UNIVERSIDAD DE SEVILLA

ESCUELA POLITECNICA SUPERIOR

TRABAJO DE FIN DE GRADO EN INGENIERIA ELECTRONICA INDUSTRIAL

Diseño Hardware de una familia de cifradores lightweight *ASCON*

Trabajo Fin de Grado presentado por David González Villalba, siendo el tutor de este el profesor Carlos Jesús Jiménez Fernández.

En Sevilla a 2 de enero de 2024

GRADO EN INGENIERIA ELECTRONICA INDUSTRIAL
ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO
CURSO ACADÉMICO [2023-2024]

TÍTULO:
Diseño Hardware de una familia de cifradores lightweight *ASCON*

AUTOR:
DAVID GONZÁLEZ VILLALBA

TUTOR:
D. CARLOS J. JIMENEZ FERNANDEZ

DEPARTAMENTO:
TECNOLOGIA ELECTRONICA

ÁREA DE CONOCIMIENTO:
TECNOLOGIA ELECTRONICA

RESUMEN

En 2018 el NIST inició un proceso para solicitar, evaluar y normalizar esquemas que proporcionen cifrado autenticado con datos asociados (AEAD) y, opcionalmente funcionalidades hash, para entornos con recursos restringidos (lightweight). Este proceso finalizó en febrero de 2023 eligiendo a ASCON como el ganador.

Este Trabajo de Fin de Grado se centra en la Criptografía lightweight, analizando, implementando en hardware y evaluando *ASCON*. *ASCON* ofrece un cifrado y descifrado eficiente junto con un mecanismo de autenticación, e incorpora también la funcionalidad de generar Hash. El objetivo principal consistió en trasladar el algoritmo matemático de *ASCON* a una descripción hardware en VHDL, diseñando un sistema funcional y eficiente. La creación de una máquina de estados en VHDL fue fundamental para asegurar el cifrado, descifrado y autenticación correcta. Se desarrolló una interfaz intuitiva que facilita la interacción con el sistema de cifrado y permite realizar pruebas exhaustivas en diversos escenarios. Finalmente, se evaluó el rendimiento del cifrador en términos de consumo de recursos y uso del espacio en la FPGA, identificando oportunidades de optimización.

PALABRAS CLAVE:

Criptografía lightweight; Algoritmo *ASCON*; VHDL; Autenticación; FPGA, NIST, CAESAR.

ABTRACT

In 2018, the NIST initiated a process to solicit, evaluate, and standardize schemes that provide Authenticated Encryption with Associated Data (AEAD) and, optionally, hash functionalities for environments with restricted resources (lightweight). This process concluded in February 2023, selecting *ASCON* as the winner.

This Final Year Project focuses on lightweight cryptography, analyzing, implementing in hardware, and evaluating *ASCON*. *ASCON* offers efficient encryption and decryption along with an authentication mechanism, and also incorporates the functionality to generate a hash. The main objective was to translate the mathematical algorithm of *ASCON* into a hardware description in VHDL, designing a functional and efficient system. The creation of a state machine in VHDL was crucial to ensure correct encryption, decryption, and authentication. An intuitive interface was developed to facilitate interaction with the encryption system and allow for comprehensive testing in various scenarios. Finally, the performance of the encryptor was evaluated in terms of resource consumption and space usage on the FPGA, identifying opportunities for optimization.

KEYWORDS:

Lightweight cryptography; *ASCON* algorithm; VHDL; Authentication; FPGA; NIST; CAESAR.

ÍNDICE

CAPITULO 1: INTRODUCCION ... 3

CAPITULO 2: Criptografía lightweight necesidad e implementaciones ... 11
- 2.1 Necesidad y características de la criptografía lightweight. ... 11
  - 2.1.1 Competencia Caesar ... 12
  - 2.1.2 Proyecto de criptografía lightweight del NIST ... 13
  - 2.1.3 Métricas ... 21
  - 2.1.4 Funciones de la criptografía lightweight ... 22
  - 2.1.5 Cifradores AEAD ... 23
- 2.2 Tecnologías de implementación ... 24
  - 2.2.1 FPGA: Características Principales ... 25
  - 2.2.2 Metodología de Diseño ... 26
  - 2.2.3 Placas de desarrollo ... 27
- 2.3 Introducción a la familia de cifradores *ASCON* ... 28
  - 2.3.1 Familia de Cifradores *ASCON*: Diseños de Encriptación Autenticada ... 29
  - 2.3.2 Fases de *ASCON* ... 33
  - 2.3.3 Permutaciones *ASCON* ... 37

CAPITULO 3: Implementación hardware del algoritmo ASCON ... 41
- 3.1 Interfaz ... 41
  - 3.1.1 Componentes Principales del Diseño ... 44
  - 3.1.2 Operación del Cifrador ... 44
  - 3.1.3 Carga de la Clave ... 45
  - 3.1.4 Carga del *nonce* ... 45
  - 3.1.5 Procesamiento del Dato Asociado ... 46
  - 3.1.6 Procesamiento del Texto Plano/Cifrado ... 47
  - 3.1.7 Proceso de Finalización ... 47
  - 3.1.8 Procesamiento de Bloques Parciales ... 47
- 3.2 Diseño Hardware *ASCON* ... 48
  - 3.2.1 Flujo de datos ... 53
  - 3.2.2 Funcionamiento detallado de cada estado del cifrador ... 60
- 3.3 Comparativa del algoritmo *ASCON-Hash* con *ASCON* ... 69
  - 3.3.1 Funcionamiento detallado de cada estado del cifrador *ASCON-Hash* ... 69
  - 3.3.2 Estados y Transiciones en el Algoritmo *ASCON-Hash* ... 71

CAPITULO 4: Test Bench y análisis de las implementaciones. 73
4.1 Test Bench .... **¡Error! Marcador no definido.**
4.2 Analisis de las implementaciones 81
4.2.1 Análisis de "Slice Registers" y "Slice *LUTs*" 81

CONCLUSIÓN... **¡Error! Marcador no definido.**

BIBLIOGRAFÍA.. 90

ANEXOS... **¡Error! Marcador no definido.**

Figura 1. Nexys4 DDR 28
Figura 2. Proceso de (a) cifrado, (b) descifrado. Fuente: https://ascon.iaik.tugraz.at .. 34
Figura 3. Adición de constantes de ronda. Fuente: https://ascon.iaik.tugraz.at. 38
Figura 4. Numero de rondas que se le asigna a una constante. Fuente: https://ascon.iaik.tugraz.at. 38
Figura 5. S-BOX. Fuente: https://ascon.iaik.tugraz.at. 39
Figura 6. S-BOX. Fuente: https://ascon.iaik.tugraz.at 39
Figura 7. Mezcla de 5 bits en cada posición de cada palabra de 64 bits. Fuente: https://ascon.iaik.tugraz.at 40
Figura 8. S-box. Fuente: https://ascon.iaik.tugraz.at. 40
Figura 9. Interfaz para ASCON 42
Figura 10. Diagrama de flujo ASCON 49
Figura 11. Diagrama de flujo Hash 72

Tabla 1. Finalistas del proyecto de criptografía lightweight del NIST. 16

Tabla 2. Variantes de los Finalistas. 20

Tabla 3. Finalistas Hash. 21

Tabla 4. Notación utilizada para la interfaz, modo y permutación de ASCON. 30

Tabla 5. Parámetros para esquemas recomendados de encriptación autenticada...... 31

Tabla 6. Parámetros para algoritmos de Hashing recomendados. 31

Tabla 7. Especificaciones Diseño ASCON. 43

Tabla 8. Señales para inicialización. 45

Tabla 9. Próximo bloque a procesar según dEop y dEob al finalizar el nonce............ 46

Tabla 10. Próximo bloque a procesar según dEop y dEob al finalizar el dato asociado. 47

Tabla 11. Próximo bloque a procesar según dEop y dEob al finalizar el texto plano/cifrado 47

Tabla 12. Valores de Partial y Partil_Bits en función del tamaño del último bloque..... 48

Tabla 13. Definición de señales principales del cifrador.. 52

Tabla 14. Ciclos de reloj en cada estado con sus diferentes versiones. 81

Tabla 15. Recursos del sistema con sus diferentes versiones. 82

Tabla 16. Comparación de recursos con original. 83

CAPITULO 1: INTRODUCCION

En plena era digital, la protección de la información ha emergido como una necesidad ineludible. Cada bit de información transmitido, ya sea personal, financiero o confidencial, merece una protección robusta frente a ojos externos. En este contexto, a lo largo de los años, estándares como DES y AES han servido como pilares en el cifrado de datos; sin embargo, la evolución constante de la tecnología demanda soluciones que no solo cifren, sino que también autentiquen la información de manera eficiente, especialmente en dispositivos con recursos limitados.

El National Institute of Standards and Technology (NIST), una agencia del Departamento de Comercio de los Estados Unidos, ha sido un líder mundial en la creación de estándares criptográficos robustos. Fundado en 1901 como el National Bureau of Standards, el NIST tiene la misión de promover la innovación y la competitividad industrial de EE. UU. a través del avance de la ciencia de la medición, los estándares y la tecnología. Este compromiso con la seguridad económica y la mejora de la calidad de vida se refleja en su continuo esfuerzo por desarrollar y actualizar estándares criptográficos que protejan la información en un mundo digital en constante cambio.

En respuesta a la creciente necesidad de criptografía optimizada para entornos con recursos limitados, el NIST lanzó una competición para desarrollar soluciones de criptografía lightweight. Esta iniciativa, similar a las competiciones anteriores de AES y SHA-3, buscaba crear un estándar criptográfico adecuado para dispositivos como etiquetas RFID, redes de sensores y nodos IoT. El objetivo no era reemplazar a AES, sino complementarlo con un estándar más adecuado para estos nuevos dispositivos.

La competición de criptografía lightweight de NIST se desarrolló en varias fases:

- Fase Inicial (julio 2015 – agosto 2018): NIST organizó talleres para entender las necesidades de la industria y recoger opiniones sobre los requisitos de seguridad para la criptografía lightweight. Publicaciones como el Informe NISTIR 8114 [3] proporcionaron pautas fundamentales para el proceso de estandarización.
- Convocatoria de Presentaciones (agosto 2018 – abril 2019): NIST publicó los requisitos de presentación y criterios de evaluación, enfocándose en la seguridad, rendimiento y optimización para mensajes cortos.
- Primera Ronda (abril 2019 – agosto 2019): Se evaluaron 56 candidatos basándose en su seguridad y rendimiento, seleccionando 32 para la siguiente ronda.

- Segunda Ronda (agosto 2019 – marzo 2021): Los 32 candidatos participaron en talleres adicionales para profundizar en el análisis de sus propuestas, reduciéndose el número a 10 finalistas.
- Ronda Final (marzo 2021 – febrero 2023): La evaluación de los diez finalistas duró aproximadamente dos años, considerando diversos criterios de seguridad y rendimiento.

Finalmente, en febrero de 2023, NIST anunció a *ASCON* como la familia ganadora de esta competición debido a su alto margen de seguridad, diseño maduro y rendimiento superior tanto en hardware como en software. *ASCON* no solo destaca por su capacidad de cifrado, sino también por su funcionalidad como algoritmo de hash. Esta versatilidad se refleja en sus tres opciones de implementación, cada una optimizada para diferentes escenarios y ofreciendo un equilibrio óptimo entre seguridad y eficiencia.

Este proyecto se centra en desentrañar las complejidades y capacidades del cifrador *ASCON*, realizando una descripción en VHDL que pueda ser implementada en hardware. En las pruebas realizadas en este trabajo se ha implementado en una tecnología FPGA, específicamente en un dispositivo Artix-7 que se integra en la Nexys 4 DDR. Esta elección refleja la relevancia de este estudio no sólo en el ámbito académico sino también en los ámbitos industrial y comercial y refleja la búsqueda constante de innovación en el campo de la criptografía. Además, el proyecto no se limita a la implementación del cifrador, sino que también incorpora la función de hash, demostrando así la versatilidad de *ASCON*.

Para destacar esta versatilidad, se han desarrollado tres versiones diferentes del cifrador *ASCON*. Estas configuraciones afectan directamente la complejidad del circuito y el rendimiento del sistema. En la configuración v1, las permutaciones de 12 y 6 rondas se realizan en 12 y 6 ciclos de reloj, respectivamente. La configuración v3 reduce estos números a 6 y 3 ciclos, mientras que la configuración v5 los disminuye aún más a 3 y 1 ciclos de reloj.

Los objetivos de este Trabajo de Fin de Grado incluyen:

- Analizar detalladamente la estructura y el funcionamiento del cifrador *ASCON*, poniendo especial énfasis en sus características como solución de criptografía lightweight.
- Implementar el cifrador *ASCON* en VHDL, aprovechando las capacidades de la tecnología FPGA para evaluar su rendimiento y eficiencia en hardware.
- Contrastar la implementación de *ASCON* con los requisitos y desafíos específicos de los dispositivos de destino con recursos limitados, resaltando su adaptabilidad y eficacia.

- Evaluar las tres versiones del cifrador mediante test bench que simulan su funcionamiento, comparando el rendimiento y eficiencia de cada configuración.

El documento está organizado para entender las bases y tener una visión general desde prácticamente cero de la criptografía lightweight y el cifrador *ASCON*.

El capítulo 2 ofrece un recorrido por el panorama de la Criptografía lightweight, subrayando la importancia de *ASCON* dentro de los esfuerzos del NIST por encontrar algoritmos que cumplan con altos estándares de seguridad y eficiencia. Se discuten los dispositivos objetivo de la Criptografía lightweight, las métricas de desempeño y las primitivas criptográficas relevantes, estableciendo el contexto en el que *ASCON* se desarrolla y destaca como la solución escogida.

El capítulo 3 constituye el corazón del trabajo, donde se examina minuciosamente el cifrador *ASCON*. Se abordan los detalles técnicos del algoritmo, su estructura, modo de operación y los principios que guían su diseño. Posteriormente, se describe el proceso de implementación del algoritmo en VHDL, detallando cada paso realizado para adaptar *ASCON* a un diseño hardware. Este análisis exhaustivo no solo ilustra la complejidad del cifrado autenticado sino también las consideraciones importantes en el diseño hardware de algoritmos criptográficos.

El capítulo 4 se enfoca en la fase de prueba y evaluación de la implementación de *ASCON*. Se detalla el diseño y ejecución del test bench, una herramienta esencial para validar la correcta funcionalidad del cifrador implementado. A través de análisis rigurosos, se evalúa la eficacia, rendimiento y eficiencia de *ASCON* en un entorno controlado, proporcionando datos valiosos sobre su comportamiento en situaciones reales y su viabilidad como solución criptográfica en dispositivos de recursos limitados.

Este trabajo no solo pretende aportar a la comunidad académica un estudio exhaustivo sobre *ASCON* y su implementación en hardware, sino también brindar un recurso valioso para futuras investigaciones y desarrollos en la Criptografía lightweight.

CAPITULO 2: Criptografía lightweight necesidad e implementaciones

En el ámbito de la seguridad digital, la criptografía juega un papel fundamental, adaptándose continuamente a las exigencias de un mundo tecnológicamente avanzado. Los sistemas criptográficos se dividen principalmente en dos tipos: los basados en claves públicas y privadas, y los sistemas de clave secreta. La criptografía asimétrica, que utiliza un par de claves pública y privada, facilita operaciones seguras como la firma digital y el intercambio de claves, esenciales para la protección de la comunicación en la web. Paralelamente, la criptografía simétrica, que emplea una clave secreta compartida para el cifrado y descifrado de mensajes, destaca por su rapidez y eficacia, siendo ideal para el manejo de grandes volúmenes de datos.

Dentro de la criptografía simétrica, el Estándar de Cifrado Avanzado (AES) representa un hito por su robustez y adaptabilidad, permitiendo el uso de claves de diversos tamaños y garantizando una protección efectiva en múltiples plataformas. No obstante, el surgimiento del Internet de las Cosas (IoT) y la proliferación de dispositivos de baja capacidad han impulsado la necesidad de sistemas criptográficos ligeros. Estos sistemas están diseñados para operar bajo severas limitaciones de recursos, proporcionando seguridad sin comprometer el rendimiento de dispositivos como etiquetas RFID, nodos de sensores y microcontroladores, lo que subraya la importancia de desarrollar soluciones que equilibren eficacia y eficiencia en entornos de recursos restringidos.

2.1 Necesidad y características de la criptografía lightweight

La criptografía lightweight se destaca por su adaptación a las necesidades de dispositivos con recursos limitados, característicos del Internet de las Cosas (IoT). Estos dispositivos, que van desde sensores ambientales hasta dispositivos de comunicación portátiles, requieren soluciones de seguridad que consuman mínimos recursos mientras mantienen altos estándares de protección de datos.

Una característica principal es su optimización para un bajo consumo energético y uso reducido de memoria. Esto es crucial para dispositivos que funcionan con baterías pequeñas o que tienen limitaciones de hardware. Además, estos sistemas criptográficos son diseñados para ser altamente eficientes, permitiendo un procesamiento rápido de datos, lo que es esencial para aplicaciones en tiempo real o dispositivos que necesitan responder rápidamente a inputs sensoriales.

Otra característica importante es la simplicidad de los algoritmos utilizados. A diferencia de la criptografía tradicional, que puede requerir complejas operaciones matemáticas y grandes cantidades de datos para garantizar la seguridad, se enfoca en algoritmos más sencillos que reducen la carga computacional sin comprometer la seguridad. Esta simplicidad también contribuye a una mayor facilidad de implementación en plataformas diversas, desde microcontroladores básicos hasta sistemas integrados más complejos.

Otra característica esencial es la escalabilidad. A medida que los dispositivos IoT evolucionan y su número crece, los sistemas criptográficos deben ser capaces de adaptarse a diferentes niveles de seguridad, dependiendo de las necesidades específicas de cada dispositivo o red.

La justificación para la adopción de la criptografía lightweight no solo reside en su adaptabilidad y eficiencia, sino también en la necesidad imperiosa de proteger información sensible en dispositivos altamente distribuidos y a menudo vulnerables. La continua evolución y mejora de estos sistemas es crucial para enfrentar las crecientes amenazas de seguridad en un mundo cada vez más conectado.

2.1.1 Competencia Caesar

La información presentada en esta sección se basa principalmente en datos obtenidos de la fuente [1] Competitions.cr.yp.to. (nd). CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. de https://competitions.cr.yp.to/caesar-call.html

La Competición de Cifrado Autenticado: Seguridad, Aplicabilidad y Robustez (CAESAR) representa una iniciativa internacional clave diseñada para impulsar el desarrollo de esquemas de cifrado autenticados que sean más seguros y eficientes. Lanzada en el taller Early Symmetric Crypto en 2013 y concluyendo con la presentación de su portafolio final en 2019, CAESAR ha tenido un impacto significativo en la evolución de la criptografía. Esta competencia se centró en identificar nuevos algoritmos que no solo protegieran la confidencialidad de los datos cifrando el contenido, sino que también verificaran su integridad y autenticidad, lo cual es vital en entornos donde la manipulación de datos puede tener consecuencias graves.

El portafolio final de CAESAR se categorizó en tres casos de uso principales, optimizados para diferentes escenarios: aplicaciones ligeras diseñadas para entornos con recursos limitados, aplicaciones de alto rendimiento para el procesamiento de datos a gran escala, y estrategias de defensa en profundidad para seguridad en múltiples niveles.

Los ganadores de esta competencia fueron:

- Ascón y BELLOTA, que se destacaron por su eficiencia en el uso de recursos, lo que los hace especialmente adecuados para dispositivos con restricciones de energía y capacidad de procesamiento.
- En la categoría de alto rendimiento, los algoritmos que se llevaron el reconocimiento fueron AEGIS-128 y OCB, ambos elogiados por su capacidad para procesar rápidamente grandes volúmenes de datos, ideal para aplicaciones que demandan una alta velocidad de procesamiento.
- Para defensa en profundidad, los algoritmos seleccionados fueron Deoxis-II y COLM, que ofrecen soluciones robustas y seguras para entornos donde la seguridad de múltiples niveles es crítica.

El comité de CAESAR, compuesto por criptógrafos renombrados como Daniel J. Bernstein y Joan Daemen, no solo evaluó las propuestas, sino que también orientó la competencia hacia soluciones que superaran los estándares existentes, como AES-GCM. Los algoritmos propuestos fueron sometidos a rigurosos criterios de rendimiento, facilidad de implementación y resistencia a ataques, buscando soluciones aplicables a una amplia gama de necesidades, desde sistemas altamente restringidos hasta aquellos que demandan una seguridad máxima.

2.1.2 Proyecto de criptografía lightweight del NIST

La información presentada en esta sección se basa principalmente en datos obtenidos de la fuente [2] National Institute of Standards and Technology. (2023). Final steps of the NIST lightweight cryptography standardization process.

El National Institute of Standards and Technology (NIST), parte del Departamento de Comercio de los Estados Unidos, fue fundado en 1901 originalmente como el National Bureau of Standards. La misión primordial de NIST es promover la innovación y la competitividad industrial de EE. UU. a través del avance de la ciencia de la medición, los estándares y la tecnología. Esta misión subraya el compromiso del instituto con la seguridad económica y la mejora de la calidad de vida a través de innovaciones que sostienen tanto el sector privado como el público.

NIST emplea a más de 3,400 trabajadores federales y trabaja con aproximadamente 3,500 asociados, incluyendo investigadores y contratistas externos. Ha sido reconocido con varios Premios Nobel, destacando su excelencia en investigación y desarrollo en diversas disciplinas científicas y tecnológicas. Un componente crucial de NIST es la Computer Security Division (CSD), que se enfoca en desarrollar estándares criptográficos robustos.

La CSD ha liderado competiciones internacionales para el desarrollo de nuevos estándares criptográficos, tales como AES (Advanced Encryption Standard) y

SHA-3. También ha jugado un papel crucial en la adopción de estándares existentes como RSA y HMAC y ha hecho propuestas sobre modos de la operación de cifrado en bloque. Estas actividades están documentadas en varias publicaciones de NIST, incluyendo los Federal Information Processing Standards (FIPS) y las NIST Special Publications (SPs), que ofrecen directrices y recomendaciones técnicas para su uso.

Los principios que guían el trabajo de NIST incluyen la transparencia, apertura, integridad, mérito técnico, y la mejora continua, asegurando que sus estándares y recomendaciones sean globalmente aceptados y utilizables.

