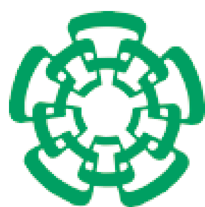


Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

UNIDAD ZACATENCO

DEPARTAMENTO DE COMPUTACIÓN

Aceleración de operaciones en el algoritmo de criptografía postcuántica CRYSTALS-KYBER

TESIS

Que presenta

Adriana Pérez Navarro

Para obtener el grado de

MAESTRA EN CIENCIAS EN COMPUTACIÓN

Directora de la Tesis:

Dra. Brisbane Ovilla Martínez

Ciudad de México Diciembre, 2024

Resumen

Hoy en día, los criptosistemas de clave pública como RSA y la criptografía de curva elíptica, representan los estándares para el intercambio seguro de claves y firmas digitales. Sin embargo con los avances en computación cuántica, se espera que se vuelvan inseguros. Ante esta preocupación, surge la Criptografía Postcuántica como una nueva área de investigación para desarrollar sistemas criptográficos resistentes a ataques convencionales y cuánticos y que puedan así sustituir las soluciones tradicionales de criptografía de clave pública. El Instituto Nacional de Estándares y Tecnología en 2017 inició un proceso para estandarizar nuevos algoritmos de clave pública que sean robustos frente a ataques cuánticos; el ganador de la tercera ronda de competencias fue el algoritmo CRYSTALS-Kyber. En este trabajo se propone diseñar e implementar un codiseño HW/SW para acelerar operaciones en el algoritmo de criptografía postcuántica CRYSTALS-Kyber. El diseño e implementación de una partición eficiente entre hardware y software maximizó el desempeño del algoritmo CRYSTALS-Kyber, en donde el hardware fue responsable de acelerar la Transformada Teórica de Números (NTT, por sus siglas en inglés) y su inversa (NTT^{-1}), así como las primitivas SHA3-256, SHA3-512, SHAKE-128 Y SHAKE-256 de la familia SHA-3, fundamentales en CRYSTALS-Kyber para la generación de claves, hashes y expansión de semillas.

Abstract

Today, public key cryptosystems such as RSA and elliptic curve cryptography represent the standards for the secure exchange of keys and digital signatures. However, with advances in quantum computing, they are expected to become insecure. Given this concern, Post-Quantum Cryptography emerges as a new area of research to develop cryptographic systems that are resistant to conventional and quantum attacks and can replace traditional public key cryptographic solutions. The National Institute of Standards and Technology in 2017 initiated a process to standardize new public-key algorithms that are robust against quantum attacks; the winner of the third round of competitions was the CRYSTALS-Kyber algorithm. In this paper, we propose to design and implement a HW/SW codesign to accelerate operations on the CRYSTALS-Kyber post-quantum cryptography algorithm. The design and implementation of an efficient partition between hardware and software maximized the performance of the CRYSTALS-Kyber algorithm, where the hardware was responsible for accelerating the Theoretical Number Transform (NTT, and its inverse (NTT^{-1}), as well as the SHA3-256, SHA3-512, SHAKE-128 and SHAKE-256 primitives of the SHA-3 family, fundamental in CRYSTALS-Kyber for key generation, hashes and seed expansion.

Agradecimientos

En primer lugar dedico esta tesis a mi familia, con todo mi corazón gracias por su amor incondicional y por estar siempre a mi lado. Su ejemplo diario me inspiran a ser una mejor persona. Eternamente los amo.

Agradezco a mi asesora, la Dra. Brisbane Ovilla Martínez, por su guía y paciencia a lo largo de este proyecto. A todos los profesores que formaron parte de mi formación, gracias por sus consejos y su tiempo. Cada una de sus enseñanzas ha dejado una huella en mi camino académico y profesional.

Agradezco a mis amigos, todos son personas increíbles. A mis amigos de toda la vida, por ser mi otra familia. A mis amigos en México por su compañía, apoyo y risas, gracias por ser mi hogar lejos de Cuba96, sin ustedes este camino hubiese sido mucho más díficil.

Finalmente, quiero agradecer al Departamento de Computación del Cinvestav por permitirme ser parte de este proyecto, y como CONAHCyT brindarme los recursos y oportunidades necesarios para realizar mis estudios de maestría.

¡A todas las personas que forman parte de mi vida por ser parte de mi historia!

Índice general

Resumen III

Abstract V

Agradecimientos VII

Índice de figuras XII

Índice de tablas XIII

1. Introducción 1
- 1.1. Planteamiento del problema 4
- 1.2. Objetivos 5
- 1.3. Propuesta de solución 5
- 1.4. Organización de la Tesis 6

2. Sistema en un Chip 9
- 2.1. Arquitectura Zynq 10
  - 2.1.1. PS 10
  - 2.1.2. FPGA 12
- 2.2. Lenguajes 16
  - 2.2.1. C Embebido 16
  - 2.2.2. VHDL 17
- 2.3. Herramientas 18
  - 2.3.1. Vivado 19
  - 2.3.2. Vitis 20

3. Criptografía Postcuántica 23
- 3.1. Antecedentes 23
- 3.2. KEM 27
- 3.3. Criptografía con base en *lattices* 28
- 3.4. El problema M-LWE 29
- 3.5. CRYSTALS-Kyber 30
  - 3.5.1. Conjuntos de parámetros de Kyber 31
  - 3.5.2. Especificaciones para Kyber.CPAPKE 32

3.5.3. Especificaciones para KYBER.CCAKEM . . . 36
3.5.4. Arquitectura KYBER . . . 39
3.5.5. Unidad Keccak . . . 40
3.5.6. NTT . . . 49

4. Diseño de CRYSTALS-KYBER 55
4.1. Propuesta . . . 55
4.2. Diseño de la partición en software . . . 56
4.3. Diseño de la partición en hardware . . . 58
4.3.1. Diseño del módulo SHA-3 . . . 58
4.3.2. Implementación de la NTT . . . 64

5. Resultados 69
5.1. Plataforma Final . . . 69
5.2. Módulo SHA-3 . . . 71
5.3. Módulo NTT/NTT^{-1} . . . 73

6. Conclusiones 81
6.1. Trabajo futuro . . . 82

Bibliografía 83

A. Instalación de Vivado ML 89
A.1. Requisitos previos . . . 89
A.2. Descarga de Vivado . . . 89
A.3. Instalación . . . 89
A.4. Activación de la licencia . . . 90

B. Descripción de la tarjeta Zybo z7 20 91

Índice de figuras

1.1. Propuesta de solución. 6

2.1. Arquitectura Zynq. 11
2.2. La estructura lógica y sus elementos constitutivos. 13
2.3. Composición de un Bloque Lógico Configurable. 14
2.4. Capacidades aritméticas del DSP48E1, encontrado en la familia Zynq-7000. 15
2.5. Arquitectura interna de una FPGA de Xilinx. 16
2.6. Ambiente de desarrollo Vivado design Suite. 19
2.7. Ambiente de desarrollo Vitis IDE. 20

3.1. Como se realiza el establecimiento de claves usando un KEM. 27
3.2. SVP y CVP en *lattices*. 29
3.3. Esquema Kyber.CCAKEM. 38
3.4. Esquema CRYSTALS-Kyber. 39
3.5. Matriz de estado de $5 \times 5 \times \omega$. 41
3.6. Construcción esponja Keccak. 43
3.7. Función f. 43
3.8. θ aplicada a un solo bit. 44
3.9. ρ aplicado a los carriles para el caso w = 8. 45
3.10. π aplicada a solo una rebanada. 46
3.11. χ aplicada a una sola fila. 47
3.12. Configuración de mariposa CT. 52
3.13. Configuración de mariposa GS. 52
3.14. Ejemplo de una multiplicación polinómica basada en NTT de 8 puntos. El gráfico de flujo de datos que incluye NTT basada en mariposa CT, multiplicación por puntos e NTT^{-1} basada en mariposa GS. El polinomio $\hat{f}$ está en el dominio NTT y f está en el dominio normal. 53

4.1. Metodología de Codiseño HW/SW para CRYSTALS-Kyber. 56
4.2. Diagrama de la arquitectura para el módulo SHA-3. 60
4.3. Diagrama de la unidad de Padding. 61
4.4. Diagrama de la unidad de XOR. 61
4.5. Diagrama del módulo Keccak completo. 62

4.6. Función de ronda Keccak. 63
4.7. Diagrama de estados de la unidad de control del módulo Keccak. . . . 63
4.8. Diagrama de estados de la unidad de control del Módulo SHA-3. . . . 64
4.9. Arquitectura NTT/NTT^{-1}. 65
4.10. Control NTT_NTT^{-1}: máquina de estados y contador de las etapas. 66
4.11. Diagrama del circuito de la unidad Mariposa Híbrida. 66
4.12. Diagrama de circuito para la Reducción Montgomery. 67

5.1. Arquitectura de la plataforma final. 77
5.2. Diagrama de tiempos de control de flujo general para el módulo SHA-3. 79
5.3. Diagrama de tiempos de control de flujo general para NTT/NTT^{-1}. . 79

B.1. Zybo z7 20, tomada de [1]. 93

Índice de tablas

3.1. Parametros de Kyber. 32
3.2. Tamaño en bytes de las llaves y el texto cifrado de Kyber. 32
3.3. Anchos de permutación KECCAK-p y cantidades ω y ℓ relacionadas. 41
3.4. Desplazamientos de ρ. 45
3.5. Valores de las constantes de ronda RC[i]. 48
3.6. Parámetros de las funciones SHA-3. 49

4.1. Funciones de las primitivas de SHA-3 en Kyber-768, entradas y salidas. 59

5.1. Resumen de utilización de recursos de la Plataforma completa. 70
5.2. Comparación del módulo SHA-3 con los trabajos relacionados en términos de área. 72
5.3. Comparación del módulo SHA-3 con los trabajos relacionados en términos de velocidad. 72
5.4. Evaluación del desempeño de NTT/NTT^{-1} en términos de área. . . . 74
5.5. Evaluación del desempeño de NTT/NTT^{-1} en términos de velocidad. 75
5.6. Resumen de utilización de recursos para `hashModule_v1_0`. 78
5.7. Resumen de utilización de recursos para IP_NTTINTT_0. 78

B.1. Recuros de la Zybo Z7-20 92
B.2. Descripción de la tarjeta Zybo Z7-20 93

List of Algorithms

1. Kyber.CPAPKE.KeyGen() 33
2. Kyber.CPAPKE.Enc(pk, m, r) 34
3. Kyber.CPAPKE.Dec(sk, c) 35
4. Kyber.CCAKEM.KeyGen() 36
5. Kyber.CCAKEM.Enc(pk) 36
6. Kyber.CCAKEM.Des(c, sk) 37
7. KECCAK-p[b, n_r] 40
8. Esponja[f, pad, r](N, d) 42
9. θ (A) 44
10. ρ (A) 45
11. π (A) 46
12. χ (A) 47
13. ι (A) 48
14. NTT(f) 51
15. $\mathrm{NTT}^{-1}(\hat{f})$ 52
16. Reducción de Montgomery 54

Glosario

CLB Bloques Lógicos Configurables. 12

CVP Problema del Vector más Cercano. 28

DSP Procesador Digital de Señales. 10

ECC Criptografía de Curva Elíptica. 1

FF Flip-Flop. 3

FPGA Arreglo de Puertas Programables en Campo . 3

Hash Función que toma una entrada y devuelve un valor de longitud fija. 2

HDL Lenguaje de Descripción de Hardware . 3

KEM Mecanismo de Encapsulación de Claves. 2

LBC Códigos Basados en Lattices. 2

LUT Tabla de Búsqueda. 3

M-LWE Aprendizaje Modular con Errores. 2

ML-KEM Mecanismo de Encapsulación de Claves basado en Rejillas Modulares . 2

NIST Instituto Nacional de Estándares y Tecnología. 2

NTT Transformada Teórica de Números. 2

PL Lógica Programable. 9

PQC Criptografía Postcuántica. 2

PS Sistema de Procesamiento. 9

RAM Memoria de Acceso Aleatorio. 13

ROM Memoria de Solo Lectura. 13

RSA Rivest-Shamir-Adleman. 1

SHA-3 Algoritmo de resumen seguro 3. 3

SoC Sistema en un Chip. 4

SVP Problema del Vector más Corto. 28

XOF Función de salida extensible. 3

Capítulo 1

Introducción

La criptografía es el arte de escribir de manera secreta, una práctica utilizada desde la época romana para ocultar información confidencial y garantizar la seguridad de los mensajes. Las funciones básicas de la criptografía son el cifrado y el descifrado. Durante el cifrado, un mensaje sencillo (texto plano) se transforma en una forma ilegible conocida como texto cifrado. En el proceso de descifrado, el texto cifrado se convierte nuevamente en el texto original (texto plano) [2]. Ambas funciones se emplean para proteger el mensaje de personas no autorizadas a acceder a su contenido.

La criptografía se divide en dos ramas principales: criptografía simétrica y asimétrica. La criptografía simétrica (también llamada criptografía de clave privada), se basa en utilizar la misma clave secreta para garantizar una comunicación segura entre el remitente y el receptor, realizando operaciones simples que utilizan esta clave para generar distribuciones pseudoaleatorias de permutaciones. El principal problema es como compartir una clave de forma segura y poder establecer una comunicación entre dos entidades sin tener contacto directo. En cambio, la criptografía asimétrica, o de clave pública, surge para solucionar el problema de distribución de la clave simétrica asegurando la comunicación mediante el uso de un par de claves: una pública y otra privada. La criptografía asimétrica se basa en problemas matemáticos donde las funciones son fáciles de resolver si se dispone del par de claves, sin embargo, resulta exponencialmente inviable calcular el resultado si se desconoce la clave privada.

En la actualidad, los criptosistemas de clave pública, tales como RSA (Rivest, Shamir y Adleman) y la Criptografía de Curva Elíptica (ECC, por sus siglas en inglés), representan los estándares que establecen mecanismos seguros para el intercambio de claves y esquemas de firma digital en nuestra infraestructura de seguridad digital. En estos algoritmos la seguridad depende del problema de factorización de números enteros grandes y el problema del logaritmo discreto en curvas elípticas, respectivamente.

Sin embargo, debido a los avances en el campo de las computadoras cuánticas es probable que estos criptosistemas se vuelvan vulnerables, por el algoritmo de Shor que puede calcular la clave privada eficientemente en un tiempo polinomial [3]. Este

algoritmo se basa en el proceso de búsqueda de períodos mediante la transformada cuántica de Fourier inversa, ésta toma una función $f(x)$ y calcula su período [4]. Lo anterior, implica que es necesario la definición y el diseño de criptosistemas alternativos a la criptografía clásica de clave pública que mantengan la seguridad frente a ataques informáticos tradicionales y garanticen al mismo tiempo la seguridad frente a ataques informáticos cuánticos. Estas nuevas alternativas han sido englobadas en una nueva subárea de investigación, nombrada Criptografía Postcuántica (PQC, por sus siglas en inglés). El objetivo de la PQC es crear sistemas criptográficos que garanticen la seguridad ante ataques clásicos y los nuevos enfoques clásicos [5]. Estos sistemas deben ser viables de ejecutarse en computadoras de arquitectura convencional y dispositivos actuales, además de ser integrables en las redes y protocolos de comunicación que se utilizan en la actualidad.

En este contexto, en diciembre de 2016, el Instituto Nacional de Estándares y Tecnología (NIST, por sus siglas en inglés) inició un proceso para solicitar, evaluar y estandarizar, nuevos algoritmos de clave pública resistente a ataques cuánticos. En particular los Mecanismos de Encapsulación de Claves (KEM, por sus siglas en inglés) y los esquemas de firma digital [6], como parte de sus esfuerzos para promover la seguridad en la criptografía de clave pública y de establecer estándares criptográficos que puedan resistir los ataques tanto de computadoras convencionales como de computadoras cuánticas. En el proceso de estandarización se inscribieron 69 esquemas de cifrado/KEM; los esquemas presentados se basan en Hash, multivariado, isogenias, códigos y *lattices* (LBC, por sus siglas en inglés), entre otras variaciones.

Un KEM permite la transmisión segura, a través de un algoritmo de clave pública de un secreto compartido, que luego puede expandirse para generar claves simétricas, lo cual es más eficiente para la transmisión de mensajes largos con respecto a un esquema PKC [7]. Tras generar un elemento aleatorio del grupo finito subyacente al esquema de clave pública implementado, este elemento se intercambia entre las dos partes de la comunicación. Posteriormente, ambas partes pueden derivar un secreto compartido aplicando una función hash al elemento del grupo finito [8]. Este secreto compartido puede utilizarse como clave para establecer una comunicación segura entre las partes.

En la tercera ronda del proceso de estandarización PQC, el NIST ha seleccionado el algoritmo CRYSTALS-Kyber como estándar del mecanismo de encapsulación de clave con base en *lattices* modulares (ML-KEM, por sus siglas en inglés) [9], por ser la mejor combinación de rendimiento y tamaños de clave, lo que permite que dos partes intercambien la clave fácilmente, así como su velocidad de operación. Se basa en la fortaleza del problema matemático Aprendizaje Modular con Errores (M-LWE, por sus siglas en inglés) [10]. Este esquema también admite la multiplicación eficiente de matriz-vector y vector-vector sobre un anillo polinómico utilizando la Transformada Teórica de Números (NTT, por sus siglas en inglés) [11]. Se han realizado varios tra-

bajos para optimizar la NTT (en software y hardware) desde diferentes perspectivas, como la utilización de recursos, el rendimiento, la eficiencia y el consumo de energía.

Actualmente, el proceso ha avanzado a la cuarta ronda de evaluación, donde se analizan en profundidad BIKE[12] , Classic McEliece[13], HQC[14] con base en código y SIKE[15] con base en isogenia. Dada la diversidad de escenarios que requieren el empleo de primitivas criptográficas. El NIST se propone asegurar que los criptosistemas postcuánticos elegidos puedan ser implementados en la mayor variedad de plataformas informáticas. En este sentido, la eficiencia en las implementaciones de software y hardware, específicamente orientadas a CPU Intel Haswell y FPGA Xilinx Artix-7 [16][17], se convierte en un aspecto crítico en la evaluación de los candidatos postcuánticos del NIST.

En particular, CRYSTAL-Kyber se han realizado varios trabajos para optimizar NTT desde diferentes perspectivas, Yaman et al. en 2021 [18] proponen un diseño implementado en HLS (High-Level Synthesis) y su evaluación comparativa de la transformación teórica de números en criptografía postcuántica con base en *lattice* utilizando el paradigma de hardware/software (HW/SW), demostrando que los gastos generales en términos de utilización de recursos de un arreglo de puertas programables en campo (FPGA, por sus siglas en inglés) son importantes, especialmente en términos del número de tablas de consulta (LUT, por sus siglas en inglés), flip-flops (FF, por sus siglas en inglés) y Slice. Además, Kyber utiliza las primitivas de la familia de funciones SHA-3: SHA3-256 y SHA3-512, así como las funciones de salida extensible (XOF, por sus siglas en inglés) SHAKE128 y SHAKE256 basados en la permutación de Keccak, Dang et al. en 2023 [19] proponen una implementación de Keccak que aprovecha la arquitectura iterativa y realiza 24 rondas en 24 ciclos de reloj.

Aceleración de Funciones Software con FPGA

Un FPGA es un dispositivo semiconductor que se puede programar después de su fabricación para realizar un diseño de aplicación específico, generalmente especificado como un sistema lógico digital [20].

El ciclo de vida de la FPGA incluye dos flujos de diseño: el diseño del arreglo base y el diseño de la aplicación [21]. El diseño de la aplicación también tiene una fase de diseño, que generalmente se realiza con herramientas de los proveedores de FPGA, a menudo complementadas con herramientas comerciales. El desarrollador de aplicaciones integra información del diseño o de la propiedad intelectual de varias fuentes en una aplicación FPGA en código de lenguaje de descripción de hardware (HDL, por sus siglas en inglés) original y reutilizado, bibliotecas del proveedor de FPGA y de otras partes. Las herramientas del proveedor de FPGA compilan el diseño de la aplicación en un flujo de bits, la programación del conjunto base de FPGA para realizar la función de la aplicación.

Además, gracias al avance de la tecnología, hoy se cuenta con dispositivos conocidos como Sistemas en un Chip (SoC, por sus siglas en inglés) que integran varios módulos de hardware, incluso procesadores, en un único circuito integrado. Este tipo de dispositivos reducen el tiempo de las comunicaciones entre los componentes y permite un mayor acoplamiento. Actualmente, es común encontrar chips que cuentan con lógica programable interconectada con microprocesadores duros [1] y otros componentes. Esto permite implementar soluciones enfocadas al paradigma del codiseño HW/SW que tiene como propósito obtener lo mejor de ambos mundos, aprovechando la versatilidad del software y la eficiencia en la concurrencia del hardware. Para una aplicación dada, el circuito en hardware específico se encarga de ejecutar la porción que consume el mayor tiempo de ejecución, mientras que el procesador convencional ejecuta la parte restante, de este modo se consigue mejorar el desempeño de tal aplicación.

1.1. Planteamiento del problema

El enfoque propuesto en esta tesis, implica la implementación de un codiseño HW/SW con VHDL y C, que ofrece una posible mejora en algún aspecto de rendimiento (tiempo, área, etc.) al abordar directamente en un HDL la optimización de algoritmos de criptografía postcuántica. Esta tesis se planea realizar el análisis del algoritmo postcuántico finalista en el concurso del NIST CRYSTALS-Kyber para identificar las operaciones más complejas y realizar un diseño e implementación en VHDL de éstas e integrarlas en un codiseño HW/SW.

Se plantea utilizar una arquitectura de sistema FPGA para optimizar el rendimiento de la ejecución de los algoritmos poscuánticos con respecto a su implementación puramente en software. Esto puede lograrse mediante la conexión de un FPGA adicional a una CPU de alto rendimiento o mediante el uso de un FPGA integrado en un SoC, como el Zynq-7000 . La aceleración resultante generalmente conlleva una significativa reducción en el consumo de energía por operación y beneficios en el tiempo de ejecución.

La pregunta de investigación de esta tesis es: ¿Se podrá mejorar el rendimiento actual de las implementaciones del algoritmo postcuántico CRYSTALS-Kyber mediante una estrategia de codiseño hardware/software realizado en VHDL/C?

Finalmente se puede formular la siguiente hipótesis:
Con un codiseño HW/SW se podrá mejorar el desempeño de las operaciones en el algoritmo postcuántico CRYSTALS-Kyber en comparación con el estado del arte existente.

[1] Nos referimos a un microprocesador duro como un procesador comercial, normalmente propietario, que se integran con la estructura de la FPGA en el mismo chip.

1.2. Objetivos

General

Diseñar e implementar un codiseño HW/SW para hacer operaciones eficientes de manera que se mejore el tiempo de ejecución en el algoritmo de criptografía postcuántica CRYSTALS-KYBER.

Particulares

1. Identificar las operaciones en el algoritmo de criptografía poscuántica CRYSTALS-KYBER que su rendimiento se pueda beneficiar de una implementación en hardware.

2. Diseñar e implementar una partición eficiente del HW/SW para mejorar el desempeño en el algoritmo de criptografía postcuántica CRYSTALS-KYBER.

3. Evaluar el desempeño de la nueva solución híbrida contra el estado del arte existente, usando métricas como el tiempo de ejecución y recursos de hardware.

1.3. Propuesta de solución

Dado que el esquema CRYSTALS-KYBER es relativamente nuevo en el ámbito de la criptografía postcuántica, la comunidad criptográfica continúa buscando formas eficientes de implementar estos algoritmos. La mayoría de las implementaciones actuales hacen uso de herramientas que no describen directamente el hardware, lo cual genera un costo adicional al momento de llevar estas implementaciones al nivel físico. Esto motivó el presente proyecto de investigación, que se enfocó en desarrollar una solución de bajo nivel que permita aprovechar al máximo los recursos del hardware en entornos heterogéneos.

El proyecto de investigación tuvo como propósito mejorar el rendimiento de operaciones computacionalmente costosas en el algoritmo de criptografía postcuántica, haciendo uso de aceleradores de hardware. Para lograr este propósito, se llevó a cabo un codiseño HW/SW (ver figura 1.1) mediante la implementación en los lenguajes C y VHDL. La meta fue contribuir con una implementación eficiente, especialmente diseñada para entornos heterogéneos. Como parte de este esfuerzo, se generó una documentación detallada que abarca tanto la arquitectura como el funcionamiento de los aceleradores desarrollados en este diseño.

Con estos resultados, se contribuye significativamente al proceso de estandarización y evaluación que el NIST está realizando en el ámbito de los algoritmos postcuánticos. Esto se alcanzó mediante una implementación HW/SW que consideró aspectos

críticos como el consumo de potencia y el tamaño del hardware. Esta propuesta exhibe un rendimiento competitivo frente al estado del arte existente, dado que las operaciones se ejecutan en hardware.

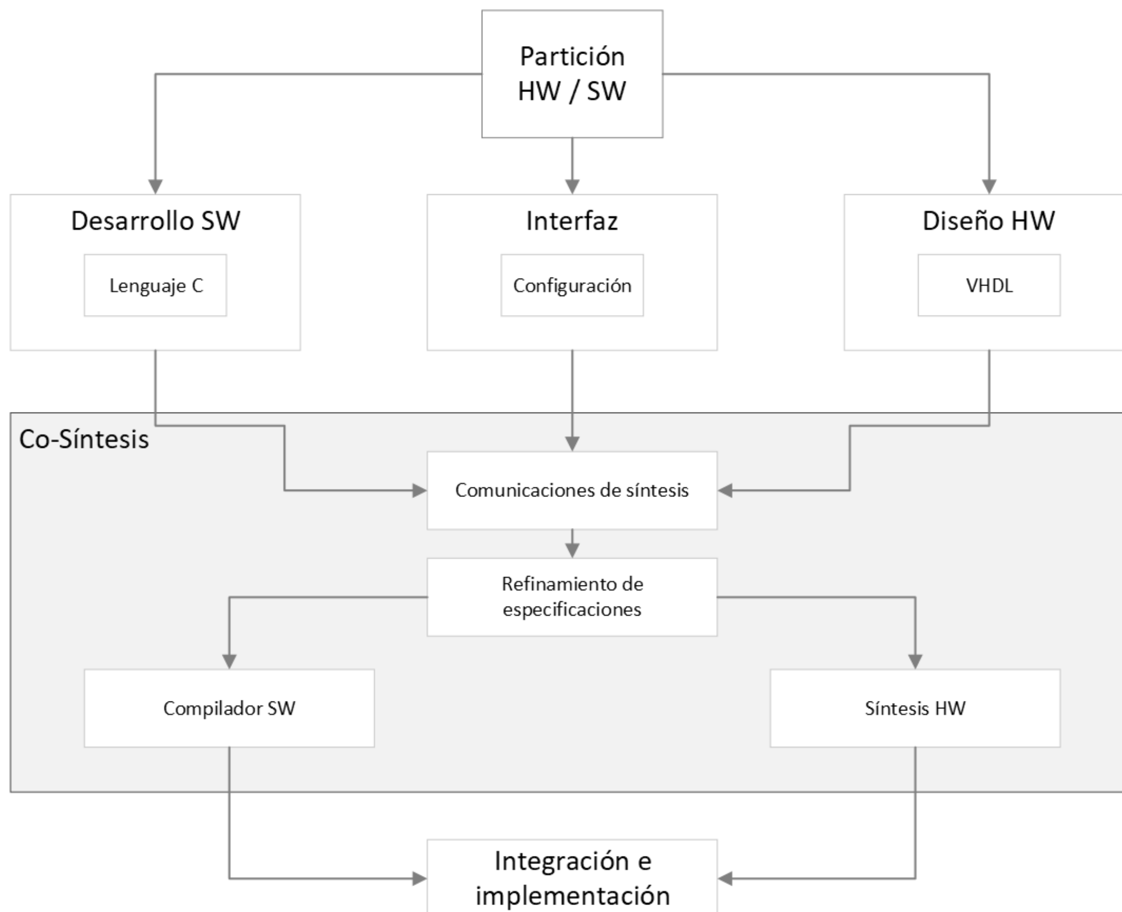


Figura 1.1: Propuesta de solución.

1.4. Organización de la Tesis

En el capítulo 2 se detallan los conceptos clave relacionados con la integración del sistema, así como las herramientas y tecnologías empleadas. Los fundamentos teóricos de criptografía postcuántica, así como la descripción de CRYSTALS-Kyber se exponen en el capítulo 3. Además se detallan las funciones de CRYSTALS-Kyber que se implementarán en hardware. El capítulo 4 describe la implementación del diseño propuesto, abordando tanto los aspectos de hardware como de software utilizados para alcanzar los objetivos. Los resultados obtenidos se presentan en el capítulo 5, incluyendo su análisis y comparación con otros trabajos previos en el área con respecto a los tiempos de ejecución del hardware y la utilización de recursos de la FPGA. Finalmente, en el capítulo 6 se discuten los resultados obtenidos, destacando las con-

tribuciones de la tesis, las conclusiones principales y las posibles líneas de trabajo futuro.

Capítulo 2

Sistema en un Chip

Un FPGA es un dispositivo de hardware que permite realizar diseños de circuitos integrados personalizados, configurándose mediante software para ejecutar funciones específicas. Su arquitectura reconfigurable facilita la creación de diseños de hardware adaptables, ideales para aplicaciones que requieren alta velocidad y procesamiento en paralelo. A medida que la tecnología FPGA ha avanzado y se ha adoptado de manera masiva, especialmente en la implementación y aceleración de algoritmos, se ha posibilitado la creación de sistemas integrados en un único chip. De esta evolución surge el concepto de SoC. El diseño se fundamenta en la reutilización de componentes previamente diseñados, módulos o núcleos IP complejos, así como arquitecturas predeterminadas, todos integrados en un solo dispositivo para la implementación de un sistema electrónico completo [22]. El auge de los SoC está directamente relacionado con la rápida reducción del costo de oportunidad asociado a la integración de aceleradores. Las plataformas más novedosas incluyen arquitecturas con procesadores multinúcleo, unidades de procesamiento gráfico de propósito general, unidades de procesamiento de inteligencia artificial y una amplia gama de aceleradores de hardware especializados.

