardware* reconfigurable en la zona programable de una FPGA.

- **FF (*Flip-Flop*)**

Elemento básico de lógica secuencial que almacena un bit de información, actualizando su estado en cada ciclo de reloj, ampliamente utilizado en la implementación de registros. En este proyecto se utilizan para almacenar datos en registros temporales, sin necesitar el uso de BRAMs.

- **Hash-and-sign**

Paradigma en esquemas de firma digital donde se aplica una función *hash* al mensaje previo a la firma, la firma se ejecuta la clave privada sobre este *hash* del mensaje. Es la metodología que sigue el esquema de firma digital Falcon, en el que se basa el diseño de esta implementación *hardware*.

- **HLS (*High Level Synthesis*)**

Técnica que convierte código software de alto nivel en descripciones HDL sintetizables, facilitando el desarrollo de circuitos complejos y optimizando a través de *pragmas*. Es el método utilizado por diversas implementaciones *hardware* con las que se compara el diseño en este proyecto.

- **IP (*Intellectual Porperty*)**

Bloques de diseño *hardware* reutilizables, como módulos personalizados que se integran en diseños mayores para realizar una implementación y proteger los derechos de autor. En este proyecto se diseña una IP personalizada del acelerador para actuar como periférico del MicroBlaze a través de interfaz AXI Lite, permitiendo una fácil reconfiguración desde *software*.

- **LUT (*Look-Up Table*)**

Estructura básica en las FPGAs que implementa funciones lógicas mediante tablas de verdad. En este proyecto, la mayoría de oepraciones y las máquinas de estado se sintetizan como tablas de verdad con entradas y salidas predefinidas.

- **MicroBlaze**

Procesador soft-core RISC de 32 bits desarrollado por AMD, utilizado en FPGAs que no disponen de sistema de procesamiento. En este proyecto se utiliza un MicroBlaze como procesador, ya que la Nexys Video no dispone de procesador.

- **NIST (*National Institute of Standards and Technology*)**

Agencia del Departamento de Comercio de EE.UU. responsable de desarrollar estándares en tecnología, como los procesos de estandarización postcuántica. Falcon, el algoritmo en el que se basa esta implementación fue elegido por el NIST para su estandarización.

- **Nexys Video**

Placa de desarrollo FPGA de recursos limitados basada en un Artix-7, dispone de interfaces de programación como JTAG, alimentación por fuente externa y comunicación UART por micro-USB. Es la plataforma sobre la que se ha implementado este diseño *hardware*.

- **Número de ciclos**

Unidad de medida para contabilizar los pulsos de reloj necesarios para ejecutar una operación, y posteriormente evaluar el rendimiento del *hardware* comparado con el *software*. En este proyecto es la métrica principal para comparar la eficiencia de la implementación *hardware*.

- **Retículo**

Subgrupo discreto de un espacio vectorial generado por combinaciones lineales entre vectores base, utilizado en criptografía para basar la seguridad en problemas resistentes a ataques cuánticos. Falcon se basa en esta estructura al utilizar NTRU.

- **Vitis**

Entorno de diseño integrado de AMD para el desarrollo de aplicaciones software de sistemas embebidos e implementación de estos en plataformas como las FPGA. Este proyecto utiliza Vitis como herramienta para la elaboración de la plataforma y la aplicación para interactuar con la implementación *hardware*.

- **Vivado**

Entorno de diseño integrado de AMD para el desarrollo de descripciones en lenguaje VHDL, generación de síntesis e implementación de *hardware* en FPGA. Este proyecto utiliza Vivado para la creación de los módulos, creación de la IP personalizada y generación del archivo XSA para el uso posterior en Vitis.

POLITÉCNICA

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES
UNIVERSIDAD POLITÉCNICA DE MADRID

José Gutiérrez Abascal, 2. 28006 Madrid
Tel.: 91 336 3060
info.industriales@upm.es

www.industriales.upm.es