El Advanced Encryption Standard (AES), publicado en 2001 bajo el Federal Information Processing Standard (FIPS) 197, es uno de los logros más significativos del National Institute of Standards and Technology (NIST). El AES fue adoptado tras una competición pública que buscaba reemplazar el antiguo estándar DES debido a sus limitaciones de seguridad y rendimiento. Tras una revisión exhaustiva después de veinte años, AES fue actualizado en mayo de 2023 para ajustarse a las necesidades modernas de seguridad, manteniendo su posición como el cifrador más utilizado globalmente, influyendo considerablemente en la economía y la seguridad digital.

El AES se implementa junto con varios modos de operación especificados en el documento SP 800-38 de NIST [11], incluyendo CBC, OFB, y GCM, entre otros. Estos modos permiten el uso de AES en diferentes contextos de seguridad, ofreciendo flexibilidad en la protección de la integridad y la confidencialidad de los datos.

A pesar del éxito de AES, la evolución continua de las tecnologías y los modelos de amenazas ha llevado a NIST a explorar nuevos primitivos criptográficos. Los nuevos desarrollos también buscan ofrecer algunas características como resistencia al mal uso de *Nonce*s, resistencia inherente a canales laterales, seguridad post-cuántica.

El rápido crecimiento de dispositivos como etiquetas RFID, redes sensores o nodos IoT ha creado una necesidad de estándares criptográficos optimizados para entornos con recursos limitados. En respuesta a esta necesidad, el NIST lanzó un proyecto similar a las competiciones AES, SHA-3 y PQC, pero centrado en la criptografía lightweight para generar un nuevo estándar. El objetivo de este nuevo estándar no es sustituir al AES, sino crear uno más adaptado a estos nuevos dispositivos.

La temporización del proyecto de criptografía lightweight del NIST es la siguiente:

Fase Inicial (julio 2015 – agosto 2018)

Durante la fase inicial, el NIST se centró en recopilar información sobre las necesidades y requisitos específicos para la criptografía lightweight. Esta fase incluyó la organización de dos talleres principales:

1. Primer Taller de Criptografía lightweight (20 – 21 de julio, 2015): Tuvo como objetivo entender las necesidades de la industria y recoger opiniones de expertos sobre las características y requisitos de seguridad deseados para la Criptografía lightweight.
2. Segundo Taller de Criptografía lightweight (17 – 18 de octubre, 2016): Continuó las discusiones sobre aplicaciones específicas y refinó los requisitos basados en la retroalimentación recibida.

Publicaciones como el Informe NISTIR 8114 sobre Criptografía lightweight [3] proporcionaron pautas fundamentales y delinearon los perfiles para el proceso de estandarización de la criptografía lightweight.

Convocatoria de Presentaciones (agosto 2018 – abril 2019)

En agosto de 2018, el NIST publicó los ‘Requisitos de Presentación y Criterios de Evaluación para el Proceso de Estandarización de la Criptografía lightweight’. El plazo para las presentaciones se fijó para febrero de 2019. Los requisitos incluyeron:

- Requisitos de seguridad: Un nivel de seguridad mínimo de 112 bits para mensajes de hasta 250 bytes, respetando el *Nonce*. El tamaño de clave debía ser al menos 128 bits.
- Requisitos de diseño: Rendimiento superior a los estándares de NIST (AES-GCM, SHA-2), optimizados para mensajes cortos.
- Requisitos de implementación: Implementación de referencia y optimizada compatible con la API (Interfaz de Programación de Aplicaciones) especificada en el documento "Hardware API for Lightweight Cryptography" [12].

Primera Ronda (abril 2019 – agosto 2019)

Durante aproximadamente cuatro meses, 56 candidatos fueron evaluados basándose en su seguridad, incluyendo aspectos como ataques de distinción, falsificaciones prácticas de etiquetas, problemas de separación de dominios, y nuevos diseños sin análisis de terceros. El **Informe NIST IR 8268** detalla las razones por las cuales sólo 32 candidatos fueron seleccionados para avanzar a la segunda ronda.

Segunda Ronda (agosto 2019 – marzo 2021)

Durante unos 20 meses, los 32 candidatos de la segunda ronda participaron en talleres adicionales para profundizar en el análisis y la mejora de sus propuestas:

- Tercer Taller de Criptografía lightweight (4 – 6 de noviembre, 2019).
- Cuarto Taller de Criptografía lightweight (19 – 21 de octubre, 2020).

El Informe NIST IR 8369 explica cómo los 10 finalistas fueron seleccionados para avanzar a la ronda final.

Ronda Final (marzo 2021 – febrero 2023)

La evaluación de los diez finalistas duró aproximadamente dos años. Los finalistas fueron los siguientes:

ASCON	Photon-Beetle
Sparkle	Grain-128AEAD
Isap	Xoodyak
Elepahnt	ROMULUS
Tinyjambu	Gift-Cob

Tabla 1. Finalistas del proyecto de criptografía lightweight del NIST.

Evaluar de manera justa a los finalistas fue una tarea completa, que involucró la asignación de diferentes pesos para distintos criterios, variadas afirmaciones de seguridad, funcionalidades, y ataques de diferentes complejidades. Los recursos limitados para el análisis de seguridad y la evaluación comparativa también presentaron desafíos.

La decisión final se basó en los análisis y los resultados de evaluación comparativa de todos los candidatos, que están disponibles públicamente, asegurando que los estándares propuestos no solo fueran técnicamente sólidos, sino también viables y aplicables en el contexto deseado.

A continuación, se resumen las principales características de los finalistas mostrados en la tabla 1:

ASCON

Tiene un esquema de cifrado y *Hashing* basado en permutaciones (con un registro de estado de 320-bit), con AEAD usando el modo MonkeyDuplex que incluye inicialización y finalización con clave, y *Hashing* de tipo esponja. No se realizaron ajustes en el diseño, pero se añadió una nueva variante en la ronda

final al incluirlo en el portafolio final de CAESAR para la encriptación autenticada lightweight.

ELEPHANT

Esquema AEAD basado en permutaciones (esponja y Keccak [200]). Utiliza un modo paralelo único de cifrar y luego MAC basado en *Nonce*. El algoritmo fue ligeramente modificado para garantizar autenticidad bajo reutilización de *Nonce*.

GIFT-COCB:

Esquema AEAD basado en el cifrado de bloque GIFT-128, utilizando el algoritmo de realimentación combinada (COFB). No se realizaron ajustes en el diseño.

GRAIN-128-AEAD

Esquema AEAD basado en un registro de desplazamiento con realimentación. Se realizó un ajuste en la parte de inicialización respecto a las versiones anteriores incluidas en el portafolio eSTREAM y en la norma ISO/IEC 29167-13:2005.

ISAP

Esquema AEAD basado en permutaciones (*ASCON* y Keccak) que puede combinarse con *ASCON Hash*. Utiliza un modo de cifrar y luego MAC basado en *Nonce*. No se realizaron ajustes en la variante principal, aunque se actualizó.

PHOTON BEETLE

Familia de esquemas de cifrado y *Hashing* basados en permutaciones (permutación Photon de 256-bit), operando en un modo similar a Esponja con realimentación combinada. No se realizaron ajustes en el diseño.

ROMULUS

Familia de esquemas AEAD y de *Hashing* basados en cifrados de bloque ajustables (Skinny). Incluye variantes que respetan y que abusan del *Nonce*, con ajustes en el diseño que reducen el número de rondas de 56 a 40, eliminación de variantes no primarias y adición de nuevas variantes.

SPARKLE

Familia de esquemas AEAD (SCHWAEMM) y de *Hashing* (ESCH) basados en permutaciones, con un diseño basado en ARX. Utiliza una construcción Esponja con realimentación combinada. Se realizó un ajuste para cambiar la variante primaria.

TINYJAMBU

Esquema AEAD basado en permutaciones con clave, utilizando un registro de desplazamiento no lineal de 128-bit. Inspirado en JAMBU (candidato de CAESAR), con un ajuste en el diseño que aumenta el número de rondas para mejorar el margen de seguridad.

XOODYAK

Familia de esquemas de cifrado y *Hashing* basados en permutaciones, utilizando la permutación Xoodoo de 384-bits. Opera en modo Cyclist, con un ajuste en la inicialización para mejorar el rendimiento en mensajes cortos.

Todos los finalistas debían cumplir con requisitos de seguridad que incluían claves de al menos 128 bits y tamaños de mensajes de entrada de hasta 250 bytes (ver Tabla 2 para los finalistas de cifradores y Tabla 3 para los finalistas del hash). Todos los finalistas cumplieron con los requisitos de seguridad y proporcionaron márgenes de seguridad suficientes:

- Ninguna de las reclamaciones de seguridad hechas por los participantes fue invalidada.
- La madurez del diseño fue uno de los factores importantes en la evaluación de la seguridad. Esto incluyó preguntas como: ¿Está el finalista basado en principios de diseño bien establecidos? ¿Recibió el diseño suficiente análisis por terceros? ¿Hay ajustes en el diseño que invaliden los análisis de seguridad anteriores? ¿Existen preocupaciones adicionales como el mal uso de *Nonce*, seguridad relacionada con la clave, seguridad RUP o seguridad post-cuántica?

Selección de *ASCON*. En febrero de 2023, el NIST anunció la familia *ASCON* como la ganadora debido a:

- Un alto margen de seguridad y un gran número de análisis por terceros, dado que fue diseñado en 2014.
- Elección principal para aplicaciones ligeras en el portafolio final de CAESAR (en 2019).
- No necesitaba ajustes en el diseño.
- Ventajas de rendimiento sobre los estándares del NIST (AES-GCM y SHA-2) tanto en hardware como en software.
- Flexibilidad de implementación y diseño.
- Mecanismo de protección a nivel de modo contra fugas y costo adicional bajo para implementaciones protegidas.
- Soporte para funcionalidades adicionales como XOF, MAC dedicado, además de *Hash*.

Variantes a estandarizar:

- Estándarizar *ASCON*-128 o ambos, *ASCON*-128 y *ASCON*-128a.
- No incluir *ASCON*-80pq.
- Estandarizar XOF en lugar de funciones *Hash*.

Tamaños de parámetros de las variantes:

- Variante AEAD: *ASCON*-128 y *ASCON*-128a con claves/*Nonce*s/tags de 128 bits.
- Variante: *ASCON*-80-pq con claves de 160 bits y *Nonce*s/tags de 128 bits.
- *Hash*: *ASCON-Hash* y *ASCON-Hash*a con resúmenes de 256 bits.
- XOF: *ASCON*-XOF y *ASCON*-XOFa con resúmenes de longitud arbitraria.

Posibles actualizaciones:

- Soporte de tags más cortos: de 64 y 96 bits.
- Soporte para cadenas de personalización.
- Codificación *little endian* de las entradas para implementaciones más eficientes.
- Soporte para funcionalidades adicionales como PRF (función pseudoaleatoria), MAC (código de autenticación de mensajes), KDF (función de derivación de clave), DRBG (generador de números pseudoaleatorios basado en la demanda), entre otros.

Finalistas	Variantes	Bloque de construcción	Modo	Key Size	*Nonce* Size	Tag Size
ASCON	*ASCON*-128	*ASCON* Permutación	MonkeyDuplex	128	128	128
	ASCON-128a	*ASCON* Permutación	MonkeyDuplex	128	128	128
	ASCON-80pq	*ASCON* Permutación	MonkeyDuplex	160	128	128
Elephant	Dumbo	Spongent-π [160]	Encrypt-then-MAC	128	96	64
	Jumbo	Spongent-π [176]	Encrypt-then-MAC	128	96	64
	Delirium	KECCAK-ρ [200]	Encrypt-then-MAC	128	96	128
GIFT-COFB	GIFT-COFB	GIFT-128	Combined Feedback	128	128	128

Grain-128AEAD	Grain-128AEAD	Feedback shift register	Encrypt-and-MAC	128	96	64
ISAP	ISAP-A-128a	*ASCON* Permutación	Encrypt-then-MAC	128	128	128
	ISAP-K-128a	KECCAK-ρ [400]	Encrypt-then-MAC	128	128	128
PHOTON-Beetle	PHOTON-Beetle-AEAD[128]	PHOTON256 Permutación	Esponja with Combined Feedback	128	128	128
Romulus	Romulus-N	Skinny-128-384+	Combined Feedback	128	128	128
	Romulus-M	Skinny-128-384+	MAC-then-Encrypt	128	128	128
	Romulus-T	Tweakable Block Cipher	Encrypt-then-MAC	128	128	128
SPARKLE	SCHWAEMM128-128	SPARKLE384	Esponja with Combined Feedback	128	128	128
	SCHWAEMM256-128	SPARKLE384	Esponja with Combined Feedback	256	128	128
	SCHWAEMM256-256	SPARKLE512	Esponja with Combined Feedback	256	256	256
TinyJAMBU	TinyJAMBU-128	Keyed Permutación	Esponja	128	96	64
	TinyJAMBU-192	Keyed Permutación	Esponja	192	96	64
	TinyJAMBU-256	Keyed Permutación	Esponja	256	128	128
Xoodyak	Xoodyak1	Xoodoo Permutación	Esponja-variant Cyclist	128	128	128

Tabla 2. Variantes de los Finalistas.

Finalistas	Variante	Bloque de Construcción	Modo	Tamaño de Digest
ASCON	*ASCON-Hash*	Permutación de *ASCON*	Esponja	256
ASCON	*ASCON-Hash*a	Permutación de *ASCON*	Esponja	256
PHOTON-Beetle	PHOTON-Beetle-*Hash*[32]	Permutación PHOTON256	Esponja	256
Romulus	Romulus-H	Skinny-128-384+	MDPH'	256
SPARKLE	ESCH256	SPARKLE384	Esponja	256
SPARKLE	ESCH384	SPARKLE512	Esponja	384
Xoodyak	Xoodyak	Permutación Xoodoo	Esponja	256

Tabla 3. Finalistas Hash.

2.1.3 Métricas

Una métrica en criptografía es una medida estándar utilizada para evaluar la efectividad y eficiencia de un algoritmo en diversos aspectos técnicos. Las métricas son herramientas esenciales que permiten a los desarrolladores y evaluadores comparar el desempeño de diferentes algoritmos bajo criterios uniformes. Esto es crucial para garantizar que los algoritmos no solo cumplan con los requisitos de seguridad, sino que también sean prácticos y viables para implementar en dispositivos con recursos limitados. Por ejemplo, métricas como el consumo de energía o la latencia influyen directamente en la viabilidad del uso de un algoritmo en dispositivos alimentados por baterías o en aplicaciones que requieren respuestas inmediatas. Así, estas evaluaciones ayudan a determinar los algoritmos más adecuados para cumplir con las normativas y asegurar que funcionen eficientemente en contextos específicos [4].

Métricas de Software

En software, se evalúa la criptografía lightweight a través del uso de memoria, que debe ser mínimo para no sobrecargar dispositivos pequeños. El *throughput* (número de bits procesados por unidad de tiempo) también es crítico, especialmente para procesar información con rapidez en situaciones de seguridad en tiempo real. Además, la flexibilidad de implementación asegura que el algoritmo pueda adaptarse a diferentes plataformas, aumentando su utilidad en una variedad de aplicaciones.

Métricas de Hardware

En hardware, se evalúa la criptografía lightweight a través del consumo de energía, que debe ser mínimo para asegurar que los dispositivos pequeños,

como aquellos alimentados por baterías, puedan operar eficientemente sin necesidad de recargas frecuentes. La latencia, que mide el tiempo que un algoritmo tarda en procesar un bloque de datos, es crucial para aplicaciones que requieren respuestas inmediatas, como la autenticación en tiempo real o la comunicación segura. Además, el área de circuito, que se refiere al espacio físico necesario para implementar el algoritmo en un chip, es vital en dispositivos donde el espacio es limitado, como las tarjetas inteligentes o los dispositivos IoT. Un área de circuito más pequeña puede reducir los costos de fabricación y permitir diseños más compactos. Por último, el costo de implementación, que incluye los costos de fabricación del chip y los materiales necesarios, es una consideración importante, ya que los algoritmos que requieren hardware costoso pueden no ser viables para aplicaciones de bajo costo.

2.1.4 Funciones de la criptografía lightweight

La criptografía lightweight abarca varias funciones clave diseñadas para proporcionar seguridad integral en dispositivos con recursos limitados. Estas funciones incluyen no solo cifradores y autenticación de mensajes, sino también una gama más amplia de algoritmos y técnicas adaptadas a diversos entornos operativos.

Cifradores de bloques y de flujo

Los cifradores de bloque son fundamentales en la criptografía por su estructura y modo de operación. Estos algoritmos procesan datos en bloques de tamaños fijos (típicamente de 128 bits), aplicando diversas transformaciones y permutaciones bajo una clave secreta. Son especialmente valorados por su robustez en la seguridad. En el contexto de criptografía tradicional el AES es el estándar aprobado por el NIST. Pero en criptografía lightweight, se desarrollan versiones más sencillas de estos cifradores, como el *ASCON* o el TinyJAMBU, que mantienen un alto nivel de seguridad, pero con menos demanda de recursos, adecuados para dispositivos como tarjetas inteligentes y pequeños sensores.

Los cifradores de flujo, por otro lado, cifran los datos bit a bit o byte a byte, lo que los hace ideales para entornos donde los datos se generan o reciben en un flujo continuo. Estos algoritmos se adaptan perfectamente a situaciones que requieren una baja latencia y donde los patrones de uso de datos son altamente variables, como en comunicaciones en tiempo real o streaming de audio/video.

Autenticación de Mensajes

Las técnicas como el Código de Autenticación de Mensajes (MAC) son importantes esenciales porque proveen un método para verificar tanto la integridad como la autenticidad de un mensaje. Un MAC funciona al generar un valor corto y fijo basado en el mensaje y una clave secreta conocida solo por el

emisor y el receptor, asegurando que cualquier alteración en el mensaje durante la transmisión sea detectable. Este proceso es importante en comunicaciones seguras, como transacciones financieras o comunicaciones confidenciales en dispositivos IoT [6].

Hash

Las funciones *Hash* ligeras están diseñadas para proporcionar una forma segura y eficiente de verificar la integridad de los datos sin necesidad de un cifrado completo, lo que es ideal para dispositivos con recursos limitados. Estas funciones toman datos de cualquier tamaño y producen un resumen fijo, o "*Hash*", que es único para cada conjunto de datos. Si los datos cambian mínimamente, el *Hash* resultante cambia de manera significativa. Este atributo es crucial para detectar alteraciones en los datos. En aplicaciones prácticas, las funciones *Hash* ligeras se utilizan para asegurar la integridad de datos en áreas como firmware de dispositivos, aplicaciones de blockchain y sistemas de detección de intrusiones, donde el rendimiento y la eficiencia son tan críticos como la seguridad [6].

2.1.5 Cifradores AEAD

Los cifradores Authenticated Encryption with Associated Data (AEAD) integran confidencialidad, integridad de los datos y autenticación en una sola operación. Esta capacidad los hace ideales para ambientes de alta seguridad, a diferencia del AES que es principalmente un cifrador de bloque enfocado en la confidencialidad y para añadirle confidencialidad hay que hacerlo de una forma externa.

Componentes del cifrador AEAD

1. Clave: La clave en AEAD debe ser robusta, generalmente generada por algoritmos de alta entropía para evitar ataques de fuerza bruta. Su longitud y complejidad deben cumplir con estándares específicos, con un mínimo de 128 bits.
2. *Nonce* (Número Utilizado Una Sola Vez): El *nonce* debe ser único para cada operación de cifrador/descifrado bajo una misma clave, evitando así la reutilización que puede llevar a vulnerabilidades como ataques de replay. Aunque no tiene que ser secreto, su generación debe ser aleatoria o secuencial.
3. Datos Asociados (Associated Data, AD): Los AD no se cifran, pero se incluyen en el cálculo del tag para verificar su integridad y autenticidad. Pueden incluir aspectos como dirección de destino u otros indicadores que se deben transmitir sin cifrar.

4. Texto Plano/Cifrado: En el cifrado, el texto plano se transforma utilizando la clave, el nonce en un texto cifrado, asegurando de esta forma que el texto cifrado no pueda ser modificado sin detectarlo.
5. Tag: Este componente es el resultado de una función de *Hash* criptográfico o MAC que asegura la integridad y autenticidad del mensaje. Es verificado al final del proceso de descifrado para confirmar que los datos recibidos son auténticos y no han sido alterados [5].

En el proceso de cifrado se combina el texto plano con la clave criptográfica y el *nonce*. La clave cifra el texto mientras que el *Nonce* asegura que cada cifrado sea único, incluso si el texto plano se repite.

En el proceso de descifrado se usa la misma clave y *Nonce* para transformar el texto cifrado de vuelta a texto plano. La reutilización del *Nonce* en este contexto es segura porque está asociada a la misma clave y mensaje.

Antes de aceptar el texto descifrado como válido, se verifica el tag calculado durante el cifrado. Si el tag del mensaje descifrado coincide con el calculado originalmente, esto confirma que el mensaje no ha sido alterado.

Si el tag no coincide, se genera una alerta indicando posible manipulación o error en la transmisión, protegiendo al usuario contra ataques que buscan alterar el mensaje.

2.2 Tecnologías de implementación

Dentro de la criptografía, la implementación puede realizarse mediante software o hardware, cada uno con características y aplicaciones distintas que los hacen adecuados para diferentes entornos y necesidades.

Implementación en Software:

- Flexibilidad: La criptografía implementada en software es altamente flexible, permitiendo actualizaciones y modificaciones rápidas para adaptarse a nuevos desafíos de seguridad sin necesidad de alterar el hardware físico.
- Costo: Generalmente, es menos costosa en términos de inversión inicial comparada con el hardware, ya que se puede ejecutar en infraestructuras de hardware general existentes.
- Escalabilidad: Se puede escalar fácilmente con el aumento de la capacidad de procesamiento del hardware o con mejoras en los sistemas operativos y plataformas.

- Dependencia del Sistema Operativo: La implementación en software puede estar limitada por las características y la seguridad del sistema operativo en el que se ejecuta, lo que podría introducir vulnerabilidades adicionales.

Implementación en Hardware:

- Eficiencia: Los dispositivos de hardware dedicados a tareas criptográficas, como los módulos de seguridad de hardware (HSMs) o las FPGAs, ofrecen un rendimiento superior para operaciones criptográficas debido a que están optimizados para este fin.
- Seguridad Incrementada: Al estar físicamente segregados del sistema principal, la criptografía en hardware es menos susceptible a ataques de software y vulnerabilidades del sistema operativo. Además, muchos chips de hardware incluyen medidas de seguridad física que protegen contra manipulaciones y ataques físicos.
- Consumo de Energía: Los chips especializados pueden ser más eficientes en el uso de la energía para tareas específicas en comparación con un CPU general, lo cual es crucial en dispositivos móviles o aquellos que operan en entornos donde la energía es una preocupación crítica.
- Costo a Largo Plazo: Aunque la inversión inicial en hardware puede ser mayor, el costo a largo plazo puede ser menor debido a la menor necesidad de actualizaciones y mantenimiento comparado con el software, además de ofrecer un mejor rendimiento energético y de procesamiento.