Xilinx-AMD fue pionera en el desarrollo de los FPGAs en la década de 1980, al ofrecer dispositivos reconfigurables capaces de adaptarse a múltiples aplicaciones. Sus gamas Virtex, Kintex y Artix están diseñadas para satisfacer diferentes necesidades de rendimiento y eficiencia, abarcando desde aplicaciones de bajo consumo hasta tareas de misión crítica en sectores como telecomunicaciones y centros de datos. Entre sus innovaciones se encuentra la familia Zynq, que combina un FPGA con un SoC integrado. La familia Zynq™ 7000 integra un sistema basado en ARM® Cortex™-A que contiene un sistema de procesamiento (PS, por sus siglas en inglés) y lógica programable (PL, por sus siglas en inglés) utilizando la arquitectura de los FPGA de Xilinx-AMD de 28 nm en un único dispositivo. Las CPU ARM Cortex-A9 son el corazón del PS que también incluyen memoria en chip, interfaces de memoria externa y un vasto conjunto de interfaces de conectividad periférica. Esta arquitectura permite ejecutar software en el procesador ARM y, simultáneamente, implementar lógica personalizada en el FPGA. Esto la convierte en una solución ideal para diversas apli-

caciones en específico aquellas requieran un procesamiento eficiente y personalizado de los datos.

2.1. Arquitectura Zynq

El diseño de un SoC se basa en aprovechar componentes previamente diseñados, como módulos o IP cores complejos, e incluso arquitecturas preestablecidas, integrándolos en un único dispositivo para implementar un sistema electrónico completo. Un SoC como el de la familia AMD Zynq™ 7000 aprovecha los recursos ya diseñados al integrar componentes como CPU, Procesador de señales digitales (DSP, por sus siglas en inglés), interfaces de entrada/salida, entre otros, optimizando el rendimiento y reduciendo el espacio y consumo de energía en comparación con sistemas que solo contienen la parte de la PL. La capacidad de hardware configurable en el FPGA permite que el SoC se adapte a necesidades específicas de procesamiento y aceleración, con la comodidad de ser controlado a través de la programación de un ARM. Zynq está formado por un ARM Cortex-A9 de doble núcleo de 32-bits y un FPGA tradicional. La interfaz de la interconexión de los distintos elementos de la arquitectura Zynq se basa en el estándar AXI, que proporciona conexiones de gran ancho de banda y baja latencia.

La figura 2.1 ilustra los bloques funcionales de la arquitectura Zynq-7000. El PS y la PL están en dominios de alimentación separados, lo que permite al usuario de estos dispositivos apagar la PL para la gestión de la alimentación si es necesario. Los procesadores del PS siempre arrancan primero, esto significa un enfoque centrado en el software para la configuración de la PL. La configuración de la PL se gestiona mediante software que se ejecuta en la CPU, por lo que arranca de forma similar a un ASSP[1].

2.1.1. PS

La inclusión del procesador duro (es decir que procesador físico que está integrado) en Zynq ofrece importantes mejoras de rendimiento, simplificando el sistema a un solo chip. Proporciona las capacidades de procesamiento de rutinas de software y/o cargar un sistema operativo [23]. El PS ofrece el entorno de ejecución necesario para ejecutar aplicaciones, reduciendo así el costo global y el tamaño físico del dispositivo.

Como se muestra en la figura 2.1, el PS consta de cuatro bloques principales [23]:

- Unidad de procesamiento de aplicaciones : Esta compuesta de dos procesadores de 32-bits ARM Cortex-A9, cada uno con unidades computacionales asocia-

[1] Producto Estándar Específico de Aplicación, por sus siglas en inglés. Es un circuito integrado diseñado para realizar funciones específicas en aplicaciones determinadas, que se produce en masa y es estándar

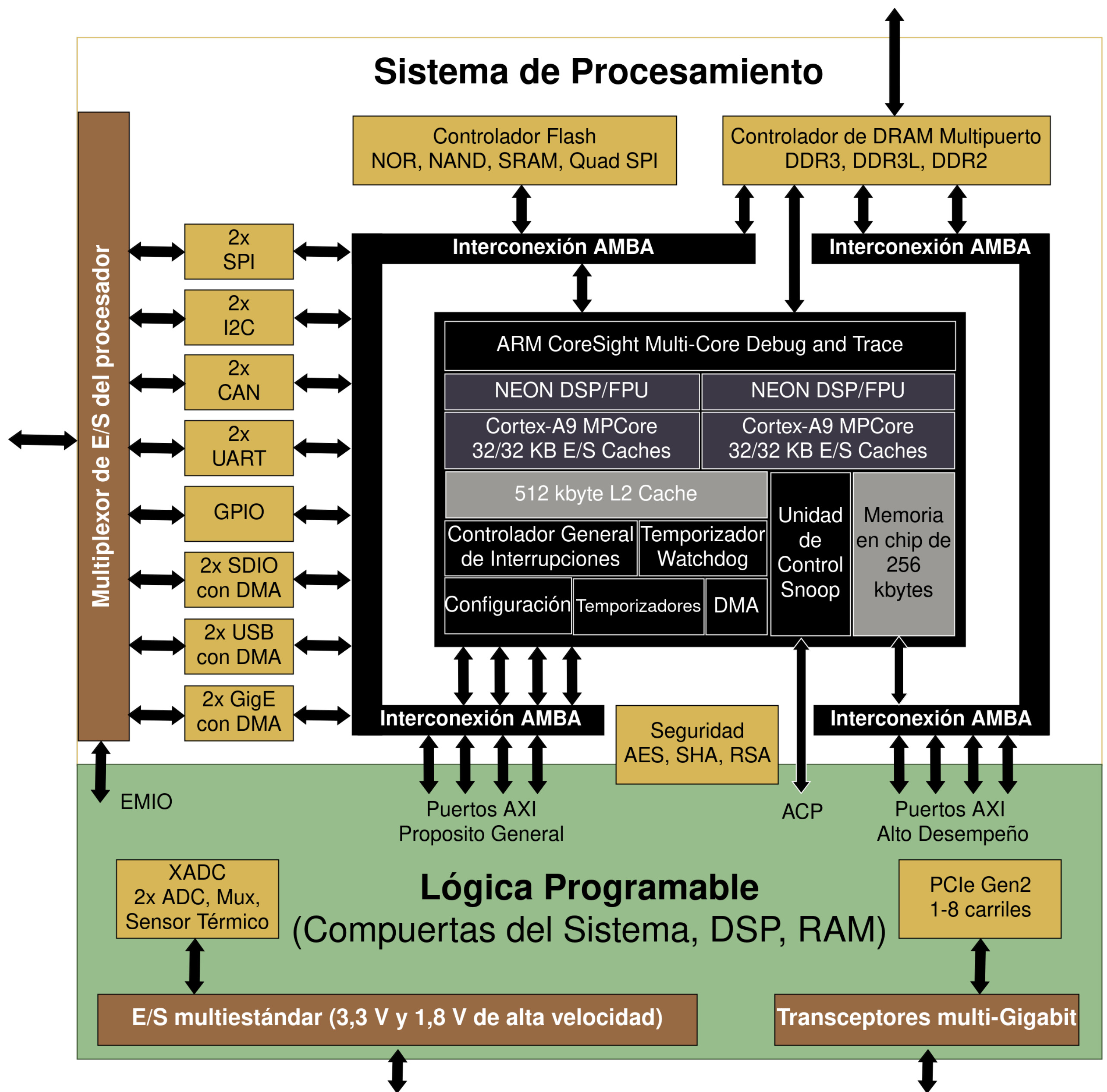


Figura 2.1: Arquitectura Zynq.

das: un motor de procesamiento de medios NEON [2] y una Unidad de Punto Flotante, una Unidad de Gestión de Memoria, y una memoria caché de Nivel 1 (dividida en dos secciones instrucciones y datos). La APU también contiene una memoria caché de Nivel 2 y una memoria adicional en chip. Finalmente, una Unidad de Control de Snoop [3] forma un puente entre los núcleos ARM y

[2]Proporciona capacidades de Instrucción Única sobre Múltiples Datos para habilitar la aceleración de algoritmos de tipo multimedia y DSP.

[3]Es el puente de comunicación entre los núcleos ARM y las memorias: caché de nivel 2 y el OCM.

las memorias caché de Nivel 2 y OCM, esta unidad también tiene cierta responsabilidad en la interfaz con la PL, que no se muestra aquí. Utiliza las interfaces de comunicaciones estándar, y como la Entrada/Salida de Propósito General .

- Interfaces de memoria: incluye un controlador de memoria dinámica y módulos de interfaz de memoria estática. El controlador de memoria dinámica es compatible con los tipos de memorias DDR3, DDR3L, DDR2 y LPDDR2. Los controladores de memoria estática soportan una interfaz de flash NAND, una interfaz de flash Quad-SPI.
- Periféricos de E/S: los cuales se comunican con la Entrada/Salida Multiplexada.
- Interconexión: la interconexión en el PS facilita transacciones de lectura y escritura entre clientes maestro y esclavo mediante canales AXI. Como parte de los buses ARM AMBA, depuración y monitoreo, y permite que los controles maestros de la PL accedan a rutas de datos de alta velocidad y coherentes con caché.

2.1.2. FPGA

En un SoC, la lógica programable se utiliza para referirse a la parte del chip que es programable y configurable, permitiéndonos implementar lógica personalizada y funcionalidades específicas en el hardware, adaptándolo para cumplir con requisitos particulares de diseño. La sección PL es ideal para implementar lógica de alta velocidad, aritmética y subsistemas de flujo de datos. La PL está compuesto predominantemente por una estructura lógica de un FPGA de propósito general, que incluye *slices* y Bloques Lógicos Configurables (CLB, por sus siglas en inglés), además de Bloques de Entrada/Salida para la interfaz (todos los términos de los componentes de la PL son específicos de las arquitecturas de Xilinx-AMD)[23].

Un FPGA es una tecnología de reciente desarrollo que se utiliza para sintetizar cualquier tipo y número de funciones lógicas, además de las aritméticas. Un chip de silicio prefabricado con una matriz de células lógicas e interconexiones, sobre el que los usuarios especifican sus propios diseños [24]. Hoy, los FPGA se usan para crear prototipos de algoritmos y verificar la solución antes de fabricar el prototipo final en los chips ASIC, sintetizarlos en silicio. Como tipo de hardware reconfigurable, pueden modelar enormes tamaños de algoritmos matemáticos que normalmente se implementarían por software, mas con una mayor densidad y velocidad, al utilizar su gran capacidad arquitectónica compleja y su potencialidad de ser paralelizado [25].

En general la lógica que contiene el PL es la siguiente: cuatro LUTs, ocho FFs y una matriz de conmutación, bloques de E/S (IOB, por sus siglas en inglés) e interconexiones programables. Además, la memoria de bloque distribuida, fuentes de reloj y varios elementos de hardware de propiedad intelectual. La terminología de

También de encarga de la interfaz con la PL.

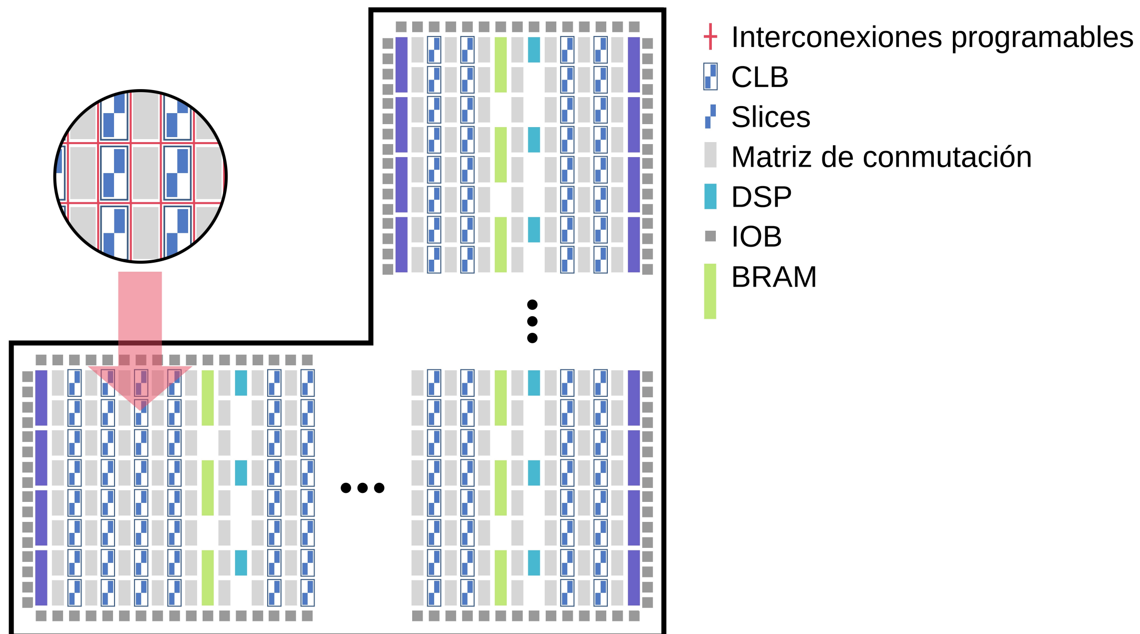


Figura 2.2: La estructura lógica y sus elementos constitutivos.

los elementos que Xilinx-AMD utiliza es la siguiente [23], la esquematización de los elementos se observa en la figura 2.2:

- CLB: Son un grupo de elementos lógicos que están sobre un arreglo bidimensional en la PL, están conectados entre sí y a otros elementos. Cada CLB contiene un dos slices. Ver el diagrama de la figura 2.3.
- *Slice*: Es una subunidad de un CLB, la cual contiene recursos para implementar lógica combinacional y secuencial. Están compuestas de cuatro LUTs, ocho FFs y otra lógica.
- LUT: Es un recurso capaz de implementar:
  - Una función lógica de seis entradas.
  - Una pequeña Memoria de Solo Lectura (ROM, por sus siglas en inglés).
  - Una pequeña Memoria de Acceso Aleatorio (RAM, pos sus siglas en inglés).
  - Un registro de corrimiento.

  Las LUTs en combinación puede implementar funciones con un mayor número de entradas, mayor cantidad de memoria, registros más grandes, como se requiera.
- FF: Es un circuito secuencial que implementa un registro de un bit, con la funcionalidad de reinicio (*reset*). Opcionalmente se puede utilizar como un *latch*.
- Matriz de conmutación: Se encuentra entre cada CLB y proporciona la facilidad de conectar CLBs entre sí y CLBs con otros recursos de la PL.
- Lógica de acarreo: Se encarga de propagar señales inmediatamente entre slices contiguos. Está formada por multiplexores.

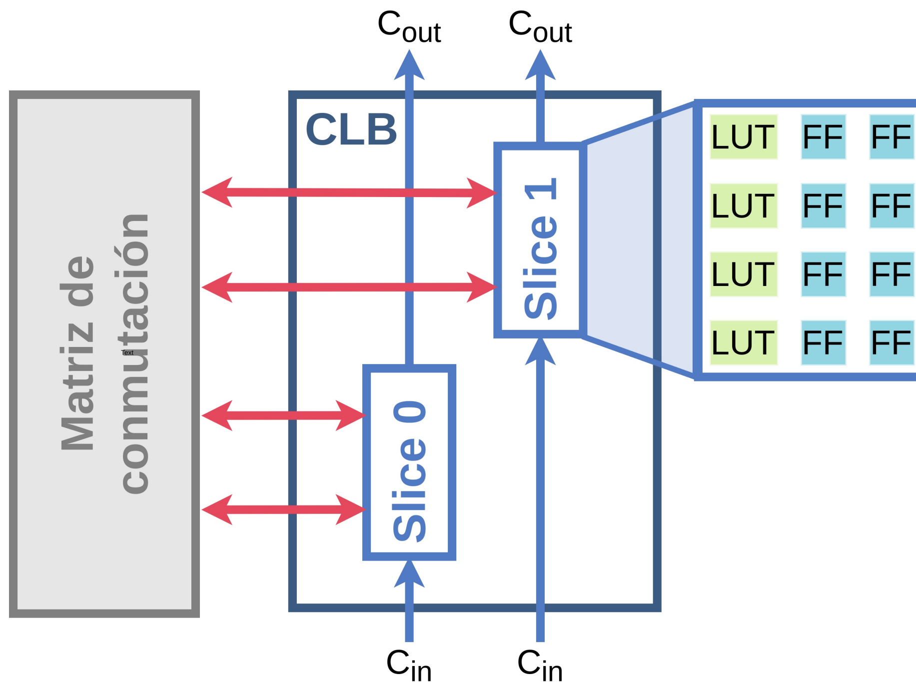


Figura 2.3: Composición de un Bloque Lógico Configurable.

- IOB: Hacen la interfaz entre lo recursos de la PL y los dispositivos físicos del dispositivo. Cada IOB puede controlar un bit de una señal de entrada o salida.
- Bloque de RAM (BRAM): Es utilizada para aplicaciones que requieran una gran densidad de memoria. Pueden ser configuradas como RAM, ROM, buffers de Primero en Entrar, Primero en Salir (FIFO, del inglés, First In, First Out), mientras da soporte a los códigos correctores de errores.
- DSP: Son utilizados para operaciones aritméticas y lógicas de alta velocidad.

Es especifico un BRAM puede almacenar hasta 36Kb de información que puede ser configurado como una unidad de 36Kb o dos unidades de 18Kb. El tamaño de palabra por defecto es de 18-bits, la cual puede ser reorganizar en otros tamaños. Cuando se requieran una mayor cantidad de memoria, las herramientas de Xilinx-AMD se encargan de interconectar varios bloques de memoria. Este tipo de elemento esta optimizado para realizar las operaciones almacenamiento en comparación de utilizar LUTs para implementar una memoria [23].

Las LUTs en el FPGA son utilizadas para implementar operadores aritméticos de tamaño arbitrario, sin embargo, son mejores para las operaciones con tamaños cortos de palabras. Conforme se aumenta el tamaño de palabra de las operaciones implementadas, el área de los recursos utilizados aumenta resultando en un circuito con una frecuencia baja en el reloj [23].

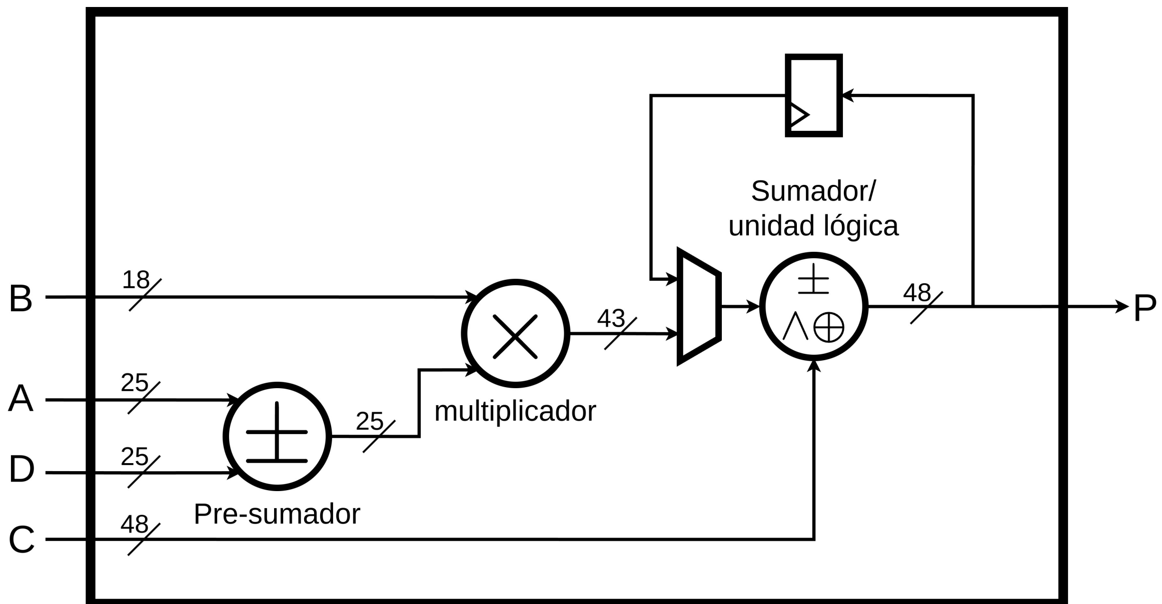


Figura 2.4: Capacidades aritméticas del DSP48E1, encontrado en la familia Zynq-7000.

Para realizar los cálculos aritméticos de tamaños de palabra mayores con una mayor eficiencia y a una frecuencia mayor, se utilizan los DSP. Son recursos de silicio dedicados que generalmente tienen los siguientes operadores aritméticos pre-sumador/restador, seguido de un multiplicador y finalizando con otra unidad que realiza la suma y restas, más las funciones lógicas (*NOT bit a bit*, AND, OR, NAND, NOR, XOR y XNOR), tal como se muestra en el diagrama de la figura 2.4. Es posible realizar varios cálculos que involucran uno, dos o todos estos operadores aritméticos. Se observa en la figura 2.4 que las entradas están etiquetadas como A, B, C y D, y que la salida está etiquetada como P. La unidad puede calcular las funciones $P = (A+D)\times B$, o $P = P'+C$, entre otras. También es capaz de realizar procesamiento SIMD, implementando 2 o 4 operaciones más cortas de suma/resta/acumulación de 24 o 12 bits, respectivamente [23].

Los nombres de los elementos anteriores de la PL mencionados son exclusivos de Xilinx-AMD, no obstante, los demás fabricantes de dispositivos FPGA tienen su propia nomenclatura y dichos elementos realizan tareas similares a los de Xilinx-AMD. Además de Xilinx-AMD, otros fabricantes tienen sus desarrollos exclusivos de FPGAs como: Intel (Altera), Microsemi (Actel) y Lattice Semiconductor, por mencionar algunos. Estos fabricantes siguen siendo los principales actores del mercado [24]. En general los FPGAs están compuestos de bloques lógicos (LUTs) que llevan a cabo las funciones lógicas, flip-flops que almacenan datos y se encargan de la implementación de circuitos secuenciales, al igual que bloques de entrada y salida que se encargan de la comunicación con elementos externos. Todos estos elementos se encuentran en un arreglo matricial configurable para implementar un diseño personalizado. La fi-

gura 2.5 muestra la arquitectura interna de la FPGA. Consta de dos componentes básicos: CLB, que contiene: FFs, LUTs y RAM y la red de interconexión.

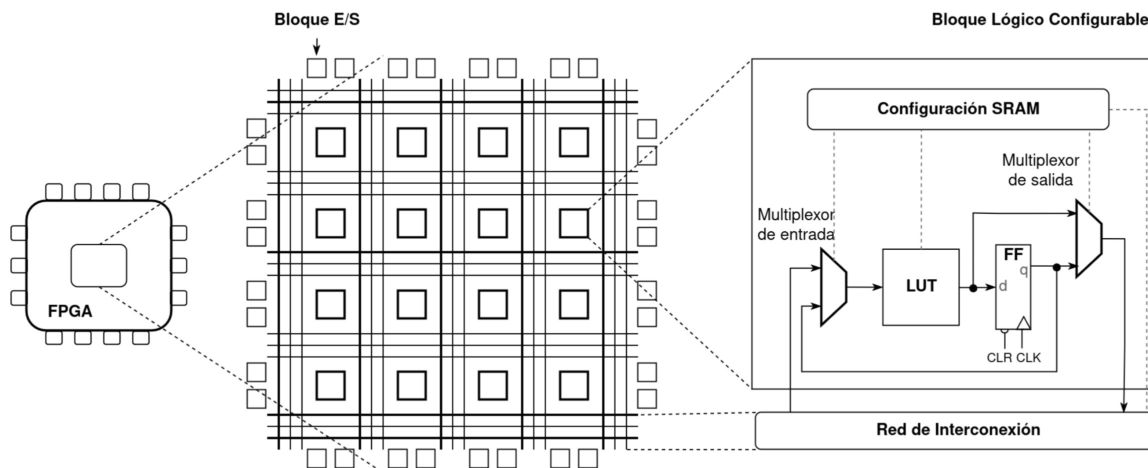


Figura 2.5: Arquitectura interna de una FPGA de Xilinx.

2.2. Lenguajes

Para configuración del SoC utilizaremos un lenguaje de programación para nuestro software y un HDL. La combinación de ambos permite aprovechar las fortalezas de cada uno logrando una integración más fluida entre el hardware y el software, ya que cada uno puede ser diseñado y probado para trabajar en conjunto de manera óptima.

2.2.1. C Embebido

La utilización de C para la programación del PS por su popularidad es conveniente para desarrollar software que debe ejecutarse con rapidez y eficacia. Se pueden extender y adaptar las capacidades de C para un dispositivo en específico haciendo uso de las bibliotecas [23]. En los sistemas embebidos, opera dentro de un espacio de memoria limitado y debe ser optimizado para aprovechar al máximo los recursos disponibles. Además, C ofrece a los programadores de sistemas embebidos un gran control directo del hardware sin sacrificar las ventajas de los lenguajes de alto nivel [26], lo que lo convierte en una opción ideal para el desarrollo de sistemas embebidos.

Las herramientas de desarrollo de los FPGA son capaces de programar el SoC o el procesador suave (un procesador implementado en la PL). Las herramientas simplifican y abstraen el desarrollo de las plataformas al contener las bibliotecas que

contienen las funciones de control de hardware, manejo de memoria, control de eventos, protocolos de comunicación, manejo de interrupciones y temporización para tarea de tiempos real, control de los dispositivos periféricos, así como el control de módulos de hardware como la aritmética de punto flotante. En general las bibliotecas proporcionan direcciones de memoria específicas y funciones predefinidas para controlar periféricos, lo que simplifica la manipulación de registros y el acceso a recursos del sistema. En específico para el SoC que tiene los núcleos ARM de Zynq, las bibliotecas realizan el control de la FPU y la unidad vectorial NEON. Las herramientas de desarrollo de software hacen la compilación cruzada, es decir que el código C se compila para una arquitectura específica distinta a la máquina que contiene las herramientas de desarrollo, en este caso se utiliza el compilador de ARM.

La combinación del PS con la PL crea oportunidades únicas para agregar periféricos y coprocesadores personalizados al PS. La lógica personalizada implementada en la PL puede utilizarse para acelerar funciones de software críticas en tiempo, reducir la latencia de la aplicación, disminuir el consumo de energía del sistema o proporcionar características de hardware específicas para la solución.

AMD proporciona herramientas de diseño para desarrollar y depurar aplicaciones de software para dispositivos SoC Zynq 7000, que incluyen un IDE[4] de software, una toolchain de compilador basada en GNU[5], un depurador JTAG [6] y utilidades asociadas. Estas herramientas permiten crear tanto aplicaciones bare-metal, que no requieren un sistema operativo, como aplicaciones para Linux de código abierto y aplicaciones con sistemas operativos de tiempo real. La lógica personalizada y el software del usuario pueden ejecutarse en diversas combinaciones de hardware físico, con la capacidad de monitorear eventos de hardware. Por ejemplo, se puede tener lógica personalizada corriendo en hardware y el software de usuario puede ejecutarse en el dispositivo objetivo[27].

2.2.2. VHDL

los FPGA se programan utilizando HDL, como Verilog y VHDL. La popularidad del uso de HDL para diseñar circuitos digitales comenzó a mediados de los noventa, cuando aparecieron las herramientas comerciales de síntesis [28]. Dos de los HDL más

[4]Entorno de Desarrollo Integrado, por su traducción del inglés, es una aplicación que proporciona herramientas y funcionalidades integradas, como editores de código, depuradores y compiladores, para facilitar el desarrollo de software en un entorno unificado.

[5]Es un sistema operativo de código abierto que proporciona herramientas y software libre, incluyendo un compilador, que permite a los desarrolladores crear y ejecutar aplicaciones en diversas plataformas.

[6]Joint Test Action Group, en inglés, es un estándar de comunicación que permite la depuración y prueba de circuitos integrados, proporcionando acceso a los registros internos y la capacidad de cargar software en dispositivos como microcontroladores y FPGAs

utilizados hoy en día por muchos ingenieros son VHDL y Verilog. Ambos permiten a los ingenieros especificar el comportamiento y la estructura de circuitos integrados, facilitando la simulación y síntesis de diseños para su implementación en hardware.

VHDL, acrónimo de VHSIC Hardware Description Language (Lenguaje de Descripción de Hardware VHSIC) y VHSIC, a su vez, de Very High Speed Integrated Circuit (Circuito Integrado de Muy Alta Velocidad), fue patrocinado y desarrollado conjuntamente por el Departamento de Defensa de EE.UU. y el IEEE a mediados de la década de 1980. Con el diseño en VDHL se genera un archivo de flujo de bits para convertir la información codificada en la configuración de la FPGA. Como la configuración se almacena en RAM, cuando no hay alimentación, la configuración se pierde. Por lo tanto, debe configurarse cada vez que se enciende.

Con VHDL se pueden ejecutar algoritmos en paralelo, al diseñarlos en hardware. En comparación con los procesadores que ejecutan instrucciones de forma secuencial, al realizarlo en hardware se puede aprovechar de una paralelización a nivel de instrucciones y datos. Se utiliza un sintetizador para traducir el código fuente a una descripción del circuito hardware real que implementa el código. A este paso de síntesis la descripción en VHDL se transforma a Nivel de Transferencia de Registros (RTL, por sus siglas en inglés). Este es un nivel de abstracción, que especifica el flujo de datos entre registros y las operaciones que se realizan sobre esos datos. Permite describir de manera clara y concisa el comportamiento del hardware. Este nivel facilita la simulación y síntesis de diseños para su implementación en circuitos integrados. Posteriormente la descripción en RTL se mapea a recursos físicos del FPGA que se le conoce como netlist. Con el netlist se puede implementar el dispositivo digital físico que realiza el algoritmo diseñado. También es posible realizar una simulación funcional y temporal precisa del código para comprobar la correctitud del circuito.

2.3. Herramientas

Para desarrollar un SoC, Xilinx-AMD ofrece una serie de herramientas y recursos que facilitan el diseño y la implementación. Una de las herramientas principales es Vivado Design Suite, que permite diseñar sistemas completos, desde la especificación de hardware, la síntesis con VHDL en un FPGA, hasta la integración del software en C con Vitis. Esto permite una programación heterogénea, donde partes de la aplicación pueden ejecutarse en el hardware programable para obtener un rendimiento optimizado, mientras que otras partes pueden ejecutarse en los procesadores de aplicaciones para una mayor flexibilidad y facilidad de programación. En esta sección pasaremos a describir las herramientas proporcionadas por Xilinx-AMD para diseñar e implementar nuestros diseños.