Dado el conjunto de todas estas características se ha preferido usar hardware, especialmente FPGAs, para la implementación de cifradores de bajo consumo. Esta preferencia asegura que los sistemas criptográficos no sólo sean seguros y eficientes, sino también capaces de operar de manera óptima en una variedad de entornos, desde infraestructuras críticas hasta dispositivos IoT que requieren una protección avanzada con un consumo de energía mínimo.

2.2.1 FPGA: Características Principales

Las FPGAs (Field-Programmable Gate Arrays) son conocidas por su capacidad de ser programadas y reprogramadas después de la fabricación, lo que permite a los diseñadores y desarrolladores alterar su configuración para adaptarla a diversas necesidades y cambios en los requisitos del sistema. Esta flexibilidad hace que sean especialmente útiles en campos de rápida evolución, como la seguridad y la comunicación, donde los sistemas deben actualizarse frecuentemente para manejar nuevos estándares o para mejorar su rendimiento y seguridad sin necesidad de rediseñar completamente el hardware [7].

Una de las ventajas más significativas de las FPGAs es su capacidad para ejecutar múltiples operaciones en paralelo. A diferencia de los procesadores tradicionales que procesan las instrucciones de manera secuencial, las FPGAs pueden configurarse para realizar cálculos complejos en paralelo, lo que aumenta enormemente la velocidad de procesamiento de las aplicaciones. Esta característica es valiosa para aplicaciones que requieren un gran volumen de procesamiento de datos y cálculos en tiempo real, como el procesamiento de señales, comunicaciones móviles y aplicaciones de video, donde el rendimiento del sistema es crítico.

Además, ofrecen una mayor eficiencia energética en comparación con los sistemas basados en CPU o GPU tradicionales. Al poder configurar exactamente las rutas de hardware necesarias para una tarea específica, los FPGAs eliminan la necesidad de componentes superfluos que consumirían energía adicional. Además, al operar a velocidades de reloj más bajas que las CPUs para muchas tareas específicas, pueden ofrecer una disminución significativa en el consumo de energía, lo cual es crucial en dispositivos portátiles y aplicaciones donde la eficiencia energética es fundamental. Esta eficiencia también se traduce en menos calor disipado, lo que simplifica los requisitos de enfriamiento y reduce los costos de mantenimiento.

Además, los FPGAs pueden ser diseñados con características de seguridad integradas, como la criptografía a nivel de hardware y la protección contra manipulaciones físicas. Estas características los hacen ideales para su uso en aplicaciones donde la seguridad es de máxima prioridad, como en sistemas de pago, aplicaciones militares y de infraestructura crítica.

2.2.2 Metodología de Diseño

La metodología de diseño de las FPGA se centra fundamentalmente en el uso de lenguajes de descripción de hardware (HDLs), como VHDL y Verilog, que son fundamentales para definir la estructura y el comportamiento de los sistemas electrónicos. Estos lenguajes permiten a los diseñadores crear modelos a nivel de transferencia de registros (RT) de componentes electrónicos que luego se implementan físicamente en un FPGA. La precisión y flexibilidad que ofrecen estos lenguajes son esenciales para explotar plenamente las capacidades de las FPGAs, permitiendo diseños complejos que pueden ser reiteradamente modificados y optimizados antes de su implementación final.

Los lenguajes de descripción de hardware proporcionan estructuras sintácticas y semánticas que facilitan la descripción detallada del circuito a nivel de bit o de dato, lo que permite simular el diseño antes de la fabricación. Esta capacidad es crucial para validar la lógica y detectar errores en las fases tempranas del desarrollo. Además, el uso de HDLs facilita la iteración rápida y la optimización

del diseño, permitiendo a los diseñadores realizar cambios y evaluar los efectos inmediatos sin necesidad de prototipos físicos costosos.

La metodología de diseño basada en HDLs tiene un impacto significativo en la velocidad y el costo de producción de las FPGA. Al permitir una descripción precisa del hardware deseado, los HDLs reducen el riesgo de errores costosos en las etapas finales de desarrollo y producción. Además, al facilitar la reutilización de componentes probados en nuevos proyectos, los HDLs pueden disminuir significativamente el tiempo de desarrollo y los costos asociados, acelerando el ciclo de innovación en la electrónica digital.

Junto con los HDLs, diversas herramientas de desarrollo juegan un papel crucial en el diseño de FPGA. Software de síntesis, herramientas de simulación y procesos de Place and Route son parte integral del desarrollo, con ejemplos populares como Quartus II de Intel y Vivado o ISE de Xilinx. Estas herramientas proporcionan entornos integrados que facilitan todo el proceso de diseño de las FPGA.

En cuanto a los fabricantes, empresas como AMD-Xilinx e Intel-FPGA lideran el mercado, ofreciendo soluciones que se adaptan a una amplia gama de aplicaciones industriales y comerciales. Xilinx, en particular, ofrece varias familias de FPGAs, como Virtex, Artix y Spartan, diseñadas para diferentes niveles de rendimiento y aplicaciones específicas, desde telecomunicaciones de alto rendimiento hasta dispositivos portátiles y aplicaciones comerciales generales.

2.2.3 Placas de desarrollo

Las placas de desarrollo FPGA son herramientas esenciales en ingeniería electrónica, proporcionando una plataforma efectiva para el desarrollo, prueba y prototipado de soluciones electrónicas. Estas plataformas son valiosas tanto en contextos educativos como profesionales, ya que permiten a los usuarios diseñar y modificar circuitos sin el alto costo y tiempo asociado con la fabricación de hardware personalizado. Son especialmente útiles para la experimentación y la validación de conceptos en etapas tempranas de diseño, acelerando el proceso de desarrollo de nuevos productos y tecnologías.

La Nexys4 DDR de Digilent es una placa de desarrollo que se destaca en el ámbito educativo y de investigación debido a su balance óptimo entre costo y funcionalidad. Está equipada con un dispositivo XC7A100T de la serie 7 de Xilinx, y viene con una variedad de características y elementos que apoyan el desarrollo avanzado de aplicaciones electrónicas.

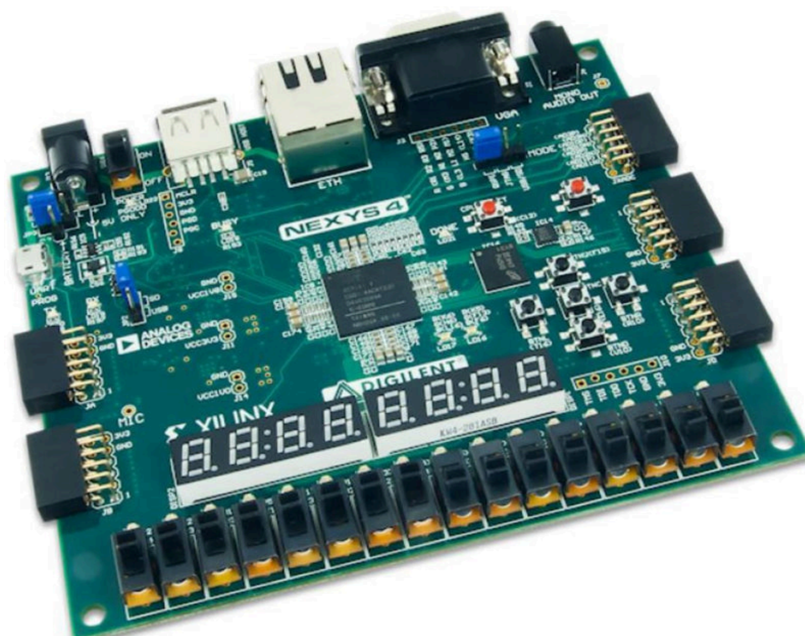

Figura 1. Nexys4 DDR

El dispositivo FPGA, que se encuentra en la placa Nexys4 DDR, es responsable de realizar todas las operaciones de procesamiento y lógica digital en la placa.

Las FPGAs de la familia Artix-7 tienen un alto rendimiento y flexibilidad, permitiendo a los usuarios configurar el dispositivo para realizar una amplia gama de funciones digitales que van desde el procesamiento de señales simples hasta operaciones complejas de procesamiento de datos.

La placa Nexys4 DDR incorpora 128MB de memoria DDR2, que es esencial para aplicaciones que manejan grandes volúmenes de datos o requieren altas velocidades de procesamiento. La memoria DDR2 es crucial para soportar aplicaciones intensivas como el procesamiento de imágenes o la ejecución de algoritmos complejos que necesitan un acceso rápido y eficiente a los datos [8].

2.3 Introducción a la familia de cifradores *ASCON*

La información que se presenta en esta sección se basa principalmente en datos obtenidos de la fuente Graz University of Technology. (n.d.). *ASCON* – Authenticated Encryption and hashing de https://ascon.iaik.tugraz.at/ [9]. Esta

propuesta es fruto del trabajo conjunto de renombrados especialistas en criptografía como Christoph Dobraunig, María Eichlseder, Florián Mendel y Martin Schläffer.

Se aborda en esta sección tres componentes clave: los diseños de encriptación autenticada de *ASCON*, las fases detalladas del proceso de encriptación, que explican cómo se aplica el algoritmo fase por fase, y las permutaciones específicas que aseguran la seguridad y optimización del algoritmo.

2.3.1 Familia de Cifradores *ASCON*: Diseños de Encriptación Autenticada

El conjunto de cifradores *ASCON* incluye una familia de diseños para encriptación autenticada y la función de *Hash ASCON-Hash*, que utiliza la función de salida extensible *ASCON*-Xof. Los diseños se definen con base en parámetros específicos y se describen a continuación en la tabla 4, incluidos los procedimientos de cifrado y descifrado.

La familia de *ASCON* están parametrizados por la longitud de la clave (k menor o igual a 160 bits), el tamaño del bloque de datos r y los números de rondas internas (a y b). A partir de estos valores se especifica un algoritmo de encriptación autenticada, *E (k, r, a, b)*, y un algoritmo de descifrado, *D (k, r, a, b)* donde se fijan los valores de los parámetros.

El procedimiento de encriptación autenticada, *E (k, r, a, b)*, toma las siguientes entradas:

- Una clave secreta *K* con *k* bits.
- Un *Nonce N* con 128 bits.
- Datos asociados *A* de cualquier longitud.
- Un texto plano *P* de cualquier longitud.

Este procedimiento produce:

- Un texto cifrado *C* de la misma longitud que el texto plano *P*.
- Una etiqueta de autenticación *T* de 128 bits que autentica los datos asociados y el mensaje cifrado.

Procedimiento de Descifrado y Verificación *D (k, r, a, b)* recibe como entradas:

- La clave *K*.
- El *Nonce N*.
- Los datos asociados *A*.
- El texto cifrado *C*.
- La etiqueta de autenticación *T*.

La salida es el texto plano *P* si la verificación de la etiqueta es correcta, o un error "?" si la verificación falla.

Encriptación: *ASCON* mezcla la clave secreta, el *Nonce* y los datos asociados para configurar el estado interno. Luego, incorpora el texto plano al estado, generando el texto cifrado y la etiqueta de autenticación.

Descifrado: Utiliza la misma clave y *Nonce*, junto con los datos asociados, para reconstruir el estado interno y procesar el texto cifrado, recuperando el texto plano y verificando la etiqueta de autenticación.

Esta configuración permite que *ASCON* se adapte a diversas necesidades de seguridad y rendimiento, asegurando una encriptación y autenticación eficaz y segura de mensajes, especialmente en entornos con recursos limitados.

K	Clave secreta K de $k \leq 160$ bits.
N, T	*Nonce* N, etiqueta T, todos de 128 bits.
P, C, A	Texto plano P, texto cifrado C, datos asociados A (en bloques de r bits P_i, C_i, A_i).
M, H	Mensaje M, valor de *Hash* H (en bloques de r bits Mi, Hi).
?	Error, la verificación del texto cifrado autenticado falló.
S	El estado de 320 bits S de la construcción de esponja.
S_r, S_c	La parte de la tasa de r bits y la capacidad de c bits del estado S.
p, p^a, p^b	Permutaciones p^a, p^b que consisten en *a, b* rondas de actualización *p*, respectivamente.
$X \in \{0,1\}^k$	Cadena de bits x de longitud k (variable si $k = \lambda$).
0^k	Cadena de bits de k bits (longitud variable si $k = \lambda$), todos 0.
$\|x\|$	Longitud de la cadena de bits x en bits.
$\|x\|_k$	Cadena de bits x truncada a los primeros (más significativos) k bits.
$[x]^k$	Cadena de bits x truncada a los últimos (menos significativos) k bits.
$x \| y$	Concatenación de cadenas de bits x e y.
$x \oplus y$	XOR de cadenas de bits x e y.
x mod y	Resto en la división entera de x por y.
$[x]$	Función techo, el entero más pequeño mayor que x.
p_c, p_s, p_L	adición de constante, sustitución y capa lineal de $p = p_L \circ p_s \circ p_c$.
$x_0, \ldots, x_4$	Las cinco palabras de 64 bits del estado S.
$x_{0,i}, \ldots, x_{4,i}$	Bit i, $0 \leq i < 64$, de las palabras x0, ..., x4, con x_0,0 el bit más a la derecha (LSB).

Tabla 4. Notación utilizada para la interfaz, modo y permutación de ASCON.

Esquemas Recomendados de Encriptación Autenticada

Aunque la propuesta del algoritmo no fija el tamaño de la clave ni el tamaño de los bloques de datos, *ASCON* propone dos configuraciones principales para la encriptación autenticada, denominadas *ASCON*-128 y *ASCON*-128a, que fueron seleccionadas en la competición CAESAR como las recomendadas para casos de uso ligeros. En ambas configuraciones el tamaño de clave es de 128 bits. Los valores de los parámetros se muestran en la tabla 5.

Nombre	Algoritmos	Tamaño de clave	*Nonce*	Tag	Data block	p^a	p^b
ASCON-128	E, D	128	128	128	64	12	6
ASCON-128ª	E, D	128	128	128	128	12	8

Tabla 5. Parámetros para esquemas recomendados de encriptación autenticada.

ASCON también especifica recomendaciones para el *Hashing*, con *ASCON-Hash* siendo la principal y única recomendación la que se muestra en la tabla 6.

Nombre	Algoritmos	Tamaño de clave	Data block	p^a	p^b
ASCON-Hash	X	256	64	12	12
*ASCON-Hash*a	X	256	12	12	8

Tabla 6. Parámetros para algoritmos de Hashing recomendados.

La instancia recomendada de *Hashing*, *ASCON-Hash*, utiliza un tamaño de bloque de datos de 64 bits y 12 rondas para la permutación, produciendo una salida de *Hash* de 256 bits. *ASCON-Hash*a se distingue de *ASCON-Hash* principalmente en el número de rondas de la permutación pb, siendo *ASCON-Hash*a más ligero con 8 rondas en lugar de las 12 rondas utilizadas en *ASCON-Hash*.

Para emparejar la encriptación autenticada con el *Hashing*, se recomienda combinar *ASCON*-128 y *ASCON-Hash*, ya que ambos comparten el mismo tamaño de bloques.

Construcciones adicionales basadas en la Permutación de *ASCON*

Además de las recomendaciones principales, la permutación de *ASCON* se puede utilizar para otros propósitos y en diferentes configuraciones de parámetros.

- *ASCON*-Xof: Se define una función de salida extensible que utiliza el algoritmo X0,64,12 con una tasa de 64 bits y 12 rondas para pa, produciendo una salida de *Hash* de longitud arbitraria.

- *ASCON*-80pq: Se define un nuevo esquema de encriptación autenticada que utiliza los algoritmos E, D160,64,12,6 con un tamaño de clave aumentado a 160 bits, un tamaño de *Nonce* y etiqueta de 128 bits, una tasa de 64 bits, 12 rondas para p^a y 6 rondas para pb.

Estado interno

La familia *ASCON* opera utilizando un estado interno de 320 bits, el cual se actualiza mediante permutaciones designadas como P_a (con a rondas) y pb (con b rondas). Este modelo de funcionamiento está basado en la construcción de esponja criptográfica, donde el estado se divide en dos partes principales: una parte externa (Sr) y una parte interna (Sc), con una tasa (r) y una capacidad (c = 320 - r) que varían según la variante de *ASCON* empleada.

El estado de 320 bits (S) se divide en:

- Parte externa (Sr): de r bits, la cual interactúa directamente con los datos durante las fases de absorción y extracción de la esponja.
- Parte interna (Sc): de c bits, que sirve para mantener la seguridad de los datos al no interactuar directamente con ellos y asegurar la no reversibilidad del proceso.

Para la descripción y aplicación de las transformaciones de ronda, el estado de 320 bits (S) se divide en cinco registros de 64 bits denominados palabras.

$$S = S_r \parallel S_c =$$

Esta disposición permite que las operaciones sobre el estado se realicen de manera eficiente, aprovechando las propiedades de la arquitectura del hardware donde se implemente *ASCON*, como procesadores de 64 bits.

Interfaz de Esponja y Codificación

Cuando se necesita interpretar el estado como un array de bytes (o cadena de bits) para la interfaz de esponja, se comienza con el byte (o bit) más significativo. Esta forma de interpretación es crucial para la conversión entre la representación del estado interno y los datos que se están procesando (por ejemplo, durante la absorción del mensaje en la fase de esponja o la generación del cifrado y la etiqueta de autenticación).

Este modelo de funcionamiento permite a *ASCON* realizar tanto la encriptación autenticada como operaciones de *Hashing* de manera segura y eficiente, optimizando el uso del estado para garantizar tanto la confidencialidad como la integridad de los datos procesados. La división del estado en registros de 64 bits facilita la implementación de las permutaciones y las transformaciones de ronda, elementos clave para la seguridad del algoritmo.

2.3.2 Fases de *ASCON*

ASCON utiliza un modo de operación basado en esponja dúplex para la encriptación autenticada, optimizado para seguridad y eficiencia en entornos con recursos limitados. La configuración recomendada para la longitud de clave, etiqueta y *Nonce* es de 128 bits. El algoritmo se caracteriza por las siguientes fases mostradas en la figura 2:

1. Inicialización: Se inicia el estado con la clave secreta K y el *Nonce* N Esta fase prepara el estado de la esponja para el procesamiento de datos.
2. Procesamiento de los datos asociados: Actualiza el estado con bloques de datos asociados A_i. Los datos asociados son procesados para integrarlos en el estado, pero no se cifran.
3. Procesamiento de Texto Plano: Inyecta bloques de texto plano P_i en el estado y extrae bloques de texto cifrado C_i. Esta es la fase donde se realiza el cifrado del mensaje.
4. Finalización: Inyecta la clave K nuevamente y extrae la etiqueta T para autenticación. La etiqueta asegura la integridad y la autenticidad del mensaje cifrado.

Después de cada bloque inyectado (excepto el último bloque de texto plano), se aplica la permutación central p^b al estado completo. Durante la inicialización y la finalización, se utiliza una permutación más fuerte p^a con más rondas. Los números de rondas a y b, así como la tasa y capacidad de la esponja, dependen de la variante de *ASCON*. Los parámetros recomendados son:

ASCON-128: 128 bits para clave, *Nonce* y etiqueta; tasa de 64 bits, capacidad de 256 bits, 12 rondas en p^a y 6 rondas en p^b.

ASCON-128a: 128 bits para clave, *Nonce* y etiqueta; tasa de 128 bits, capacidad de 192 bits, 12 rondas en p^a y 8 rondas en p^b.

Estas configuraciones permiten un equilibrio entre seguridad y rendimiento, haciendo a *ASCON* adecuado para una amplia gama de aplicaciones,

especialmente en dispositivos donde el espacio y el consumo de energía son consideraciones críticas.