2.3.1. Vivado

El entorno de desarrollo AMD Vivado™ ML Design es una suite de software para la síntesis y el análisis de diseños en lenguaje de descripción de hardware (HDL). Es el software de diseño para SoCs y FPGAs adaptables de AMD. Vivado representa una reescritura desde cero y un replanteamiento de todo el flujo de diseño[29]. Incluye: Entrada de diseño, Síntesis, herramientas de Verificación/Simulación.

El ambiente de desarrollo se ilustra en la figura 2.6.

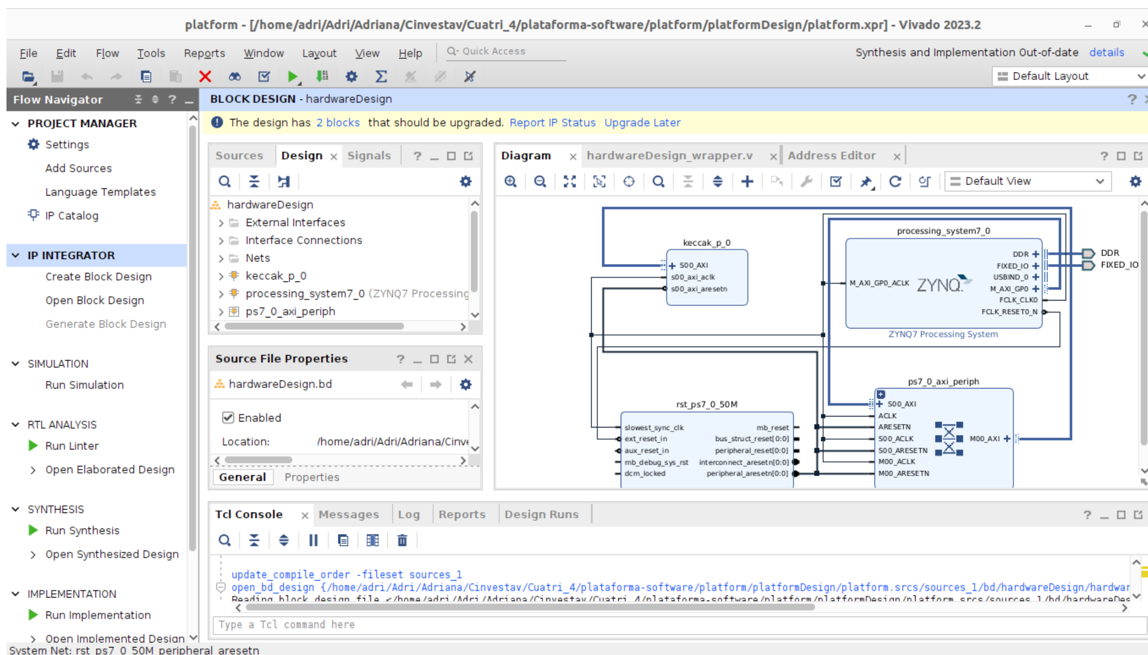


Figura 2.6: Ambiente de desarrollo Vivado design Suite.

Dentro de las principales características de AMD Vivado™ ML Design se encuentran:

- Entrada y ejecución del diseño: Vivado admite la entrada de diseño en VHDL y una herramienta basada en interfaz gráfica de usuario denominada IP Integrator. ofrece los mejores resultados de síntesis e implementación de su clase utilizando funciones avanzadas, incluidos algoritmos de aprendizaje automático, para el cierre de temporización.
- Verificación y depuración: permiten una validación eficaz de la funcionalidad del diseño.
- Intercambio dinámico de funciones: permite a los diseñadores modificar dinámicamente secciones de sus diseños sobre la marcha. Los diseñadores pueden descargar secuencias de bits parciales en sus dispositivos AMD mientras el resto de la lógica sigue funcionando.

2.3.2. Vitis

La adopción de la PS basada en ARM también aporta una amplia gama de herramientas de terceros y proveedores de IP en combinación con el ecosistema PL existente de Xilinx-AMD.

Vitis Unified Software Platform es un entorno de desarrollo de software creado por Xilinx-AMD [30] que simplifica y acelera el desarrollo de aplicaciones para dispositivos SoC de Xilinx-AMD. Las herramientas Vitis funcionan junto con AMD Vivado™ ML Design Suite para proporcionar un mayor nivel de abstracción para el desarrollo de diseños permitiendo a los desarrolladores aprovechar tanto el hardware programable (FPGA) como los procesadores de aplicaciones en un solo flujo de trabajo unificado. El ambiente de desarrollo se ilustra en la figura 2.7.

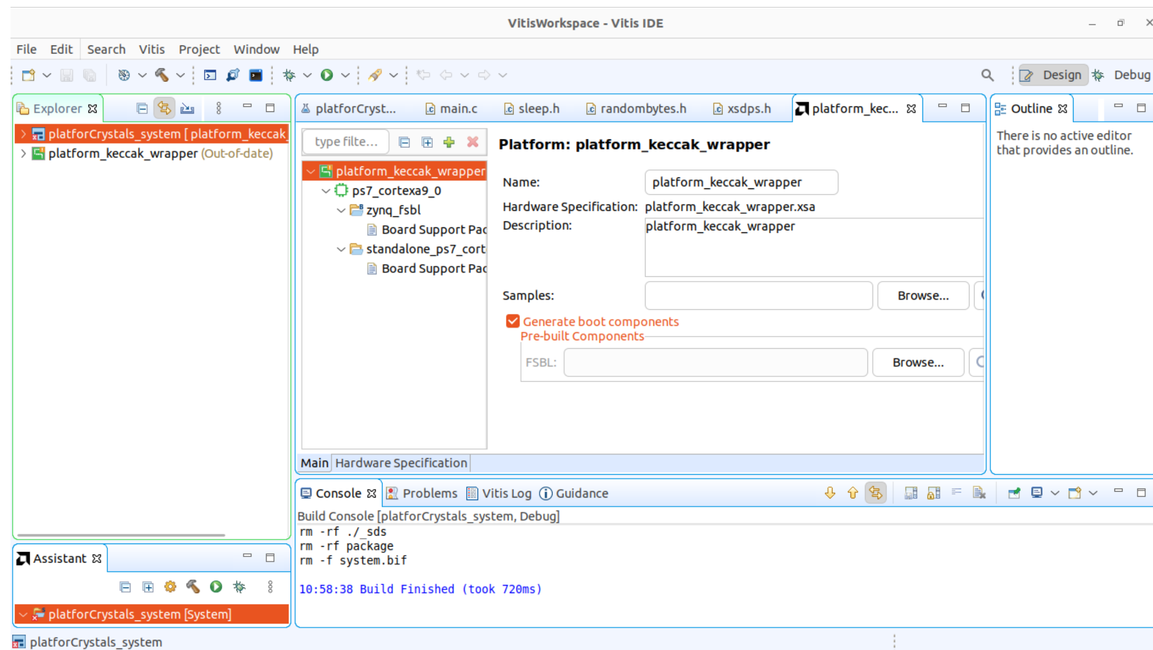


Figura 2.7: Ambiente de desarrollo Vitis IDE.

La plataforma de software Vitis incluye las siguientes herramientas:

- Vitis Embedded: para desarrollar código de aplicación C/C++ que se ejecuta en procesadores Arm integrados.
- Compilador y simuladores: para implementar diseños utilizando la matriz AI Engine.
- Vitis HLS: para desarrollar bloques IP basados en C/C++ que apuntan a la estructura FPGA.
- Vitis Model Composer: una herramienta de diseño basada en modelos que permite una rápida exploración de diseños dentro del entorno MathWorks Simulink®.

- Un conjunto de funciones de biblioteca de código abierto y rendimiento optimizado, como DSP, Vision, Solver, Ultrasound, BLAS y muchas más, que se pueden implementar en la estructura FPGA o utilizando motores de IA.

Capítulo 3

Criptografía Postcuántica

3.1. Antecedentes

Los ordenadores cuánticos son máquinas que se basan en los principios de la mecánica cuántica. La idea de crear las computadoras cuánticas viene de Richard Feynman en 1982 en su artículo "*Simulating Physics with Computers*", donde propone que las computadoras que simulen sistemas físicos complejos, como los de los sistemas cuánticos, deberían ser construidas con los mismos principios para hacer una simulación eficiente.

Las computadoras cuánticas dependen fuertemente de los principios de superposición y en entrelazamiento. La propiedad de superposición es lo que les da el poder, ya que pueden representar múltiples combinaciones a la vez, que se traduce en realizar cálculos en paralelo. Mientras que el entrelazamiento se utiliza para que un estado dependa de otro. Sin embargo, las computadoras cuánticas presentan el problema de la medición, que de una superposición solo se obtiene un resultado o estado a la vez (en mecánica cuántica se dice que la superposición colapsa a un estado base) con cierta probabilidad. Por ello los algoritmos que se ejecuten en las computadoras cuánticas tienen que amplificar las respuestas para que tengan la mayor probabilidad de ser medidas.

A diferencia de las computadoras clásicas, que operan con bits que son 0 o 1, las computadoras cuánticas se basan en bits cuánticos (o qubits) como su unidad de información, físicamente el qubit está representado sistemas de dos estados como iones, fotones, superconductores, puntos cuánticos, entre otras tecnologías. Un qubit representa dos estados pueden ser 0 y 1 simultáneamente, cuando se encuentra en una superposición. Los qubits utilizan la notación de Dirac, para representar los dos estados se tiene $|0\rangle$ y $|1\rangle$. Una superposición general de un qubit esta expresada de la siguiente forma:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

donde los coeficientes α y β, llamados amplitudes, pertenecen al campo de los números complejos $\alpha, \beta \in \mathbb{C}$. El qubit se encuentra en un espacio de Hilbert de dos dimensiones

$\mathcal{H}_2$, el espacio de Hibert es un espacio vectorial complejo que define el producto punto, el producto escalar y una norma. Los sistemas de qubits siguen la condición de normalización es decir que su norma es 1. La norma de la superposición indica la probabilidad que al medir el sistema se obtenga cierto estado, $P(|0\rangle) = |\alpha|^2$ y $P(|1\rangle) = |\beta|^2$. La norma se define de la siguiente manera:

$$|||\psi\rangle||^2 = |\alpha|^2 + |\beta|^2 = 1$$

El poder de las computadoras cuánticas se da al agregar más qubits. El espacio de Hilbert aumenta exponencialmente por cada qubit que se añada al sistema, es decir se tendría un espacio de 2^n dimensiones $\mathcal{H}_{2^n}$, donde n es el número de qubits presentes en el sistema. Esto se traduce a poder representar 2^n distintos valores en superposición y aplicarles operaciones unitarias simultáneamente. El estado en superposición de un sistema de n-qubits se representa con la siguiente ecuación, sus estados son todas las cadenas de 0s y 1s de tamaño n:

$$|\psi\rangle = \sum_{i \in \{0,1\}^n} \alpha_i |i\rangle, \alpha_i \in \mathbb{C}$$

Todos los estados de n-qubits igualmente cumplen la condición de normalización, es decir, su norma es 1:

$$|||\psi\rangle|| = \sum_{i \in \{0,1\}^n} |\alpha_i|^2 = 1$$

Esto indica que todas las operaciones que se lleven a cabo por los algoritmos van a ser unitarias, es decir que el vector del estado solo rota delimitado por la superficie de una hiperesfera de 2^n dimensiones complejas. Al medir el estado este colapsa a una sola base, la probabilidad de que el estado colapse en una base en particular está determinada de la siguiente manera:

$$P(|i\rangle) = |\alpha_i|^2, i \in \{0,1\}^n$$

En 1992 Deutsch y Jozsa introducen el primer algoritmo cuántico y la aplicación de la computación cuántica. Este algoritmo tiene una complejidad $O(1)$ y es capaz de indicar si una función booleana[1] de n entradas produce una salida balanceada o constante. Los algoritmos cuánticos se derivan de la complejidad exponencial de los sistemas cuánticos, al poder realizar 2^n operaciones en un solo paso. Comienzan desde un estado de superposición inicial, manipulan los qubits hasta llegar a una superposición final, se mide y se obtiene un resultado [31]. Los algoritmos cuánticos ajustan la distribución de probabilidad para que un elemento del espacio muestra con la mayor probabilidad sea la solución de un problema [32].

Los algoritmos cuánticos, en general, son aplicables solo a ciertos problemas específicos donde las propiedades cuánticas pueden aprovecharse para mejorar la eficiencia

[1] Una función booleana toma una o más variables de entrada que tienen como valor un elemento del conjunto $\{0,1\}$ y produce una única salida del mismo conjunto $\{0,1\}$, $f : \{0,1\}^n \mapsto \{0,1\}$

de su contraparte clásica. Cuatro de los principales y más famosos algoritmos pueden resolver problemas convenientemente en computadoras cuánticas estos son: el algoritmo de Shor que trata con la factorización de números primos y el logaritmo discreto, el algoritmo de Grover que busca un dato en una lista no ordenada, el algoritmo de HHL (por las siglas de sus creadores Aram W. Harrow, Avinatan Hassidim, y Seth Lloyd) resuelve sistemas lineales de ecuaciones y los algoritmos de optimización cuántica que tratan de encontrar la mejor solución [33].

Específicamente el algoritmo de Shor, publicado en 1994 [34], es un algoritmo cuántico que provoca una aceleración exponencial al resolver problemas de factorización, logaritmo discreto y logaritmo discreto de curva elíptica, con una complejidad $O(\log_2(N)^3)$, donde N es el número por factorizar. El algoritmo de Shor resuelve problemas más generales que la factorización y los logaritmos discretos. Específicamente, si una función $f(x)$ es periódica, es decir, si hay un r (el período) tal que $f(x + r) = f(x)$ para cualquier x, el algoritmo de Shor encontrará r de manera eficiente.

Esto significa que se podría usar una computadora cuántica para resolver cualquier algoritmo criptográfico que dependa de esos problemas, incluidos RSA, Diffie-Hellman, criptografía de curva elíptica y todos los mecanismos de criptografía de clave pública implementados actualmente los cuales su dificultad consiste en la factorización de números grandes, o el cálculo de logaritmos discretos. Como ejemplo en el caso del esquema RSA; su fortaleza se basa en producto de dos primos grandes p y q que generan el número público $N = p \cdot q$. Para romper este esquema el algoritmo de Shor escoge un número a, tal que $1 < a < N$. Posteriormente busca el periodo r de la función $f(x) = a^x \mod N$, que es el menor entero positivo r tal que $a^r \equiv 1 \pmod{N}$. Calcular el periodo de forma clásica es exponencial, debido a que las claves aumentan a razón del número de bits. Finalmente se comprueba el periodo, que necesariamente tiene que ser par y se obtienen los factores de N utilizando el Máximo Común Divisor (MCD, por sus siglas en inglés):

$$\text{factor}_1 = MCD(a^{r/2} + 1, N), \quad \text{factor}_2 = MCD(a^{r/2} - 1, N)$$

IBM implementó exitosamente el algoritmo de Shor por primera vez en su máquina cuántica, en 2001. En un procesador de 7 qubits, la técnica se utilizó para calcular factores del dígito 15. Después, se logró un avance sustancial hacia una máquina cuántica comercialmente factible. La tasa de progreso en esta investigación ha aumentado considerablemente en los últimos años. Para finales de 2025, IBM afirmó que pronto lanzará una computadora cuántica de más de 4000 qubits [35]. La ingeniería de construir una computadora cuántica presenta tres principales retos, es crear ambientes libres de ruido de tal forma que el estado cuántico sea estable y no colapse (decoherencia), reducir las acumulaciones de errores de los qubits y sus operaciones, por último, mejorar la escalabilidad de la potencia de cómputo al agregar más qubits. En la construcción de computadoras cuánticas se implementan qubits a los que se les

denomina físicos, estos no son ideales ya que no son tolerantes a los fallos y decoherencia. Para mantener estos qubits físicos estables se emplean códigos de corrección de errores, usando cientos o miles de qubits físicos para formar un solo qubit lógico. Los qubits adicionales meten redundancia al sistema y pueden detectar y corregir errores a lo largo de la ejecución de un algoritmo [32].

Con forme la tecnología avanza y se descubren nuevas técnicas para estabilizar los qubits, se podrán crear computadoras cuánticas estables y resistentes a fallos. Cuando las computadoras tengan el suficiente poder podrán ejecutar el algoritmo de Shor sin errores y de forma eficiente. Así romperán los esquemas de criptografía pública y podrán factorizar rápidamente los grandes números que protegen las claves privadas de RSA, resolver el logaritmo discreto en Diffie-Hellman y la ECC. Esto significaría que un atacante podría, recuperar el intercambio de claves, descifrar comunicaciones encriptadas, firmar documentos falsos, y suplantar identidades digitales. Podría parecer que las computadoras cuánticas no representan en la actualidad, si no hasta un cierto futuro y que no hay que implementar medidas contra ellas aún, sin embargo, existe la amenaza de “almacena y descifra después”. Esto implica que los datos interceptados y almacenados hoy podrían ser descifrados en el futuro.

Esto hace que se planteen la exploración de esquemas criptográficos alternativos resistentes a los ataques cuánticos, asegurando la seguridad a largo plazo de la información confidencial. Así surge la PQC, que trata del diseño de algoritmos de clave pública que no puedan descifrarse por una computadora cuántica, ni clásica; que sean seguros para la computación cuántica y puedan reemplazar los algoritmos basados en curvas elípticas y RSA en un futuro en el que las computadoras cuánticas listas puedan descifrar módulos RSA de 4096 bits en un instante. Estos algoritmos no deberían depender de un problema difícil que se sabe que puede resolverse de manera eficiente mediante el algoritmo de Shor, que elimina la dificultad de los problemas de factorización y de logaritmo discreto.

Por este motivo, la comunidad científica ha aumentado los esfuerzos por definir nuevos estándares para los protocolos de firma digital, cifrado y establecimiento de claves en los esquemas criptográficos postcuánticos. En el 2016 el NIST convocó un concurso en el que todas las instituciones académicas e industriales podían participar, con el fin de buscar estándares de criptografía alternativos resistentes a los ataques cuánticos, asegurando la seguridad a largo plazo de la información confidencial. El concurso tenía dos categorías firma digital y KEM. Actualmente, el concurso se encuentra en la cuarta ronda, con un algoritmo seleccionado para el KEM y 3 finalistas para la firma digital.

3.2. KEM

Un KEM representa un método para transmitir de forma segura claves secretas a través de canales de comunicación no seguros o comprometidos [36], utilizando técnicas de cifrado de clave pública. Son diseñados para ser seguros en computadoras cuánticas y clásicas, ofrecen una solución a largo plazo para la criptografía segura.

Un KEM consta de tres algoritmos y una colección de conjuntos de parámetros. Los tres algoritmos son [9]:

1. Un algoritmo probabilístico de generación de claves, denominado KeyGen.

2. Un algoritmo probabilístico de encapsulación denominado Encaps.

3. Un algoritmo determinista de desencapsulación denominado Desencaps.

La colección de conjuntos de parámetros se utiliza para seleccionar un compromiso entre seguridad y eficacia. Este proceso permite el intercambio de una clave segura sin una clave secreta compartida preestablecida.

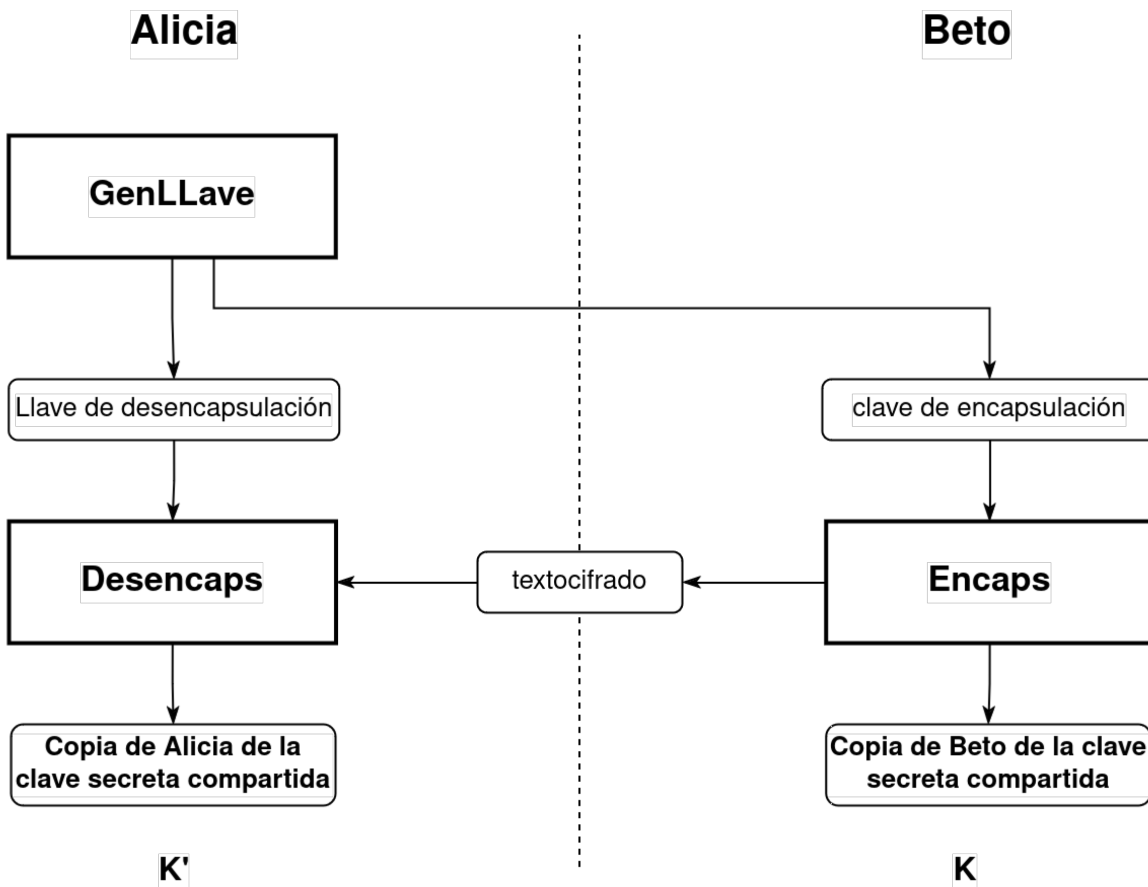


Figura 3.1: Como se realiza el establecimiento de claves usando un KEM.

En un escenario común, un KEM se emplea para que dos partes (llamadas Alicia y Beto) acuerden una clave secreta compartida, como se ilustra en la figura 3.1. Alice comienza generando una clave pública de encapsulación y una clave privada de desencapsulación mediante el algoritmo KeyGen. Con la clave pública de Alice, Bob ejecuta el algoritmo Encaps, lo que le permite obtener su copia de la clave secreta compartida (K) y un texto cifrado correspondiente. Bob luego envía el texto cifrado a Alice, quien utiliza su clave privada junto con el texto cifrado para ejecutar el algoritmo Decaps y generar su propia copia de la clave secreta compartida (K'). Tras este proceso, ambos esperan que K y K' sean iguales, asegurando que han establecido una clave secreta segura y aleatoria.

3.3. Criptografía con base en *lattices*

Hasta el momento, se han creado diversas familias de algoritmos criptográficos postcuánticos, entre las cuales está la LBC, esto debido a que los problemas sobre los que se basan estos algoritmos son difíciles de resolver incluso para las computadoras cuánticas.

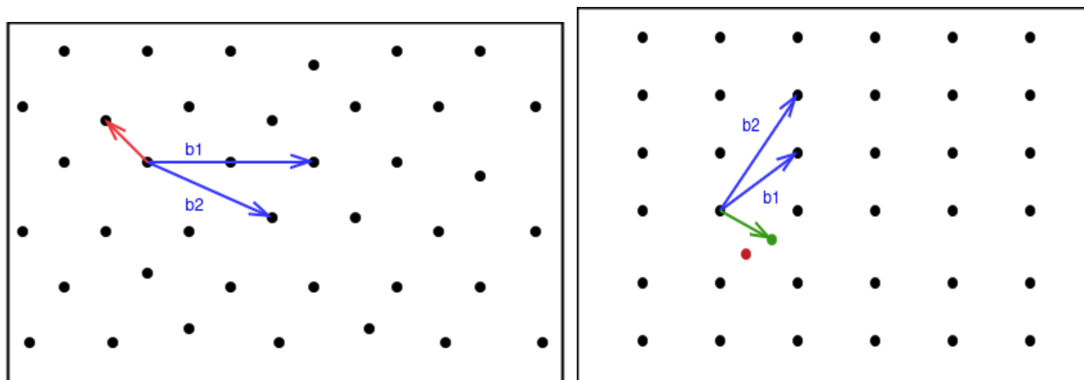
LBC se basa en el hecho de resolver problemas matemáticos difíciles en *lattices* de alta dimensión, conocidos como problema del vector más corto (SVP, por sus siglas en inglés) o problema del vector más cercano (CVP, por sus siglas en inglés) [34]. Éstos son problemas de caso peor, lo que quiere decir que no existe un algoritmo eficiente que resuelva cualquier instancia, aunque puede ser que alguna instancia del problema sea fácil de resolver.

Estos problemas, que se basan en el uso de la norma euclídea en $\mathbb{R}^n$, se definen como sigue:

- **SVP**: Dada una base $\mathbf{B}$, encontrar el vector más corto distinto de cero en la *lattice* $L(\mathbf{B})$ (figura 3.2a).
- **CVP**: Dada una base $\mathbf{B}$ y un vector $\mathbf{t}$, encontrar el elemento del retículo $\mathbf{v} \in L(\mathbf{B})$ más cercano a $\mathbf{t}$ (figura 3.2b).

Estos problemas son difíciles de resolver para todas las instancias, incluso con un ordenador cuántico, cuando n es grande. Esto quiere decir que no existe un algoritmo que los resuelva en tiempo polinomial. Sobre la base de la dificultad de las situaciones de peor caso, los protocolos criptográficos con base en *lattices* suelen tener demostraciones de seguridad muy sólidas. Se ha demostrado que un problema de caso promedio tiene una dificultad al menos igual a la del problema de peor caso [37], que se considera desafiante y cuyas soluciones están correlacionadas con un adversario del protocolo.

La dificultad en el peor de los casos de algunos problemas de *lattice* se ha utilizado para crear esquemas criptográficos seguros. Los tres algoritmos criptográficos basados



(a) SVP (vectores base en azul, vector más corto en rojo).

(b) CVP (vectores base en azul, vector externo en verde, vector más cercano en rojo).

Figura 3.2: SVP y CVP en *lattices*.

en *lattices* más utilizados en la criptografía postcuántica son: aprendizaje con error, aprendizaje con error en anillo y aprendizaje con error en módulo.

3.4. El problema M-LWE

Propuestas anteriores de criptosistemas basados en LWE utilizaban el problema muy estructurado Ring-LWE o LWE estándar [17]. Las variantes estructuradas de LWE basadas en anillos polinómicos proporciona eficiencia en términos de velocidad, tamaños de clave y texto cifrado y a la vez esta estructura adicional podría permitir ataques más eficientes y que las compensaciones entre eficiencia y seguridad solo se pueden escalar de manera bastante burda. Mientras que las ventajas de LWE estándar serían la falta de esta estructura y la fácil escalabilidad, aunque a costa de reducir significativamente la eficiencia. M-LWE ofrece un equilibrio entre estos dos extremos.

El problema de M-LWE establece que: en un Módulo k, para un elemento secreto s y un mensaje M, donde s y $M \in R_q^k$, y $e \in R_q$, al seleccionar cada coeficiente del polinomio R_q al azar y uniformemente de $\mathbb{Z}_q$, la versión de búsqueda de M-LWE establece que: "Supongamos que se nos han dado m muestras de $(M, \langle M, s\rangle + e) \in R_q^k \times R_q$, entonces sería inviable recuperar el secreto s". La versión de decisión de M-LWE establece que: "Supongamos que se nos han dado m instancias de la forma $(M, d) \in R_q^k \times R_q$, entonces sería difícil distinguir si todas las instancias provienen de $(M, \langle M, s\rangle + e)$ o si provienen de una distribución $R_q^k \times R_q$".

Problema M-LWE[38]. Sean $k, m \in \mathbb{Z}_{>0}$ y χ una distribución de probabilidad sobre el anillo R . Dados $(A, b) \in R^{m\times k} \times R^m$ donde:

- A es aleatorio y uniforme en $R_K^{m\times k}$.

- A partir de $s \leftarrow \chi^k$ y $e \leftarrow \chi^m$, se calcula $b = As + e$.

El problema M-LWE consiste en hallar s.

En el caso específico de los parámetros de M-LWE utilizados en KYBER[17], la estructura algo reducida en comparación con Ring-LWE, una escalabilidad mucho mejor y, al cifrar mensajes de un tamaño fijo de 256 bits, un rendimiento muy similar a los esquemas basados en Ring-LWE.

3.5. CRYSTALS-KYBER

En julio de 2022, el Instituto Nacional de Estándares y Tecnología (NIST) seleccionó oficialmente a CRYSTALS-KYBER para la estandarización PQC. KYBER es un algoritmo de LBC que se utiliza como KEM. En comparación con otros esquemas KEM postcuánticos, KYBER ofrece importantes ventajas en términos de eficiencia, al tiempo que garantiza la resistencia contra ataques cuánticos y clásicos conocidos [39].

La seguridad de KYBER se basa en la presunta dureza del llamado problema de M-LWE, que es una generalización del problema de LWE introducido por Regev en 2005. La dureza del problema M-LWE se fundamenta en la supuesta dificultad de ciertos problemas computacionales en *lattices* de módulos [9].

Los procesos de generación de claves, cifrado y descifrado para el algoritmo KYBER son los siguientes:

- **Generación de claves**: $\mathbf{A} \in R_q^{k\times k}$, $\mathbf{s}$ y $\mathbf{e} \in R_q^k$. $\mathbf{pk} = \mathbf{A} \circ \mathrm{NTT}(s) + \mathrm{NTT}(e)$, $sk = \mathrm{NTT}(s)$, donde $\circ$ representa la multiplicación de matrices de polinomios.
- **Encapsulación**: $\mathbf{A} \in R_q^{k\times k}$, $\mathbf{r}$ y $\mathbf{e1} \in R_q^k$, $e2 \in R_q$. Para un mensaje $\mathtt{msg}$, $\mathbf{u} = \mathrm{NTT}^{-1}(\mathbf{A}^T \circ \mathrm{NTT}(\mathbf{r})) + \mathbf{e1}$, $v = \mathrm{NTT}^{-1}(\mathbf{pk}^T \circ \mathrm{NTT}(\mathbf{r})) + e2 + \mathtt{msg}$, donde T representa la operación de transposición.
- **Desencapsulación**: $\mathtt{msg} = v - \mathrm{NTT}^{-1}(\mathbf{sk}^T \circ \mathrm{NTT}(\mathbf{u}))$.

Básicamente, el protocolo KYBER incluye tres algoritmos: generación de claves, encapsulación y descencapsulación. Durante la generación de claves, se toma una muestra de una matriz de una distribución uniforme, mientras que una clave secreta se toma de una distribución binomial. La clave pública se obtiene realizando una multiplicación polinómica entre A y s en el dominio NTT, seguida de la adición de ruido al producto resultante. Para la encapsulación, se combina un mensaje m con el producto de la clave pública y un vector r muestreado aleatoriamente en el dominio normal, lo que da como resultado un vector. Se realiza otra multiplicación polinómica entre r y una matriz distribuida uniformemente, lo que da como resultado una matriz U. La salida del cifrado, denominada texto cifrado ct, se forma comprimiendo tanto U y v. Para descifrar el mensaje, se recupera una aproximación de v calculando el producto de la clave secreta s y U. Estos tres algoritmos se utilizan para crear un

esquema de cifrado de clave pública (PKE, por sus siglas en inglés) que sea seguro contra ataques de texto simple elegido (CPA, por sus siglas en inglés). Además, al emplear una transformación Fujisaki-Okamoto adaptada, se puede construir el KYBER KEM seguro para ataques de texto cifrado elegido (CCA, por sus siglas en inglés).

La construcción de KYBER se realiza en dos fases principales. Primero, se utiliza el problema M-LWE como base para desarrollar un esquema PKE. Luego, este esquema PKE se transforma en un mecanismo de encapsulación de claves mediante la conocida transformación Fujisaki-Okamoto (FO, por sus siglas en inglés) [40]. Gracias a ciertas características de la transformación FO, el KEM resultante es seguro incluso bajo un modelo de ataque mucho más amplio que el PKE inicial. Por lo tanto, se considera que el KYBER cumple con la seguridad conocida como IND-CCA2 [41].

3.5.1. Conjuntos de parámetros de KYBER

KYBER.CPAPKE se define a partir de los parámetros enteros n, k, q, η_1, η_2, d_u y d_v. Los valores de n y q definen el anillo R y el anillo R_q, los cuales se expresan como:

$$\mathbb{Z}[X]/(X^n + 1)$$

$$\mathbb{Z}_q[X]/(X^n + 1)$$

donde $n = 2^{n'} - 1$, de modo que $X^n + 1$ representa el polinomio ciclotómico de orden $2^{n'}$ [41]. Estos parámetros se mantienen constantes.

Todos los polinomios constan de 256 coeficientes sobre vectores de dimensión k, donde k = 2, 3 o 4, que representan los tres niveles de seguridad postcuánticos diferentes. El módulo primo está fijado para los tres niveles de seguridad de KYBER.

- n: los polinomios tienen un grado uniforme de $n = 256$, y los coeficientes pertenecen al campo primo $\mathbb{Z}_q$, siendo $q = 3329$ para todos los niveles de seguridad. Este valor de n se selecciona para alcanzar una encapsulación de claves con 256 bits de entropía: si n fuera menor, sería necesario codificar varios bits de clave en un único coeficiente del polinomio, lo cual exigiría un nivel de ruido inferior y, en consecuencia, menor seguridad. Aumentar n reduciría la flexibilidad para escalar la seguridad a través del parámetro k.
- q es un primo pequeño que satisface n | (q - 1); esto es necesario para habilitar la multiplicación rápida basada en NTT. Hay dos primos más pequeños para los que se cumple esta propiedad, a saber, 257 y 769. Sin embargo, para esos primos no podríamos lograr la probabilidad de falla insignificante requerida para la seguridad de CCA, por lo que elegimos el siguiente más grande, es decir, $q = 3329$
- k es la dimensión de la red como un múltiplo de n. El cambio de k es el mecanismo principal en KYBER para escalar la seguridad (y, como consecuencia, la eficiencia) a diferentes niveles.

- Los parámetros restantes η_1, η_2, d_u y d_v se eligieron para equilibrar la seguridad, el tamaño del texto cifrado y la probabilidad de falla. El parámetro derivado δ, que es la probabilidad de que falle la desencapsulación de un texto cifrado KYBER.CCAKEM válido.
- El parámetro η_1 define el ruido de s y e en el Kyber.CCAKEM.KeyGen() y de r en el Kyber.CCAKEM.Enc(pk). El parámetro η_2 define el ruido de e1 y e2 en el Algoritmo 5.

KYBER ofrece tres niveles de seguridad diferentes, cada uno con su propio conjunto de parámetros de configuración. Cada conjunto de parámetros (ver la tabla 3.1) está asociado con un nivel de seguridad requerido para la generación de aleatoriedad, KYBER512, KYBER768 y KYBER1024, proporcionan niveles de seguridad comparables a AES-128, AES-192 y AES-256, respectivamente [42]. Los tamaños de las claves y los textos cifrados para cada conjunto de parámetros se resumen en la tabla 3.2.

Tabla 3.1: Parametros de KYBER.

	Nivel-NIST	n	k	q	η_1	η_2	(d_u, d_v)	δ
KYBER512	1 (AES-128)	256	2	3329	3	2	(10,4)	2^{-139}
KYBER768	3 (AES-192)	256	3	3329	2	2	(10,4)	2^{-164}
KYBER1024	5 (AES-256)	256	4	3329	2	2	(10,5)	2^{-174}

Tabla 3.2: Tamaño en bytes de las llaves y el texto cifrado de KYBER.

	llave encapsulación	llave desencapsulación	ciphertext	llave secreta compartida
KYBER512	800	1632	768	32
KYBER768	1184	2400	1088	32
KYBER1024	1568	3168	1568	32

3.5.2. Especificaciones para KYBER.CPAPKE

KYBER.CPAPKE es un esquema de cifrado de clave pública seguro bajo el criterio IND-CPA. Este esquema permite cifrar mensajes de una longitud fija de 32 bytes y se compone de tres algoritmos:

- Generación de claves (Kyber.CPAPKE.KeyGen());
- Cifrado (Kyber.CPAPKE.Enc(pk, m, r));
- Descifrado (Kyber.CPAPKE.Dec(sk, c)).

En la Generación de Claves Kyber.CPAPKE.KeyGen(), se genera aleatoriamente una matriz de polinomios A, y los vectores polinomiales s y e se seleccionan de acuerdo con la distribución B_{η_1}. Normalmente, la clave secreta es s y la clave pública es $As+e$. Sin embargo, para optimizar la implementación, la multiplicación As se realiza en el

dominio NTT generando A en el dominio NTT (es decir, $\hat{A}$) y transformando s en $\hat{s} = \text{NTT}(s)$. Para evitar la operación NTT^{-1}, e también se transforma en $\hat{e}$ y se suma a $\hat{A} \circ \hat{s}$.

Por lo tanto, los valores de las claves secreta y pública permanecen en el dominio NTT y se codifican como sk y pk, respectivamente. Además, la semilla utilizada para generar la aleatoriedad se adjunta a la clave pública para que el receptor pueda generar la matriz A.

Algoritmo 1 Kyber.CPAPKE.KeyGen()

1: **Salida:** Clave secreta $sk \in \mathbb{B}^{12 \cdot k \cdot n/8}$
2: **Salida:** Clave pública $pk \in \mathbb{B}^{12 \cdot k \cdot n/8+32}$
3: $d \leftarrow \mathbb{B}^{32}$
4: $(\rho, \sigma) := G(d)$
5: $N := 0$
6: **for** i desde 0 hasta $k-1$ **do** ▷ Generar la matriz $\hat{\mathbf{A}} \in \mathbb{R}_q^{k \times k}$ en el dominio **NTT**
7: **for** j desde 0 hasta $k-1$ **do**
8: $\hat{A}[i][j] := \text{Parse}(XOF(\rho, j, i))$
9: **end forend for**
10: **end forend for**
11: **for** i desde 0 hasta $k-1$ **do** ▷ Muestrear $\mathbf{s} \in \mathbb{R}_q^k$ de B_{η_1}
12: $s[i] := \text{CBD}_{\eta_1}(\text{PRF}(\sigma, N))$
13: $N := N + 1$
14: **end forend for**
15: **for** i desde 0 hasta $k-1$ **do** ▷ Muestrear $\mathbf{e} \in \mathbb{R}_q^k$ de B_{η_1}
16: $e[i] := \text{CBD}_{\eta_1}(\text{PRF}(\sigma, N))$
17: $N := N + 1$
18: **end forend for**
19: $\hat{s} := \mathbf{NTT}(\mathbf{s})$
20: $\hat{e} := \mathbf{NTT}(\mathbf{e})$
21: $\hat{t} := \hat{A} \circ \hat{s} + \hat{e}$
22: $pk := (\text{Encode}_{12}(\hat{t}\, mod^+ q) \| \rho)$ ▷ pk := $\mathbf{As} + \mathbf{e}$
23: $sk := \text{Encode}_{12}(\hat{s}\, mod^+ q)$ ▷ sk := s
24: **return** (pk, sk)

El algoritmo Kyber.CPAPKE.Enc(pk, m, r) Cifrado comienza extrayendo el vector t y la semilla de la clave de cifrado. Luego, la semilla se expande para regenerar la matriz A, de la misma forma en que se hizo en Kyber.CPAPKE.KeyGen(). Si t y A se derivan correctamente de una clave de cifrado generada por Kyber.CPAPKE.KeyGen(), entonces serán iguales a sus valores correspondientes en Kyber.CPAPKE.KeyGen().

En Kyber.CPAPKE.Dec(sk, c) Descifrado, el vector de polinomios u y el polinomio v se obtienen del texto cifrado mediante un proceso de decodificación y descompresión. El vector s se obtiene de la clave secreta. Luego, el mensaje m se calcula como $v - s^T u$. Nuevamente, las multiplicaciones se realizan en el dominio NTT y se transforman al

Algoritmo 2 Kyber.CPAPKE.Enc(pk, m, r)

1: **Entrada:** Clave pública $pk \in \mathbb{B}^{12 \cdot k \cdot n/8+32}$
2: **Entrada:** Mensaje $m \in \mathbb{B}^{32}$
3: **Entrada:** Monedas aleatorias $r \in \mathbb{B}^{32}$
4: **Salida:** Texto cifrado $c \in \mathbb{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
5: $N := 0$
6: $\hat{t} := \text{Decode}_{12}(pk)$
7: $\rho := pk + 12 \cdot k \cdot n/8$
8: **for** i desde 0 hasta $k-1$ **do** ▷ Generar la matriz $\hat{A} \in \mathbb{R}_q^{k \times k}$ en el dominio NTT
9: **for** j desde 0 hasta $k-1$ **do**
10: $\hat{A}^T[i][j] := \text{Parse}(XOF(\rho, i, j))$
11: **end forend for**
12: **end forend for**
13: **for** i desde 0 hasta $k-1$ **do** ▷ Muestrear $r \in \mathbb{R}_q^k$ de B_{η_1}
14: $r[i] := \text{CBD}_{\eta_1}(\text{PRF}(r, N))$
15: $N := N + 1$
16: **end forend for**
17: **for** i desde 0 hasta $k-1$ **do** ▷ Muestrear $e_1 \in \mathbb{R}_q^k$ de B_{η_2}
18: $e_1[i] := \text{CBD}_{\eta_2}(\text{PRF}(r, N))$
19: $N := N + 1$
20: **end forend for**
21: $e_2 := \text{CBD}_{\eta_2}(\text{PRF}(r, N))$ ▷ Muestrear $e_2 \in \mathbb{R}_q$ de B_{η_2}
22: $\hat{r} := \text{NTT}(r)$
23: $u := \text{NTT}^{-1}(\hat{A}^T \circ \hat{r}) + e_1$ ▷ $u := \hat{A}^T \hat{r} + e_1$
24: $v := \text{NTT}^{-1}(\hat{t} \circ \hat{r}) + e_2 + \text{Decompress}_q(\text{Decode}_1(m), 1)$ ▷ $v := \hat{t}\hat{r} + e_2 + \text{Decompress}_q(m, 1)$
25: $c_1 := \text{Encode}_{d_u}(\text{Compress}_q(u, d_u))$
26: $c_2 := \text{Encode}_{d_v}(\text{Compress}_q(v, d_v))$
27: **return** $c = (c_1 \| c_2)$ ▷ $c := (\text{Compress}_q(u, d_u), \text{Compress}_q(v, d_v))$

dominio normal usando NTT^{-1}.

Algoritmo 3 Kyber.CPAPKE.Dec(sk, c)

1: **Entrada:** Clave secreta $sk \in \mathbb{B}^{12 \cdot k \cdot n/8}$
2: **Entrada:** Texto cifrado $c \in \mathbb{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
3: **Salida:** Mensaje $m \in \mathbb{B}^{32}$
4: $u := \mathrm{Decompress}_q(\mathrm{Decode}_{d_u}(c), d_u)$
5: $v := \mathrm{Decompress}_q(\mathrm{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$
6: $\hat{s} := \mathrm{Decode}_{12}(sk)$
7: $m := \mathrm{Encode}_1(\mathrm{Compress}_q(v - \mathbf{NTT}^{-1}(\hat{s}^T \circ \mathbf{NTT}(u)), 1))$
8: **return** m $\triangleright$ $m := \mathrm{Compress}_q(v - s^T u, 1)$

3.5.3. Especificaciones para Kyber.CCAKEM

A partir de la descripción de la sección anterior se define IND-CCA2 de Kyber.CCAKEM un KEM seguro. Este esquema de cifrado logra la seguridad de “Indistinguibilidad bajo ataque de texto cifrado elegido” (IND-CCA). La seguridad IND-CCA implica que un adversario, incluso cuando se le otorga acceso a un oráculo de cifrado, no puede discernir entre dos textos cifrados correspondientes al mismo texto simple.

En los algoritmos Kyber.CCAKEM.KeyGen(), Kyber.CCAKEM.Enc(pk) y Kyber.CCAKEM.Des(c, sk), definimos la generación de claves, la encapsulación y la desencapsulación de Kyber.CCAKEM.

El algoritmo Kyber.CCAKEM.KeyGen() (algoritmo 4) acepta dos semillas aleatorias como entrada, y produce una clave de encapsulación y una clave de desencapsulación.

Algoritmo 4 Kyber.CCAKEM.KeyGen()

1: **Salida:** Clave pública $pk \in \mathbb{B}^{12 \cdot k \cdot n/8+32}$
2: **Salida:** Clave secreta $sk \in \mathbb{B}^{24 \cdot k \cdot n/8+96}$
3: $z \leftarrow \mathbb{B}^{32}$
4: $(pk, sk_0) :=$ Kyber.CPAPKE.KeyGen()
5: $sk := (sk_0 \| pk \| H(pk) \| z)$
6: **retornar** (pk, sk)

Algoritmo 5 Kyber.CCAKEM.Enc(pk)

1: **Entrada:** Clave pública $pk \in \mathbb{B}^{12 \cdot k \cdot n/8+32}$
2: **Salida:** Texto cifrado $c \in \mathbb{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$
3: **Salida:** Clave compartida $K \in \mathbb{B}^*$
4: $m \leftarrow \mathbb{B}^{32}$
5: $m := H(m)$
6: $(\bar{K}, r) := G(m \| H(pk))$
7: $c :=$ Kyber.CPAPKE.Enc(pk, m, r)
8: $K :=$ KDF$(\bar{K} \| H(c))$
9: **retornar** (c, K) ▷ No enviar la salida del RNG del sistema

La figura 3.3 muestra un diagrama de flujo del protocolo de intercambio de claves de un esquema de cifrado Kyber.CCAKEM, entre dos partes, Alicia y Beto.

Algoritmo 6 Kyber.CCAKEM.Des(c, sk)

1: **Entrada:** Texto cifrado $c \in \mathbb{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
2: **Entrada:** Clave secreta $sk \in \mathbb{B}^{24 \cdot k \cdot n/8 + 96}$
3: **Salida:** Clave compartida $K \in \mathbb{B}^*$
4: $pk := sk + 12 \cdot k \cdot n/8$
5: $h := sk + 24 \cdot k \cdot n/8 + 32 \in \mathbb{B}^{32}$
6: $z := sk + 24 \cdot k \cdot n/8 + 64$
7: $m_0 :=$ Kyber.CPAPKE.Dec$(s, (u, v))$
8: $(\bar{K}_0, r_0) := G(m_0 \| h)$
9: $c_0 :=$ Kyber.CPAPKE.Enc(pk, m_0, r_0)
10: **if** $c = c_0$ **then**
11:     **retornar** $K :=$ KDF$(\bar{K}_0 \| H(c))$
12: **else**
13:     **retornar** $K :=$ KDF$(z \| H(c))$
14: **end ifend if**
15: **retornar** K

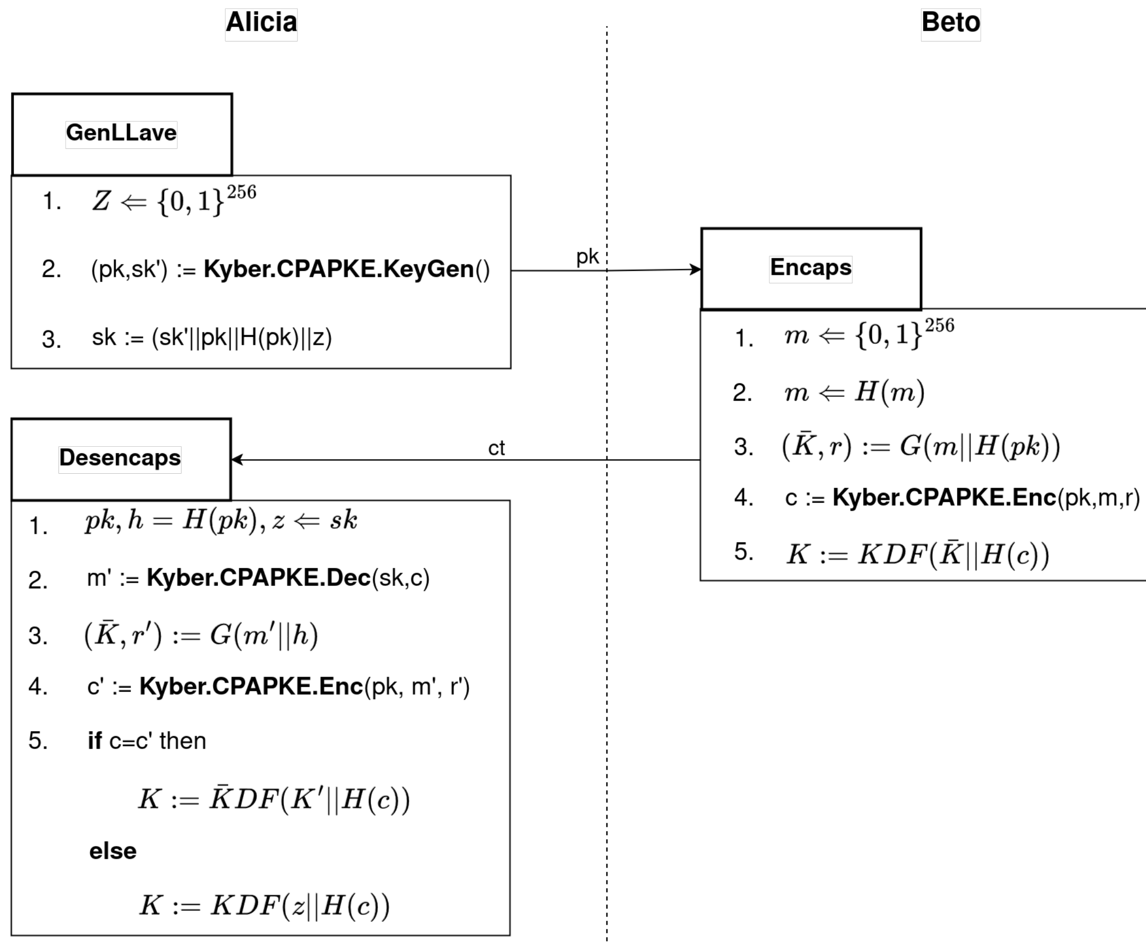


Figura 3.3: Esquema Kyber.CCAKEM.

3.5.4. Arquitectura KYBER

La figura 3.4 muestra las operaciones principales de la arquitectura de intercambio de claves en KYBER para dos entidades que desean comunicarse, representadas como Alicia y Beto. En este esquema, las funciones basadas en Keccak, SHA-3 y la NTT se implementan como componentes clave del algoritmo CRYSTALS-KYBER, formando parte de los tres algoritmos principales: Kyber.CPAPKE.KeyGen(), Kyber.CPAPKE.Enc(pk, m, r) y Kyber.CPAPKE.Dec(sk, c).

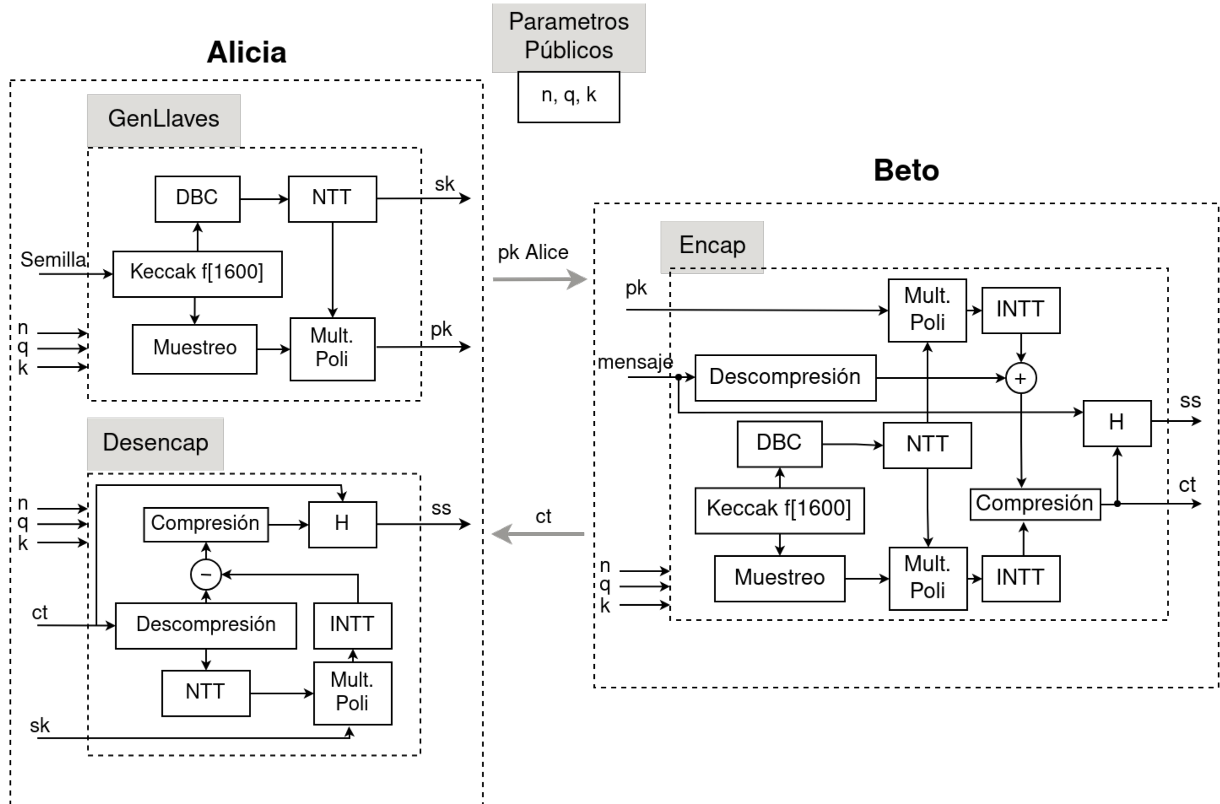


Figura 3.4: Esquema CRYSTALS-KYBER.

Los diseñadores de KYBER han incorporado NTT y su NTT^{-1} en su algoritmo para frustrar un cuello de botella computacional primario [42], a saber, las multiplicaciones modulares polinómicas. Las operaciones NTT y NTT^{-1} en el algoritmo KYBER aparece en la especificación de KYBER.CPAPKE en algoritmo 1, algoritmo 2 y algoritmo 3, respectivamente.

El segundo componente más crítico para el rendimiento de la implementación de software es el núcleo Keccak [39], basado en los recuentos de ciclos perfilados presentados en [11]. De hecho, más de la mitad de los ciclos de reloj registrados en las pruebas comparativas SW y HW/SW se utilizan para calcular Keccak. Sin embargo, este núcleo puede acelerarse en una arquitectura de hardware pura, ya que Keccak es un diseño de SHA compatible con hardware.

3.5.5. Unidad Keccak

SHA-3 [2] es una familia de funciones que se basan en una instancia del algoritmo Keccak. La familia SHA-3 consta de cuatro funciones hash criptográficas y dos funciones de salida extensible. Estas seis funciones comparten la estructura de la construcción esponja; las funciones con esta estructura se denominan funciones esponja.

Una función hash es una función sobre datos binarios (es decir, cadenas de bits) para la que la longitud de la salida es fija. La entrada de una función hash se denomina mensaje, y la salida se denomina resumen (del mensaje) o valor hash. El resumen suele servir como representación condensada del mensaje [43]. Las dos funciones hash SHA-3 que utiliza CRYSTALS-Kyber son: SHA-3-256 y SHA-3-512.

Una XOF es una función sobre cadenas de bits en la cual la longitud de la salida se puede extender a cualquier longitud deseada para cumplir con los requisitos de cada aplicación. Las dos XOFs SHA-3 utilizadas CRYSTALS-Kyber son: SHAKE128 y SHAKE256.

Las permutaciones KECCAK-p se diseñaron para que fueran adecuadas como componentes principales para una variedad de funciones criptográficas, incluidas las funciones con clave para autenticación y/o cifrado. Las seis funciones SHA-3 pueden considerarse modos de operación de la permutación KECCAK-p[1600,24].

Algoritmo 7 KECCAK-p[b, n_r]

Entrada: cadena S de longitud b
número de rondas n_r
Salida: String S' de longitud b.

Etapas:
1- Conviertir S en un arreglo de estados A.
2- Para i_r in $12 + 2\ell - n_r$ to $12 + 2\ell - 1$, sea A = Rnd(A, i_r).
3- Convertir A en una cadena S' de longitud b.
4- Devuelve S'.
end

El estado de la permutación KECCAK-p[b, nr] se compone de b bits. Las especificaciones de esta norma contienen otras dos cantidades relacionadas con b: b/25 y $\log_2(b/25)$, indicadas por ω y ℓ, respectivamente. Los siete valores posibles para estas variables que se definen para las permutaciones KECCAK-p se dan en la tabla 3.3.

Dada una matriz de estado A y un índice de ronda i_r, la función de ronda Rnd es la transformación que resulta de aplicar las asignaciones de pasos θ, ρ, π, χ y ι, en ese orden, es decir:

[2] Secure Hash Algorithm-3

Tabla 3.3: Anchos de permutación KECCAK-p y cantidades ω y ℓ relacionadas.

b	25	50	100	200	400	800	1600
ω	1	2	4	8	16	32	64
ℓ	0	1	2	3	4	5	6

$$Rnd(A, i_r) = \iota(\chi(\pi(\rho(\theta(A)))), i_r).$$

La permutación KECCAK-p[b, nr] consta de $n_r = 12 + 2\ell$ iteraciones de Rnd como se especifica en el algoritmo 7.

La permutación se especifica en términos de una matriz de valores (figura 3.5) para b bits que se actualiza repetidamente, llamada estado; el estado se establece inicialmente en los valores de entrada de la permutación.

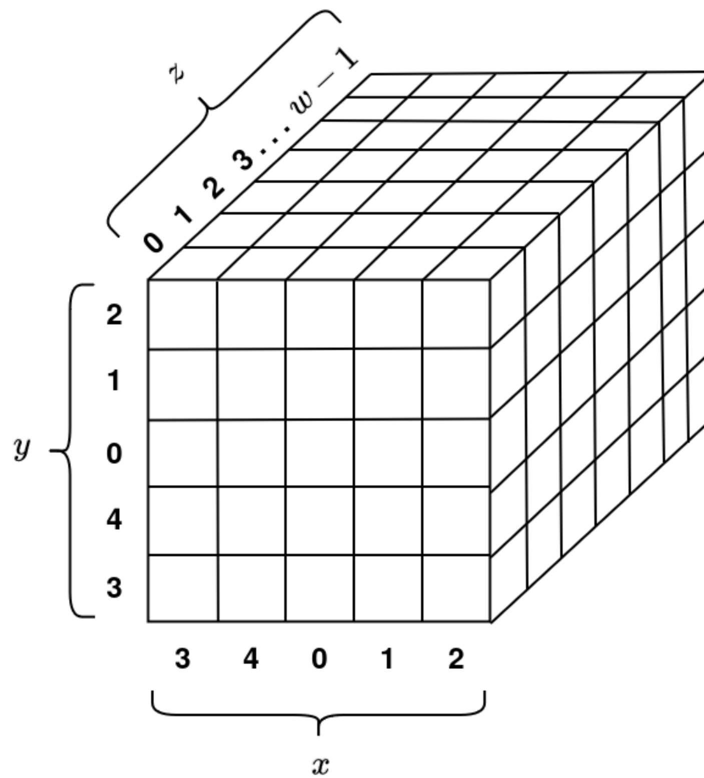


Figura 3.5: Matriz de estado de $5 \times 5 \times \omega$.

Permutación Keccak-f [1600]

La construcción emplea los siguientes tres componentes:

- Una función subyacente en cadenas de longitud fija, denotada por f.
- Un parámetro llamado tasa, denotado por r.

- Una regla de relleno, denotada por pad.

La función que produce la construcción a partir de estos componentes se denomina función de esponja, denotada por ESPONJA[f, pad, r]. Una función esponja recibe dos datos como entrada: una secuencia de bits identificada como N, y la longitud en bits de la cadena de salida, identificada como d, ESPONJA[f, pad, r](N, d).

La construcción de la esponja depende de la elección de variantes de hash y utiliza la tasa de bits (r) y la capacidad (c) adecuadas como parámetros principales para determinar el nivel de seguridad global.