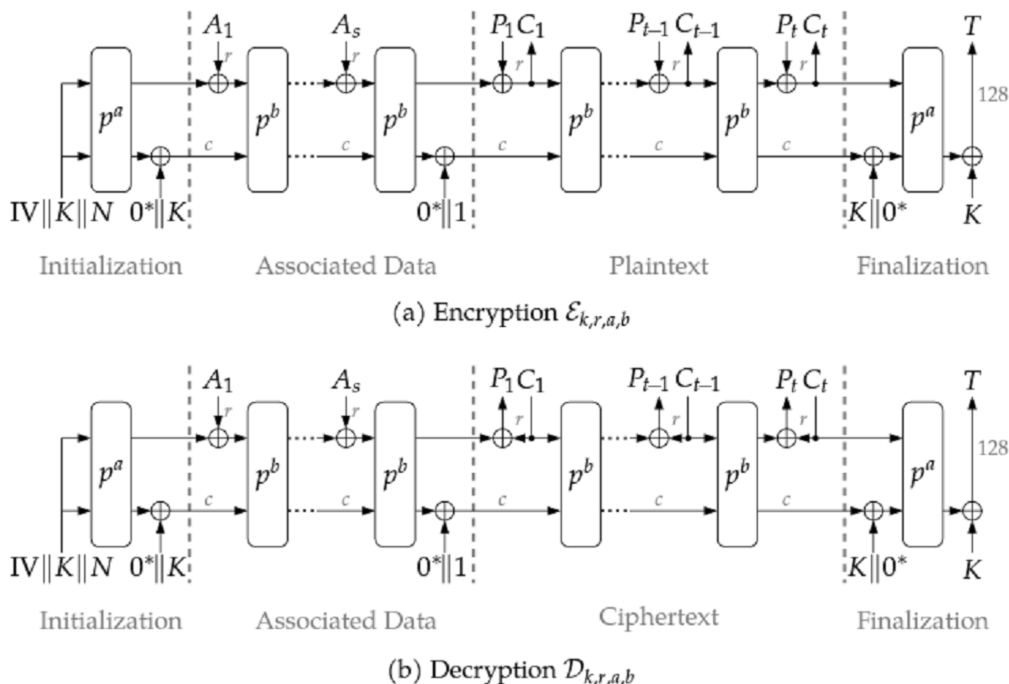

Figura 2. Proceso de (a) cifrado, (b) descifrado.
Fuente: https://ascon.iaik.tugraz.at

Inicialización

El estado inicial de 320 bits de *ASCON* se forma por la clave secreta *K* de *k* bits y el *Nonce N* de 128 bits, así como un IV que especifica el algoritmo (incluyendo el tamaño de la clave *k*, la tasa *r*, el número de rondas de inicialización y finalización a, y el número de rondas intermedias *b*, cada uno escrito como un entero de 8 bits):

$IV_{k,r,a,b}$ = k | r | a | b | 0^{160-k} =
- → 0x80400c0600000000 para *ASCON*-128
- → 0x80800c0800000000 para *ASCON*-128a
- → 0xa0400c06 para *ASCON*-80pq

Donde S ← IV {k, r, a, b} | K | N

En la inicialización, se aplican a rondas de la transformación de ronda p al estado inicial, seguido por un XOR de la clave secreta K:

$$S \leftarrow p^a(S) \oplus (0^{320-k} \| K)$$

Procesado del dato asociado

ASCON procesa los datos asociados A en bloques de r bits. Añade un único 1 y el menor número de 0s a A para obtener un múltiplo de r bits y lo divide en s bloques de r bits, $A_1||\ldots||A_s$. En caso de que A esté vacío, no se aplica ningún relleno y s = 0:

$$A_1,\ldots,A_s \leftarrow r \text{ bit blocks of } A \,||\, 1 \,||\, 0^{r-1-(|A| \bmod r)} \text{ si } |A| > 0$$
$$A_1,\ldots,A_s \leftarrow \emptyset \text{ si } |A| = 0$$

Cada bloque A_i con $i = 1,\ldots,s$ se hace un XOR con los primeros r bits S_r del estado S, seguido por la aplicación de la permutación de b rondas p_b a S:

$$S \leftarrow p^b((S_r \oplus A_i)\,||\,S_c)$$

Después de procesar A_s (también si $s=0$), se hace un XOR a S con una constante de separación de dominio de 1 bit:

$$S \leftarrow S \oplus (0^{319}\,||\,1)$$

Procesado del texto plano / texto cifrado

ASCON procesa el texto plano P en bloques de r bits. El proceso de relleno añade un único 1 y el menor número de *0s* al texto plano P de tal manera que la longitud del texto plano rellenado sea un múltiplo de r bits. El texto plano rellenado resultante se divide en t bloques de r bits, P_1 hasta P_t.

$$P_1,\ldots,P_s \leftarrow r \text{ bit blocks of } P \,||\, 1 \,||\, 0^{r-1-(|P| \bmod r)}$$

Encriptación

En cada iteración, un bloque de texto plano rellenado Pi, con i = 1 hasta t, se hace XOR con los primeros r bits Sr del estado interno S, seguido de la extracción de un bloque de texto cifrado Ci. Para cada bloque excepto el último, todo el estado interno S es transformado por la permutación pb usando b rondas.

$$C_i \leftarrow S_r \oplus P_i$$

$$S \leftarrow p^b (C_i \,||\, S_c) \quad \text{si} \quad 1 \leq i < t$$

$$S \leftarrow C_i \,||\, S_c \quad \text{si} \quad 1 \leq i = t$$

El último bloque de texto cifrado Ct se trunca a la longitud del último fragmento de bloque de texto plano sin rellenar, de tal manera que su longitud esté entre 0 y r-1 bits, y la longitud total del texto cifrado C sea exactamente la misma que la del texto plano P original.

$$C_t \leftarrow |C_t|_{|P| \bmod r}$$

Descifrado

En cada iteración, excepto la última, el bloque de texto plano Pi se calcula haciendo XOR del bloque de texto cifrado Ci con los primeros r bits Sr del estado interno. Luego, los primeros r bits del estado interno, Sr, son reemplazados por Ci. Finalmente, para cada bloque de texto cifrado excepto el último, el estado interno se transforma por la permutación pb de b rondas.

$$P_i \leftarrow S_r \oplus C_i$$

$$S \leftarrow p^b (C_i \,||\, S_c) \quad \text{si} \quad 1 \leq i < t$$

Para el último bloque de texto cifrado truncado, con 0 ≤ L < r, el procedimiento es diferente.

$$P_t \leftarrow |S_r|_L \oplus C_t$$

$$S \leftarrow (S_r \oplus (P_t \,||\, 1 \,||\, 0^{r-1-L}))||S_c)$$

Este proceso asegura que el texto cifrado tenga la misma longitud que el texto plano original, manteniendo la integridad del mensaje.

Finalización

En la fase de finalización de *ASCON*, se hace una operación XOR de la clave secreta *K* con el estado interno, seguido de la aplicación de la permutación P_a utilizando *a* rondas para transformar el estado. Este proceso termina con la generación del *TAG,* el cual se obtiene haciendo *XOR* entre los últimos 128 bits

menos significativos del estado transformado y los últimos 128 bits de la clave secreta *K*.

$$S \leftarrow P_a\,(S \oplus (0^{r-1-L} \,||\, K \,||\, 0^{c-k}\,)\,)$$

$$T \leftarrow [S]^{128} \oplus [K]^{128}$$

El algoritmo de encriptación de *ASCON* devuelve el tag ***T*** junto con el texto cifrado, concatenando $C_1||\ldots||C_T$ De manera similar, el algoritmo de descifrado solo devuelve el texto plano, $P_1||\ldots||P_T$, si el valor del tag calculado coincide con el valor del tag recibido. Esta verificación del tag es esencial para asegurar la integridad y autenticidad del mensaje: solo cuando el tag calculado durante el proceso de descifrado coincide exactamente con el tag proporcionado junto con el mensaje cifrado, se puede confiar en que el mensaje no ha sido alterado y que el remitente es legítimo, proporcionando así una robusta seguridad en la comunicación.

2.3.3 Permutaciones *ASCON*

ASCON, se basa en una serie de permutaciones cuidadosamente construidas para proporcionar la mezcla y difusión necesarias para una robusta seguridad criptográfica.

Estas transformaciones incluyen la adición de constantes de ronda, una capa de sustitución no lineal y una capa de difusión lineal. Cada una de estas etapas desempeña un papel único, manteniendo la eficiencia necesaria para su uso en dispositivos con recursos limitados.

Las constantes de ronda fueron seleccionadas con el objetivo de prevenir ataques como los de tipo *slide* y rotacionales, entre otros. Estas constantes se eligen de manera sencilla, utilizando un contador creciente y decreciente para las dos mitades del byte afectado, lo que simplifica su cálculo a través de operaciones básicas de cuenta e inversión.

La baja entropía de estas constantes asegura que no se utilicen para implementar puertas traseras en el algoritmo, manteniendo así la confianza en la seguridad del cifrado. Además, el patrón de las constantes de ronda puede extenderse hasta 16 rondas si se busca un margen de seguridad muy alto. Añadir más de 16 rondas no se espera que mejore significativamente este margen.

El posicionamiento estratégico de las constantes de ronda, particularmente en la palabra x2 del estado, fue elegido para permitir el encadenamiento con las operaciones siguientes o previas. Esto incluye la inyección de mensajes en la primera ronda o las instrucciones subsiguientes de la implementación de la S-box en trozos de bits (*bit-sliced*). Estas transformaciones incluyen 3 etapas:

Adición de Constantes de Ronda

Introduce variabilidad y previene estructuras repetitivas en el proceso de cifrado.

Capa de Sustitución No Lineal: Aporta complejidad y resistencia contra ataques lineales y diferenciales.

Capa de Difusión Lineal: Asegura que los cambios en una parte del estado se propaguen rápidamente a través del algoritmo.

Adición de Constantes de Ronda

Esta etapa involucra una operación XOR aplicada a una constante específica de ronda de 1 byte contra la palabra x2 del estado del cifrado como se muestra en la figura 3.

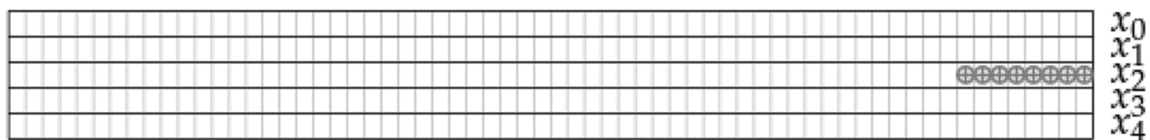

Figura 3. Adición de constantes de ronda. Fuente: https://ascon.iaik.tugraz.at.

Su funcionamiento se basa en que cada ronda del proceso de permutación incluye la adición de una constante de ronda única. Esta constante es un valor de 1 byte que se combina con la palabra x2 (una de las cinco palabras de 64 bits que componen el estado de 320 bits de *ASCON*) mediante una operación XOR.

El objetivo principal es que la adición de constantes de ronda ayuda a evitar patrones repetitivos o predecibles en el proceso de cifrado, aumentando así la resistencia del algoritmo contra diversos tipos de ataques criptográficos.

Las constantes específicas de cada ronda se eligen cuidadosamente para asegurar la integridad y seguridad del proceso de cifrado. Esta elección estratégica es crucial para evitar vulnerabilidades como ataques de similitud rotacional.

p^{12}	p^8	p^6	Constant c_r	p^{12}	p^8	p^6	Constant c_r
0			000000000000000000f0	6	2	0	00000000000000000096
1			000000000000000000e1	7	3	1	00000000000000000087
2			000000000000000000d2	8	4	2	00000000000000000078
3			000000000000000000c3	9	5	3	00000000000000000069
4	0		000000000000000000b4	10	6	4	0000000000000000005a
5	1		000000000000000000a5	11	7	5	0000000000000000004b

Figura 4. Numero de rondas que se le asigna a una constante. Fuente: https://ascon.iaik.tugraz.at.

Capa de sustitución no lineal

La capa de sustitución en *ASCON* es fundamental para su seguridad y efectividad como algoritmo de cifrado. Actúa como la única parte no lineal de la transformación de ronda, mezclando cinco bits en cada una de las mismas posiciones de bits en una de las cinco palabras del estado. La S-box utilizada es invertible y no tiene puntos fijos, lo que significa que cada salida es única para una entrada dada (Figura 5), y no hay entrada para la cual la salida sea igual a

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$\mathcal{S}(x)$	4	b	1f	14	1a	15	9	2	1b	5	8	12	1d	3	6	1c

x	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
$\mathcal{S}(x)$	1e	13	7	e	0	d	11	18	10	c	1	19	16	a	f	17

Figura 5. S-BOX. Fuente: https://ascon.iaik.tugraz.at.

la entrada. Este diseño es esencial para prevenir estructuras predecibles y aumentar la resistencia del algoritmo contra diversos tipos de ataques criptográficos, manteniendo la eficiencia del diseño global.

La capa de sustitución mezcla cinco bits en cada una de las mismas posiciones de bits en una de las cinco palabras de estado. Esto significa que se aplican operaciones no lineales a grupos de cinco bits en cada palabra del estado de *ASCON*, que se compone de cinco palabras de 64 bits cada una.

Capa de difusión Lineal

Dentro de cada palabra del estado de *ASCON* (cada una de las 5 palabras de 64 bits), se realizan operaciones XOR (OR exclusivo) con diferentes versiones rotadas de la misma palabra. Esto se hace de manera horizontal, es decir, dentro de cada palabra.

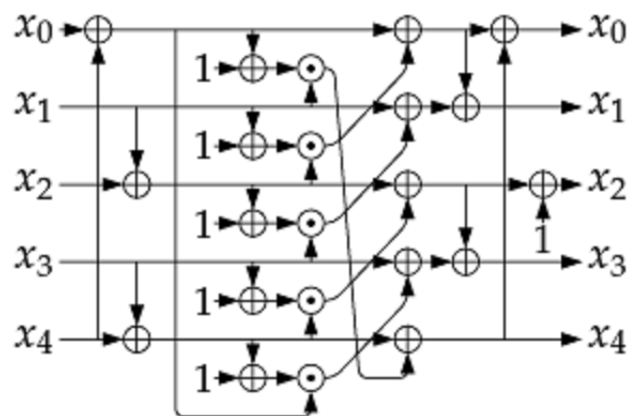

Figura 6. S-BOX. Fuente: https://ascon.iaik.tugraz.at

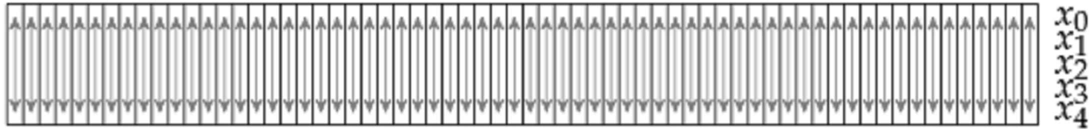

Figura 7. Mezcla de 5 bits en cada posición de cada palabra de 64 bits. Fuente: https://ascon.iaik.tugraz.at

Figura 8. S-box. Fuente: https://ascon.iaik.tugraz.at.

Rotaciones Específicas

Cada palabra del estado se combina con sus propias versiones rotadas a la derecha por cantidades específicas de bits. Las rotaciones son diferentes para cada palabra del estado y están diseñadas para maximizar la difusión a lo largo de la palabra.

Las operaciones se especifican de la siguiente manera:

$X_0 := x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)$

$x_1 := x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)$

$x_2 := x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6)$

$x_3 := x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)$

$x_4 := x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)$

Esta capa de difusión lineal es importante porque ayuda a dispersar la influencia de un bit de entrada a través de toda la palabra, aumentando así la resistencia del cifrado frente a varios tipos de ataques criptográficos. La difusión efectiva es un componente esencial para asegurar que los cambios en una parte del estado se propaguen rápidamente a través de todo el estado, una característica clave para un cifrado fuerte y seguro.

CAPITULO 3: Implementación hardware del algoritmo ASCON

En este capítulo se presenta la implementación hardware del algoritmo de cifrado *ASCON*, abordando tanto su diseño como la comprobación de su funcionamiento. Se comienza con una descripción exhaustiva de la interfaz del cifrador, detallando los componentes principales del diseño y su operación, así como los procedimientos específicos para la carga del *Nonce*, el procesamiento de datos asociados y de texto plano/cifrado, y el proceso de finalización y manejo de bloques parciales.

A continuación, se profundiza en el diseño hardware del cifrador *ASCON*, proporcionando una visión integral de su arquitectura y los elementos que la componen. Este análisis incluye la implementación del código en hardware, ilustrando cómo se han traducido los conceptos teóricos en un diseño tangible y funcional.

Finalmente, se realiza una comparativa del algoritmo *ASCON* en su función de cifrado con su variante generación de *Hash*, *ASCON_Hash*. Esta sección tiene como objetivo destacar las diferencias y similitudes entre ambas implementaciones, evaluando su rendimiento y eficiencia en distintos contextos de uso.

3.1 Interfaz

Para el desarrollo de este diseño, se ha tomado como referencia la interfaz desarrollada en el trabajo de fin de máster de Carlos Fernández titulado “Diseño Microelectrónico de Cifradores *AEAD* con Introducción de Técnicas de Reducción del Consumo de Potencia” [10], y más específicamente la definición de las señales de control para el cifrador Ascon. Este documento es una buena guía para la identificación y comprensión de las señales de entrada y salida necesarias para la implementación. La descripción detallada, incluyendo las señales de control, datos y señales indicadoras, ha facilitado la tarea de establecer un marco coherente y funcional para el diseño. A través de la adaptación y reinterpretación de estas señales, se ha podido configurar una interfaz mostrada en la figura 9 y explicadas en la tabla 7 totalmente eficaz con modificaciones que responden a las necesidades específicas del proyecto referido al cifrador y al hash.

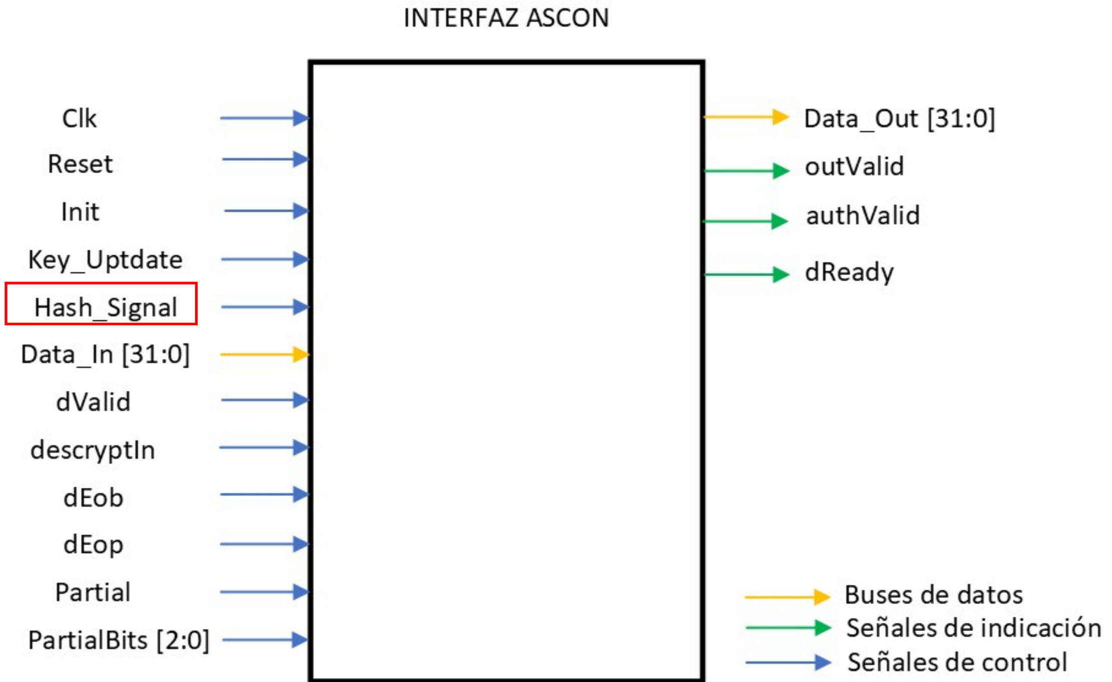

Figura 9. Interfaz para ASCON

SEÑALES DE ENTRADA		
Señal	Tamaño (Bits)	Descripción
Clk	1	Señal de reloj
Reset	1	Señal de reset activo en alta
Init	1	Señal de inicio
Key_Update	1	Señal de inicio de carga de la clave
*Hash*_Signal	1	Señal de inicio para el *Hash*
Data_In	32	Bus de datos de entrada
dValid	1	Señal de que el dato de entrada es valido
descryptIn	1	Señal que indica si ciframos/desciframos
dEob	1	Señal que indica que se está procesando el último bloque
dEop	1	Señal que indica que se está procesando el último paquete.
Partial	1	Señal que indica que el ultimo bloque es parcial
PartialBits	3	Señal que indica el tamaño (bytes) que tiene el ultimo bloque
SEÑALES DE SALIDA		
Señal	Tamaño (Bits)	Descripción
Data_Out	32	Bus de datos de salida
OutValid	1	Señal que indica el dato en el bus de salida es valido
authValid	1	Señal que indica la validez de la operación de descifrado
dReady	1	Señal que indica que el cifrador está listo para recibir dato de entrada.

Tabla 7. Señales de entrada y salida del Diseño ASCON.

3.1.1 Componentes Principales del Diseño

El diseño para el cifrado/descifrado se compone de dos componentes esenciales:

- Top: Se encarga de orquestar la operación del cifrador. Aquí reside la máquina de estados, que dirige las fases de cifrado, junto con la lógica combinacional necesaria para procesar las señales de entrada y salida.
- *ASCON*Permutation: Este módulo ejecuta la permutación específica que utiliza el algoritmo de cifrado, esencial para la seguridad del sistema.

La interfaz está compuesta por distintos tipos de señales:

- Buses de Datos: como dataIn y dataOut, que manejan la transferencia de los datos cifrados y a cifrar. Ambos de 32 bits.
- Señales de Control: que incluyen keyUpdate, dValid, dEob, dEop, init, partial, partialBits y decryptIn, que controlan el proceso de carga de datos y la realización del proceso de cifrado/descifrado.
- Señales de Indicación: como outValid, authValid y dReady, que ofrecen información sobre el estado y la validez de los datos procesados.

El bus de entrada tiene un ancho de 32 bits, pero según el algoritmo, los datos se tienen que procesar en bloques de 64 bits. Esto implica que los datos son introducidos en dos ciclos de reloj, con una adaptación específica para bloques parciales.

3.1.2 Operación del Cifrador

El sistema con reset asíncrono activo en alto que pone al diseño en un estado de espera de inicio de una operación. En este estado se pueden realizar tres operaciones mostrados en la tabla 8:

- Carga de la Clave: Se activa mediante la señal keyUpdate, con una prioridad superior que permite su preeminencia sobre otras señales.
- Inicio del Cifrado/Descifrado: Señales como init y decryptIn dictan el comienzo y la naturaleza de la operación a ejecutar.
- Inicio de la generación de un Hash: La señal hash_signal marca el inicio de esta operación, para la que sólo se necesitan los datos para los que generara el hash.

INIT	KEY_UPDATE	*HASH*_SIGNAL	ESTADO
0	0	0	N/A
-	1	-	LOAD_KEY
1	0	0	CIFRADO/DESCIFRADO
1	0	1	*HASH*
0	0	1	N/A

Tabla 8. Señales para inicialización.

Nos centramos ahora en el proceso de cifrado y descifrado. El modo hash se verá más adelante.

La inclusión de un *Nonce* y la gestión de bloques de datos (asociados, texto plano/cifrado, y tag) se realizan a través de un esquema detallado que abarca la carga de datos, la señalización precisa para el procesamiento de bloques completos o parciales, y la finalización adecuada del proceso de cifrado o descifrado.

Iniciamos el proceso con una señal de activación que prepara el sistema para las operaciones de cifrado o descifrado. Esta activación se señala mediante las entradas init y decryptIn, que marcan el inicio de la operación y definen su naturaleza, respectivamente. La operación comienza oficialmente cuando init se pone a “1”, momento en el cual el sistema está listo para recibir el *Nonce* a través del bus de entrada, marcando este dato como válido con la señal dValid.

3.1.3 Carga de la Clave

La carga de la clave se activa cuando el cifrador está en estado de espera y el valor de la entrada keyUpdate cambia a '1'. Es importante destacar que esta señal tiene mayor prioridad que la señal init. Por lo tanto, si ambas señales (keyUpdate e init) están en '1' simultáneamente, se procederá a cargar la clave en lugar de iniciar el proceso de cifrado/descifrado indicado por la señal decryptIn.