La función f asigna cadenas de una sola longitud fija, indicada por b, a cadenas de la misma longitud. La tasa r es un número entero positivo que es estrictamente menor que el ancho b. La capacidad, denotada por c, es el entero positivo b-r. Por lo tanto, r + c = b.

Algoritmo 8 Esponja[f, pad, r](N, d)

Entrada: Cadena N
entero d
Salida: cadena Z tal que len(Z) = d

Etapas:
1- Sea P = N || pad(r, len(N)).
2- Sea n = len(P)/r.
3- Sea c = b - r.
4- Sea $P_0, ..., Pn_1$ la secuencia única de cadenas de longitud r tal que $P = P0||...||P_{n-1}$.
5- Sea $S = 0^b$
6- For i de 0 a n_1, sea $S = f(S \oplus (P_i||0^c))$
7- Sea Z la cadena vacía
8- Sea $Z = Z||Truncarr(S)$
9- If $d \leq |Z|$, devuelve Trunc d (Z); **else** continuar.
10- Sea S = f(S), y continúe con el Paso 8.
end

La función se compone de dos fases que se muestran en la figura 3.6.

1. Fase de absorción: se descompone el mensaje en bloques de r bits generando $M = M_1||...||M_s$. Si el último bloque es incompleto, M_s se rellena con la regla correspondiente. La tasa de bits del estado inicializado se somete a XOR con la primera parte de la entrada. La nueva tasa de bits y junto con la capacidad de la matriz de estado inicializada formarán un nuevo estado que se utiliza en la permutación f. El estado resultante servirá como el nuevo estado inicial para la siguiente ronda y el proceso continúa durante 24 rondas de iteraciones.

2. Fase de exprimir: la salida del hash es generada a partir del estado interno de la esponja. Si la longitud deseada del hash es menor o igual a la longitud de la tasa r, entonces este valor de salida se toma directamente de la tasa del estado interno. Cuando la longitud del hash deseada es mayor que r, se aplica la función de permutación de Keccak al estado interno hasta producir suficiente salida para alcanzar la longitud del hash deseada.

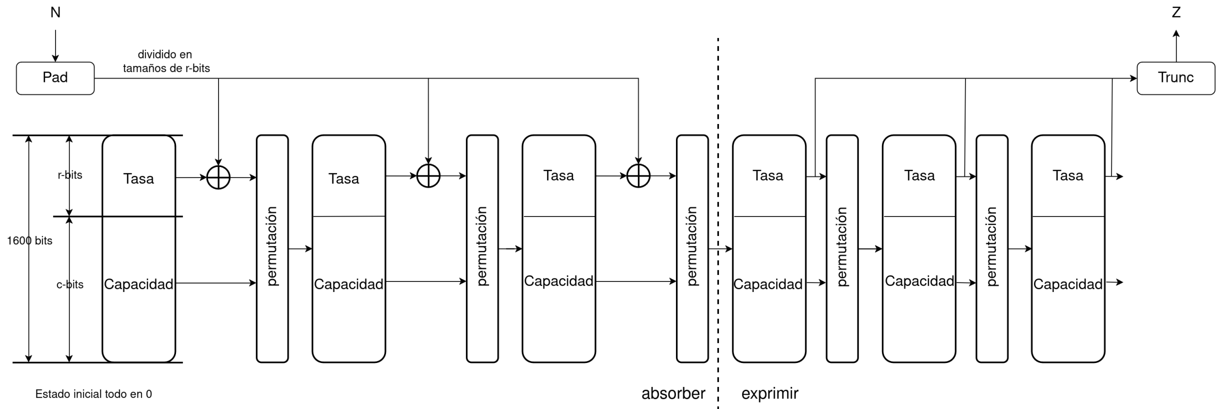


Figura 3.6: Construcción esponja Keccak.

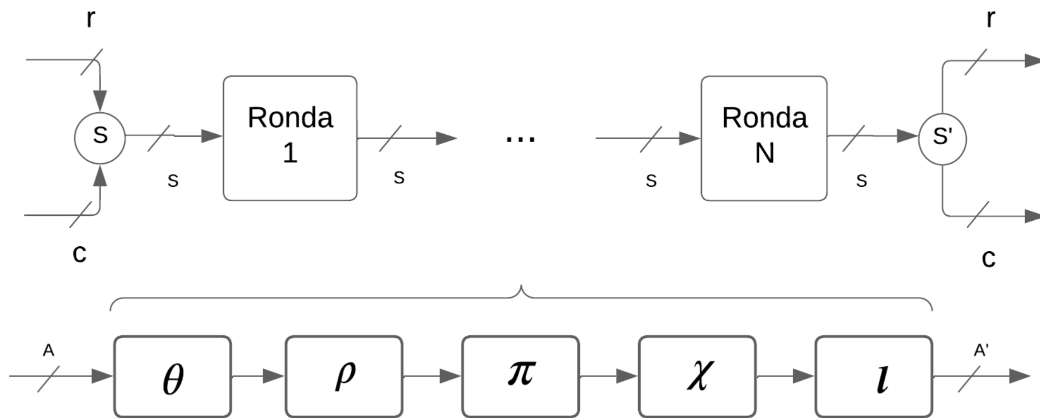


Figura 3.7: Función f.

Funciones de ronda

Una ronda de KECCAK-p[b, nr] está compuesta de cinco pasos separados: theta (θ), rho (ρ), pi (π), chi (χ) e iota (ι). El proceso de cada iteración del algoritmo, mostrado en la figura 3.7, toma una matriz que representa el estado actual del sistema, llamada A, como entrada, y produce una nueva matriz de estado actualizado, llamada A', como salida. Excepto la función de mapeo ι tiene una segunda variable de entrada,

la cual es un número entero llamado índice redondo, representado por i_r. Las otras asignaciones de pasos no dependen del índice redondo.

Detalles para Theta θ

El efecto de θ es XOR cada bit en el estado con las paridades de dos columnas en la matriz como podemos ver en la figura B.1. El operador matemático $\sum$, que representa la suma, indica la paridad de una columna, es decir, el resultado de la operación lógica XOR aplicada a todos los bits que conforman la columna.

Algoritmo 9 θ (A)

Entrada: Matriz de estado A
Salida: Matriz de estado A'

ETAPAS:

1- Para todo par (x,z) dado que $0 \leq x < 5$ y $0 \leq z < w$

$$C[x,z] = A[x,0,z] \oplus A[x,1,z] \oplus A[x,2,z] \oplus A[x,3,z] \oplus A[x,4,z]$$

2- Para todo par (x,z) dado que $0 \leq x < 5$ y $0 \leq z < w$

$$D[x,z] = C[(x-1)\,mod\,5, z] \oplus C[(x+1)\,mod\,5, (z-1)\,mod\,w]$$

3- Para todos los triples (x, y, z) tales que $0 \leq x < 5$, $0 \leq y < 5$ y $0 \leq z < w$

$$A'[x,y,z] = A[x,y,z] \oplus D[x,z].$$

end

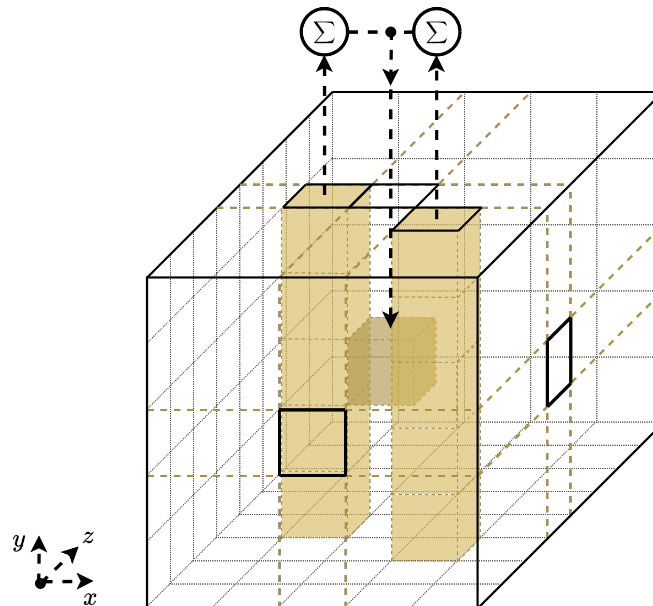


Figura 3.8: θ aplicada a un solo bit.

Especificaciones para Rho ρ

El efecto de ρ (ver el algoritmo 10), es rotar los bits de cada carril por una longitud, llamada desplazamiento, que depende de las coordenadas x e y fijas del carril (ver tabla 3.4). De manera equivalente, para cada bit en el carril, la coordenada z se modifica agregando el desplazamiento, módulo el tamaño del carril.

Tabla 3.4: Desplazamientos de ρ.

	y=2	**y=1**	**y=0**	**y=4**	**Y=4**
x=3	25	55	28	56	21
x=4	39	20	27	14	8
x=0	3	36	0	18	41
x=1	10	44	1	2	45
x=0	43	6	62	61	15

Algoritmo 10 ρ (A)

Entrada: Matriz de estado A
Salida: Matriz de estado A'

Etapas:

1- Para todo z tal que $0 \leq z < w$, sea A' [0, 0, z] = A[0, 0, z].
2- Sea (x, y) = (1, 0).
3- For t de 0 a 23:

a. For todo z tal que 0 ≤ z < w, sea A'[x, y, z] = A[x, y, (z − (t + 1)(t+2)/2) mod w];
b. Sea (x, y) = (y, (2x + 3y) mod 5).

4- Devuelve A'.

end

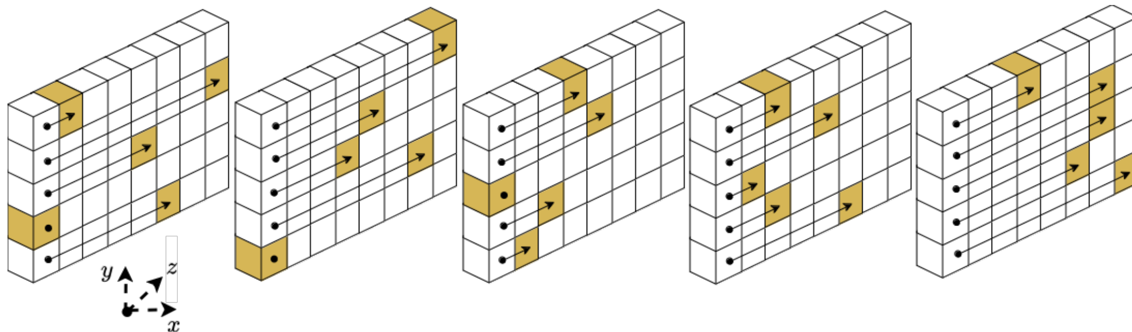


Figura 3.9: ρ aplicado a los carriles para el caso w = 8.

Para cada carril (ver figura 3.9), el punto negro indica el bit cuya coordenada z es 0, y el cubo sombreado indica la posición de ese bit después de la ejecución de ρ. Los

otros bits del carril se desplazan con el mismo desplazamiento, y el desplazamiento es circular.

Especificaciones para π

El efecto de π (ver el algoritmo 11), es reorganizar las posiciones de los carriles, como se ilustra para cualquier corte en la figura 3.10.

Algoritmo 11 π (A)

Entrada: Matriz de estado A
Salida: Matriz de estado A'

Etapas:

1- Para todos los triples (x, y, z) tales que $0 \leq x < 5$, $0 \leq y < 5$ y $0 \leq z < w$, sea

$$A'[x, y, z] = A[(x + 3y) \, mod \, 5, x, z].$$

2- Devuelve A'.

end

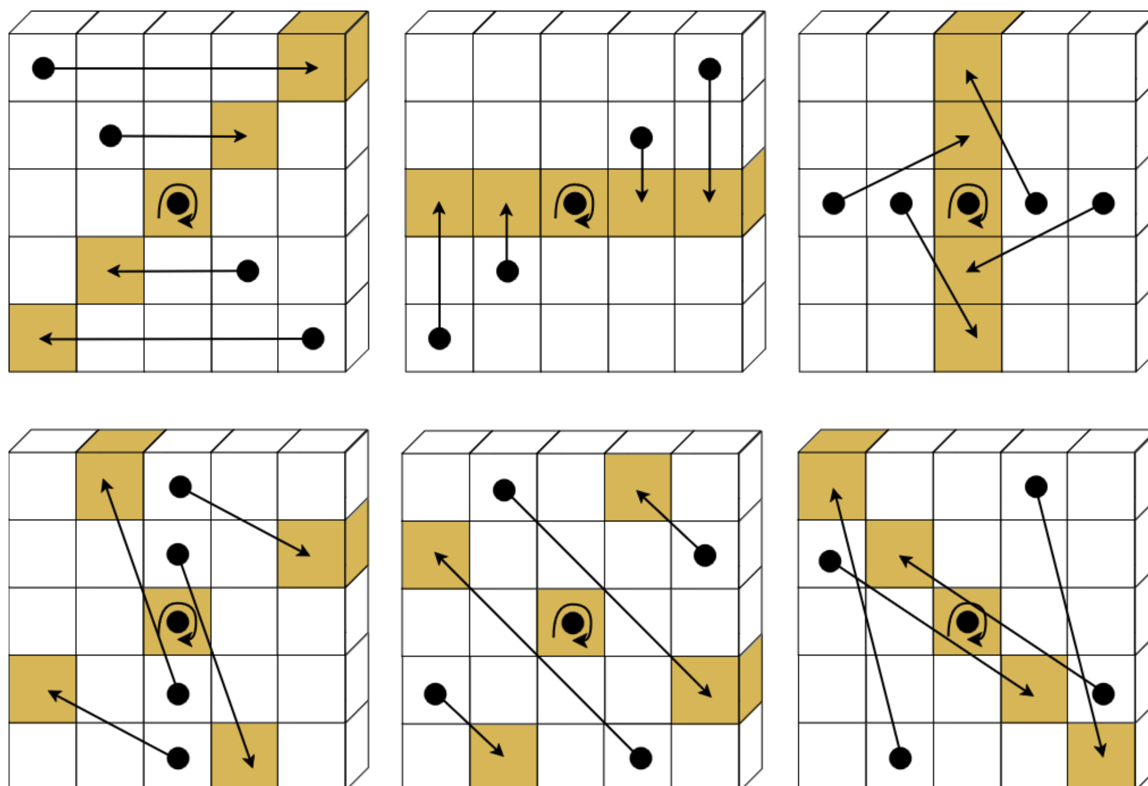


Figura 3.10: π aplicada a solo una rebanada.

Especificaciones para χ

El efecto de χ, ver algoritmo 12, es XOR cada bit con una función no lineal de otros dos bits en su fila, como se ilustra en la figura 3.11.

Algoritmo 12 χ (A)

Entrada: Matriz de estado A
Salida: Matriz de estado A'

Etapas:

1- Para todos los triples (x, y, z) tales que $0 \leq x < 5$, $0 \leq y < 5$ y $0 \leq z < w$, sea

$$A'[x, y, z] = A[x, y, z] \oplus ((A[(x + 1)\, mod\, 5, y, z] \oplus 1) \cdot A[(x + 2)\, mod\, 5, y, z]).$$

2- Devuelve A'.

end

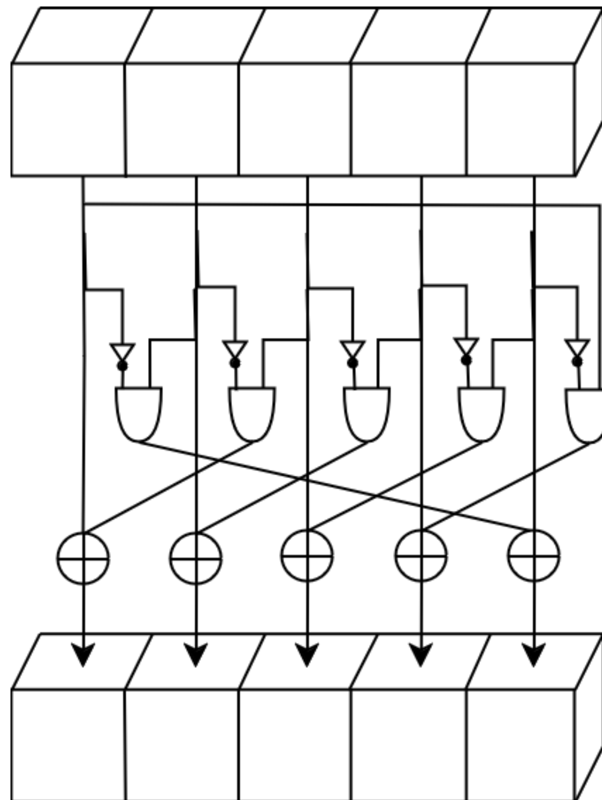


Figura 3.11: χ aplicada a una sola fila.

Especificaciones para ι

El efecto de ι es modificar algunos de los bits de carril (0, 0) de una manera que depende del índice de ronda i_r. Los otros 24 carriles no se ven afectados por ι. ι está parametrizado por el índice de ronda, i_r. Dentro de la especificación de ι en el algoritmo 12, este parámetro determina $\ell + 1$ bits de un valor de carril denominado constante de ronda, indicado por la constante de ronda (RC[i]) cuyos valores se muestran en la tabla 3.5.

Algoritmo 13 ι (A)

Entrada: matriz de estado A
constante de ronda rc
Salida: matriz de estado A'

Etapas:

1- Para todos los triples (x, y, z) tales que $0 \leq x < 5$, $0 \leq y < 5$ y $0 \leq z < w$, sea

$$A'[x, y, z] = A[x, y, z].$$

2- Para todo z tal que $0 \leq z < w$, sea

$$A'[x, y, z] = A'[x, y, z] \oplus RC[z].$$

3- Devuelve A'.

end

Tabla 3.5: Valores de las constantes de ronda RC[i].

RC[0]	0x0000000000000001	RC[1]	0x0000000000008082
RC[2]	0x800000000000808a	RC[3]	0x8000000080008000
RC[4]	0x000000000000808b	RC[5]	0x0000000080000001
RC[6]	0x8000000080008081	RC[7]	0x8000000000008009
RC[8]	0x000000000000008a	RC[9]	0x0000000000000088
RC[10]	0x0000000080008009	RC[11]	0x000000008000000a
RC[12]	0x000000008000808b	RC[13]	0x800000000000008b
RC[14]	0x8000000000008089	RC[15]	0x8000000000008003
RC[16]	0x8000000000008002	RC[17]	0x8000000000000080
RC[18]	0x000000000000800a	RC[19]	0x800000008000000a
RC[20]	0x8000000080008081	RC[21]	0x8000000000008080
RC[22]	0x0000000080000001	RC[23]	0x8000000080008008

Parametrización para CRYSTALS-Kyber

La parametrización de los cuatro algoritmos de la familia de funciones SHA-3 para CRYSTALS-Kyber implica seleccionar los parámetros de la tabla 3.6 adecuados

para SHA-3-256, SHA-3-512, SHAKE128, SHAKE256. Para SHA-3-256, SHA-3-512, SHAKE128, SHAKE256 b será 1600. Instanciación de PRF, XOF, H, G y KDF, las primitivas simétricas. Instanciamos todas esas primitivas con funciones del estándar FIPS-202 [43] de la siguiente manera:

- XOF con SHAKE-128;
- H con SHA-3-256;
- G con SHA-3-512;
- PRF(s, b) con SHAKE-256(s||b); y
- KDF con SHAKE-256.

Tabla 3.6: Parámetros de las funciones SHA-3.

Algoritmo	Tamaño en Bits: Taza r	Capacidad c	Pad Mensaje[1]	Salida Hash	Rondas n^r
			Hash SHA-3		
SHA3-256	1088	512	$M \parallel 0x06 \parallel 0x00 \ldots \parallel 0x80$	256	24
SHA3-512	576	128		128	24
			Salida extensibles SHA-3		
SHAKE-128	1344	256	$M \parallel 0x1F \parallel 0x00 \ldots \parallel 0x80$	arbitraria	24
SHAKE-256	1088	512		arbitraria	24

[1] para tamaños de padding mayor a 2 bytes

3.5.6. NTT

La multiplicación de polinomios representa el cuello de botella en la criptografía basada en *lattices*, y puede realizarse mediante la NTT o el algoritmo tradicional de multiplicación de polinomios. La NTT permite realizar esta operación de manera eficiente sobre un anillo de polinomios $\mathbb{Z}_q[X]/\langle X^n + 1\rangle$ en CRYSTALS-KYBER. La multiplicación utilizando la NTT presenta varias ventajas: es muy veloz, no necesita memoria extra (a diferencia de métodos como Karatsuba o Toom), y puede implementarse en un espacio de código muy reducido[43].

La NTT es una generalización de la transformada rápida de Fourier (FFT) definida en un campo finito. Sea f un polinomio de grado n, donde $f = \sum_{i=0}^{n-1} f_i X^i$ y $f_i \in \mathbb{Z}_q$, y sea ζ_n la raíz primitiva n-ésima de la unidad, tal que $\zeta_n^n = 1 \mod q$. La NTT directa se define como $\hat{f} = \mathrm{NTT}(f)$, donde $\hat{f}_i = \sum_{j=0}^{n-1} f_j \zeta_n^{ij} \mod q$. Además, la NTT inversa se expresa como $f = \mathrm{NTT}^{-1}(\hat{f})$, donde $f_i = n^{-1} \sum_{j=0}^{n-1} \hat{f}_j \zeta_n^{-ij} \mod q$.

Para nuestro primo q $= 3329$ con $q - 1 = 2^8 \cdot 13$, el cuerpo base $\mathbb{Z}_q$ contiene raíces primitivas 256-ésimas de la unidad, aunque no raíces primitivas 512-ésimas. Por lo tanto, el polinomio definitorio $X^{256} + 1$ de $\mathbb{R}$ se factoriza en 128 polinomios de grado 2 módulo q y el NTT de un polinomio $f \in \mathbb{R}_q$ es un vector de 128 polinomios de

grado uno. Las implementaciones simples en el sitio de la NTT sin reordenamiento generan estos polinomios en orden inverso de bits y definimos la NTT de esta manera. Concretamente, sea $\zeta = 17$ la primera raíz primitiva 256-ésima de la unidad módulo q, y $\zeta, \zeta^3, \zeta^5, ..., \zeta^{255}$ el conjunto de todas las raíces 256-ésimas de la unidad. El polinomio $X^{256} + 1$ puede escribirse, por tanto, como:

$$X^{256} + 1 = \prod_{i=0}^{127}(X^{256} - \zeta^{2i+2}) = \prod_{i=0}^{127}(X^{256} - \zeta^{2br_7(i)+1}),$$

donde br_7(i) para i = 0,..., 127 es la inversión de bits del entero de 7-bits sin signo i. Entonces, el NTT de $f \in R_q$ está dado por

$$(f \bmod X^2 - \zeta^{2br_7(0)+1}), ..., f \bmod X^2 - \zeta^{2br_7(127)+1}). \tag{3.1}$$

Este vector de polinomios lineales se serializa entonces a un vector en $\mathbb{Z}_q^{256}$ de forma canónica. Además, para no introducir tipos de datos adicionales y facilitar las implementaciones en el sitio de la NTT definimos NTT : $R_q \to R_q$ como la biyección que mapea $f \in R_q$ al polinomio con el vector de coeficientes mencionado. Por lo tanto,

$$NTT(f) = \hat{f} = \hat{f}_0 + \hat{f}_1 X + ... + \hat{f}_{255} X^{255}.$$

$$\hat{f}_2 i = \Sigma_{j=0}^{127} f_{2j} \zeta^{(2br_7(i)+1)j}, \tag{3.2}$$

$$\hat{f}_2 i + 1 = \Sigma_{j=0}^{127} f_{2j+1} \zeta^{(2br_7(i)+1)j}. \tag{3.3}$$

La representación algebraica natural de $NTT(f) = \hat{f}$ es como 128 polinomios de grado 1 como en la ecuación (3.1) usando las definiciones para $\hat{f}$ de la ecuación (3.2) y ecuación (3.3). Es decir,

$$NTT(f) = \hat{f} = (\hat{f}_0 + \hat{f}_1 X + \hat{f}_2 + \hat{f}_3 X + ... + \hat{f}_{254} + \hat{f}_{255} X).$$

Usando NTT y su inversa NTT^{-1} podemos calcular el producto $f \cdot g$ de dos elementos f, $g \in R_q$ de manera muy eficiente como $NTT^{-1}(NTT(f) \circ NTT(g))$ donde $NTT(f) \circ NTT(g) = \hat{f} \circ \hat{g} = \hat{h}$ denota la multiplicación del caso base que consiste en los 128 productos

$$\hat{h}_{2i} + \hat{h}_{2i+1} X = (\hat{f}_{2i} + \hat{f}_{2i+1} X)(\hat{g}_{2i} + \hat{g}_{2i+1} X) \, mod \, X^2 - \zeta^{(2br_7(i)+1)}$$

de polinomios lineales.

Las operaciones NTT y NTT^{-1} del algoritmo KYBER se presentan en los algoritmos NTT(f) y $\text{NTT}^{-1}(\hat{f})$, respectivamente. En este contexto, denominamos BitRev7(k) a la representación en orden inverso del bit i en una secuencia de k bits.

Algoritmo 14 NTT(f)

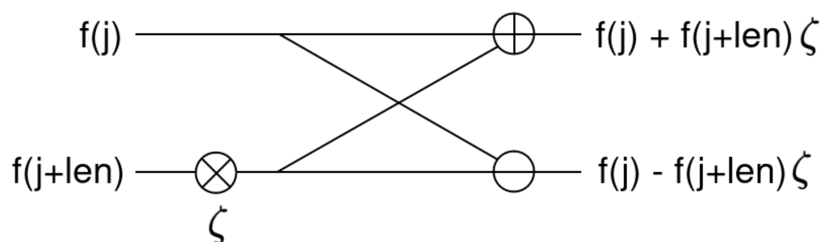
Entrada: Un array $f \in \mathbb{Z}_q^{256}$ ▷ los coeficientes del polinomio de entrada
Salida: Un array $\hat{f} \in \mathbb{Z}_q^{256}$ ▷ los coeficientes de la NTT del polinomio de entrada

```
1: f̂ ← f                    ▷ Calcular la NTT in-place en una copia del array de entrada
2: k ← 1
3: for (len ← 128; len ≥ 2; len ← len/2) do
4:     for (start ← 0; start < 256; start ← start+2 ·len) do
5:         zeta ← ζ^BitRev7(k) mód q
6:         k ← k + 1
7:         for (j ← start; j < start + len; j++) do
8:             t ← zeta · f̂[j + len]          ▷ Pasos 8-10 hechos módulo q
9:             f̂[j + len] ← f̂[j] − t
10:            f̂[j] ← f̂[j] + t
11:        end for
12:    end for
13: end for
14: return f̂
```

Para la multiplicación polinómica, se pueden utilizar mariposas Cooley-Tukey (CT) [44], y transformar ambas entradas al dominio NTT, luego realizar la multiplicación por elementos para las salidas NTT. A continuación, el resultado se transforma de nuevo utilizando mariposas Gentleman-Sande (GS) [45] para realizar NTT^{-1}. Como las mariposas reducen la operación matemática en una escala cuasilineal, la complejidad de la multiplicación polinómica se reduce de $O(n^2)$ a $O(n \log n)$. Cuanto mayor sea el grado del polinomio, mayor será la ganancia en velocidad y coste [46].

En el algoritmo NTT(f) que presenta la computación de la NTT, el cálculo del núcleo mariposa de CT (Ver figura 3.12) corresponde a las líneas 8-10.

Figura 3.12: Configuración de mariposa CT.



Algoritmo 15 $\text{NTT}^{-1}(\hat{f})$

Entrada: Un arreglo $\hat{f} \in \mathbb{Z}_q^{256}$ ▷ los coeficientes de la representación NTT dada
Salida: Un arreglo $f \in \mathbb{Z}_q^{256}$ ▷ los coeficientes de la inversa-NNN del input

```
1:  f ← f̂                    ▷ Calcular in-place en una copia del arreglo de entrada
2:  k ← 127
3:  for (len ← 2; len ≤ 128; len ← len · 2) do
4:      for (inicio ← 0; inicio < 256; inicio ← inicio+2 ·len) do
5:          zeta ← ζ^BitRev7(k) mód q
6:          k ← k − 1
7:          for (j ← inicio; j < inicio + len; j++) do
8:              t ← f[j]
9:              f[j] ← t + f[j + len]             ▷ Pasos 8-10 hechos módulo q
10:             f[j + len] ← zeta · (f[j + len] − t)
11:         end for
12:     end for
13: end for
14: f ← f · 3303 mód q        ▷ Multiplicar cada entrada por 3303 ≡ 128^−1 mód q
15: return f
```

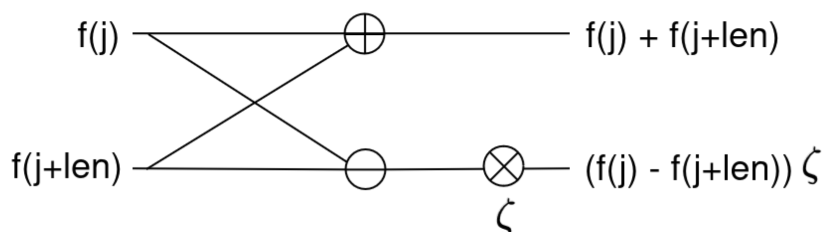


Figura 3.13: Configuración de mariposa GS.

La figura 3.14 ilustra el flujo de datos para la operación $\text{NTT}/\text{NTT}^{-1}$ de ocho puntos. Esta operación se organiza en tres etapas, cada una dedicada al procesamien-

to de cálculos de mariposa en el esquema CT. En el centro de la figura, los puntos de intersección representan cada cálculo de mariposa. Es evidente que cada etapa de la NTT de ocho puntos implica la realización de cuatro cálculos de mariposa. Además, los datos de entrada y salida de estos cálculos, dentro de cada etapa, son independientes entre sí, lo que permite diseñar múltiples cálculos de mariposa para ejecutarse en paralelo.

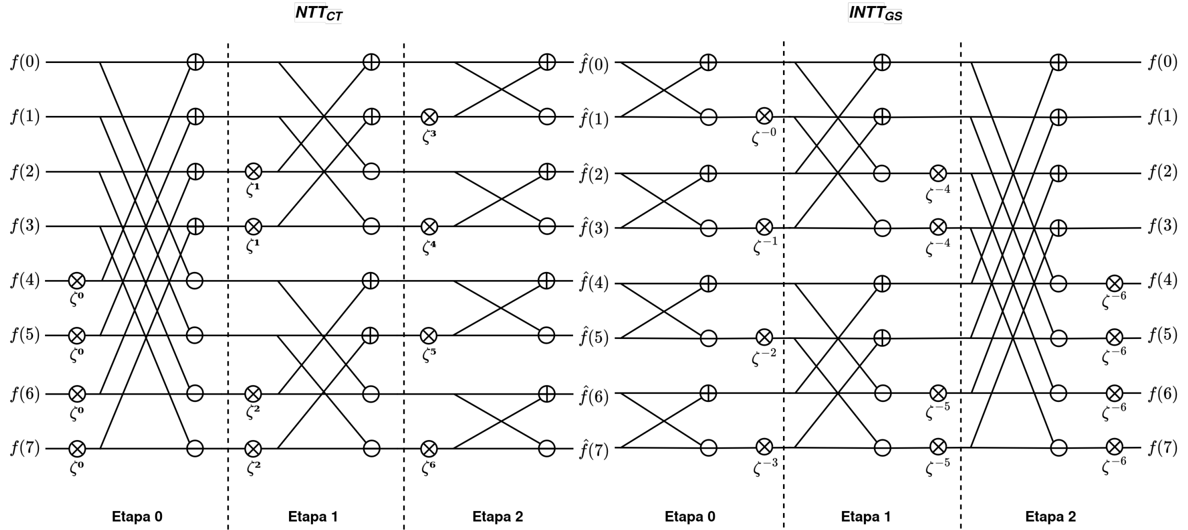


Figura 3.14: Ejemplo de una multiplicación polinómica basada en NTT de 8 puntos. El gráfico de flujo de datos que incluye NTT basada en mariposa CT, multiplicación por puntos e NTT^{-1} basada en mariposa GS. El polinomio $\hat{f}$ está en el dominio NTT y f está en el dominio normal.

Diferentes reducciones modulares pueden implementarse en el núcleo de mariposa, incluyendo la reducción de Barrett y la reducción de Montgomery. Una variante de la reducción de Montgomery fue introducida por [47] en el algoritmo Reducción de Montgomery.

En el núcleo de nuestro diseño, la unidad mariposa, empleamos de manera estratégica el algoritmo de Reducción de Montgomery, adaptado específicamente para las necesidades del algoritmo Kyber, donde $q = 3329$.

Algoritmo 16 Reducción de Montgomery

Entrada: $C[23:0] = A[11:0] \cdot B[11:0]$

Salida: $res = C \cdot R^{-1} \mod q$, $(q = 3329,\ R = 2^{12})$

1: $sum0 = C[11:0] \ll 11 + C[11:0] \ll 10 + C[11:0] \ll 8 - C[11:0]$

2: $sum1 = C + (sum0[11:0] \ll 11) + (sum0[11:0] \ll 10) + (sum0[11:0] \ll 8) + sum0[11:0]$

3: $res = sum1[24:12]$

4: **return** $result - q \geq 0 ? result - q : result$

Capítulo 4

Diseño de CRYSTALS-Kyber

4.1. Propuesta

Este diseño está pensado dentro de una arquitectura SoC que combina un PS y un PL, la cual se observa en la parte baja de la figura 4.1. Esto es para integrar el criptosistema CRYSTALS-Kyber, aprovechando así las ventajas de ambos mundos. Específicamente, la interfaz AXI contiene los registros manejan las entradas y salidas, permitiéndonos establecer una comunicación eficiente entre el PS y el PL. Los registros actúan como búferes de datos que facilitan la transferencia de información sin necesidad de mecanismos de comunicación complejos o costosos en términos de recursos.

El enfoque propuesto en esta tesis implicó la implementación de un codiseño HW/SW con VHDL y C, que ofrece un equilibrio entre las propuestas del estado del arte existente con respecto al rendimiento (tiempo, área, energía, etc.) al abordar directamente en un lenguaje de descripción de hardware la optimización de algoritmos de criptografía postcuántica como CRYSTALS-Kyber. Esta tesis se planteó realizar el análisis del algoritmo postcuántico CRYSTALS-Kyber finalista en la 3ra ronda del concurso del NIST para identificar las operaciones más complejas y realizar un diseño e implementación en VHDL de éstas e integrarlas en un codiseño HW/SW.

La figura 4.1 muestra el flujo del planteamiento de codiseño HW/SW. El codiseño de HW/SW aprovecha las compensaciones entre HW y SW para lograr objetivos a nivel de sistema como el rendimiento, el tiempo de comercialización o una integración más rápida y mejor mediante el diseño simultáneo [48]. Este enfoque de codiseño para la computación reconfigurable en FPGAs ofrece mayores posibilidades de personalización, adaptándose de manera óptima a las necesidades específicas de cada aplicación.

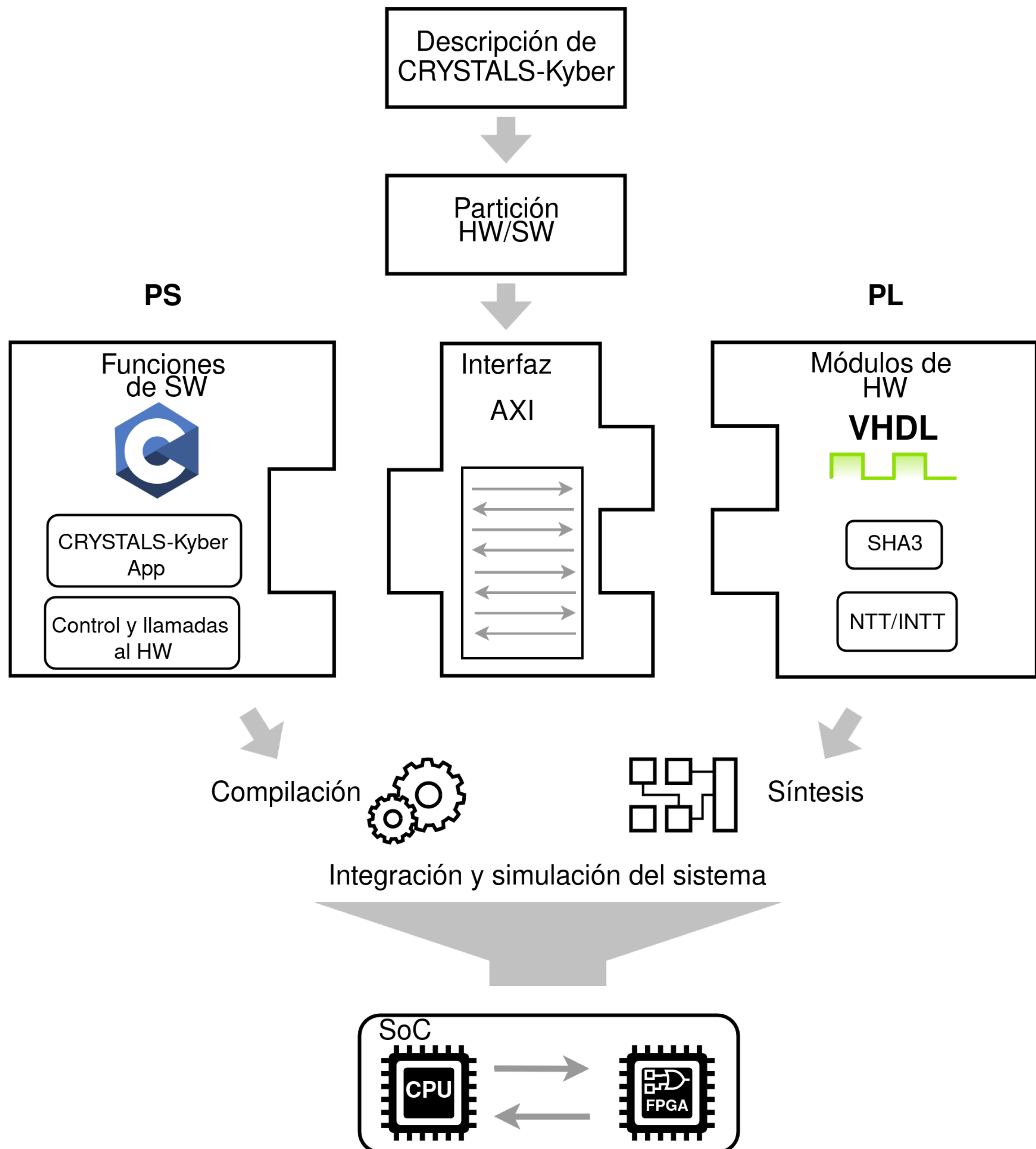


Figura 4.1: Metodología de Codiseño HW/SW para CRYSTALS-Kyber.

4.2. Diseño de la partición en software

El esquema CRYSTALS-Kyber implementa tres funciones principales: crypto_kem_keypair, crypto_kem_enc, y crypto_kem_dec, las cuales operan a un nivel más alto en el

protocolo conformando la base del esquema. Dentro de estas funciones se derivan múltiples operaciones como:

- **crypto_kem_keypair:** genera la clave pública y privada para el mecanismo de encapsulación de claves KYBERcon seguridad CCA.
    - **randombytes:** valor z para salida pseudoaleatoria en rechazo.
    - **indcpa_keypair:** genera la clave pública y privada del esquema de cifrado de clave pública segura CPA subyacente a KYBER.
- **crypto_kem_enc:** genera texto cifrado y secreto compartido para una clave pública dada.
    - **indcpa_enc:** función de cifrado del esquema de cifrado de clave pública seguro CPA, subyacente a KYBER.
- **crypto_kem_dec:** genera un secreto compartido para un texto cifrado y una clave privada dados.
    - **indcpa_dec:** función de descifrado del esquema de cifrado de clave pública seguro CPA, subyacente a KYBER.
    - **indcpa_enc_cmp:** Compara el texto cifrado recifrado con el texto cifrado original byte por byte. La comparación se realiza en tiempo constante.

En las implementaciones en software se encuentra la aplicación CRYSTALS-KYBER que contiene los algoritmos básicos listados anteriormente. Estos algoritmos utilizan además funciones para manejar polinomios, que nos permiten la serialización, deserialización, compresión, y comparación de estos. Estas funciones se enlistan a continuación:

- **indcpa_dec**: función de descifrado del esquema de cifrado de clave pública seguro CPA subyacente a KYBER.
- **poly_compress**: serializa un polinomio y lo comprime subsecuentemente.
- **poly_decompress**: deserializa un polinomio y lo descomprime subsecuentemente, siendo el inverso aproximado de **poly_compress**.
- **poly_packcompress**: serializa y comprime subsecuentemente un polinomio de un polyvec, escribiendo a una representación en cadena de bytes del polyvec completo. Se usa para comprimir un polyvec de uno en uno en un bucle.
- **poly_unpackdecompress**: deserializa y descomprime subsecuentemente un polinomio de un polyvec, usándose para descomprimir un polyvec de uno en uno en un bucle.
- **cmp_poly_compress**: serializa y compara un polinomio con un polinomio serializado.

- **cmp_poly_packcompress**: serializa y compara un polinomio de un polyvec con un polyvec serializado, debiendo llamarse en un bucle sobre todos los polinomios de un polyvec.
- **poly_tobytes**: serializa un polinomio.
- **poly_frombytes**: deserializa un polinomio; es el inverso de **poly_tobytes**.
- **poly_frombytes_mul**: multiplica un polinomio con la deserialización de otro polinomio.
- **poly_frommsg**: convierte un mensaje de 32 bytes en un polinomio.
- **poly_tomsg**: convierte un polinomio en un mensaje de 32 bytes.
- **poly_zeroize**: pone todos los valores de un polinomio en cero.
- **polyvec_compress**: comprime y serializa un vector de polinomios.
- **polyvec_decompress**: deserializa y descomprime un vector de polinomios; es el inverso aproximado de **polyvec_compress**.

Estas funciones configuran valores iniciales como constantes, dimensiones de matrices o vectores, o valores precomputados necesarios para otras partes del algoritmo. Transforman datos entre representaciones binarias, polinomiales o vectoriales. Utilizan compresión de polinomios o vectores para reducir el tamaño de los datos transmitidos. Realizan operaciones básicas como copiar, mover, concatenar, o reorganizar datos, y sin realizan funciones complejas, es decir, preparan los datos para poder llamar a la NTT o las primitivas de la familia SHA-3 implementadas en hardware que nos permiten completar el funcionamiento del criptosistema.

4.3. Diseño de la partición en hardware

Esta sección presenta en detalle el diseño de las dos funciones, SHA-3 y NTT, que fueron seleccionadas para ser diseñadas e implementadas en hardware mediante el lenguaje VHDL.

Ambos módulos fueron diseñados para ser conectados al procesador a través de la interfaz AXI. Los detalles de conexión se presentan en el siguiente capítulo por considerarse parte de los resultados.

4.3.1. Diseño del módulo SHA-3

El desarrollo de un módulo SHA-3 a nivel de IP para CRYSTALS-Kyber plantea el reto de optimizar el establecimiento de claves criptográficas. Dado que CRYSTALS-Kyber es un esquema postcuántico, su implementación en servidores con muchas peticiones simultáneas puede generar una alta carga. El reto es garantizar un proceso de establecimiento de claves eficiente, minimizando la latencia y el uso de recursos

sin comprometer la seguridad frente a amenazas cuánticas.

La arquitectura del módulo SHA-3 utilizado en KYBER, ha sido diseñada considerando el 3er nivel de seguridad de CRYSTALS-KYBER para dispositivos embebidos: KYBER 768 con las diferentes características técnicas (ver la tabla 4.1).

Tabla 4.1: Funciones de las primitivas de SHA-3 en KYBER-768, entradas y salidas.

Primitiva	**Función**	**Entrada [bits]**	**Salida [bits]**
SHA3-256	H(m)	256	256
SHA3-512	G(d)	256	512
SHA3-512	G(m||H(pk))	512	512
SHAKE-128	XOF(ρ, i, j)	272	768
SHAKE-128	XOF(ρ, j, i)	272	768
SHAKE-256	PRF(σ, N)	264	1024
SHAKE-256	PRF(r, N)	264	1024
SHAKE-256	KDF(K||H(c))	512	256
SHAKE-256	KDF(z||H(c))	512	256

Cada una de estas primitivas tiene una configuración diferente como es la descrita en la Tabla 3.6. Estas primitivas comparten la misma función de permutación KECCAK-$p[b, nr]$ descrita en la subsección 3.5.5, en la pág. 49.

La figura 4.2 muestra el diagrama del módulo SHA-3 es el diseño principal de nuestra propuesta para implementar las cuatro primitivas de la familia SHA-3 empleadas en CRYSTALS-KYBER. La arquitectura de SHA-3 consiste principalmente en los siete módulos:

- Pad
- Estado Zero
- XOR
- Keccak-*p*
- CtrlSHA-3
- Truncar

Cada módulo, realiza funciones distintas, las cuales se describen en las siguientes subsecciones.

Pad

Este módulo garantiza que los datos de entrada cumplen con las especificaciones del estándar de *padding* para todas las primitivas de la familia SHA-3 antes de ser procesados por la función Keccak. El diseño de la unidad de Padding se presenta en la figura 4.3.

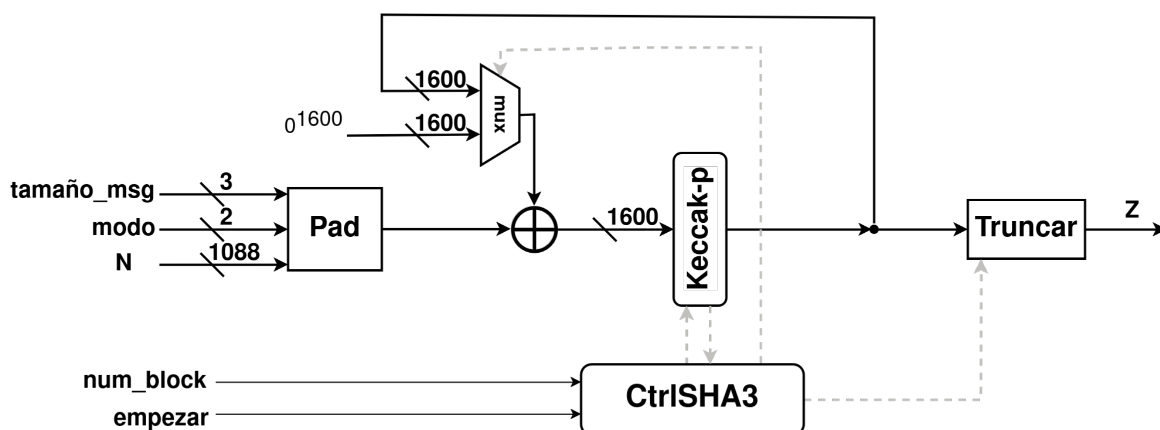


Figura 4.2: Diagrama de la arquitectura para el módulo SHA-3.

Estado Zero

La primera iteración del algoritmo, independientemente de las primitivas utilizadas, comienza con un estado inicial en ceros, almacenado en un registro que tiene una capacidad de 1600 bits. Para configurar el estado Zero, se introduce el vector identificado como 0^{1600} que actúa como una de las entradas del multiplexor, un dispositivo de 2 a 1 empleado para la realimentación cuando se procesa un mensaje compuesto por múltiples bloques. La señal de control de este multiplexor es generada por la unidad de control del componente principal. Todas las señales de control se presenta con una línea discontinua en gris claro.

XOR

Se trata del módulo de XOR a partir de la versión de la primitiva de SHA-3 que se va procesar. El diagrama de esta unidad observa en la figura 4.3 y está formado por 1344 bits XOR para el almacenamiento inicial y cuatro bloques de concatenación (SHA-3 256, SHA-3 512, SHAKE128 y SHAKE 256) que se encargan de construir el estado adecuado por primitiva.

Las entradas XOR son:

- MENSAJE_SHA-3: es el mensaje de entrada al núcleo SHA-3. Este mensaje es recolectado y reorganizado adecuadamente por la unidad de control, como se describe en la Sección 5.2. La cantidad de operaciones XOR necesarias varía según la primitiva utilizada: 1088 para SHA-3-256 y SHAKE256, 576 para SHA-3-512 y 1344 para SHAKE128, representando el rate;
- ESTADO_RETROALIMENTACIÓN: corresponde al estado cero en la primera iteración y al estado de retroalimentación retrasado por un ciclo de reloj, derivado del módulo Keccak-p en el caso de mensajes multibloque. La selección de la versión se realiza únicamente después de la operación XOR. Independientemente de la primitiva utilizada, se procesan 1344 evaluaciones, lo

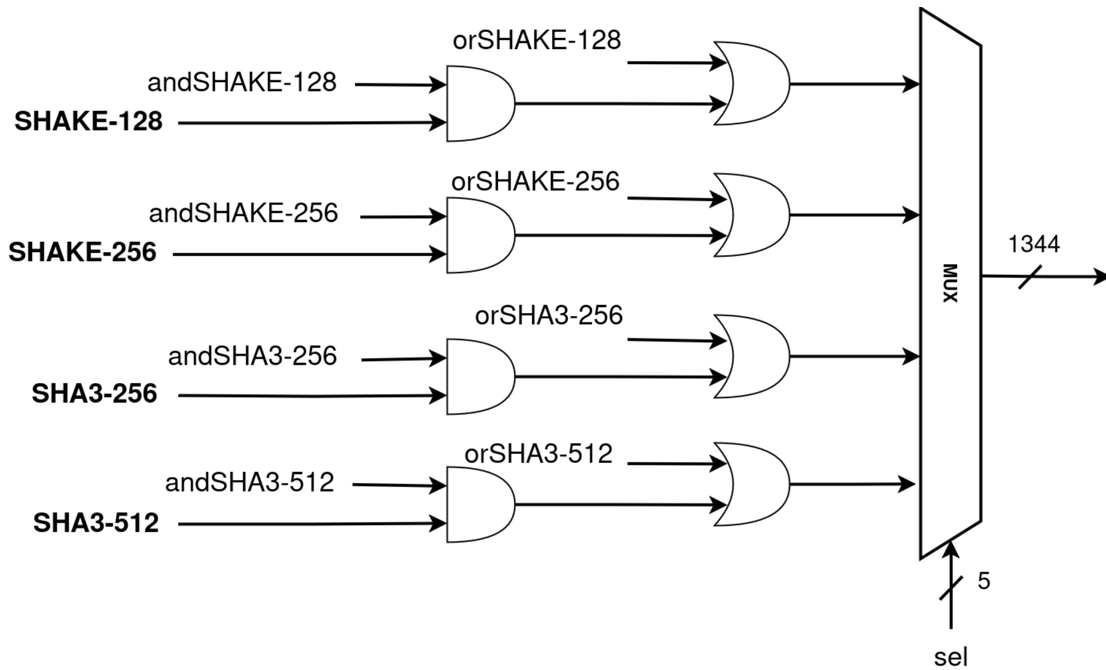


Figura 4.3: Diagrama de la unidad de Padding.

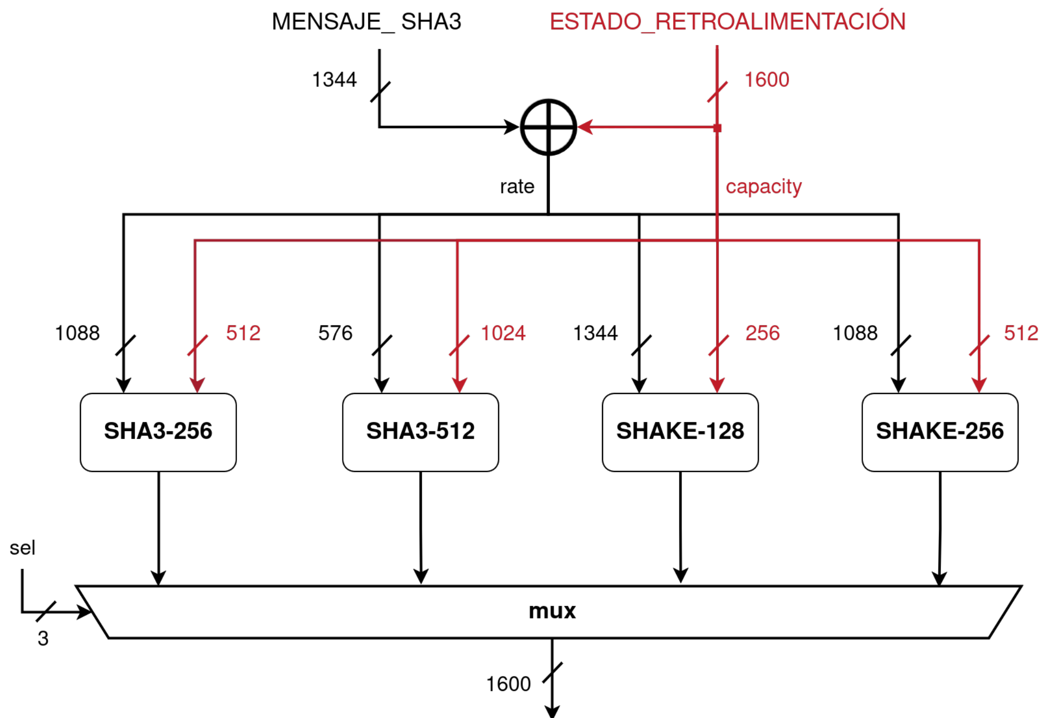


Figura 4.4: Diagrama de la unidad de XOR.

que permite implementar un menor número de puertas XOR en comparación con un diseño donde la selección se realiza al inicio. Inicialmente, se toman r bits de la salida de la puerta XOR, los cuales se concatenan con c bits del `ESTADO_RETROALIMENTACIÓN`.

Keccak-p

Para lograr un correcto funcionamiento del núcleo Keccak-p, deben definirse los mecanismos que controlan las entradas, salidas y almacenamiento de estados. La figura 4.5 ilustra los bloques principales de la arquitectura para el núcleo Keccak-p a partir del diagrama general. Tiene un registro de 1600 bits (Reg. Estado), una unidad de control de permutación (CtrlPermutation) con tres estados (reset, esperar, rondas), un multiplexor que controla la entrada a la permutación, un memoria ROM con las constantes de ronda, y el módulo de Keccak-Permutación. Estos módulos controlan las rondas a dar, los datos de entrada y salida, y el reinicio junto con las señales de control.

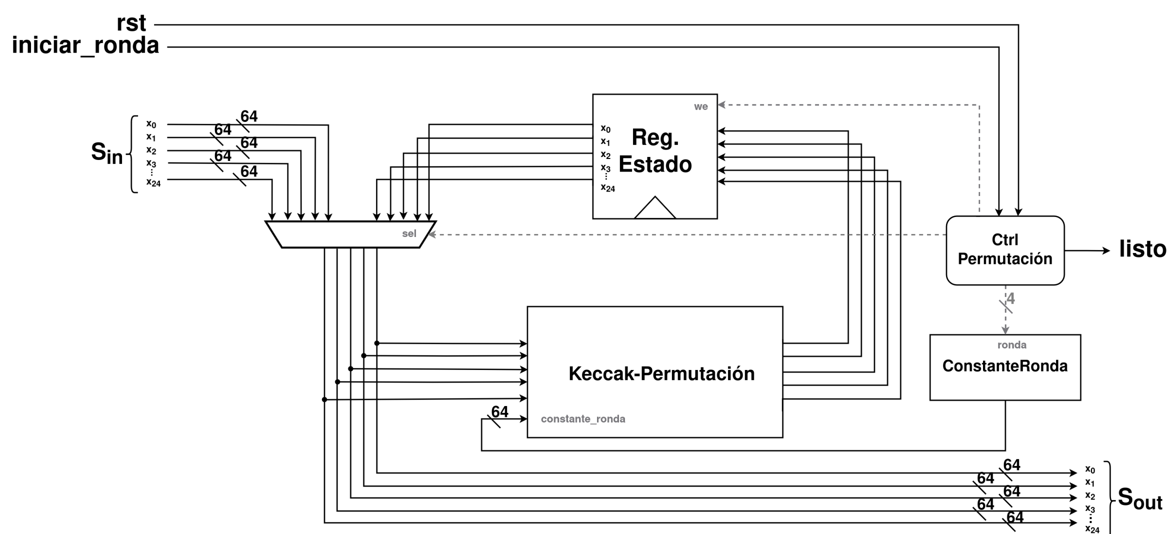
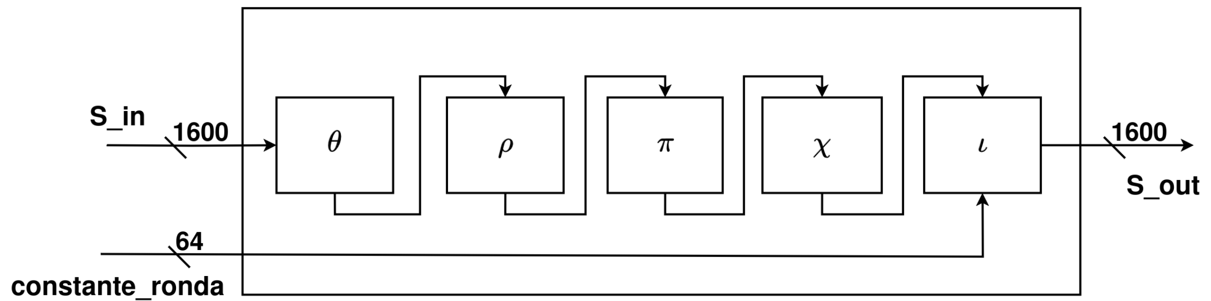


Figura 4.5: Diagrama del módulo Keccak completo.

- Keccak-Permutación: también conocida como función de ronda, esta se muestra en la figura 4.6, toma como entrada una matriz que representa el estado actual del sistema, una ronda constante de 64 bits y produce una nueva matriz de estado actualizada. Dependiendo del número de rondas, éstas se inicializan utilizando una constante diferente. Realiza la función de permutación núcleo de las primitivas a implementar. Los módulos circundantes controlan las entradas, salidas, inicios y rondas realizadas.
- Ctrl Permutación (figura 4.7): la función principal de esta máquina es controlar el flujo de datos a la permutación del estado de entrada o el estado almacenado



Figura

Figura 4.6: Función de ronda Keccak.

en los registros, así como el número de rondas. Cuando se han completado las rondas, se envía la señal `valido` para indicar que la salida contiene datos válidos, y `listo` alta para indicar que se puede iniciar una nueva serie de rondas. También pasa al estado `esperar` para esperar una nueva señal `iniciar_ronda`.

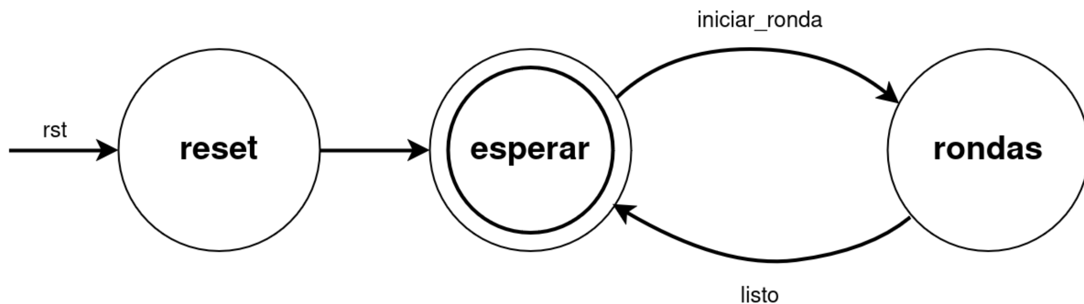


Figura 4.7: Diagrama de estados de la unidad de control del módulo Keccak.

- ConstanteRonda: este módulo consiste en una memoria ROM que almacena y proporciona las constantes necesarias para cada una de las 24 rondas de la permutación Keccak. Tiene como entrada una señal de tipo integer `ronda` que indica la ronda actual. Este índice seleccionará una de las 24 constantes almacenadas. Da una salida `constante_ronda` de tipo *std_logic_vector*(63 downto 0) que entrega la constante de 64 bits correspondiente a la ronda especificada por `ronda`.

CtrlSHA-3

Esta máquina de estados, ilustrada en la figura 4.8 fue diseñada para controlar el Módulo SHA-3. Inicia en el estado `reset`, donde se reinician las variables del sistema, y luego pasa a `idle`, el estado de espera. Si se activa la señal `iniciar = 1`, el sistema transita al estado `inicializar`, encargado de configurar los parámetros iniciales. Posteriormente, en `procesando mensaje`, se leen los datos de entrada. Una vez listos, el sistema avanza al estado `rondas` y envía una señal `iniciar_ronda` a la máquina de

estados que controla la permutación. Al finalizar las rondas, el flujo puede transitar a `hash` si `listo = 1`, donde se genera el `hash` final, para luego regresar al estado `idle` o volver a procesar otro mensaje si `offset = 1`. El estado intermedio `esperar`, se utiliza para asegurar un flujo correcto del proceso. Esta máquina de estados controla directamente la máquina de estados que controla la permutación.