El proceso de carga de la clave, ilustrado en la Figura 3.6, consiste en introducir, mediante el bus de entrada Data_In, cuatro palabras de 32 bits que constituyen los 128 bits de la clave que se usará durante la operación del cifrador. Estas palabras deben asignarse al bus de entrada durante los 4 ciclos de reloj que dura

el periodo de carga tras la señal keyUpdate comenzando por los 32 bits más significativos.

3.1.4 Carga del *nonce*

La carga del *nonce* se realiza de forma similar a la carga de la clave, actualizando el bus de entrada con las porciones del *Nonce* en los siguientes cuatro ciclos de reloj tras recibir la señal init, comenzando también por los 32 bits más significativos. Posteriormente, el sistema está listo para procesar el siguiente bloque de datos, cuya naturaleza se determina por las señales dEob y dEop.

El cifrador maneja varios tipos de bloques de datos en un orden específico, utilizando principalmente señales de control como dEob, dEop, partial, y partialBits para el procesamiento como podemos ver en la tabla 9.

DEob	dEop	Próximo bloque a tratar
0	0	Dato asociado
0	1	Texto plano/cifrado
1	-	Tag

Tabla 9. Próximo bloque a procesar según dEop y dEob al finalizar el nonce

3.1.5 Procesamiento del Dato Asociado

El procesamiento del Dato Asociado comienza inmediatamente después de la carga del *Nonce*. La continuidad del proceso depende de la señal dReady, que indica cuándo se puede actualizar el bus de entrada con las siguientes palabras del Dato Asociado. Como el procesado se hace en bloques de 64 bits, pero el bus de entrada del ASCON es de 32 bits, se preparan los 64 bits, introduciendo primero los 32 bits menos significativos y después los 32 más significativos. En el último bloque de datos hay que poner, en el caso de sea un bloque parcial la entrada partial = “1” y el valor del parcial según la tabla 10.

DEob	dEop	Próximo bloque a tratar
0	0	Dato asociado
0	1	Texto plano/cifrado
1	1	Tag

Tabla 10. Próximo bloque a procesar según dEop y dEob al finalizar el dato asociado.

3.1.6 Procesamiento del Texto Plano/Cifrado

Similar al Dato Asociado, el procesamiento de Texto Plano/Cifrado sigue la misma metodología, con la diferencia en la gestión de las señales dEob y dEop al final del bloque, ya que este puede ser el último bloque opcional antes del Tag. La señal outValid se activa para indicar que el bus de salida contiene el resultado del proceso de cifrado o descifrado. Se muestra en la tabla 11.

DEob	dEop	Próximo bloque a tratar
0	0	Texto plano/cifrado
1	0	Tag

Tabla 11. Próximo bloque a procesar según dEop y dEob al finalizar el texto plano/cifrado

3.1.7 Proceso de Finalización

La operación termina con la generación o verificación del Tag, según se esté cifrando o descifrando. Este paso es diferente en el descifrado, ya que requiere la asignación del Tag al bus de entrada, similar a cómo se manejan el Dato Asociado y el Texto Plano/Cifrado. La señal authValid se utiliza para confirmar la validez del descifrado.

3.1.8 Procesamiento de Bloques Parciales

Una característica notable del sistema es su capacidad para manejar bloques parciales, es decir, aquellos cuyo último componente tiene un tamaño inferior a 64 bits. En estos casos, las señales partial y partialBits se ajustan para indicar la presencia y el tamaño de estos bloques parciales(ver tabla 12), asegurando un tratamiento adecuado durante el ciclo de reloj correspondiente.

Tamaño del último bloque (Bits)	Partial	Partial_Bits
64	0	000
56	1	001
48	1	010
40	1	011
32	1	100
24	1	101
16	1	110
8	1	111

Tabla 12. Valores de Partial y Partil_Bits en función del tamaño del último bloque.

3.2 Diseño Hardware *ASCON*

En este apartado nos adentraremos en el núcleo principal del trabajo: el diseño del cifrador *ASCON* mediante VHDL. Este apartado del trabajo se centra en detallar el diseño realizado, el cual se ha abordado a través de una máquina de estados finitos diseñada específicamente para optimizar y ejecutar el proceso de cifrado de manera eficaz, eficiente y fácil de analizar.

En este apartado se aborda la implementación de *ASCON*-128, una versión específica del algoritmo *ASCON*. Es importante señalar que esta discusión se limita a las capacidades de cifrado y descifrado de *ASCON*-128, dejando de lado las funcionalidades relacionadas con la generación de *Hash*, las cuales se detallan en el siguiente apartado.

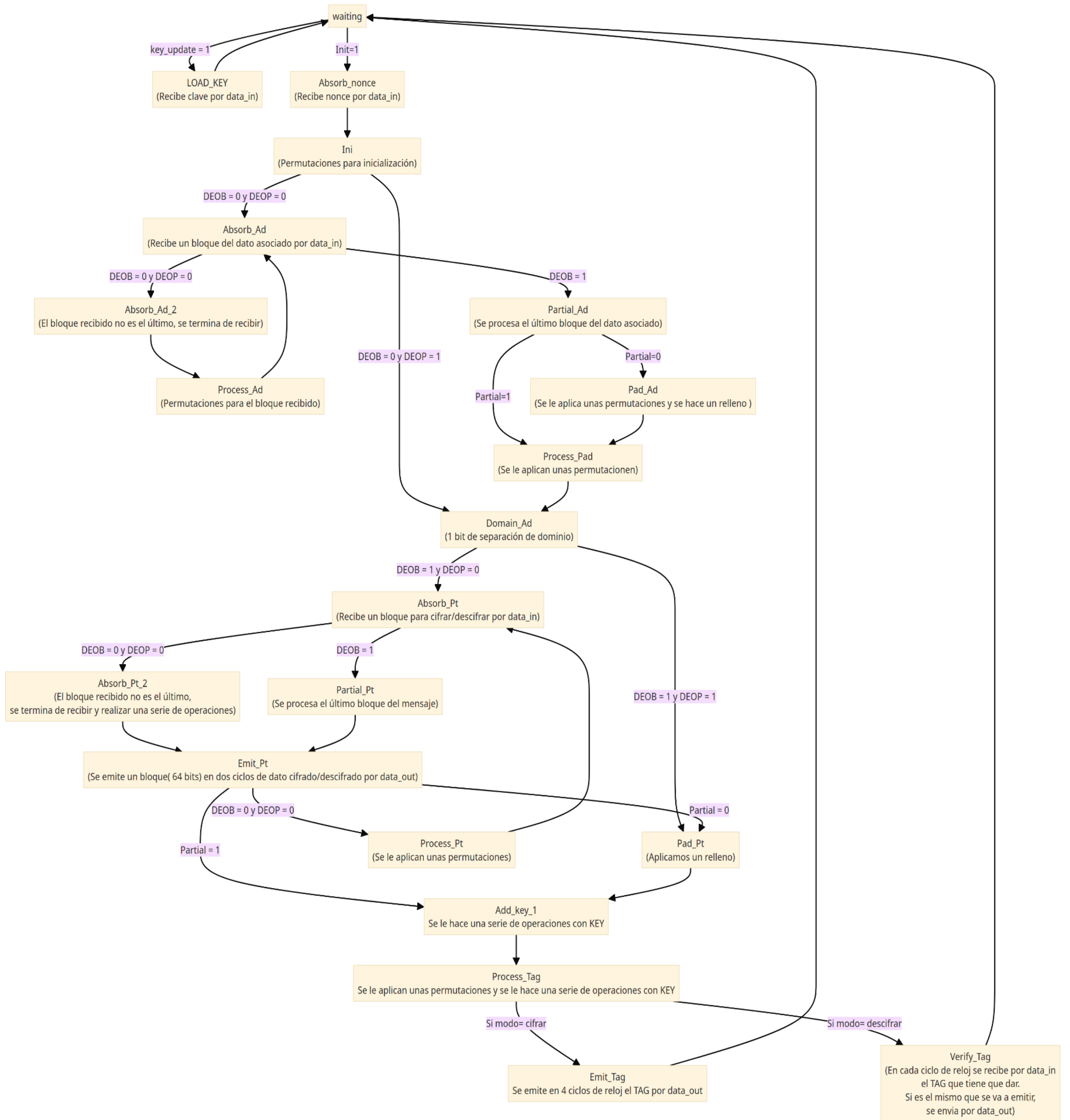

Figura 10. Diagrama de flujo ASCON

El diagrama de flujo mostrado en la Figura 10 ilustra de manera visual el funcionamiento detallado del cifrador. A continuación, se ofrece una descripción general de las principales etapas representadas en el diagrama donde más adelante se verán detalladamente cada etapa:

Inicio y carga de clave:

- El proceso comienza en el estado "Waiting", esperando una señal de actualización de clave (key_update = 1).
- Una vez recibida la señal, se carga una clave dentro del estado "LOAD_KEY", utilizando data_in para recibir la clave. Después de cargar la clave, el proceso vuelve al estado inicial "Waiting".

Inicialización con *nonce*:

- Cuando estando en el estado “Waiting” se activa la señal Init, se pasa al proceso de absorver un nonce (un valor que se utiliza una única vez) mediante "Absorb_nonce". El nonce se recibe también a través de data_in.
- Tras recibir el nonce, el algoritmo ejecuta una serie de permutaciones iniciales en el estado "Ini" para configurar el sistema.

Absorción de datos asociados:

- La absorción de datos asociados se maneja en múltiples estados, comenzando con "Absorb_Ad", que recibe un bloque de datos asociados.
- Dependiendo de los bits de fin de absorción de datos (DEOB) y de operación de dominio (DEOP), el flujo puede seguir diferentes caminos:
    - Si ambos bits son cero, se absorben bloques adicionales en "Absorb_Ad_2".
    - Si DEOB es “1”, se procesa el último bloque de datos asociados en "Partial_Ad".
    - Se aplican permutaciones y rellenos en "Pad_Ad" y "Process_Pad" según sea necesario.
- Un bit de separación de dominio se gestiona en "Domain_Ad".

Absorción y Procesamiento del Mensaje:

- Los bloques de mensaje para cifrar o descifrar se reciben en "Absorb_Pt".
- Si el bloque no es el último (DEOB = 0 y DEOP = 0), el flujo continúa en "Absorb_Pt_2", donde se completan operaciones adicionales.

- Si es el último bloque (DEOB = 1), se procesa en "Partial_Pt".
- Los bloques cifrados o descifrados se emiten en "Emit_Pt".

Relleno y Operaciones con la Clave:

- Los bloques de mensaje se procesan con permutaciones en "Process_Pt".
- Se aplica un relleno en "Pad_Pt" y operaciones adicionales con la clave en "Add_key_1".

Generación y Verificación del TAG:

- En la fase final, se generan y verifican los TAGs (etiquetas) para garantizar la integridad del mensaje.
- Si el modo es cifrar, se emite el TAG en "Emit_Tag".
- Si el modo es descifrar, se verifica el TAG recibido en "Verify_Tag".
- Finalmente, el proceso retorna al estado "Waiting".

En el diseño del cifrador, se han detallado las señales de entrada y salida mostradas en la tabla 13 que rigen las operaciones fundamentales de cifrado y descifrado. Sin embargo, para comprender el mecanismo interno y la eficiencia del sistema, es esencial entender también las señales intermedias. Estas señales juegan un papel crítico en facilitar las transiciones de estado, controlar los procesos de permutación, y gestionar los datos de entrada y salida a lo largo del ciclo de cifrado. A continuación, se presenta una descripción detallada de estas señales intermedias.

Variable /Señal	Tipo / Tamaño (Bits)	Descripción
estado	tipo_estado	Mantiene el estado actual del cifrador.
estado_siguiente	tipo_estado	Determina el próximo estado al que transitará el sistema.
estado_anterior	tipo_estado	Almacena el estado anterior para referencias de flujo de control.
modo	tipo_cifrado	Indica si el sistema está cifrando o descifrando.

reg_state	319 downto 0	Registro principal que almacena el estado del cifrado.
state_out	319 downto 0	Almacena el estado después de aplicar operaciones específicas.
reg_permu_out	319 downto 0	Registro de salida de permutaciones.
IV	63 downto 0	Constante de inicialización específico para este modelo según el algoritmo.
Rounds_A	3 downto 0	Número de rondas para una fase específica del procesamiento (12 rondas).
Rounds_B	3 downto 0	Número de rondas para otra fase del procesamiento (6 rondas).
rondas	3 downto 0	Contador de rondas de permutación restantes en el proceso actual.
contador	2 downto 0	Utilizado para contar operaciones específicas o elementos procesados.
data_in_64	63 downto 0	Almacena datos de entrada temporalmente para procesamiento.
Key	127 downto 0	Clave de cifrado/descifrado.
Tag	127 downto 0	Almacena el Tag de autenticación generado o verificado.
partialBits_signal	2 downto 0	Indica el tamaño en bytes del último bloque si es parcial.
toggle	std_logic	Controla la alternancia para la emisión de datos.
toggle_Tag	integer	Controla la secuencia de emisión del tag de autenticación.
data_out_64	63 downto 0	Almacena temporalmente datos de salida para emisión.

Tabla 13. Definición de señales principales del cifrador.

3.2.1 Flujo de datos

En este apartado se incluye una lista que detalla la incorporación del flujo de los datos en el hardware, permitiendo una comprensión precisa de las operaciones del cifrador en todo momento. Esta lista proporciona una guía clara sobre el manejo de cada bit a lo largo del proceso de cifrado, desde la entrada de los datos hasta la generación de la etiqueta. Al desglosar cada paso del flujo de datos, se facilita la visualización de cómo el cifrador procesa la información.

Inicialización:

1. Cargar key en bloques de 32 bits.
2. Cargar *Nonce* en reg_state (319 downto 192) en bloques de 32 bits.
3. Asignar key a reg_state (191 downto 64).
4. Asignar IV a reg_state (63 downto 0).
5. Realizar 12 rondas de permutaciones en reg_state.
6. Aplicar XOR entre reg_state (319 downto 192) y key(127 downto 0).

Dato asociado:

1. Si no hay dato asociado
   - Añadir dominio: reg_state (312) <= reg_state (312) XOR '1'.
   - Pasar a procesar el mensaje PT o CT
2. Empieza el proceso del dato asociado:
   - Recoger data_in en data_in_64 en dos ciclos de reloj:

```
data_in_64 <= data_in & data_in_64 (63 downto 32)
```

3. Aplicar XOR entre reg_state (63 downto 0) y data_in_64:

```
reg_state (63 downto 0) <= reg_state (63 downto 0) XOR
data_in_64.
```

4. Realizar 6 rondas de permutaciones en reg_state.
5. Recoger otro bloque de dato asociados: Volver al punto 2.
6. Si es el último bloque del dato asociado y el bloque tiene partial:
   - Si falta 1 byte:

```
reg_state (63 downto 0) <= reg_state (63 downto 0) XOR
("10000000" & data_in_64(55 downto 0))
```

- Si faltan 2 bytes:

```
reg_state (63 downto 0) <= reg_state (63 downto 0) XOR
(x"00" & "10000000" & data_in_64(47 downto 0)).
```

- Si faltan 3 bytes:

```
reg_state (63 downto 0) <= reg_state (63 downto 0) XOR
(x"0000" & "10000000" & data_in_64(39 downto 0)).
```

- Así sucesivamente para otros valores de bytes faltantes.
- Realizar 6 rondas de permutaciones en reg_state.
- Añadir dominio:

```
reg_state (312) <= reg_state (312) XOR '1'
```

- Pasar a procesar el mensaje PT o CT.

7. Si es el último bloque y está completo:
   - Aplicar XOR entre reg_state (63 downto 0) y data_in_64.
   - Realizar 6 rondas de permutaciones en reg_state.
   - Aplicar XOR entre reg_state (7 downto 0) y x"80":

```
reg_state (7 downto 0) <= reg_state (7 downto 0) XOR
x"80"
```

- Realizar 6 rondas de permutaciones en reg_state.
- Añadir dominio:

```
reg_state (312) <= reg_state (312) XOR '1'
```

- Pasar a procesar el mensaje PT o CT.

Texto plano/cifrado:

1. Si no hay texto:
   - `reg_state (7 downto 0) <= reg_state (7 downto 0) xor x"80"`
   - Pasar a generar TAG.
2. Recoger data_in en data_in_64 en dos ciclos de reloj:

```
data_in_64 <= data_in & data_in_64(63 downto 32)
```

3. Modo cifrar:

- Aplicar XOR entre data_in_64 (63 downto 0) y reg_state (63 downto 0):

```
data_out_64(63 downto 0) <= reg_state (63 downto 0)
XOR data_in_64
```

- Aplicar XOR entre reg_state (63 downto 0) y data_in_64:

```
reg_state (63 downto 0) <= reg_state (63 downto 0) XOR
data_in_64
```

- Emitir el dato cifrado en bloques de 32 bits por data_out:

  a) Primer ciclo de reloj:

  ```
  data_out <= data_in_64(31 downto 0)
  ```

  b) Segundo ciclo de reloj:

  ```
  data_out <= data_in_64(63 downto 32)
  ```

- Realizar 6 rondas de permutaciones en reg_state.
- Volver a absorber el siguiente bloque de PT/CT.
- Si el bloque recibido es el último bloque, es cifrar y no tiene partial:

  a) Aplicar XOR entre reg_state (63 downto 0) y data_in_64:

  ```
  reg_state (63 downto 0) <= reg_state (63 downto 0)
  XOR data_in_64
  ```

  b) Preparar data_out_64 para emisión:
     - data_out_64 <= (others => '0').
     - data_out_64(63-8*0 downto 0) <= reg_state (63-0 downto 0) XOR data_in_64(63-0 downto 0).

  c) Emitir el dato cifrado en bloques de 32 bits por data_out:
     - Primer ciclo de reloj:

     ```
     data_out <= data_out_64(31 downto 0)
     ```

     - Segundo ciclo de reloj:

     ```
     data_out <= data_out_64(63 downto 32)
     ```

  d) Realizar 6 rondas de permutaciones en reg_state.

  e) Aplicar XOR entre reg_state (7 downto 0) y x"80":

  ```
  reg_state (7 downto 0) <= reg_state (7 downto 0)
  XOR x"80"
  ```

f) Pasar a sacar el TAG.

- Si el bloque recibido es el último bloque, es cifrar y tiene partial

a) Si falta 1 byte:

- `reg_state (63 downto 0) <= reg_state (63 downto 0) XOR ("10000000" & data_in_64(55 downto 0)).`
- `data_out_64 <= (others => '0').`
- `data_out_64(63-8*1 downto 0) <=reg_state (63-8*1downto 0) XOR data_in_64(63-8*1 downto 0).`

b) Si falta 2 byte:

- `reg_state (63 downto 0) <= reg_state (63 downto 0) XOR (x"00" & "10000000" & data_in_64(47 downto 0)).`
- `data_out_64 <= (others => '0').`
- `data_out_64(63-8*2 downto 0) <=reg_state (63-8*2downto 0) XOR data_in_64(63-8*2 downto 0).`

c) Si falta 3 byte:

- `reg_state (63 downto 0) <= reg_state (63 downto 0) XOR (x"0000" & "10000000" & data_in_64(39 downto 0)).`
- `data_out_64 <= (others => '0').`
- `data_out_64(63-8*3 downto 0) <=reg_state(63-8*3downto 0) XOR data_in_64(63-8*3 downto 0).`

d) Y así sucesivamente dependiendo de cuantos bytes falten

e) Emitir el dato descifrado en bloques de 32 bits por data_out:

- Primer ciclo de reloj:

```
data_out <= data_out_64(31 downto 0)
```

- Segundo ciclo de reloj:

```
data_out <= data_out_64(63 downto 32)
```

f) Pasar a generar el TAG.

4. Modo descifrar:

- Guardamos el valor de data_in64 en reg_state(63 downto 0):

```
reg_state(63 downto 0) <= data_in_64
```

- Aplicar XOR entre reg_state (63 downto 0) y data_in_64:

```
data_out_64(63 downto 0) <= reg_state (63 downto 0)
XOR data_in_64
```

- Emitir el dato descifrado en bloques de 32 bits por data_out:
  a) Primer ciclo de reloj:

  ```
  data_out <= data_in_64 (31 downto 0)
  ```

  b) Segundo ciclo de reloj:

  ```
  data_out <= data_in_64 (63 downto 32)
  ```

- Realizar 6 rondas de permutaciones en reg_state.
- Volver a absorber el siguiente bloque de PT/CT.
- Si el bloque recibido es el último bloque, es descifrar y no tiene partial:
  a) Guardamos el valor de data_in64 en reg_state(63 downto 0):

  ```
  reg_state(63 downto 0) <= data_in_64;
  ```

  b) Preparar data_out_64 para emisión:
     - `data_out_64 <= (others => '0').`
     - `data_out_64(63-8*0 downto 0) <= reg_state (63-0 downto 0) XOR data_in_64(63-0 downto 0).`

  c) Emitir el dato cifrado en bloques de 32 bits por data_out:
     - Primer ciclo de reloj:

     ```
     data_out <= data_out_64(31 downto 0)
     ```

     - Segundo ciclo de reloj:

     ```
     data_out <= data_out_64(63 downto 32)
     ```

  d) Realizar 6 rondas de permutaciones en reg_state.
  e) Aplicar XOR entre reg_state (7 downto 0) y x"80":

  ```
  reg_state (7 downto 0) <= reg_state (7 downto 0) XOR
  x"80"
  ```

  f) Pasar a sacar el TAG.
- Si el bloque recibido es el último bloque, es descifrar y tiene partial

a) Si falta 1 byte:

- `reg_state (63 downto 0) <= reg_state (63 downto 0) XOR ("10000000" & data_in_64(55 downto 0))`
- `reg_state (63-8*1 downto 0) <= data_in_64(63-8*1 downto 0)`
- `data_out_64 <= (others => '0')`
- `data_out_64(63-8*1 downto 0) <= reg_state (63-8*1 downto 0) XOR data_in_64(63-8*1 downto 0)`

b) Si faltan 2 bytes:

- `reg_state (63 downto 0) <= reg_state (63 downto 0) XOR (x"00" & "10000000" & data_in_64(47 downto 0))`
- `reg_state (63-8*2 downto 0) <= data_in_64(63-8*2 downto 0)`
- `data_out_64 <= (others => '0')`
- `data_out_64(63-8*2 downto 0) <= reg_state (63-8*2 downto 0) XOR data_in_64(63-8*2 downto 0)`

c) Si faltan 3 bytes:

- `reg_state (63 downto 0) <= reg_state (63 downto 0) XOR (x"0000" & "10000000" & data_in_64(39 downto 0))`
- `reg_state (63-8*3 downto 0) <= data_in_64(63-8*3 downto 0)`
- `data_out_64 <= (others => '0')`
- `data_out_64(63-8*3 downto 0) <= reg_state (63-8*3 downto 0) XOR data_in_64(63-8*3 downto 0)`

d) Y así sucesivamente dependiendo de cuantos bytes falten.
e) Emitir el dato descifrado en bloques de 32 bits por data_out:

- Primer ciclo de reloj:

```
data_out <= data_out_64(31 downto 0)
```

- Segundo ciclo de reloj:

```
data_out <= data_out_64(63 downto 32)
```

f) Pasar a generar el TAG.

Generar tag

- Aplicar XOR entre reg_state (191 downto 64) y key:

```
reg_state (191 downto 64) <= reg_state (191 downto 64)
XOR key
```

- Realizar 12 rondas de permutaciones en reg_state.
- Aplicar XOR entre reg_state (127 downto 0) y key:

```
reg_state (127 downto 0) <= reg_state (319 downto 192)
XOR key (127 downto 0)
```

- Si estamos en proceso de cifrar:

a) Primer ciclo:

```
data_out <= reg_state (31 downto 0)
```

b) Segundo ciclo:

```
data_out <= reg_state (63 downto 32)
```

c) Tercer ciclo:

```
data_out <= reg_state (95 downto 64)
```

d) Cuarto ciclo:

```
data_out <= reg_state (127 downto 96)
```

- Si estamos en proceso de descifrar, antes de generar el TAG se comprueba que por el bus de entrada el Tag coincide con el generado.

a) Primer ciclo:

```
if data_in = reg_state (31 downto 0) then data_out
<= reg_state (31 downto 0); else generar error
```

b) Segundo ciclo:

```
if data_in = reg_state (63 downto 32) then data_out
<= reg_state (63 downto 32); else generar error
```

c) Tercer ciclo:

```
if data_in = reg_state (95 downto 64) then data_out
<= reg_state (95 downto 64); else generar error
```

d) Cuarto ciclo:

```
if data_in = reg_state (127 downto 96) then data_out
<= reg_state (127 downto 96); else generar error
```

3.2.2 Funcionamiento detallado de cada estado del cifrador

En este apartado, se ofrece una explicación exhaustiva de cada estado del algoritmo de cifrado, con un enfoque en las operaciones específicas y las transiciones que ocurren a lo largo del proceso. Se examina cómo el sistema gestiona las señales y los datos en cada etapa, abarcando desde la espera inicial hasta la generación y verificación del TAG.

Waiting

Este es el estado inicial, donde se determina la acción que va a ocurrir. Al reiniciar algunas señales a '0', se prepara para cualquier operación nueva. Dos eventos principales pueden sacar al sistema de este estado: la señal Key_Update activada a '1', que mueve al sistema al estado load_Key para la carga de una nueva clave, indicando así la necesidad de una actualización de clave; o la señal init activada a '1', sin requerir actualización de clave, lo que lleva al sistema al estado Absorb_*Nonce* para comenzar la absorción del *Nonce*, marcando el comienzo del proceso de cifrado o descifrado según lo determinado por la señal descryptIn, la cual se activa a la vez que init.

Load_Key

En este estado se carga la clave en el cifrador. Primero verifica el contador para determinar si aún hay partes de la clave (Key) que cargar con los datos entrantes (data_in). Concatena data_in con la clave existente, decrementando el contador después de cada operación. Cuando el contador llega a 0, indicando que la clave está completamente cargada (4 ciclos de reloj), desactiva dReady y, si init es '1', reinicia el contador a "100" y cambia el estado a Absorb_*Nonce* para comenzar la absorción del *Nonce*, ajustando también el modo de operación (cifrar o descifrar) según descryptIn.

Absorb_*nonce*

En este estado, el sistema se prepara para absorber el *Nonce*, activando la señal *dReady* para indicar que está listo para recibir datos. Si *d_valid* es '1', indicando que hay datos válidos para procesar, el sistema verifica si el contador aún es mayor a 0 para incorporar estos datos en el segmento específico de *reg_state*. La lógica decrementa el contador después de cada absorción de datos. Este proceso lo hace durante 4 ciclos de reloj que son los 128 bits del *Nonce*.

Cuando el contador llega a "001", equivalente al último bloque, el sistema evalúa las señales *dEob* (end of block) y *dEop* (end of packet) para determinar el flujo siguiente del proceso:

- Si *dEob* es '0' y *dEop* es '0', el *estado_siguiente* se establece en *absorb_Ad*, preparándose para absorber datos asociados.
- Si *dEob* es '0' y *dEop* es '1', el *estado_siguiente* se establece en Domain_*Ad*, con *estado_anterior* configurado a *Absorb_pT* para indicar de dónde viene la transición.
- Si *dEob* es '1', el *estado_anterior* se establece igualmente en *Domain_Ad*, pero con *estado_anterior* ajustado a *Add_Key_1,* reflejando un tratamiento diferente del bloque de datos.

El contador se reinicia a "010" para un nuevo ciclo de procesamiento, la señal *dReady* se desactiva, y se inicia una preparación de estado con *estado* cambiando a *Ini*.. Una vez absorbido el *Nonce* ya podremos a proceder a la incialización del cifrador. El *reg_state* será nuestro registro principal, que se usará para las permutaciones, incialización, finalización y posibles operaciones de procesado. Habiendo absorbido previamente la clave, *reg_state* se configurará de la siguiente forma:

- reg_state(191 downto 64) almacenará la clave (key).
- reg_state(63 downto 0) contendrá el IV.
- reg_state(319 downto 192) se asignará al Nonce.

INI

Se realizan las permutaciones necesarias utilizando la instancia Inst_ASCONp del módulo ASCONp, que afecta a reg_state según el contador de rondas, en este caso será 12 rondas. Cada permutación mezcla los datos de forma segura, con el resultado actualizándose en reg_state.

Esta instancia se configura con una asociación de puertos que incluye:

- *state_in*: La entrada del estado actual *reg_state*, que se somete a la permutación.
- *rcon*: Un contador de *rondas* utilizado para controlar el proceso de permutación.
- *state_out*: La salida del estado permutado *reg_permu_out*, que refleja el resultado de aplicar la permutación al estado de entrada.

Después se realiza una operación XOR entre una sección de reg_state y la clave key, como un paso final en el proceso de inicialización. Luego, se reinicia el

contador a "010", preparando el sistema para el próximo estado, y se transita al estado_siguiente determinado previamente.

Absorb_Ad

En este estado se absorbe los datos asociados (AD) y los prepara para permutaciones subsiguientes. Se activa la señal *dReady* para indicar que el sistema está listo para recibir datos. Si *d_valid* es 1, mostrando que hay datos válidos para procesar, y el contador es mayor a 0, se forma un bloque de 64 bits concatenando el dato entrante con una parte de *data_in_64*. Luego, se decrementa el contador.

Al llegar el contador a “001”, se evalúan las señales *dEob* y *dEop* para determinar el próximo paso:

Actualización de *estado_anterior:*

- Si *dEob* y *dEop* son 1, indicando el último bloque de datos, se ajusta *estado_anterior* a *Add_Key_1*.
- Si *dEob* es 1 y *dEop* es 0, se configura *estado_anterior* a *Absorb_pT*.

Actualización de *estado*:

- Si *partial* y *dEob* son 1, indicando un bloque final parcial, el estado cambia a *partial_AD*, se desactiva *dReady*, y se asigna a *partialBits_signal* el valor de *partialBits* que nos dirá cuál es el tamaño del bloque parcial.
- Si *partial* es 0 y *dEob* es 1, mostrando un bloque completo y siendo el último, se establece *partialBits_signal* a 000 y se cambia al estado *partial_AD*, desactivando también *dReady*.

En otros casos, el proceso continúa en *Absorb_Ad_2* para más tratamiento de datos antes de las permutaciones.

Absorb_Ad_2

Al no ser el último bloque de datos asociados se finaliza la absorción del bloque y se prepara para el procesamiento de éstos mediante la desactivación de la señal *dReady*. Se establece el número de rondas a *Rounds_B*, (6 rondas). Luego, se realiza una operación XOR entre los últimos 64 bits de *reg_state* y *data_in_64*, integrando así los datos recibidos en el estado del registro. Finalmente, se actualiza *estado* a *Process_Ad*.

Partial_AD

Este estado se ocupa del procesamiento del bloque de datos asociados (AD) que es parcial o completo, pero necesita ser rellenado antes de las

permutaciones. Dependiendo del valor de *partialBits_signal*, que indica cuánto del bloque está lleno, se realiza una operación XOR específica para mezclar los datos entrantes con el estado actual de *reg_state*, y se ajusta el relleno necesario según el caso:

- "000": Se aplica directamente un XOR entre *reg_state(63 downto 0)* y *data_in_64*, sin necesidad de relleno adicional y se guardará en *reg_state*. Luego, se actualiza el estado a *Pad_Ad*.
- "001" a "111": En estos casos, se ajusta el relleno según la cantidad de bits efectivamente usados en *data_in_64*, desde"001" en adelante, se añade un relleno progresivo: "001" conlleva añadir un prefijo de 8 bits (x"80") seguido del dato de entrada reducido en 8 bits, "010" implica añadir un prefijo de 8 ceros y x"80" seguido del dato de entrada menos 16 bits, y así sucesivamente. Esta estrategia asegura que cada bloque parcial se procese de manera consistente, extendiendo el dato con un bit de parada (x"80") y ceros necesarios para completar el bloque antes de aplicar la operación XOR, garantizando la integridad y la correcta alineación de los datos en el *reg_state*.

Un ejemplo para el caso “011” (24 bits de relleno):

```
reg (63 a 0) <= reg (63 a 0) xor (x"0000" & "10000000" &
data_in_64(39 a 0))
```

En todos los casos, se establece el número de rondas a *Rounds_B* (6 rondas), y se cambia el estado a *Process_Pad* o *Pad_Ad*, dependiendo de si el bloque es parcial o completo.

Process_Ad

En este estado, después de realizar las permutaciones necesarias con *Inst_ASCONp* para el procesamiento de un bloque de datos asociados, se vuelve a actualizar el estado a *Absorb_Ad* para comenzar a procesar el siguiente bloque de datos asociados, si lo hubiere. Activa la señal *dReady* a '1', indicando que el sistema está listo para recibir más datos asociados. También reinicia el contador a "010", preparando el sistema para procesar el nuevo bloque de datos asociados.

Pad_Ad

En este estado, después de realizar las permutaciones necesarias con *Inst_ASCONp* para el procesamiento de relleno del dato asociado, se vuelve a actualizar el estado a *Process_Ad,* Además, realiza una operación XOR entre los 8 bits más bajos de *reg_state* y x"80", introduciendo así un bit de relleno específico al final del estado del registro. El contador de rondas se reinicia a

Rounds_B (6 rondas), preparando el sistema para la próxima fase de permutaciones.

Process_Pad

En este estado, después de volver a realizar las permutaciones necesarias con *Inst_ASCONp* para el procesamiento de relleno del dato asociado sobre el registro que ahora incluye tanto los datos asociados (AD) como el relleno específico, se actualiza el estado a *Domain_Ad*.

Domain_Ad

En este estado se finaliza el procesamiento del paquete de datos asociados (AD) aplicando una operación XOR en el bit 312 de *reg_state* con '1', marcando así el bloque con información de dominio específica para diferenciarlo claramente. Tras esta marcación, reinicia el contador a "010" y decide el próximo estado basándose en *estado_anterior*: si era *Absorb_Pt*, el estado pasa a *absorb_pt* para continuar el procesamiento de la parte del texto plano/cifrado; de lo contrario, si no hay texto plano o cifrado avanza hacia *Pad_Pt*, preparándose para realizar un padding adicional antes de proceder a la generación del tag de autenticación.

Absorb_Pt

Se prepara para recibir el texto plano o cifrado, activando la señal *dReady* para indicar que está listo para procesar los datos entrantes. Cuando detecta datos válidos *d_valid* = '1', comienza a absorberlos, concatenando el dato entrante *data_in* con el contenido previo de *data_in_64* para formar bloques de 64 bits, y decrementa el contador que lleva la cuenta de cuántos bloques o segmentos de datos faltan por procesar.

Al llegar el contador a “001”, se evalúan las señales *dEob* y *dEop* para determinar el próximo paso:

- Si es el último bloque y no es el fin del paquete *dEob* = '1' y *dEop* = '0', se prepara para pasar al estado *ADD_key_1*, indicando que se necesita añadir la clave nuevamente para continuar con el procesamiento del cifrado o descifrado.
- Si el último bloque es parcial *partial* = '1', el sistema se dirige al estado *partial_Pt* para manejar este último segmento de datos parciales, asignando a *partialBits_signal* el valor actual de *partialBits* para procesar adecuadamente la cantidad de datos válidos restantes.
- En caso de que el último bloque esté completo (*partial* = '0'), el sistema también transita a *partial_Pt*, pero se asume que no hay datos válidos

restantes *partialBits_signal* = "000", preparándose para aplicar un relleno estándar antes de continuar.

- Si no se cumplen las condiciones anteriores, es decir, aún hay más datos por procesar, el sistema pasa al estado *Absorb_Pt_2* para seguir absorbiendo y procesando el texto.

Absorb_Pt_2

Al no ser el último bloque de datos se finaliza la absorción del bloque y se prepara para el procesamiento de éstos mediante la desactivación de la señal *dReady*. Se establece el número de rondas a *Rounds_B*, (6 rondas), preparando el sistema para una serie de permutaciones que se aplicarán antes de emitir el texto procesado. El sistema transita al estado *Emit_P*t, donde se manejarán los datos procesados para su salida, y se actualiza *estado_anterior* a *Absorb_Pt*.

Antes de proceder al estado *Emit_Pt* se realizan una serie de operaciones según sea, cifrar o descifrar:

- Si el modo es cifrar, realiza una operación XOR entre el segmento de 64 bits menos significativo de *reg_state* y *data_in_64*, actualizando tanto *reg_state* como *data_out_64* con el resultado. Esto mezcla los datos de entrada con el estado actual del sistema antes de su emisión, cifrando efectivamente el texto plano.
- Si el modo es descifrar, asigna directamente *data_in_64* al segmento de 64 bits menos significativo de *reg_state* y luego establece *data_out_64* como el resultado de una operación XOR entre *reg_state* y *data_in_64*. Este paso revierte el cifrado, recuperando el texto plano original a partir del texto cifrado.

Partial_Pt

Este estado se ocupa del procesamiento del bloque de datos que es parcial o completo, pero necesita ser rellenado antes de las permutaciones. Dependiendo del valor de *partialBits_signal*, que indica cuánto del bloque está lleno, se realiza una operación XOR específica para mezclar los datos entrantes con el estado actual de *reg_state*, y se ajusta el relleno necesario según el caso.

Primero, desactiva la señal *dReady*, indicando que no está listo para recibir más datos. Luego, según si el modo es descifrar o cifrar, el sistema procede de manera diferente:

Descifrar

Dependiendo del valor de *partialBits_signal* se realiza una de las siguientes operaciones:

- "000": Asigna directamente *data_in_64* al segmento de 64 bits más bajo de *reg_state*. Tambien realiza una operación XOR entre el segmento de 64 bits más bajo de *reg_state* y *data_in_64*, actualizando *data_out_64.*

  Tras procesar, el estado cambia a *Emit_Pt* también actualiza *estado_anterior* a *Pad_Pt.*

- "001" a "111": En estos casos, se ajusta el relleno según la cantidad de bits efectivamente usados en *data_in_64*, desde"001" en adelante, se añade un relleno progresivo: "001" conlleva añadir un prefijo de 8 bits (x"80") seguido del dato de entrada reducido en 8 bits, "010" implica añadir un prefijo de 8 ceros y x"80" seguido del dato de entrada menos 16 bits, y así sucesivamente.

Un ejemplo para el caso “011” (24 bits de relleno):

```
reg (63 a 0) <= reg (63 a 0) xor (x"0000" & "10000000" &
                 data_in_64(39 a 0))
```

Se realiza la operación XOR entre *reg_state* y data_in_64, ajustando *data_out_64* para contener solo la porción válida de los datos descifrados.

Un ejemplo para el caso “100” (32 bits de relleno):

```
data_out_64(63 -8*4 a 0) <=reg_state (63-8*4 a 0) XOR
                 data_in_64(63-8*4 a 0)
```

Tras procesar, el estado cambia a *Emit_Pt* para emitir el texto descifrado. También actualiza estado_anterior a *Partial_Pt.*

Cifrar

Dependiendo del valor de *partialBits_signal* se realiza una de las siguientes operaciones:

- "000": Realiza una operación XOR entre el segmento de 64 bits más bajo de *reg_state* y *data_in_64*, actualizando *data_out_64 y reg_state.*
- Tras procesar, el estado cambia a *Emit_Pt* también actualiza *estado_anterior* a *Pad_Pt.*
- "001" a "111": En estos casos, se ajusta el relleno según la cantidad de bits efectivamente usados en *data_in_64*, desde"001" en adelante, se añade un relleno progresivo: "001" conlleva añadir un prefijo de 8 bits (x"80") seguido del dato de entrada reducido en 8 bits, "010" implica añadir un prefijo de 8 ceros y x"80" seguido del dato de entrada menos 16 bits, y así sucesivamente.

Un ejemplo para el caso “011” (24 bits de relleno):

reg (63 a 0) = reg (63 a 0) xor (x"0000" & "10000000" & data_in_64(39 a 0))

Se realiza la operación XOR entre *reg_state* y data_in_64, ajustando *data_out_64* para contener solo la porción válida de los datos descifrados.

Un ejemplo para el caso “011” (24 bits de relleno):

*data_out_64(63 -8*3 a 0) =reg_state (63-8*3 a 0) XOR data_in_64(63-8*3 a 0)*

Tras procesar, el estado cambia a *Emit_Pt* para emitir el texto descifrado. También actualiza estado_anterior a *Partial_Pt.*

Emit_Pt

En este estado se emite el texto procesado, ya sea cifrado o descifrado. Se activa la señal *outValid* para indicar que los datos de salida son válidos y están listos para ser transmitidos o almacenados. Aquí, se utiliza una lógica de alternancia (*toggle*) para determinar qué parte del bloque de 64 bits (*data_out_64*) se envía en cada ciclo:

- Si *toggle* está en '0', se emiten los 32 bits inferiores de *data_out_64* y luego *toggle* se cambia a '1', preparándose para emitir la siguiente mitad del bloque en el próximo ciclo.
- Si *toggle* está en '1', se emiten los 32 bits superiores de *data_out_64* y *toggle* se reinicia a '0', completando la emisión del bloque de 64 bits actual.

Después de emitir la parte correspondiente de *data_out_64*, el sistema decide el próximo estado basándose en cuál fue el *estado anterior*:

- Si viene del estado *Absorb_Pt*, se transita a *Process_Pt* para continuar procesando más texto plano o cifrado.
- Si el estado anterior fue *Pad_Pt*, indica que se estaba procesando *p*adding, por lo que el sistema regresa a *Pad_Pt*
- Si el estado anterior fue *Partial_Pt*, significa que se ha completado el procesamiento de un bloque parcial, y el sistema avanza a *Add_Key_1* para añadir la clave de cifrado y seguir con el proceso de finalización.

Process_Pt

En este estado, después de realizar las permutaciones necesarias con *Inst_ASCONp* para las operaciones del procesamiento del texto plano/cifrado, se desactiva la señal *outValid*, se vuelve a actualizar el estado a *Absorb_Pt* para retomar la fase de absorción de más texto plano o cifrado, señala que está listo para recibir más datos activando la señal *dReady* a “1”, y reinicia el contador a "010", preparándose así para el procesamiento del siguiente bloque de datos.

Pad_Pt

En este estado, después de realizar las permutaciones necesarias con *Inst_ASCONp* para las operaciones del último bloque del texto plano/cifrado, antes de proceder a la fase de finalización, se desactiva la señal *outValid*, se vuelve a actualizar el estado a *Add_Key_1* y se aplica un último paso de padding específico al bloque actual mediante una operación XOR entre los 8 bits menos significativos de *reg_state* y el valor hexadecimal x"80".

Add_Key_1

Añadimos la clave de cifrado al estado del registro para reforzar la seguridad del proceso de cifrado. La señal *outValid* se desactiva, indicando que no hay datos válidos de salida

Se realiza una operación XOR entre la porción del estado del registro que va de los bits 191 a 64 (*reg_state(191 downto 64)*) y la clave de cifrado (*key*).

El contador de rondas se establece en *Rounds_A*, (12 rondas). Esto prepara el sistema para una serie de permutaciones adicionales que se realizarán en el estado siguiente.

Se cambia el estado a *Process_Tag,* donde se llevará a cabo el procesamiento final del tag de autenticación o la finalización del cifrado.

Process_Tag

En este estado, después de realizar las permutaciones necesarias con *Inst_ASCONp* para las operaciones de permutaciones del procesamiento del tag.Se ejecuta una operación XOR entre la sección específica de *reg_state* (bits 319 a 192) y la clave de cifrado, actualizando los 128 bits más significativos de *reg_state*. Luego, dependiendo de si el sistema está cifrando o descifrando, transita al estado *Emit_Tag* para emitir el tag de autenticación o al estado *Verify_Tag* para comparar el tag recibido con el calculado, terminando así el proceso de cifrado o validación de la integridad del mensaje.

Verify_Tag

Se verifica la autenticidad del Tag para determinar si el mensaje cifrado recibido es auténtico. Primero, se activan las señales *dReady* y *authValid*, indicando que el sistema está listo para recibir datos y presuponiendo la validez de la autenticación. La verificación del tag se realiza comparando los segmentos de 32 bits del tag recibido (*data_in*) con los correspondientes en *reg_state*, siguiendo una secuencia determinada por la variable *toggle_Tag*.

El proceso de verificación se ejecuta en cuatro pasos, correspondientes a los valores de *toggle_Tag* de 0 a 3, donde cada paso compara un segmento de 32 bits de *data_in* con el segmento correspondiente en *reg_state*. Si los datos

coinciden, *data_out* se actualiza con el segmento verificado, y *toggle_Tag* avanza al siguiente valor para continuar la verificación. Si en cualquier punto los datos no coinciden, esto indica un fallo en la verificación del tag y el sistema desactiva *authValid y dReady* indicando así que los datos de salida no son válidos y que la autentificación ha fallado. Si por el caso contrario el Tag sigue emitiendo los siguientes 4 ciclos de reloj, la autentificación ha sido satisfactoria.