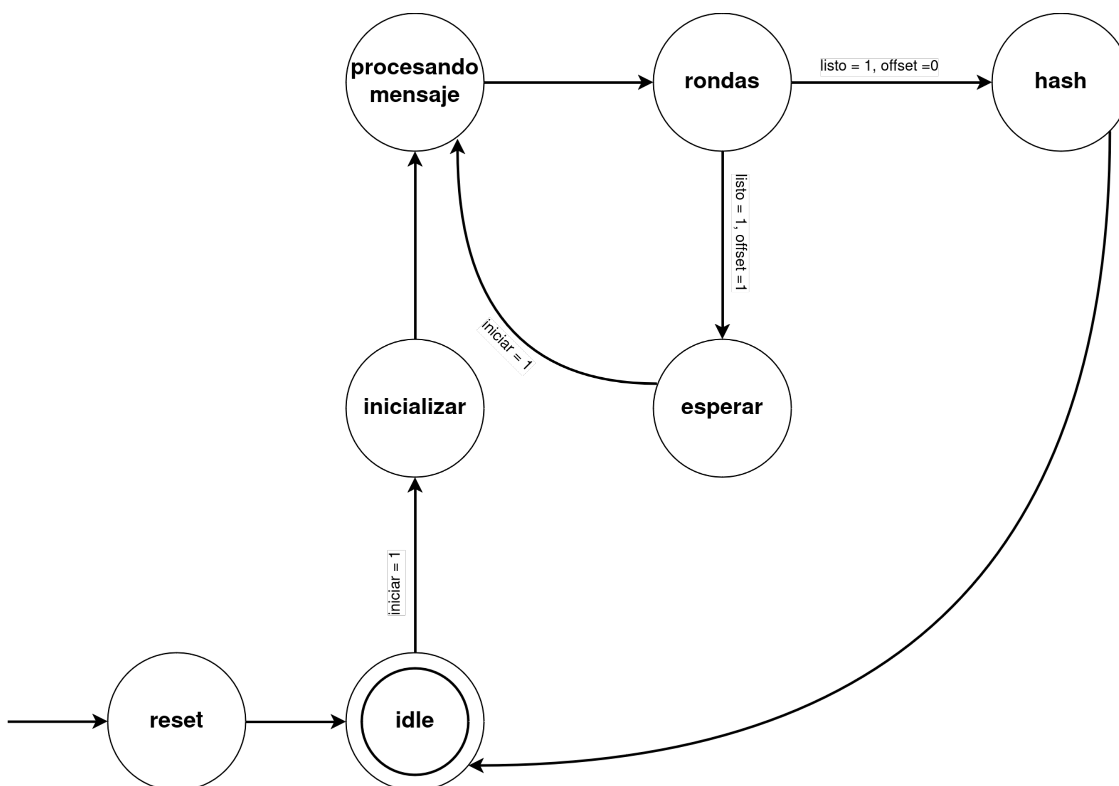


Figura 4.8: Diagrama de estados de la unidad de control del Módulo SHA-3.

Truncar

El módulo de truncamiento se utiliza para reorganizar correctamente el valor de la hash a la salida del módulo SHA-3. Esta unidad recibe como entrada una matriz tridimensional de tamaño $5 \times 5 \times 64$. Independientemente de la longitud de salida requerida, para obtener el resultado adecuado, esta matriz se reorganiza primero, y luego se envía a la salida como un único flujo de bits. Dado que la longitud de salida máxima requerida es de 1024 bits para la primitiva SHAKE256, sólo 1024 de 1600 se envían.

4.3.2. Implementación de la NTT

Nuestra solución propuesta maximiza la utilización del paralelismo inherente al NTT. Si nos fijamos en el flujo de datos representado en la figura 3.14, resulta evidente que existe un potencial significativo para los cálculos paralelos. Una NTT de

n puntos requiere $n/2$ operaciones de mariposa independientes por etapa. Para aprovechar esta ventaja, nuestro enfoque sugiere procesar cada una de las siete etapas simultáneamente.

Para ello, la arquitectura propuesta (ver la figura 4.9) utiliza unidades mariposa para llevar a cabo las 128 multiplicaciones modulares, 128 sustracciones modulares y 128 sumas modulares necesarias en cada una de las siete etapas, diseñado tanto para operaciones NTT como NTT^{-1}. Las entradas a cada etapa son coordinadas mediante una máquina de estados (Ver figura 4.10), que asegura la correcta secuenciación de las operaciones en cada etapa de la $\mathrm{NTT}/\mathrm{NTT}^{-1}$ conjunto con un mux.

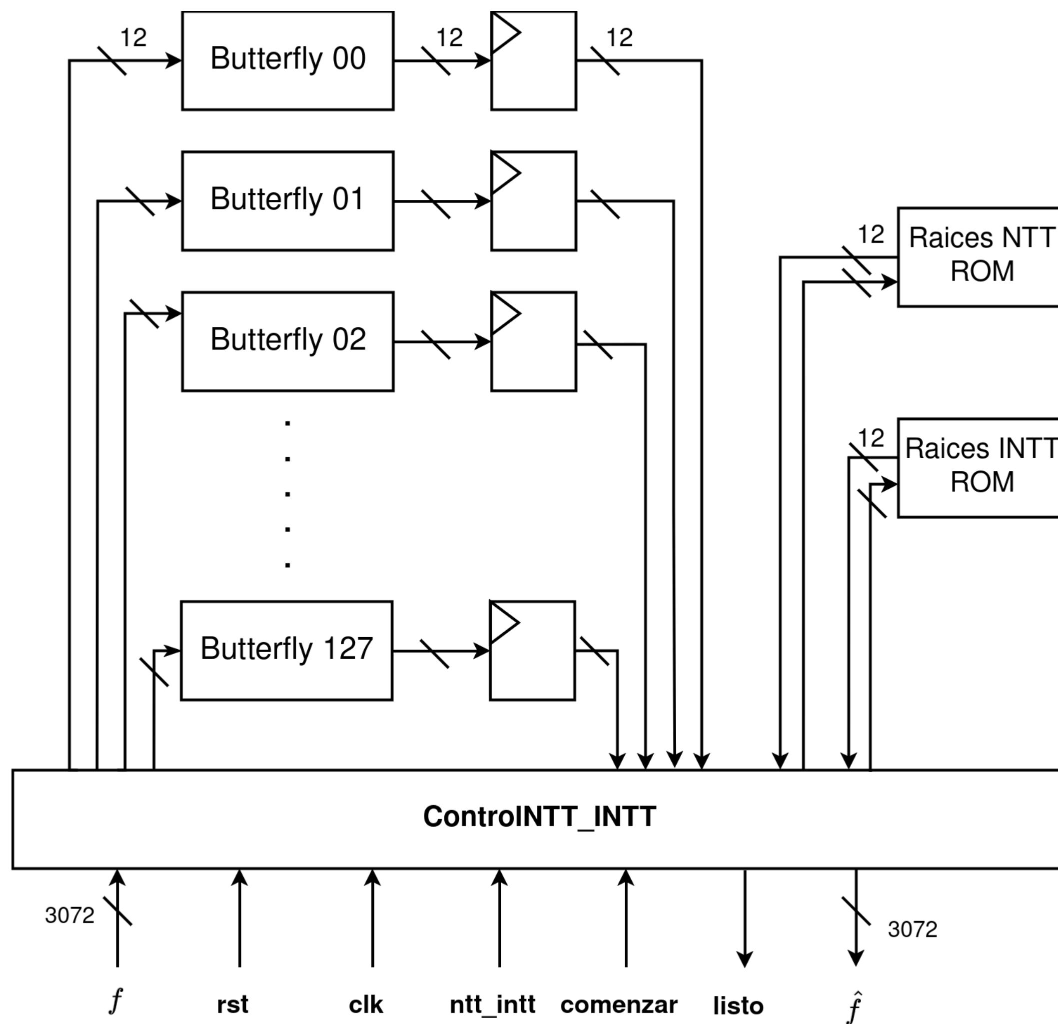


Figura 4.9: Arquitectura $\mathrm{NTT}/\mathrm{NTT}^{-1}$.

En este diseño, se implementó una unidad mariposa híbrida que combina las características de las mariposas CT y GS, con el objetivo de optimizar el uso de recursos

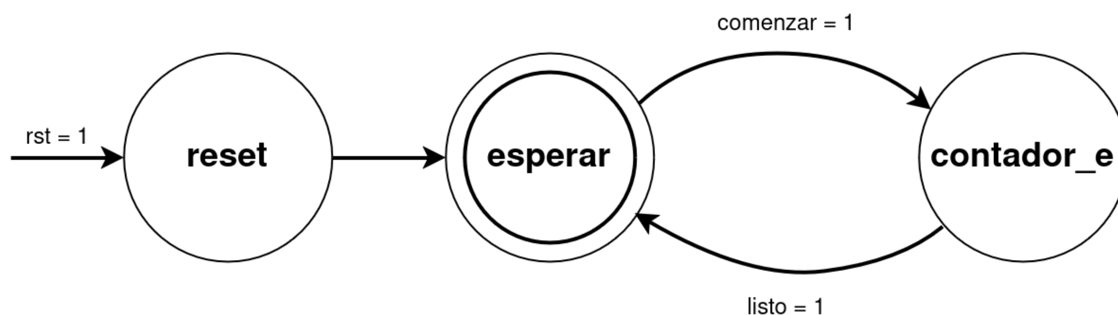


Figura 4.10: Control NTT_NTT^{-1}: máquina de estados y contador de las etapas.

en las operaciones aritméticas de la NTT e NTT^{-1}.El diseño de nuestra unidad mariposa híbrida se detalla en la figura 4.11. Solo utilizamos una unidad de reducción para la multiplicación modular en medio de la operación mariposa y empleamos un sumador/substractor modular en las configuraciones propuestas.

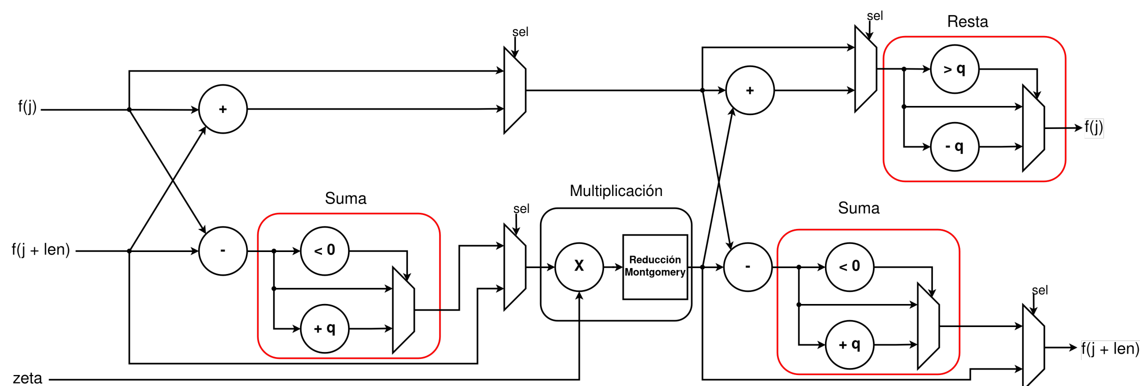


Figura 4.11: Diagrama del circuito de la unidad Mariposa Híbrida.

La arquitectura propuesta también implementa la reducción de Montgomery para llevar a cabo las multiplicaciones modulares dentro de la unidad mariposa híbrida. Este enfoque nos permitió realizar las multiplicaciones de manera eficiente al implementar una reducción específica para Kyber. El diseño esquemático del circuito se muestra en la figura 4.12. En nuestro diseño, el DSP reemplaza al multiplicador para realizar cálculos rápidos de datos de 12 bits, y las operaciones de desplazamiento sustituyen las operaciones de multiplicación del algoritmo original de Reducción de Montgomery.

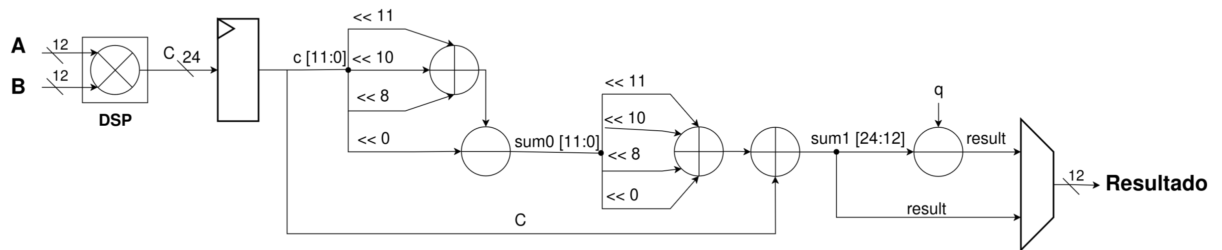


Figura 4.12: Diagrama de circuito para la Reducción Montgomery.

Capítulo 5

Resultados

5.1. Plataforma Final

El modelado de un sistema completo en Vivado sigue una metodología con base en plataformas. Esta metodología permite que el sistema esté compuesto por diferentes bloques IP, los cuales se interconectan mediante buses y interfaces AMBA AXI4 para garantizar que el diseño cumpla con los requisitos de la aplicación objetivo.

Además de los bloques personalizados desarrollados utilizando la metodología de síntesis de alto nivel, como los módulos de filtrado y cifrado, es necesario integrar otros módulos complementarios. Estos módulos adicionales aseguran que el sistema completo funcione de acuerdo con el comportamiento esperado. A continuación, se describen los bloques IP incluidos en la plataforma, destacando las características clave que fueron relevantes para este diseño.

- **Processing_system7_0**: Este bloque es el núcleo esencial de toda plataforma basada en el dispositivo Xilinx Zynq, ya que representa el sistema de procesamiento integrado. Su función principal es coordinar el funcionamiento del sistema, gestionando tareas como la comunicación entre la FPGA y las interfaces de entrada/salida, el control de la memoria DDR externa, la administración de temporizadores e interrupciones, y la ejecución del software embebido en los núcleos de propósito general que incorpora el dispositivo.
- **Rst_processing_system7_0_50M**: Este módulo se encarga de gestionar las señales de reinicio de todos los bloques IP que conforman la plataforma. Garantiza que, después de cada reinicio, todos los módulos comiencen desde un estado inicial predefinido y consistente, asegurando una configuración adecuada del sistema y evitando errores derivados de estados indefinidos.
- **ps7_0_axi_periph**: Este bloque actúa como un nodo central de interconexión AXI que vincula los periféricos y la lógica personalizada con el sistema de procesamiento del Zynq-7000. No solo establece los enlaces adecuados entre los diversos bloques IP de la plataforma, sino que también adapta las frecuencias

de operación entre diferentes módulos conectados. Esta capacidad de adaptación permite que dispositivos maestro y esclavo que operan a tasas de datos distintas puedan comunicarse de manera eficiente, mejorando así la flexibilidad y compatibilidad del diseño.

- **IP_NTTINTT_0**: Este bloque IP implementa la NTT y su inversa NTT^{-1}. Su incorporación permite llevar a cabo cálculos intensivos en paralelo dentro del hardware, reduciendo significativamente el tiempo de ejecución en aplicaciones como la criptografía postcuántica. Además, su diseño modular facilita la integración con otros bloques del sistema.
- **hashModule_0**: Este módulo IP implementa las primitivas de la familia SHA-3, diseñado específicamente para operar con el protocolo CRYSTALS-Kyber. Al estar dedicado exclusivamente a la generación de hashes y funciones XOF en hardware, mejora el rendimiento en comparación con una implementación puramente software. También se integra de manera eficiente en la plataforma mediante la interconexión AXI, asegurando la interoperabilidad con otros componentes del sistema.

En la tabla 5.1 destaca el módulo **IP_NTTINTT_0**, que realiza las operaciones de la NTT/NTT^{-1}, consumiendo la mayor cantidad de recursos (33226 LUTs y 128 DSP). Otros submódulos como **hashModule_0** implementan funciones de la familia SHA-3 consumiendo 5606 LUTs, mientras que **ps7_0_axi_periph** y **processing_system7_0** gestionan la comunicación con el procesador ARM integrado mediante un bus AXI. Además, el módulo **rst_ps7_0_50M** controla tareas básicas como el reinicio del sistema, utilizando recursos mínimos.

Tabla 5.1: Resumen de utilización de recursos de la Plataforma completa.

Nombre	**Slice LUTs**	**Slice Registers**	**F7 Muxes**	**F8 Muxes**	**Slice**	**LUT como Lógica**	**LUT como Memoria**	**DSPs**	**IOB Conectados**	**BUFGCTRL**
IP_NTT_INTT_wrapper	39358	20102	6722	3201	12476	39296	62	128	130	3
IP_NTT_INTT_i (IP_NTT_INTT)	39358	20102	6722	3201	12476	39296	62	128	0	3
hashModule_0 (IP_NTT_INTT)	5606	6543	288	128	2188	5606	0	0	0	2
IP_NTTINTT_0 (IP_NTT_INTT)	33226	12873	6434	3073	10135	33226	0	128	0	0
processing_system7_0 (IP_NTT_INTT)	0	0	0	0	0	0	0	0	0	1
ps7_0_axi_periph (IP_NTT_INTT)	509	653	0	0	241	448	61	0	0	0
rst_ps7_0_50M (IP_NTT_INTT)	17	33	0	0	7	16	1	0	0	0

5.2. Módulo SHA-3

En esta sección, presentamos nuestros resultados de implementación en lenguaje VHDL para una FPGA AMD Zynq-7000 y los comparamos con los encontrados en la literatura. En particular, consideramos trabajos previos que utilizan dispositivos como Artix-7 y Zynq-7000 FPGA.

La tabla 5.6 muestra un resumen detallado de la utilización de recursos de hardware para la implementación del módulo hashModule_v1_0. Se proporciona un desglose detallado de los recursos utilizados en cada módulo del diseño, permitiendo identificar las áreas de mayor consumo de recursos. Aunque la instancia hashModule_v1_0 utiliza recursos significativos, las instancias y módulos internos tienen un uso menor, lo que demuestra una estructura optimizada.

El flujo temporal del módulo de hardware para SHA3-3 se muestra en la figura 5.2. Una vez que reset se desactiva, el módulo entra en un estado de reposo `idle`, esperando a que se configure el algoritmo (en este caso, SHA3-256) mediante la señal `modo` y se proporcione el mensaje de entrada a través de la señal `msg`. Este mensaje tiene su tamaño especificado en `tamaño_msg`, mientras que la señal `offset` gestiona si se procesará otro bloque de texto posterior. Cuando se activa la señal `iniciar`, el módulo sale del estado de reposo y entra en el estado de inicio (inic), donde el mensaje se prepara para el procesamiento. A continuación, el módulo pasa al estado p_m (procesando mensaje), donde comienza el núcleo del algoritmo. El procesamiento se realiza a través de 24 rondas, representadas por la señal `ronda`. Al finalizar las 24 rondas, el módulo entra en el estado hash, indicando que el cálculo ha concluido y que el resultado final está disponible en la señal de salida `hash`. Durante este tiempo, la señal `listo` se activa, notificando al sistema externo que el hash de 256 bits calculado está listo para ser utilizado.

Comparación con el estado del arte

La tabla 5.2 compara exhaustivamente el diseño SHA-3 propuesto con trabajos relacionados. La utilización de recursos en términos de área en el contexto de Kyber puede servir para evaluar el rendimiento computacional. La elección del Zynq-7000 nos ofrece ventajas en términos de integración de hardware y software debido a su arquitectura SoC.

Bisheh-Niasar et al. [11] proponen un conjunto de instrucciones de componentes para realizar muestreo polinómico, NTT y multiplicación punto por punto, con el fin de acelerar los sistemas criptográficos basados en redes de *lattices* PQC. La unidad Keccak se configura para realizar cuatro funciones: SHA3-256, SHA3-512, SHAKE-128 y SHAKE-256. Bisheh-Niasar y su equipo. Desarrollan un búfer dedicado para la interfaz con el núcleo de Keccak de 64 bits. En nuestra implementación, este búfer se ha diseñado con un ancho de 32 bits y su longitud ha sido ajustada al dato más

Tabla 5.2: Comparación del módulo SHA-3 con los trabajos relacionados en términos de área.

Diseño	Método	Dispositivo	Área		
			#LUTs	#FFs	#Slices
Guo et al. [49]	HW	Artix-7	4614	1771	1407
Bisheh et al. [11]	HW	Artix-7	4405	1629	1825
Banerjee et al. [50]	HW/SW	Artix-7	5784	1605	1716
Dolmeta et al. [51]	HW	Artix-7	9651	8697	3413
Este trabajo	**HW**	**Zynq-7000**	**4673**	**2638**	**2212**

extenso requerido, es decir, 1344 bits.

Banerjee et al. [50] presenta Sapphire, un procesador de criptografía basado en *lattices* con parámetros configurables. En este caso, se ha implementado un núcleo SHA-3 de bajo consumo. Un núcleo Keccak-f[1600] dentro de Sapphire se puede acceder de manera independiente a través del software RISC-V y se utiliza para acelerar el hash SHA-3 y las funciones de salida extendida según los requisitos del protocolo.

En En cuanto a la utilización del área, se ha producido una mejora del 11 % respecto al valor medio de las otras implementaciones, utilizando más slices que los diseños de Guo et al. [49] y Banerjee et al. [50], aunque menos que el de Dolmeta et al. [51].

Con respecto a los Ciclos de reloj, Frecuencia Máxima y Latencia en la tabla 5.3 se hace una comparación con los trabajos del estado del arte.

Tabla 5.3: Comparación del módulo SHA-3 con los trabajos relacionados en términos de velocidad.

Diseño	Frec.Max.	Ciclos	Latencia
	[MHz]	[CCs]	[μs]
Guo et al. [49]	159	-	-
Bisheh et al. [11]	115	24	0.20
Banerjee et al. [50]	25	24	0.96
Dolmeta et al. [51]	250	39	0.15
Este trabajo	**145**	**28**	**0.19**

Para calcular el número de ciclos de reloj transcurridos durante el tiempo de simulación, se utilizó la siguiente fórmula. En este caso, esimulaciónl tiempo total de simulación es de 560.000 ns, y el período del reloj es de 20.000 ns.

$$\text{Ciclos de reloj} = \frac{510.000\,\text{ns}}{20.000\,\text{ns}} = 28.000\,\text{ciclos de reloj}$$

La latencia (us) se calcula con la fórmula:

$$\text{Latencia} = \frac{\text{Ciclos Totales de Reloj}}{\text{Frecuencia del Reloj (MHz)}}$$

Donde:

- **Ciclos Totales de Reloj:** Número total de ciclos que tarda un proceso en completarse (28 ciclos).
- **Frecuencia del Reloj (MHz):** Se calcula como:

$$\text{Frecuencia (MHz)} = \frac{1}{\text{Periodo del Reloj (us)}} = \frac{1}{6.897} = 145 \text{ MHz}$$

Sustituyendo los valores:

$$\text{Latencia} = \frac{28}{145} = 0.193 \text{ us}$$

En cuanto a la frecuencia máxima, nuestro trabajo alcanza los 145 MHz, que es competitiva. Es superior a la de Bisheh et al.[11] y Banerjee et al. [50], siendo inferior a la de Guo et al. [49] y Dolmeta et al. [51]. La alta frecuencia de Dolmeta et al. [51] está asociada a su mayor consumo de recursos.

En ciclos de reloj, nuestra implementación requiere 28 ciclos, que es un número moderado en comparación con otros trabajos. Es más que los 24 ciclos requeridos por Bisheh et al. [11] y Banerjee et al. [50], y menos que los 39 ciclos de Dolmeta et al. [51].

5.3. Módulo NTT/NTT^{-1}

La tabla 5.7 proporciona un resumen de la utilización de recursos de hardware para un diseño IP_NTTINTT_v1_0, muestra cómo se distribuyen los recursos dentro de diferentes módulos del sistema. El diseño más grande es el For_Hybrid_Butterfly, que utiliza una gran cantidad de recursos, especialmente LUTs y registros, junto con DSPs e I/O conectados.

La figura 5.3 presenta una simulación del flujo obtenida mediante el entorno de diseño Vivado. utilizada para analizar el comportamiento temporal de un diseño implementado en VHDL para el cálculo de la NTT. Una vez desactivado el `reset`, el sistema espera la activación de la señal inicio, que marca el comienzo del procesamiento. Durante la operación, la señal etapa avanza progresivamente, indicando las fases del algoritmo NTT, mientras que el `estado` refleja el flujo de control interno, pasando por estados como reset", .esperarz çontador_etapa". La entrada principal del diseño, representada por la señal `A_f_j`, contiene los datos iniciales en formato hexadecimal. Una vez completadas todas las etapas del algoritmo, la señal `listo` se activa, indicando que la `salida` contiene los datos transformados.

Comparación con el estado del arte

La tabla 5.4 compara este trabajo con otras implementaciones de CRYSTALS-Kyber. Todas estas implementaciones eligen el mismo conjunto de parámetros, a saber, n=256 y q=3329. Las principales distinciones entre estos diseños son sus diferentes números de radix de NTT y diferentes algoritmos de reducción modular. Además, en la literatura se reportan implementaciones en dispositivos como Artix-7, Zynq-7000 FPGA y microcontroladores ARM Cortex-M4. Esto es particularmente relevante, ya que permite analizar el impacto de las características específicas de cada plataforma, como su capacidad de paralelismo, recursos disponibles, frecuencia operativa y consumo energético, sobre el desempeño global del nuestra propuesta.

Tabla 5.4: Evaluación del desempeño de NTT/NTT^{-1} en términos de área.

Diseño	Método	Mariposa	Área			
			#LUTs	#FFs	#DSPs	#BRAM
Botros et al. [52](2019) [a]	SW	Baja	-	-	-	-
Xing et al. [53](2021)[c]	HW	2	1,737	1,167	2	3
Fritzmann et al. [54](2020)[b]	HW	2	2908	170	9	0
Bisheh et al. [11] (2021)[c]	HW	2 x 2	360	145	3	2
Itabashi et al. [55] (2022)[c]	HW	Media	274	181	1	1.5
Saoudi et al. [42] (2024)[c]	HW	64	18296	12134	64	0
Este trabajo[b]	HW	128	33420	12640	128	0

[a] *Implementado en Cortex M4.*
[b] *Implementado en FPGA AMD Zynq-7000.*
[c] *Implementado en FPGA AMD Artix-7.*

Xing et al. [53] presentan una implementación completamente en hardware de CRYSTAL-Kyber, logrando un rendimiento decente con recursos de hardware limitados para una programación elaborada de muestreo y cálculos relacionados con la NTT.

Bisheh et al. [11] proponen una arquitectura de hardware de conjunto de instrucciones sintetizadas para Xilinx Artix-7 FPGA para evaluación y pruebas de rendimiento. Proporcinan un conjunto de componentes eficiente y de alto rendimiento para realizar muestreo polinómico, NTT y multiplicación puntual para acelerar la PQC con base en *lattices*. La implementación de componentes eficientes, incluidos núcleos de muestreo, NTT y arquitecturas de multiplicación puntual, aumenta el rendimiento en comparación con las implementaciones de SW y HW/SW.

Mientras que Itabashi et al. [55], presentan un diseño de hardware que realiza eficientemente la NTT a través de una ruta de datos de unidad mariposa de NTT

equipada con un multiplicador modular propuesto por ellos. Los resultados muestran que los aceleradores NTT propuestos logran hasta un 3 % y un 26 % más de eficiencia área-tiempo en términos de LUT y FF, respectivamente que Bisheh-Niasar et al. [11]. Evalúan su rendimiento en Xilinx Artix-7.

A diferencia de otros trabajos Saoudi et al. [42], nuestra una solución que maximiza la utilización del paralelismo inherente al NTT. Para ello, proponen una arquitectura que abarca 64 multiplicaciones modulares, 64 sustracciones modulares y 64 sumas modulares necesarias para cada una de estas siete etapas.

Tabla 5.5: Evaluación del desempeño de NTT/NTT^{-1} en términos de velocidad.

Diseño	Frec.Max.	Ciclos	Latencia
	[MHz]	[CCs]	[μs]
Botros et al. [52](2019) [a]	100	7 725/9 347	77.25
Xing et al. [53](2021)[c]	161	512	3.18
Fritzmann et al. [54](2020)[b]	-	1 935/1 930	-
Bisheh et al. [11] (2021)[c]	115	940	8.17
Itabashi et al. [55] (2022)[c]	208	902	4.34
Saoudi et al. [42] (2024)[c]	210	85/104	0.40/0.50
Este trabajo[b]	53	9/9	0.17

Una comparación con los trabajos del estado del arte en términos de velocidad (ciclos de reloj, frecuencia máxima y latencia) se muestra en la tabla 5.5. Nuestro diseño demuestra un desempeño destacado, especialmente en términos de ciclos y latencia. Con solo 9 ciclos, este diseño presenta una arquitectura altamente eficiente en comparación con los demás trabajos evaluados en el estado del arte. Además, alcanza una latencia de 0.17 μs, lo que lo posiciona como el mejor resultado en latencia siendo 2.3 veces más rápido que el mejor resultado de la literatura.

En comparación con Saoudi et al. (2024), también logra una baja latencia (0.40/0.50 μs), pero requiere un mayor número de ciclos (85/104) y opera a una frecuencia más alta (210 MHz). Por otro lado, Xing et al. (2021) alcanza una latencia de 3.18 μs, a pesar de operar a una frecuencia más alta (161 MHz) y tener menos ciclos que algunos diseños (512).

En comparación con Botros et al. (2019), nuestro diseño muestra una clara superioridad. Mientras que este requiere **7 725 ciclos** y alcanza una latencia de **77.25** μ**s**, nuestro diseño supera ampliamente este resultado, demostrando una eficiencia muy superior.

Nuestro diseño destaca como el mejor en términos de latencia debido a su arquitectura optimizada que minimiza los ciclos necesarios para completar las operaciones. Este resultado resalta su eficiencia y su potencial para aplicaciones donde el tiempo de ejecución es crítico.