En cualquiera de los dos casos se reinicia *toggle_Tag* a 0 y cambia el estado a *waiting*, preparándose para recibir un nuevo mensaje. También desactiva *dReady*, señalando que no está listo para recibir más datos inmediatamente.

Emit_Tag

El sistema emite secuencialmente los 128 bits del tag de autenticación en segmentos de 32 bits, controlados por el valor de toggle_Tag. Comienza con los 32 bits menos significativos y, en cada paso, incrementa toggle_Tag para avanzar al siguiente segmento: primero los bits 31 a 0, luego los bits 63 a 32, seguidos de los bits 95 a 64, y finalmente los bits 127 a 96. Tras emitir el último segmento, toggle_Tag se reinicia y el sistema transita al estado waiting, marcando la finalización de la emisión del tag.

3.3 Comparativa del algoritmo *ASCON-Hash* con *ASCON*

ASCON-Hash sigue el mismo modelo de esponja que el cifrador *ASCON*, pero con una distinción importante: no requiere datos de entrada como clave, *nonce* o datos asociados. En lugar de eso, *ASCON-Hash* procesa secuencias de datos de tamaño indefinido y produce un resultado de tamaño fijo de 256 bits. Al igual que en otros aspectos de *ASCON*, los bloques de datos se manejan en bloques de 64 bits, y las permutaciones aplicadas son de 12 rondas. Esta unificación de características, especialmente el tamaño de los bloques facilita la integración de *ASCON-Hash* con la encriptación autenticada de *ASCON-128*, optimizando así su eficiencia.

3.3.1 Funcionamiento detallado de cada estado del cifrador *ASCON-Hash*

Inicialización

El estado inicial de 320 bits para *ASCON-Hash* se define mediante una constante IV (Valor Inicial), la cual establece los parámetros del algoritmo en un formato similar, pero de valor diferente utilizado para *ASCON-128*. Para el IV del ASCON-Hash el valor de k es 0, la tasa ***r*** que es de 64 bits, y los números de rondas de permutaciones a de 12 rondas y b que no se usa. Cada uno expresado en un entero de 8 bits. A esto le sigue la longitud máxima de salida de h bits como un entero de 32 bits (con h = 256 para *ASCON-Hash* y h = 0 para una salida ilimitada

en el caso de *ASCON*-Xof) y 256 bits a cero. Para inicializar el registro s se le aplica unas permutaciones de rondas a:

$IV_{k,r,a,b} = 0^8 \,||\, r \,||\, a \,||\, 0^8 \,||\, h =$ → 0x00400c0000000000 para *ASCON*-xof
→ 0x00400c0000000100 para *ASCON-Hash*

Donde $S \leftarrow P^{\,a} (IV_{h,r,a} \,||\, 0^{256})$

El proceso de relleno es idéntico al utilizado para el texto plano de *ASCON-128*: se añade un 1 seguido del menor número de 0s necesario para que la longitud del mensaje rellenado sea un múltiplo de r bits. Cada bloque M_i se combina mediante una operación XOR con los primeros r bits S_r del estado S seguido por la aplicación de la permutación de a rondas $P^{\,a}$ a S.

$$M_1,\ldots,M_s \leftarrow_r \text{bit blocks of } M \,||\, 1 \,||\, 0^{r-1-(|M| \bmod r)}$$

Este proceso se repite bloque por bloque, garantizando que el mensaje completo sea absorbido eficazmente en el estado interno, preparando el sistema para la siguiente fase del proceso de *Hash* o generación de salida extensible.

$$S \leftarrow p^b ((S_r \oplus M_i) \,||\, S_c) \quad si \quad 1 \leq i < s$$

Extracción

En la etapa de extracción, el resultado del *Hash* se obtiene del estado en bloques de r bits, denominados H_i hasta completar la longitud de salida solicitada $L \leq H$ después de $t=[L/r]$ bloques. Tras cada extracción, el estado interno S se transforma mediante la permutación de a rondas $P^{\,a}$.

$$Hi \leftarrow S_r$$
$$S \leftarrow p^b (S) \; si \; 1 \leq i < t=[L/r]$$

El último bloque de texto cifrado H_t se trunca a L mod r bits y $H = H_1 \,||\, \ldots \,||\, \tilde{H}_t$ devolviendo:

$$H_t \leftarrow |H_t|_{|L| \bmod r}$$

En el diseño Hardware del algoritmo *ASCON-Hash* se ha hecho igual que en el cifrador, desarrollando una máquina de estados para facilitar así su compresión y sencillez.

3.3.2 Estados y Transiciones en el Algoritmo *ASCON-Hash*

En el diseño Hardware del algoritmo *ASCON-Hash* se ha hecho igual que el cifrador siguiendo una máquina de estados para facilitar así su compresión y sencillez.

En el diagrama mostrado a continuación en la Figura 11 proporciona una visión clara y detallada del funcionamiento del algoritmo de *ASCON-Hash* pudiéndose ver las diferencias con el cifrador.

A continuación, se presenta una descripción general de las principales etapas representadas en el diagrama:

Inicio y Aplicación de Permutaciones:

- El proceso se inicia en el estado "Waiting", donde espera una señal de *Hash* (*Hash*_signal = 1) y la señal de inicialización (Init = 1).
- Una vez recibidas estas señales, se aplican las permutaciones iniciales con el vector de inicialización correspondiente en el estado "Ini_*Hash*".

Absorción del *Hash*:

- El algoritmo procede a recibir bloques del *Hash* dentro del estado "Absorb_*Hash*", a través de la entrada data_in.
- Dependiendo de si el bloque recibido es el último (DEOB = 1) y si es un bloque parcial (partial), el flujo puede seguir diferentes caminos:
  a) Si partial = 1 y DEOB = 1, o si partial = 0 y DEOB = 1, el proceso sigue al estado "Partial_*Hash*", donde se procesa el último bloque del *Hash*.
  b) Si partial = 0 y DEOB = 0, el flujo continúa en "Absorb_*Hash*_2", donde se termina de recibir el bloque y se realizan operaciones adicionales.

Procesamiento del *Hash*:

1. En "Absorb_*Hash*_2", se aplica una serie de permutaciones al bloque recibido en el estado "Process_*Hash*".
2. Después de procesar el bloque, el flujo vuelve al estado "Absorb_*Hash*" para continuar con los siguientes bloques.

Relleno y Extracción del *Hash*:

- Si el bloque es parcial (Partial = 0), se aplica un relleno y permutaciones en el estado "Pad_*Hash*" antes de proceder a la extracción.
- En ambos casos (bloque parcial o no), se aplican permutaciones adicionales antes de emitir el *Hash* en el estado "Extract_*Hash*".

Emisión del *Hash*:

- Finalmente, el *Hash* se emite en bloques de 64 bits a través del bus data_out (de 32 bits) en el estado "Emit_*Hash*".
- Tras emitir el *Hash*, el proceso retorna al estado inicial "Waiting".

El diagrama de flujo mostrado en la Figura 11 ilustra de manera visual el funcionamiento detallado del algoritmo Hash.

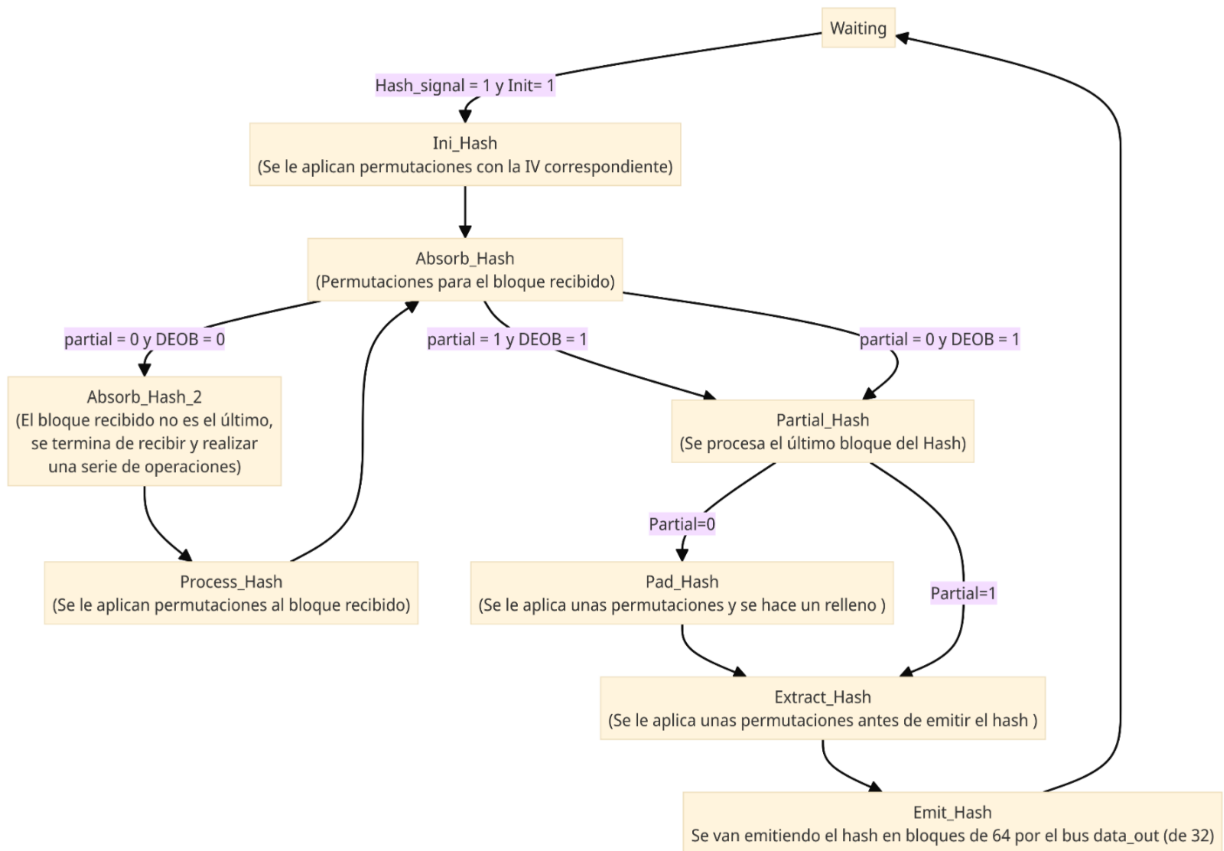

Figura 11. Diagrama de flujo Hash

CAPITULO 4: Test Bench y análisis de las implementaciones

En este último capítulo se desarrolla y evalúa los test de prueba para el cifrador ASCON-128 y ASCON-Hash desarrollados, así como el análisis detallado de sus diversas implementaciones en sistemas SoC-FPGA.

Se destacará cómo las implementaciones específicas de hardware pueden ofrecer ventajas significativas sobre las soluciones de software convencionales en términos de velocidad y eficacia.

A través del test bench diseñado, se ha facilitado un control detallado sobre cada ciclo de operación del cifrador, permitiendo un análisis profundo y una evaluación de cada componente y su funcionamiento bajo diversas condiciones. Este enfoque ha incrementado la eficiencia del proceso de prueba y ha permitido identificar y rectificar errores de forma eficaz, asegurando así un diseño robusto y optimizado.

El análisis en este capítulo no solo abarca los aspectos técnicos y prácticos del test bench, sino también los resultados de rendimiento obtenidos a partir de las distintas configuraciones del cifrador, proporcionando una visión sobre la escalabilidad y adaptabilidad de las implementaciones en un entorno real de FPGA.

4.1 Test Bench inicial ASCON-128

Para desarrollar el test bench del cifrador ASCON-128, se optó inicialmente por escribir manualmente los valores en cada ciclo de reloj. Este método permitió un control detallado de la secuencia de prueba, asegurando que cada fase del proceso de cifrado se gestionara con precisión. Sin embargo, este enfoque también presentó diversos desafíos, que se analizarán a continuación.

4.1.1 Estructura y secuencia

El proceso de prueba del ASCON-128 se divide en varias fases, cada una con un conjunto específico de acciones y señales:

1. **Cargar la clave (Key)**

- Se activa la señal `Key_Update` en '1' durante un ciclo de reloj para iniciar la carga de la clave.
- La clave se envía en 4 bloques de 64 bits, comenzando por el último bloque.

2. **Inicialización**

- Tras un ciclo de espera, se activan las señales `init` y `descryptIn` en '1' para iniciar el proceso de descifrado, restableciéndose en '0' en el siguiente ciclo.

3. **Cargar el *Nonce***

- Se espera a que `dReady` esté en '1' para indicar que el sistema está listo para recibir el nonce.
- Se activa `d_valid` y se cargan los 4 bloques de 64 bits del Nonce, desactivando `d_valid` después del último bloque.

4. **Cargar datos asociados (AD)**

- Similar al nonce, se espera a que `dReady` esté en '1'.
- Se activa `d_valid`, se cargan los datos asociados y se marca el final del bloque con `dEob` en '1', luego `d_valid` se desactiva.

5. **Cargar el texto cifrado (CT) o texto plano (PT)**

- Se cargan los distintos bloques de datos a cifrar o descifrar, asegurándose que `dReady` esté en '1', activando `d_valid` y desactivándolo tras marcar el final del bloque con `dEob`.

6. **Verificar TAG o Emit Tag:**

- Cuando `authValid` está en '1', se activa `d_valid` para cargar la etiqueta de autenticación (TAG), desactivándolo después del último bloque.

4.1.2 Desafíos del enfoque inicial

Al implementar este enfoque manual, se encontraron varios problemas:

1. **Tediosidad y margen de error**
    - Especificar manualmente cada señal en el momento exacto era tedioso y propenso a errores, lo que incrementaba el tiempo de prueba y la posibilidad de fallos.
2. **Visibilidad y depuración**
    - La depuración se dificultaba debido a la gestión manual de cada aspecto del test, haciendo complicada la identificación de problemas específicos y prolongando las sesiones de depuración sin garantías de una solución rápida.

3. **Rigidez y escalabilidad**

- La rigidez del método limita la capacidad de adaptar o escalar el test bench, ya que cada ajuste requería una revisión completa de los ciclos de reloj y señales involucradas.

4. **Comunicación Limitada**

- Explicar la configuración específica de las señales a colaboradores, como el tutor, resultaba complejo y propenso a malentendidos o errores de interpretación.

5. **Falta de Automatización**

- La falta de automatización impedía realizar pruebas exhaustivas de manera eficiente, limitando la profundidad y amplitud de las pruebas.

Tras completar el diseño del cifrador, se elaboró un test bench considerablemente más automatizado, recurriendo a un conjunto de funciones y procedimientos diseñados para optimizar el flujo de trabajo.

4.2 Test Bench inicial ASCON-Hash

Tras detallar el funcionamiento del test bench general para el cifrador ASCON-128, a continuación, se describe una variante específica de este test bench, realizada para probar el ASCON_Hash. Esta versión se centra exclusivamente en la generación de hash a partir de los datos de entrada, siguiendo un proceso que, aunque similar en estructura al descrito previamente, posee características únicas adaptadas para el manejo de hashes. Este test bench simplifica algunas de las operaciones mientras se enfoca en la eficiencia y precisión del cálculo de hashes.

Descripción del Test Bench ASCON_Hash:

El funcionamiento del test bench para ASCON_Hash es esencialmente idéntico al del cifrador general, pero con un enfoque exclusivo en la manipulación y procesamiento de hashes. El proceso se inicia con la activación de la señal Hash_Signal y la señal Init, que preparan el sistema para recibir datos y comenzar el proceso de hashing. A continuación, los datos son introducidos secuencialmente en bloques específicos, con cada bloque procesado durante un ciclo de reloj determinado.

Inicio del Proceso:

Hash_Signal y Init se establecen en '1' para indicar el comienzo del hashing.

Se espera un ciclo de reloj para que las configuraciones tomen efecto.

Carga de Datos:

Los datos se cargan en el sistema mediante la señal data_in, mientras que d_valid se activa para indicar que los datos son válidos.

Cada bloque de datos es procesado en su respectivo ciclo de reloj, seguido por la desactivación temporal de d_valid para preparar el siguiente bloque.

Finalización y Generación de Hash:

Una vez todos los datos han sido introducidos, se manejan las señales de fin de operación (dEob y partial) para asegurar que el sistema procese correctamente el último bloque.

El proceso concluye con una espera final durante el cual el hash es calculado y almacenado.

Este test bench específico para ASCON_Hash permite realizar pruebas de hashing de manera efectiva, asegurando que el algoritmo funcione correctamente bajo diversas condiciones de entrada y generando el TAG correspondiente solo cuando se procesa la totalidad de los datos.

4.3 Test Bench final ASCON-128

Este enfoque se basó en la utilización de fichero de texto, que contenía un compendio de todas las pruebas a realizar, así como sus respectivos resultados. De manera meticulosa, se examinaba cada dato del cifrador (clave, *Nonce*, datos asociados, texto cifrado/plano y etiqueta de autenticación), extrayendo y almacenando la información relevante en variables. Este proceso permite manipular los datos de forma automatizada, facilitando la evaluación de aspectos cruciales como los tamaños de los datos y la cantidad de bloques enviados y recibidos.

Una vez almacenadas todas las variables y usando los procedimientos, el sistema podía ejecutar de manera autónoma el cifrado y descifrado para cada conjunto de prueba, verificando los resultados automáticamente. Este avance cambió completamente la forma en que se realizaban las pruebas, permitiendo no solo hacerlas más rápido sino también con mayor precisión al encontrar y arreglar fallos. Con la automatización, se pudo profundizar más en el análisis, lo que realmente mejoró la eficiencia y efectividad al probar el cifrador.

Para comprender mejor cómo opera este test bench y la lógica detrás de su construcción, aquí se explica paso a paso sus componentes y la función que cumplen dentro del proceso de prueba.

Primero se explicarán algunos componentes iniciales. Estos incluyen archivos para el manejo de estímulos y resultados, señales específicas para representar

diferentes tipos de datos, y una variedad de funciones y procedimientos diseñados para automatizar y facilitar el proceso de pruebas.

4.3.1 Estructura

Archivos de estímulo y resultados

Se emplean dos archivos principales: datos.txt y resultados.txt. El archivo datos.txt se utiliza para leer los datos de prueba, mientras que resultados.txt sirve para registrar los resultados obtenidos de las pruebas.

El archivo datos.txt se estructura siguiendo un formato específico que incluye los vectores de prueba, que son esenciales para verificar la correcta funcionalidad del cifrador. Estos vectores de prueba contienen una variedad de datos de entrada, como claves, nonces, mensajes de texto plano y los correspondientes textos cifrados y etiquetas de autenticación (TAG) valores de Hash y etiquetas de autenticación del Hash mostrados en la figura 12. Estos vectores son cruciales para evaluar tanto la seguridad como la eficiencia del cifrador bajo diferentes condiciones y tipos de datos.

El contenido específico de estos vectores de prueba proviene del archivo test_vectors.txt, disponible en la versión v1 del repositorio de GitHub de *ASCON* [13].

```
#### Msg 400
key      = 14B20392AD5C9CA15BD6FF29197B5718
npub     = 10976FF540FE6C94F65B92425C3B7922
ad       = BC92
pt       = DF1DFBEEBE1A1E4CDE795FC53F84C47BE1B65C235FBB06F0
ct       = 437E3F9F9D0E320F7B98BA487D31F868FC894328C18AC756
tag      = F367EB1581A9BBC955A9EBE28C082BF7

#### Msg 401
hash     = 4C3CDC13A56B5BCE91D2291017C47E15DCA26DAF48BD3FECD470548F2859E9
hash_tag = E781A0AFA17292DA93A6B4F9660FEC333A3F29DC76A8FA359B62E18599A968C7

#### Msg 402
key      = 1F10010F550ECD49D1B4599548607F8B
npub     = 8926797D1B4EF84F314D6B037672B173
ad       = 724CF8E89D419ADF4A87C795B357ED4B1C60F23578E36D62C6
pt       = 09231A494766
ct       = 7CCED59334F4
tag      = 4343587AB0D429043FFC58B2B52A34BA

#### Msg 403
hash     = EF74
hash_tag = E74F981D499F6F6BC3BD6D4471854DA7A46505DE7FAC30124EE6EB9AB71CB7E3

#### Msg 404
hash     =
hash_tag = 7346BC14F036E87AE03D0997913088F5F68411434B3CF8B54FA796A80D251F91

#### Msg 405
key      = 92BCBC484742F09AE10DAE19FF3596C8
npub     = EFFF6691408CA294DE86044F03E5705C
ad       = A39ED9488A6FB768E4FEC1853A068431083204636007BE3C0C8F48DAE90450EC
pt       = 580D9DDFCA6CE60EF60E98BA66EF61EB28FFC23A5382E2
ct       = 807DCD7F5071B912113E9D6EF7AB47FFF851A9EB373D74
tag      = 44C7EFC84CE22F6E264B1E4EC2DEC336
```

Figura 12.Captura parcial del archivo de texto "test_vectors"

Señales

Se definen varias señales, como key,tag,*Nonce*,ad (datos asociados), `pt` (texto plano), y `ct` (texto cifrado), junto con sus tamaños correspondientes para poder guardarlas.

A continuación, se detallan las funciones y procedimientos diseñados específicamente para manipular y procesar estas señales dentro del test bench. Cada función y procedimiento tiene un rol específico en la preparación y envío de los datos.

Funciones definidas

Una función actúa como una herramienta especializada que procesa datos de entrada y siempre retorna un resultado específico, sin modificar señales externas; es ideal para realizar cálculos u operaciones repetitivas donde se requiere un resultado directo y claro.

- Función reverse_byte:

Esta función invierte los bytes de un vector lógico, asegurando que los datos se procesen en el formato correcto (little/big endian), es necesario para la coherencia entre los datos de prueba y cómo los interpreta el cifrador.

- Función hex_to_slv:

Convierte una cadena hexadecimal a un vector lógico estándar (***std_logic_vector***). Se necesita traducir los datos de prueba escritos en formato hexadecimal a un formato que el test bench y el diseño del cifrador puedan manejar directamente.

Procedimientos definidos

Un procedimiento es un conjunto de instrucciones que ejecuta acciones más complejas dentro del diseño, pudiendo alterar tanto señales internas como externas y no necesariamente retorna un valor; se emplea para ejecutar tareas como configurar registros o manejar secuencias de control, agrupando varias operaciones que influyen en el estado del circuito. Se han definido los siguientes procedimientos:

- Procedimiento SEND_KEY:

Inicia invirtiendo los bytes de la clave de cifrado mediante la función *reverse_byte*, adaptando el formato de la clave a las necesidades del cifrador. Divide la clave en 4 bloques de 32 bits y los envía secuencialmente en cada ciclo de reloj, imitando cómo se transmitirían en un escenario real.

- Procedimiento SEND_*NONCE:*

Similar al procedimiento de la clave, además de invertir y enviar el *nonce* en 4 bloques de 32 bits, este procedimiento maneja señales adicionales como *dEob* y *dEop*, que indican que se va a procesar a continuación si es dato asociado, o texto plano/ texto cifrado. Dependiendo de si existe o no para ese cifrado.

- Procedimientos SEND_AD, SEND_PT, y SEND_CT:

Estos procedimientos envían los datos asociados, el texto plano, y el texto cifrado, respectivamente. Cada uno toma el respectivo conjunto de datos, lo invierte para alinear con el formato esperado y lo envía en bloques de 32 bits. Durante este proceso, se utilizan señales como *d_valid* para indicar cuándo los datos son válidos y se deben procesar. Además, se emplean señales como *partia*l, *partialBits*, *dEob* y *dEop* para gestionar particularidades como bloques de datos parciales y marcar el final de los datos, asegurando que el cifrador procese correctamente toda la información.

- Procedimiento SEND_TAG:

Envía la etiqueta de autenticación al sistema. Al igual que con los otros componentes, la etiqueta se invierte y se envía en segmentos, completando así el ciclo de pruebas al validar que los mensajes descifrados sean auténticos y no hayan sido alterados.

- Procedimiento Configuración:

Este procedimiento configura cómo se deben tratar los bloques parciales de datos (es decir, los últimos bloques que no llenan completamente un bloque de datos estándar).

4.3.2 Secuencia

El código del test bench para el cifrador *ASCON* se organiza en una serie de pasos secuenciales. A continuación, se detalla cómo se estructura este proceso:

Lectura de documento de texto

El proceso se inicia con la lectura de un archivo de texto (`*datos.txt*`), donde se almacenan los datos de prueba, incluyendo las claves, *nonce*s, datos asociados, textos planos y cifrados, y etiquetas de autenticación. Se leen todos los datos relativos a cifrado o descifrado.

Análisis y recolección de señales

Una vez leído los datos, el siguiente paso es analizar esta información y recoger las señales relevantes. Se extrae cada elemento necesario para el proceso de prueba, como la clave de cifrado, el *Nonce*, y demás, almacenándolos en

variables adecuadas para su posterior uso. Se leen todos los datos relativos a cifrado o descifrado.

Cifrado

Con todas las señales preparadas, el proceso continúa con el cifrado del mensaje. Aquí, se simula cómo el cifrador *ASCON* procesaría los datos, utilizando las claves, *nonce*s, y datos asociados recolectados usando las funciones y procedimientos previamente explicados.

Descifrado

Después del cifrado, se procede a descifrar el mensaje cifrado para verificar si el proceso ha sido exitoso y coherente, comprobando que el mensaje descifrado coincide con el texto plano original usando las funciones y procedimientos previamente explicados.

Preparación para el siguiente mensaje:

Una vez completados los pasos de cifrado y descifrado para un conjunto de datos, el test bench se prepara para procesar el siguiente mensaje, repitiendo este ciclo hasta haber evaluado todos los mensajes en el fichero de texto.

El diseño implementa estos pasos dentro de un proceso llamado *stim_pro*`. Este proceso incluye un bucle principal *L1* que gestiona el flujo de ejecución para cada caso de prueba.

Dentro de este bucle, se utiliza un sub-bucle *L2* para leer y procesar cada línea del archivo de estímulo, extrayendo las señales necesarias y asignándolas a las variables correspondientes.

Dentro del bucle *L2,* se encuentra el bucle *L3*. La función de *L3* es leer y procesar cada carácter de la línea actual obtenida por *L2*. Esto incluye la extracción de los caracteres que conforman los datos de prueba (como la clave, *Nonce*, etc.) y la preparación de estos datos para su posterior conversión a *std_logic_vector*. Si la longitud de los datos extraídos no es un múltiplo de 16 caracteres (que son 64 bits), *L3* también se encarga de aplicar el *padding* necesario. Las señales se procesan y preparan para el cifrado/descifrado mediante una serie de casos (`case`), dependiendo del primer carácter de cada línea leída, que indica el tipo de señal (por ejemplo, clave, *Nonce*, datos asociados, etc.).

Finalmente, el proceso se encarga de cerrar el archivo de resultados una vez completadas todas las pruebas, marcando el fin del test bench. Este enfoque estructurado y secuencial asegura una evaluación detallada y efectiva del cifrador *ASCON*, facilitando la identificación de cualquier anomalía o comportamiento inesperado en su funcionamiento.

Con este test bench se pudo comprobar que el ASCON desarrollado, tanto en su versión de cifrado/descifrado como en su versión hash, funcionaban correctamente.

4.4 Analisis de las implementaciones

Las simulaciones se han realizado utilizando el programa Xilink ISE versión 14.7, y las implementaciones se han realizado en un dispositivo Artix-7 incorporado en la placaNexys 4 DDR. Esta placa ha sido utilizada en el entorno académico y específicamente seleccionado para este trabajo por estar disponible en la Universidad.

Con estos análisis se buscó conocer los recursos consumidos por las implementaciones (en diferentes casos) y también la frecuencia máxima de operación.

4.4.1 Análisis de "Slice Registers" y "Slice *LUTs*"

El código que se ha desarrollado cuenta con una variable llamada “UROL” que permite cambiar el número de permutaciones que se hacen en un ciclo de reloj. Los diferentes valores de esta variable afectan directamente a la complejidad del circuito que realiza las permutaciones y al número de ciclos de reloj que se necesitan para realizar dichas permutaciones, afectando el rendimiento del sistema. En la tabla 14 se muestran las tres configuraciones de la variable UROL que se han utilizado. La configuración v1 realiza las permutaciones de 12 y 6 rondas en 12 y 6 ciclos de reloj, la configuración v3 las realiza en la mitad de los ciclos 6 y 3 ciclos de reloj y por último la configuración v5 las realiza en 3 y 1 ciclos de reloj. Recordamos que el número de permutaciones a realizar varía en función de qué elemento se esté procesando.

Estado	Ciclos en V1	Ciclos en V3	Ciclos en V5
Ini	12	6	3
Process_Ad	6	3	1
Padding_Ad	6	3	1
Process_Pt	6	3	1
Padding_Pt	6	3	1
Process_Tag	12	6	3

Tabla 14. Ciclos de reloj en cada estado con sus diferentes versiones.

Modelo	V1 (12 y 6)		V3 (6 y3)		V5 (3 y 1)	
Utilización lógica	Usado	Utilización	Usado	Utilización	Usado	Utilización
Número de registros de corte	643	0%	637	0%	639	0%
Número de LUTs de corte	2315	3%	3145	5%	4085	6%
Número de pares FF completamente LUT	449	17%	446	13%	447	10%
Número de IOB vinculados	79	37%	79	37%	79	37%
Número de BUFG/BUFGCTRLs	1	3%	1	3%	1	3%

Tabla 15. Recursos del sistema con sus diferentes versiones.

En la tabla 15 se analizan los recursos consumidos por cada una de las configuraciones.

Slice Registers: El número de flip-flops que consumen las tres configuraciones se mantienen casi constantes, con valores muy parecidos en todas ellas. Esto es algo esperado, porque el cambio en el número de permutaciones a realizar en un ciclo sólo afecta a la lógica combinacional y no a la secuencial, por lo tanto, el almacenamiento el almacenamiento de datos y la gestión de estados no experimentan variaciones significativas con los cambios en el número de rondas de permutación, siendo un factor estable en el análisis del rendimiento del cifrador.

Slice *LUTs*: Aquí es donde se observa una diferencia más marcada entre las versiones. Comenzando con la versión v1 que utiliza 2315 *LUTs*, pasando a la v3 con 3145 *LUTs*, y finalmente la versión v5 que aumenta el uso hasta 4085 *LUTs*. Este incremento refleja una mayor complejidad lógica que va en paralelo con la mejora en la eficiencia del procesamiento conforme se reduce el número de ciclos de reloj.

La elección de la "mejor" versión depende de las prioridades entre la minimización de la complejidad lógica y la maximización de la eficiencia. Si se prefiere una implementación más sencilla y con menor uso de recursos, la versión v1 sería la adecuada. Por otro lado, si se busca una mayor rapidez en el procesamiento, aceptando un uso más intensivo de recursos, las versiones v3 y especialmente la v5 se presentan como opciones óptimas. La versión v5, en particular, ofrece el mayor rendimiento en términos de velocidad de cifrado y descifrado, a costa de un mayor consumo de recursos lógicos.

Se ha realizado también una comparación de los recursos consumidos por el ASCON desarrollado en este trabajo, con una versión desarrollada por los creadores del ASCON. Los datos de esta comparación se muestran en la tabla 16. En esta tabla se comparan únicamente versiones que realizan una permutación en un ciclo de reloj (versión v1). Se observan algunas diferencias interesantes en términos de utilización de recursos.

Modelo	**V1**		**Original**	
Utilización de la lógica	Usada	Utilización	Usada	Utilización
Número de registros de segmentos	643	0%	496	0%
Número de *LUTs de corte*	2315	3%	2356	3%
Número de pares FF completamente LUT	449	17%	492	20%
Número de IOB vinculados	79	37%	136	64%
Número de BUFG/BUFGCTRLs	1	3%	1	3%

Tabla 16. Comparación de recursos con original.

La versión original utiliza 496 registros, mientras que la versión creada v1 utiliza 643. Este menor uso en la versión original sugiere una optimización eficiente en términos de almacenamiento temporal y gestión de estados, pero también puede deberse a que algunos datos se tienen que almacenar en registros externos al cifrador. Es importante destacar que alcanzar este nivel de optimización en la utilización de recursos puede aumentar significativamente la complejidad del diseño y la dificultad para realizar cambios o entender el funcionamiento interno del sistema.

Esta optimización, aunque resulta en un sistema más eficiente desde el punto de vista del hardware, también introduce una capa de complejidad adicional en el diseño no sólo del cifrador en sí, sino también de los circuitos de interfaz que se comunican con el cofrador. Durante el proceso de este proyecto, quedó claro que alcanzar un alto nivel de optimización puede hacer que sea más complicado comprender y modificar el cifrador. Por esto se consideró especialmente importante llevar un balance entre la claridad y la optimización del diseño.

Además, trabajar para optimizar los recursos supuso un aprendizaje sobre el equilibrio entre eficiencia y accesibilidad del diseño. Aunque inicialmente el enfoque estaba puesto en mejorar el rendimiento y reducir el uso de recursos,

también se aprendió sobre la importancia de mantener el diseño lo suficientemente simple como para permitir ajustes y comprensión futura.

4.4.2 Frecuencia máxima de operación

La realización de pruebas para averiguar la frecuencia máxima de operación es esencial para evaluar la eficacia del diseño, incidiendo directamente en su capacidad para procesar datos eficientemente. Para estas medidas, se ha adoptado un procedimiento sistemático y escalonado para determinar la frecuencia máxima, variando la frecuencia de la señal de reloj en el test bench. A través de la modificación del período de la señal del reloj, comenzando por un valor alto y disminuyéndolo progresivamente, fue posible comprobar de forma automática la máxima frecuencia de operación.

Se ha realizado una síntesis con restricciones de reloj de 300 MHz para permitir que el programa actúe y alcance su máximo rendimiento, a pesar de que el diseño no podría operar a esa frecuencia. Posteriormente, se ha simulado reduciendo el período de 10 ns hacia abajo, y se han tomado datos tanto estáticos como de simulación.

Para la versión v1, el diseño se simuló inicialmente con un período de reloj de 10 ns, estableciendo un punto de partida conocido. Gradualmente, el período se fue ajustando a la baja, comprobando el comportamiento del diseño para detectar el momento exacto de fallo. Para la versión v1, el diseño funcionó correctamente hasta un período de reloj de 4.9 ns, equivalente a una frecuencia de aproximadamente 204 MHz. En las estadísticas de diseño, se obtuvo un período mínimo de 4.038 ns, lo que corresponde a una frecuencia máxima de 247.647 MHz.

Para la versión v3, el diseño funcionó correctamente hasta un período de reloj de 5.6 ns, equivalente a una frecuencia de aproximadamente 178.571 MHz. En las estadísticas de diseño, se obtuvo un período mínimo de 5.281 ns, lo que corresponde a una frecuencia máxima de 189.358 MHz.

Para la versión v5, el diseño funcionó correctamente hasta un período de reloj de 9.8 ns, equivalente a una frecuencia de aproximadamente 102.041 MHz. En las estadísticas de diseño, se obtuvo un período mínimo de 9.424 ns, lo que corresponde a una frecuencia máxima de 106.112 MHz.

Este análisis se realizó mediante la simulación temporal post-route, donde se redujo gradualmente el período del reloj hasta identificar el punto de fallo. Si el resultado no coincidía con las expectativas, se ajustaba nuevamente la frecuencia del reloj.

A pesar de que la simulación con Xilinx ISE ofrece una estimación precisa de la frecuencia máxima, es fundamental entender que no todas las variables y

condiciones de un entorno de hardware real pueden ser replicadas en simulación. Aspectos como el ruido eléctrico, fluctuaciones térmicas, integridad de señal y limitaciones de los materiales impactan el desempeño efectivo.

Lograr una frecuencia de operación máxima cercana a 204 MHz para la versión v1 es notable, reflejando una optimización significativa. Las versiones v3 y v5 también muestran un buen desempeño, aunque con frecuencias máximas más bajas. No obstante, es crucial evaluar la factibilidad de alcanzar y mantener estas velocidades en aplicaciones reales, considerando las limitaciones de las tecnologías de semiconductores y los procesos de fabricación actuales.

A continuación, se presenta la tabla 17, con los resultados de las pruebas realizadas en diferentes versiones del cifrador:

Versión	Período mínimo estático (ns)	Frecuencia máxima estática (MHz)	Frecuencia máxima simulación (MHz)
Ascon-128 v1	4.038	247.647	204
Ascon-128 v3	5.281	189.358	178.571
Ascon-128 v5	9.424	106.112	102.041

Tabla 17. Resultados de las pruebas de frecuencia máxima.

CONCLUSIONES

Este trabajo de fin de grado ha abordado de manera exhaustiva la implementación y análisis de un cifrador *lightweight AEAD* basado en el algoritmo *ASCON-128* y su correspondiente función de *Hash*, *ASCON-Hash*. A lo largo del proyecto, se ha profundizado en la necesidad y las características de la criptografía ligera, resaltando su relevancia para dispositivos con recursos limitados y aplicaciones que requieren alta eficiencia energética y bajas latencias.

El primer paso en de desarrollo del trabajo fue el análisis del algoritmo ASCON, entendiendo las operaciones que debe realizar y las señales de entrada y salida que posee. En este sentido una tarea importante fue el análisis de las señales de entrada que posee y de cómo hay que introducir los valores de dichas señales, ya que algunas como la clave y el nonce tienen longitud fija, pero tanto el AD como el plaintext o el ciphertext no tienen una longitud fija.

Otra tarea importante fue la transcripción de las expresiones del algoritmo a señales binarias. Las expresiones del algoritmo no siempre son fáciles de interpretar en términos de señales binarias, por lo que la correcta interpretación es fundamental para el correcto desarrollo del diseño.

Y respecto a las señales de interfaz de entrada y salida, para el desarrollo de este diseño se tomó como referencia la interfaz desarrollada en el trabajo de fin de máster de Carlos Fernández. Gracias a este trabajo se estableció un marco funcional, adaptado a las necesidades específicas de este tipo de cifradores.

El diseño del cifrador *ASCON* se tiene dos componentes: el módulo Top, que orquesta la operación del cifrador mediante una máquina de estados finitos, y el módulo *ASCON Permutation*, que ejecuta la permutación específica del algoritmo. En el proceso de diseño se detallaron las señales principales y las intermedias, las cuales facilitan las transiciones de estado, controlar los procesos de permutación y gestionar los datos de entrada y salida a lo largo del ciclo de cifrado.

La implementación del cifrador se llevó a cabo utilizando VHDL y se verificó mediante un test bench detallado. Inicialmente, se utilizó un enfoque manual para escribir los valores en cada ciclo de reloj, lo que resultó en un proceso tedioso y propenso a errores. Posteriormente, se adoptó un enfoque más automatizado, utilizando un conjunto de funciones y procedimientos que optimizaron el flujo de trabajo, permitiendo una evaluación más rápida y precisa del cifrador.

El test bench automatizado permitió realizar pruebas exhaustivas y sistemáticas, desde la carga de la clave y el nonce, hasta la verificación del texto cifrado y la

etiqueta de autenticación. Este enfoque no solo mejoró la eficiencia del proceso de prueba, sino que también facilitó la identificación de errores y la validación del funcionamiento correcto del cifrador.

Se realizaron simulaciones utilizando el programa Xilinx ISE versión 14.7, y para la implementación se utilizó el dispositivo de la familia Artix XC7A100T. El análisis de prestaciones se centró en varios aspectos clave, como la utilización de recursos y la frecuencia máxima de operación. Se evaluaron diferentes versiones del cifrador, ajustando el número de rondas de permutación para encontrar un equilibrio óptimo entre la complejidad lógica y el rendimiento.

Los resultados mostraron que las versiones con menos rondas de permutación (v3 y v5) ofrecen una mayor rapidez en el procesamiento a costa de un mayor consumo de recursos lógicos. Sin embargo, estas versiones también demostraron la flexibilidad del diseño para adaptarse a diferentes necesidades y restricciones de recursos.

Se realizó una comparación con la versión original del cifrador *ASCON* creada por los autores. La versión original utiliza menos registros y *LUTs* en comparación con las versiones desarrolladas en este proyecto, lo que refleja una optimización eficiente, pero también una mayor complejidad en el diseño. Como la interfaz de entrada-salida es diferente para la versión original tampoco pueden establecerse conclusiones definitivas, dado que algunos de los datos que en el diseño realizado en este trabajo se almacenan internamente, en el diseño original puede ser almacenado fuera del mismo.

Se ha llevado a cabo la evaluación de la frecuencia máxima mediante simulaciones temporales (post-route) ajustando gradualmente el período del reloj para determinar el punto de fallo y mediante un análisis temporal estático. El diseño para la versión v1 funcionó correctamente hasta una frecuencia cercana a 204 MHz, para la versión v3, 178.571 MHz, y, por último, para la versión v5 102.041.

Finalmente concluir que este trabajo ha permitido aplicar de forma práctica los conocimientos adquiridos durante el grado, incluyendo el diseño de hardware digital, el uso de flujos de diseño y herramientas para tecnologías FPGAs, y el conocimiento de técnicas de implementación. A través de este proyecto, se ha demostrado la viabilidad y eficiencia de la implementación hardware de algoritmos de criptografía lightweight como *ASCON*, cumpliendo con las normativas y requerimientos específicos para dispositivos con recursos limitados.

Además, este estudio ha proporcionado una base sólida para futuras investigaciones y desarrollos en el campo de la criptografía lightweight. La experiencia adquirida en la optimización de recursos y la implementación de

técnicas de prueba avanzadas contribuirá significativamente a la mejora continua y la innovación en este ámbito.

En resumen, este proyecto no solo ha logrado los objetivos propuestos, sino que también ha ampliado la comprensión y las capacidades en el diseño y análisis de sistemas criptográficos ligeros, sentando un precedente para futuros trabajos en esta área crítica de la seguridad informática.

BIBLIOGRAFÍA

[1] Competitions.cr.yp.to. (n.d.).

CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. https://competitions.cr.yp.to/caesar-call.html.

Accedido el 31 de enero de 2024.

[2] National Institute of Standards and Technology. (2023).

Final steps of the NIST lightweight cryptography standardization process [PDF]. https://csrc.nist.gov/csrc/media/Presentations/2023/final-steps-of-the-nist-lightweight-cryptography-s/images-media/talk-cesar-2023-meltem-nov2023.pdf.

Accedido el 31 de enero de 2024.

[3] National Institute of Standards and Technology. (n.d.).

NISTIR 8114 Report on Lightweight Cryptography. https://doi.org/10.6028/NIST.IR.8114.

Accedido el 31 de enero de 2024.

[4] Stallings, W. (2017).

Cryptography and Network Security: Principles and Practice.

Pearson. https://www.pearson.com/store/p/cryptography-and-network-security-principles-and-practice/P100000337223.

Accedido el 31 de enero de 2024.

[5] Rogaway, P. (2004).

Authenticated Encryption with Associated Data. https://web.cs.ucdavis.edu/~rogaway/papers/ad.pdf.

Accedido el 31 de enero de 2024.

[6] NIST. (2016).

Lightweight Cryptography. https://csrc.nist.gov/projects/lightweight-cryptography.

Accedido el 31 de enero de 2024.

[7] Digikey. (n.d.).

Fundamentals of FPGAs Part 4: Getting Started with Xilinx FPGAs., de https://www.digikey.com/es/articles/fundamentals-of-fpgas-part-4-getting-started-with-xilinx-fpgas.

Accedido el 31 de enero de 2024.

[8] Digilent. (n.d.).

Nexys 4 DDR Resource Center. https://digilent.com/reference/programmable-logic/nexys-4-ddr/start.

Accedido el 31 de enero de 2024.

[9] Graz University of Technology. (n.d.).

ASCON – Authenticated Encryption and Hashing. https://ASCON.iaik.tugraz.at/.

Accedido el 31 de enero de 2024.

[10] Fernández García, C. (2022).

Diseño microelectrónico de cifradores AEAD con introducción de técnicas de reducción del consumo de potencia . Repositorio de la Universidad de Sevilla [Trabajo de Fin de Máster]. https://idus.us.es/bitstream/handle/11441/141649/FERNANDEZ%20GARCIA%2c%20CARLOS.pdf?sequence=1&isAllowed=y.

Accedido el 31 de enero de 2024.

[11] NIST SP 800-38A.

Recommendation for Block Cipher Modes of Operation: Methods and Techniques. https://doi.org/10.6028/NIST.SP.800-38A

Accedido el 31 de enero de 2024.

[12] Hardware API for Lightweight Cryptography

https://cryptography.gmu.edu/athena/LWC/LWC_HW_API.pdf.

Accedido el 31 de enero de 2024.

[13] Ascon hardware. (n.d.). Test vectors for ASCON LWC. GitHub. https://github.com/ascon/ascon-hardware/blob/master/hardware/ascon_lwc/KAT/v1/test_vectors.txt

Accedido el 31 de enero de 2024.