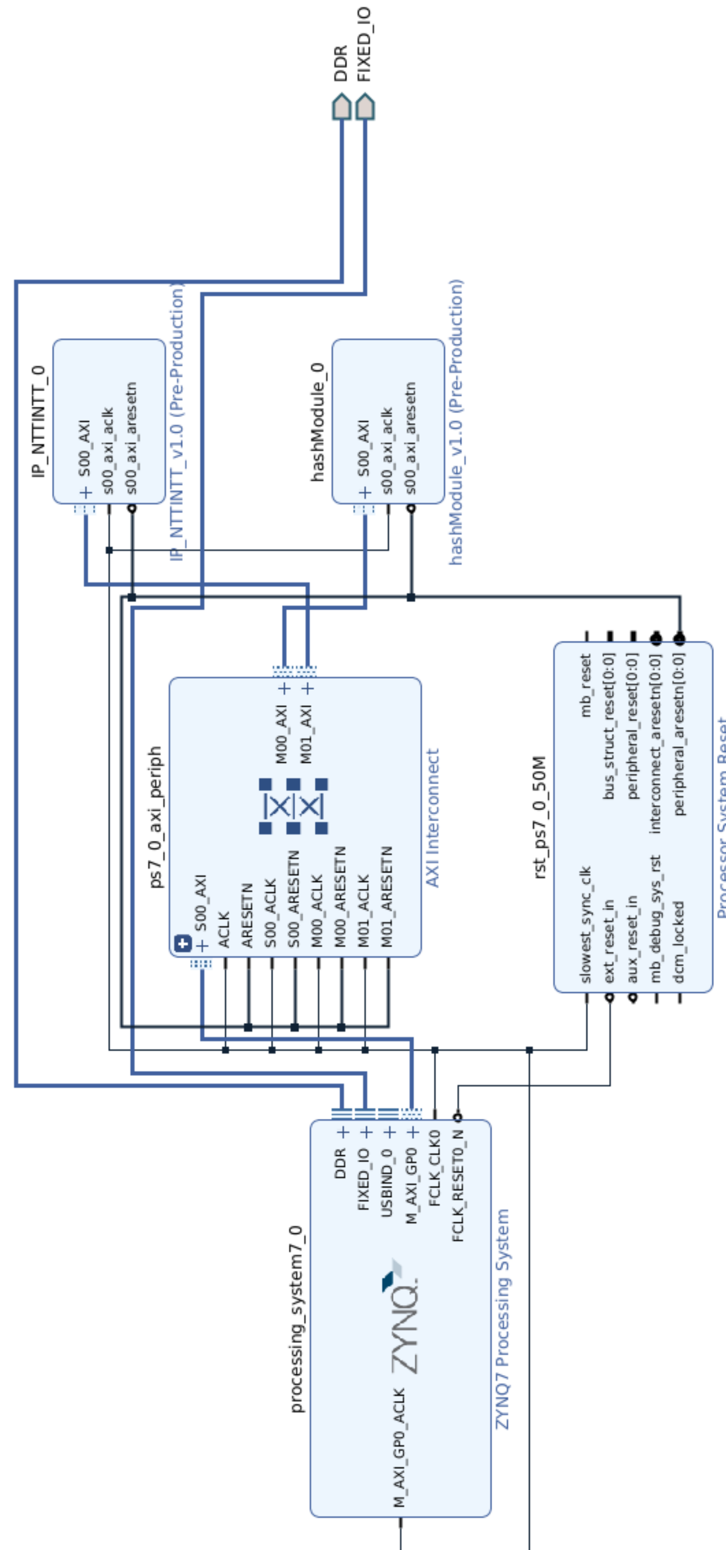


Figura 5.1: Arquitectura de la plataforma final.

Tabla 5.6: Resumen de utilización de recursos para `hashModule_v1_0`.

Nombre	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	Slice	LUT como Lógica	DSPs	IOB Conectados	BUFGCTRL
hashModule_v1_0	5614	6543	288	128	2048	5614	0	98	3
hashModule_v1_0_AXI_inst	5614	6543	288	128	2048	5614	0	0	3
uutHashM (hashModule)	4852	5331	0	0	1767	4852	0	0	0
StateM_Ctrl (SM_Ctrl)	7	6	0	0	6	7	0	0	0
u_P (KeccakF1600_StatePermute)	3884	1610	0	0	1130	3884	0	0	0
u_pad (pad)	961	1090	0	0	511	961	0	0	0
u_reg_hash (reg_1024bits)	0	1025	0	0	261	0	0	0	0
u_XOR (XORBytes)	0	1600	0	0	544	0	0	0	0

Tabla 5.7: Resumen de utilización de recursos para IP_NTTINTT_0.

Nombre	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	Slice	LUT como Lógica	DSPs	IOB Conectados	BUFGCTRL
IP_NTTINTT_v1_0	33420	12640	6434	3073	10486	33420	128	100	1
IP_NTTINTT_v1_0_S00_AXI	33420	12640	6434	3073	10486	33420	128	0	0
utt (For_Hybrid_Butterfly)	32216	9479	6050	2881	10204	32216	128	0	0
Control (NTT_Control)	9698	7	3984	1008	4821	9698	0	0	0
Registro3072	13206	3072	2066	1873	5629	13206	0	0	0
Mariposa	73	50	0	0	45	73	1	0	0

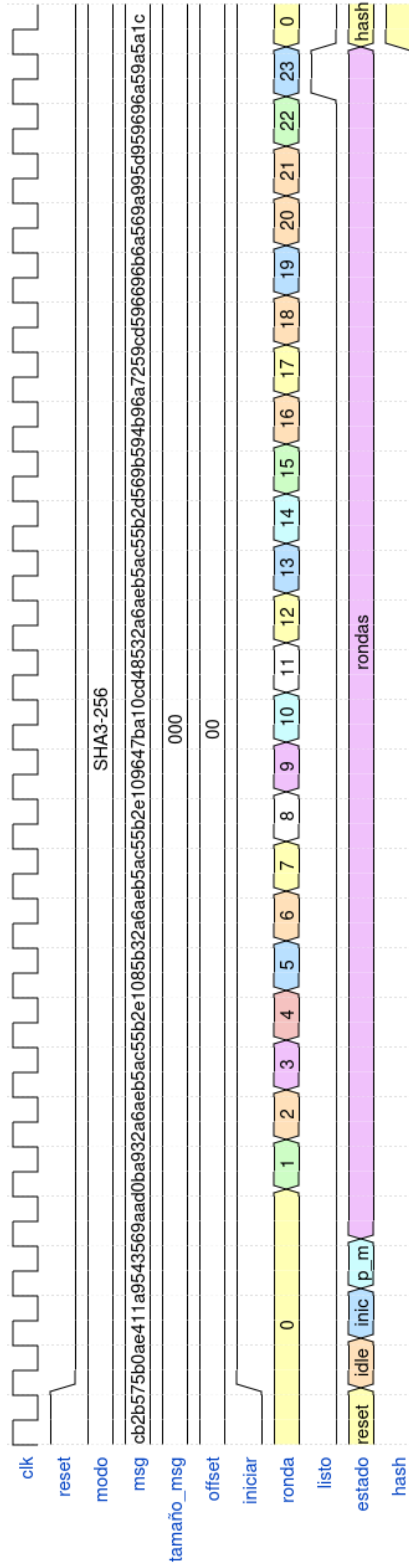


Figura 5.2: Diagrama de tiempos de control de flujo general para el módulo SHA-3.

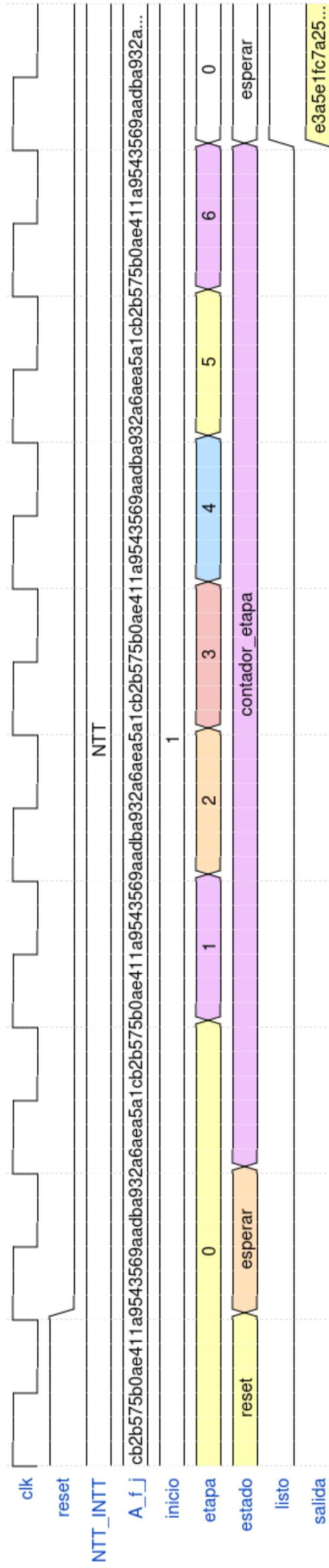


Figura 5.3: Diagrama de tiempos de control de flujo general para NTT/NTT^{-1}.

Capítulo 6

Conclusiones

En esta tesis se abordó el diseño e implementación de un codiseño HW/SW con VHDL/C para mejorar la eficiencia del algoritmo CRYSTALS-Kyber en una placa de desarrollo Zybo z7 20 que combina un SoC Zynq-7000 de Xilinx, el cual integra un procesador ARM Cortex-A9 de doble núcleo con lógica programable FPGA de la serie 7. Para el desarrollo de este trabajo se hizo un análisis detallado del flujo de datos del algoritmo, el diseño y verificación de los módulos hardware, así como la integración con el software. A continuación, se presentan las conclusiones obtenidas de este proceso:

Se identificaron las operaciones críticas en el algoritmo de criptografía postcuántica CRYSTALS-Kyber que podrían beneficiarse significativamente de una implementación en hardware. En particular, las operaciones relacionadas con la NTT y el cálculo de *hash* y XOF utilizando la familia SHA-3 fueron seleccionadas debido a su alto impacto en el tiempo de ejecución general del algoritmo. Este análisis permitió priorizar aquellas tareas donde el hardware podría aportar mayor eficiencia en términos de velocidad y paralelismo.

Se diseñó e implementó una partición eficiente entre hardware y software que maximiza el desempeño del algoritmo CRYSTALS-Kyber. El hardware fue responsable de acelerar la NTT y la familia SHA-3, mientras que el software se encargó del flujo lógico y del manejo de los datos. Este enfoque híbrido demostró un balance óptimo entre flexibilidad y desempeño, reduciendo la carga computacional del software y explotando las ventajas de la paralelización en hardware. Se evaluó exhaustivamente el desempeño de la solución híbrida propuesta frente a implementaciones previas del estado del arte. Las métricas empleadas, como el tiempo de ejecución y la utilización de recursos de hardware, mostraron mejoras significativas. En particular, se logró una implementación balanceada en cuanto a los tiempos de ejecución y el área, lo que valida la efectividad del enfoque propuesto. Además, se optimizó el uso de recursos del hardware en el módulo SHA-3, obteniendo un resultado equilibrado entre área y velocidad competente con el estado del arte. En cuanto al módulo NTT/NTT^{-1}, se alcanzó una latencia de 0.17 μs, destacándose como el mejor en este aspecto, con solo

9 ciclos. La comparación con otros trabajos de la literatura muestra que la propuesta es significativamente más rápida y eficiente, especialmente en términos de latencia.

El codiseño HW/SW desarrollado para el algoritmo de criptografía postcuántica CRYSTALS-Kyber logró un desempeño competitivo en comparación con el estado del arte, logrando cumplir con el objetivo general del proyecto. La integración eficiente entre hardware y software demostró ser una solución robusta y escalable, capaz de competir con el estado del arte en términos de desempeño y eficiencia. Este trabajo no solo contribuye al desarrollo de criptografía postcuántica más eficiente, sino que también establece una base para futuras investigaciones y mejoras en algoritmos similares.

6.1. Trabajo futuro

- Realizar una implementación de CRYSTALS-Kyber completamente en hardware.
- Proteger nuestra implementación contra Ataques Laterales [1].
- Evaluar la integración de CRYSTALS-Kyber con soluciones de criptografía ligera, como el uso de algoritmos postcuánticos adaptados para entornos de baja potencia.
- Implementación de un sistema de comunicaciones postcuánticas utilizando CRYSTALS-Kyber.
- Simular el rendimiento de nuestra implementación en escenarios reales, como comunicaciones seguras en redes 5G, almacenamiento seguro en la nube, o blockchain

[1] Side-Channel Attacks, como comúnmente se conoce en inglés.

Bibliografía

[1] D. Inc., *Zybo Z7 Reference Manual*, 2024. Último acceso: 25 de noviembre de 2024.

[2] F. Maqsood, M. Ahmed, M. M. Ali, and M. A. Shah, "Cryptography: a comparative analysis for modern techniques," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 6, 2017.

[3] T. Monz, D. Nigg, E. A. Martinez, M. F. Brandl, P. Schindler, R. Rines, S. X. Wang, I. L. Chuang, and R. Blatt, "Realization of a scalable shor algorithm," *Science*, vol. 351, no. 6277, pp. 1068–1070, 2016.

[4] C. Ugwuishiwu, U. Orji, C. Ugwu, and C. Asogwa, "An overview of quantum cryptography and shor's algorithm," *Int. J. Adv. Trends Comput. Sci. Eng*, vol. 9, no. 5, 2020.

[5] D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature*, vol. 549, no. 7671, pp. 188–194, 2017.

[6] National Institute of Standards and Technology (NIST), "Post-quantum cryptography," 2021.

[7] V. Shoup, "A proposal for an iso standard for public key encryption," *Cryptology ePrint Archive*, 2001.

[8] A. Galimberti, G. Montanaro, W. Fornaciari, and D. Zoni, "An evaluation of the state-of-the-art software and hardware implementations of bike," *arXiv preprint arXiv:2212.10636*, 2022.

[9] N. I. of Standards and Technology, "Module-Lattice-Based Key-Encapsulation Mechanism Standard," 2024. Date Published: August 13, 2024.

[10] H. Li, Y. Tang, Z. Que, and J. Zhang, "Fpga accelerated post-quantum cryptography," *IEEE Transactions on Nanotechnology*, vol. 21, pp. 685–691, 2022.

[11] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, "Instruction-set accelerated implementation of crystals-kyber," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 11, pp. 4648–4659, 2021.

[12] M. R. Nosouhi, S. W. A. Shah, L. Pan, and R. Doss, "Bit Flipping Key Encapsulation for the Post-Quantum Era," *IEEE Access*, vol. 11, pp. 56181–56195, 2023.

[13] D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, *et al.*, "Classic mceliece," 2017.

[14] C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zémor, and I. Bourges, "Hamming quasi-cyclic (HQC)," *NIST PQC Round*, vol. 2, no. 4, p. 13, 2018.

[15] R. Azarderakhsh, M. Campagna, C. Costello, L. D. Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa, *et al.*, "Supersingular isogeny key encapsulation," *Submission to the NIST Post-Quantum Standardization project*, vol. 152, pp. 154–155, 2017.

[16] V. B. Dang, F. Farahmand, M. Andrzejczak, and K. Gaj, "Implementing and benchmarking three lattice-based post-quantum cryptography algorithms using software/hardware codesign," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 206–214, IEEE, 2019.

[17] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-kyber algorithm specifications and supporting documentation," *NIST PQC Round*, vol. 2, no. 4, pp. 1–43, 2019.

[18] F. Yaman, A. C. Mert, E. Öztürk, and E. Savaş, "A hardware accelerator for polynomial multiplication operation of crystals-kyber pqc scheme," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1020–1025, 2021.

[19] V. B. Dang, K. Mohajerani, and K. Gaj, "High-speed hardware architectures and fpga benchmarking of crystals-kyber, ntru, and saber," *IEEE Transactions on Computers*, vol. 72, no. 2, pp. 306–320, 2023.

[20] S. Trimberger, *Field-Programmable Gate Array Technology.* Springer Science & Business Media, 1994.

[21] S. M. Trimberger and J. J. Moore, "Fpga security: Motivations, features, and applications," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1248–1265, 2014.

[22] A. B. Abdallah, *Advanced Multicore Systems-On-Chip.* Springer, 2017.

[23] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq book: embedded processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 all programmable SoC.* Strathclyde Academic Media, 2014.

[24] P. Raja, D. K. Singh, and H. Jerath, "Introduction to hardware description languages (HDLs)," in *Advanced VLSI Design and Testability Issues*, pp. 93–109, CRC Press, 2020.

[25] M. H. Al Meer, "Programmable SoC for an XTEA encryption algorithm using a co-design environment replication performance approach," *Journal of Computer and Communications*, vol. 5, no. 11, pp. 40–59, 2017.

[26] M. Barr and A. Massa, *Programming embedded systems: with C and GNU development tools*. .O'Reilly Media, Inc.", 2006.

[27] Xilinx, *Zynq 7000 SoC Software Developers Guide (UG821)*, 2023. UG821.

[28] B. J. LaMeres, *Introduction to logic circuits & logic design with VHDL*. Springer Nature, 2023.

[29] Xilinx, "What's New - 2023.2 Release Highlights." https://www.xilinx.com/products/design-tools/vivado.html. Consultado el 30 de abril de 2024.

[30] Xilinx, "Vitis Unified Software Platform." https://www.xilinx.com/products/design-tools/vitis.html, 2024. Consultado el 30 de abril de 2024.

[31] E. National Academies of Sciences, Medicine, *et al.*, *Quantum computing: progress and prospects*. 2018.

[32] R. S. Sutor, *Dancing with Qubits: How quantum computing works and how it can change the world*. Packt Publishing Ltd, 2019.

[33] *Efficient Implementation of NTT Based Polynomial Multiplier for Fast PQC Multiplication*. PhD thesis, ResearchGate, 2024.

[34] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th annual symposium on foundations of computer science*, pp. 124–134, Ieee, 1994.

[35] IBM, "IBM Unveils New Roadmap to Practical Quantum Computing Era; Plans to Deliver 4,000+ Qubit System," 2022. [Consultado el 22 de julio de 2024].

[36] M. Harmalkar, K. Jain, and P. Krishnan, "A survey of post quantum key encapsulation mechanism," in *2024 5th International Conference on Mobile Computing and Sustainable Informatics (ICMCSI)*, pp. 141–149, IEEE, 2024.

[37] C. Peikert *et al.*, "A decade of lattice cryptography," *Foundations and trends® in theoretical computer science*, vol. 10, no. 4, pp. 283–424, 2016.

[38] O. Prieto Fernández *et al.*, "Crystals: Kyber y dilithium, los próximos estándares de criptografía post-cuántica," 2023.

[39] G. Alagic, G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, Y.-K. Liu, C. Miller, *et al.*, "Status report on the third round of the nist post-quantum cryptography standardization process," 2022.

[40] E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes," *Journal of Cryptology*, vol. 26, no. 1, pp. 80–101, 2013.

[41] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-kyber: Algorithm specifications and supporting documentation," tech. rep., Third-round submission to the NIST's post-quantum cryptography standardization process, 2020. Available at https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[42] M. Saoudi, A. Kermiche, O. H. Benhaddad, N. Guetmi, and B. Allailou, "Low latency fpga implementation of ntt for kyber," *Microprocessors and Microsystems*, vol. 107, p. 105059, 2024.

[43] N. I. of Standards and Technology, "Sha-3 standard: Permutation-based hash and extendable-output functions (fips pub 202)," Tech. Rep. FIPS PUB 202, National Institute of Standards and Technology, 2015.

[44] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[45] W. M. Gentleman and G. Sande, "Fast fourier transforms: for fun and profit," in *Proceedings of the November 7-10, 1966, fall joint computer conference*, pp. 563–578, 1966.

[46] P. Heckbert, "Fourier transforms and the fast fourier transform (fft) algorithm," *Computer Graphics*, vol. 2, no. 1995, pp. 15–463, 1995.

[47] H. Yang, R. Chen, Q. Wang, Z. Wu, and W. Peng, "Hardware acceleration of ntt-based polynomial multiplication in crystals-kyber," in *International Conference on Information Security and Cryptology*, pp. 111–129, Springer, 2023.

[48] C. Pham-Quoc, X.-Q. Nguyen, and T. N. Thinh, "Hardware/software co-design for convolutional neural networks acceleration: a survey and open issues," in *Context-Aware Systems and Applications: 10th EAI International Conference, ICCASA 2021, Virtual Event, October 28–29, 2021, Proceedings 10*, pp. 164–178, Springer, 2021.

[49] W. Guo, S. Li, and L. Kong, "An efficient implementation of kyber," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 3, pp. 1562–1566, 2022.

[50] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols." arXiv preprint arXiv:1910.07557, 2019.

[51] A. Dolmeta, M. Martina, and G. Masera, "Hardware architecture for crystals-kyber post-quantum cryptographic sha-3 primitives," in *2023 18th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, pp. 209–212, IEEE, 2023.

[52] L. Botros, M. J. Kannwischer, and P. Schwabe, "Memory-efficient high-speed implementation of kyber on cortex-m4," in *Progress in Cryptology–AFRICACRYPT 2019: 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9–11, 2019, Proceedings 11*, pp. 209–228, Springer, 2019.

[53] Y. Xing and S. Li, "A compact hardware implementation of cca-secure key exchange mechanism crystals-kyber on fpga," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 328–356, 2021.

[54] T. Fritzmann, G. Sigl, and J. Sepúlveda, "Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 239–280, 2020.

[55] Y. Itabashi, R. Ueno, and N. Homma, "Efficient modular polynomial multiplier for ntt accelerator of crystals-kyber," in *2022 25th Euromicro Conference on Digital System Design (DSD)*, pp. 528–533, IEEE, 2022.

Apéndice A

Instalación de Vivado ML

A.1. Requisitos previos

Antes de instalar Vivado ML, asegúrate de cumplir con los siguientes requisitos[1]:

Código A.1: Liberias necesarias antes de instalar vivado en Ubuntu.

```
sudo apt-get install libtinfo5
sudo apt install libncurses5
```

A.2. Descarga de Vivado

1. Visita el sitio oficial de Xilinx en https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2023-2.html.

2. Descargar y ejecutar el instalador "MD Unified Installer for FPGAs & Adaptive SoCs 2023.2: Linux Self Extracting Web Installer".

3. Navega hasta la sección de descargas y selecciona la versión de Vivado que deseas instalar.

4. Descarga el archivo instalador correspondiente a tu sistema operativo.

A.3. Instalación

Sigue estos pasos para instalar Vivado:

1. Ejecuta el instalador descargado con permisos de administrador.

[1] **Nota**: Al instalar Vivado ML se queda bloquea el progreso y es necesario volver a instalar si no se cuenta con dichas bibliotecas.

2. Selecciona la ruta de instalación y los componentes adicionales que deseas instalar.
3. Acepta los términos de la licencia y haz clic en Ïnstalar".
4. Espera a que el proceso de instalación se complete. Esto puede tomar varios minutos.

A.4. Activación de la licencia

Para utilizar Vivado, es necesario activar la licencia:

1. Inicia Vivado y abre el *License Manager*.
2. Selecciona la opción de .ºbtener licenciaz carga el archivo .lic.
3. Una vez activada la licencia, reinicia Vivado si es necesario.

Apéndice B

Descripción de la tarjeta Zybo z7 20

La Zybo Z7 es completamente compatible con la herramienta de diseño Vivado® Design Suite de alto rendimiento de Xilinx. Este conjunto de herramientas combina el diseño lógico para FPGA con el desarrollo de software integrado en procesadores ARM, ofreciendo un flujo de diseño fácil de usar e intuitivo.

Procesador Zynq

- Procesador Cortex-A9 de doble núcleo a 667 MHz.
- Controlador de memoria DDR3L con 8 canales DMA y 4 puertos esclavos AXI3 de alto rendimiento.
- Controladores periféricos de alta velocidad: Ethernet 1G, USB 2.0,SDIO.
- Controladores periféricos de baja velocidad: SPI, UART, CAN, I2C.
- Programable mediante JTAG, memoria flash Quad-SPI y tarjeta microSD.
- Lógica programable equivalente a un FPGA Artix-7.

Memoria

- 1 GB DDR3L con bus de 32 bits a 1066 MHz.
- 16 MB de memoria Quad-SPI Flash con número aleatorio de 128 bits programado de fábrica e identificador único global EUI-48/64™ compatible de 48 bits.

Alimentación

- Alimentación mediante USB o cualquier fuente de energía externa de 5V.

USB y Ethernet

- PHY Ethernet Gigabit.
- Circuitería de programación USB-JTAG.
- Puente USB-UART.
- PHY USB 2.0 OTG con soporte para host y dispositivo.

Audio y Video

- Conector de cámara Pcam con soporte para MIPI CSI-2.
- Puerto de entrada HDMI (sink) con CEC.
- Puerto de salida HDMI (source) con CEC.
- Códec de audio con soporte para auriculares estéreo, entrada de línea estéreo y conectores para micrófono.

Interruptores, Botones y LEDs

- 6 botones pulsadores (2 conectados al procesador).
- 4 interruptores deslizantes.
- 5 LEDs (1 conectado al procesador).
- 2 LEDs RGB .

Conectores de Expansión

- 6 puertos Pmod:
  - 8 E/S totales para el procesador.
  - 40 E/S totales para el FPGA.
  - 4 pares diferenciales analógicos con rango de 0-1.0V para XADC.

Tabla B.1: Recuros de la Zybo Z7-20

Característica	Zybo Z7-20
Parte Zynq	XC7Z020-1CLG400C
ADC integrado (1 MSPS)	Sí
LUTs	53,200
FF	106,400
Memoria RAM en bloques	630 KB
Bloques de gestión de reloj	4
Puertos Pmod totales	6
Conector de ventilador	Sí
Disipador de calor para Zynq	Sí

La Zybo Z7-20 es ideal para aplicaciones que requieren un potente procesador ARM combinado con una lógica FPGA altamente configurable (ver tabla B.1), lo que te permite ejecutar algoritmos complejos de manera eficiente.

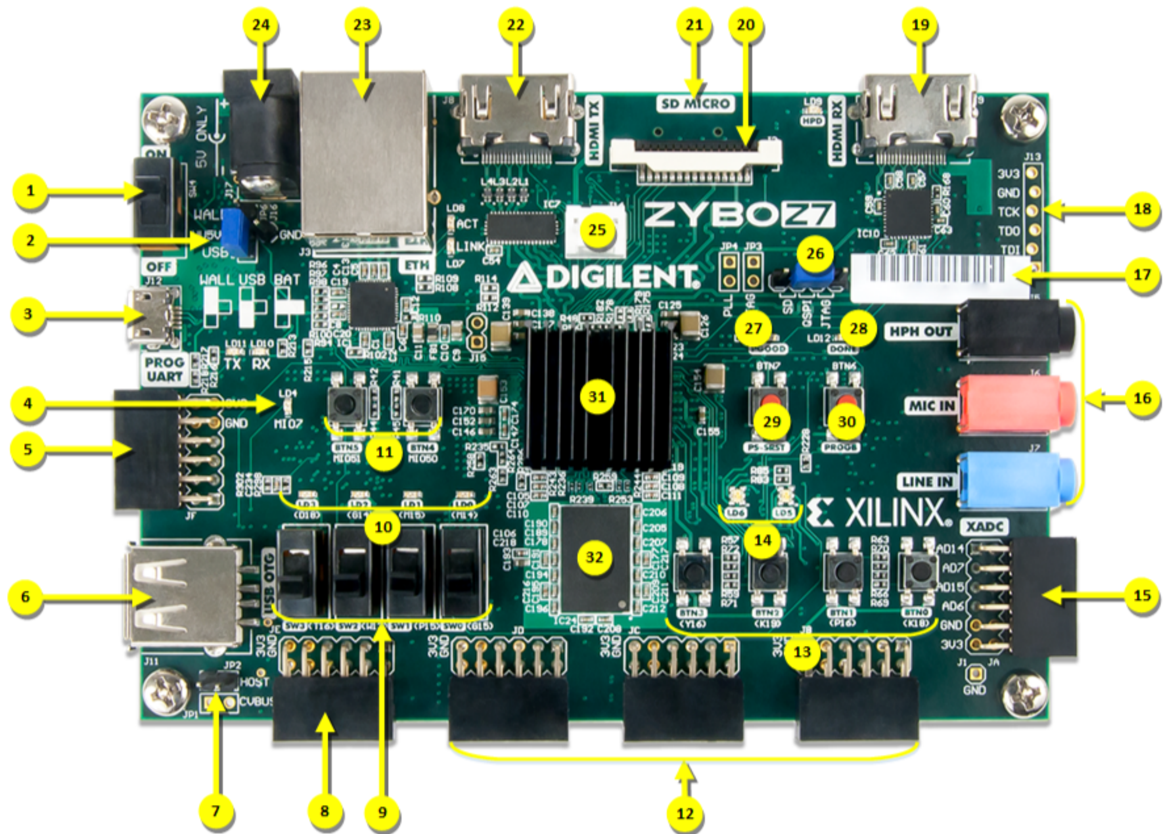


Figura B.1: Zybo z7 20, tomada de [1].

Tabla B.2: Descripción de la tarjeta Zybo Z7-20

Núm	Descripción	Núm	Descripción
1	Interruptor de encendido	**17**	Etiqueta de dirección MAC única
2	Jumper de selección de energía	**18**	Puerto JTAG externo
3	Puerto USB JTAG/UART	**19**	Puerto de entrada HDMI
4	LED de usuario MIO	**20**	Puerto Pcam MIPI CSI-2
5	Puerto Pmod MIO	**21**	Conector microSD (lado opuesto)
6	Puerto USB 2.0 Host/OTG	**22**	Puerto de salida HDMI
7	Jumper de habilitación de energía USB Host	**23**	Puerto Ethernet
8	Puerto Pmod estándar	**24**	Conector de alimentación externa
9	Interruptores de usuario	**25**	Conector para ventilador (5V, tres cables)
10	LEDs de usuario	**26**	Jumper de selección de modo de programación
11	Botones de usuario MIO	**27**	LED de energía correcta
12	Puertos Pmod de alta velocidad	**28**	LED de programación de FPGA
13	Botones de usuario	**29**	Botón de reinicio del procesador
14	LEDs RGB de usuario	**30**	Botón para borrar la configuración del FPGA
15	Puerto Pmod XADC	**31**	Zynq-7000
16	Puertos del códec de audio	**32**	Memoria DDR3L