Trabajo Fin de Grado en Ingeniería Electrónica Industrial

Diseño en VHDL del cifrador lightweight TinyJAMBU

Autor: Carlos Ruiz Rubio.

Tutores: Carlos Jesús Jiménez Fernández y Francisco Eugenio Potestad Ordóñez.

Fecha: 16/12/2022.

Agradecimientos

En esta parte, me gustaría agradecer a todas las personas que me han acompañados estos años de carrera, que sinceramente, han sido muy duros y bonitos. Sobre todo, para este trabajo, agradecer a mi tutor Carlos la paciencia, y la clara vocación que tiene por esto, que ha conseguido motivarme en numerosas ocasiones.

También agradecer a todas las personas que he conocido a lo largo de estos años, que especialmente estos últimos, han sido en ciertos momentos necesarios, sin duda una fuente de inspiración.

Por último, destacar a mi familia, que han sido también fuente de motivación cuando esta más flojeaba, sin duda, años extraños, pero con un aprendizaje bestial.

Resumen

Debido al auge de la tecnología, así como de las comunicaciones digitales, también ha sido necesario el aumento de la seguridad de dichas comunicaciones. Es ahí donde surge la necesidad de la criptografía, que garantice, además de la confidencialidad, la autenticidad de los mensajes.

La criptografía está cada vez más presente en diferentes ámbitos, teniendo una evolución que parece no parar. Debido a la incorporación de dispositivos IoT, está tomando mucho auge un tipo de criptografía denominada criptografía lightweight. Esta es un tipo de criptografía enfocada a dispositivos con recursos limitados y donde las restricciones de procesado y de potencia son muy elevadas.

El objetivo principal de este trabajo, es mostrar información acerca de este tipo de criptografía, y diseñar un cifrador basado en uno de los algoritmos ligtweight y que incluya autenticación, llamado TinyJAMBU. Este es uno de los diez finalistas del proyecto del National Institute of Standards and Technology (NIST) que tiene como objetivo establecer nuevos estándares de criptografía lightweight, y complementar los que ya existen en la criptografía convencional.

El diseño del algoritmo ha sido realizado en lenguaje VHDL, y utilizando la aplicación de Xilinx de ISE Design. Además del propio código, se han llevado a cabo verificaciones, en las que se comprueba el correcto funcionamiento del código mediante simulaciones.

Además, se han llevado a cabo dos diseños más, sirviendo como ejemplos del correcto funcionamiento del código. El primero de ellos se trata de una demostración visual del correcto funcionamiento, mediante el uso de una FPGA (Field Programmable Gate Array) de Xilinx, utilizando la placa Nexys4DDR. El segundo de ellos, ha sido un ejemplo de aplicación mediante la emisión y recepción de datos, mediante el uso de una UART.

Con todo ello, se pretende mostrar los mecanismos de uso de la criptografía mediante el diseño de cifradores y de ejemplos de aplicación.

Abstract

Due to the increase in technology, as well as digital communications, it has also been necessary to increase the security of said communications. It is there where the need for cryptography arises, which guarantees, in addition to confidentiality, the authenticity of messages.

Cryptography is increasingly present in different environments, having an evolution that does not seem to stop. Due to the incorporation of IoT devices, a type of cryptography called lightweight cryptography is taking off. This is a type of cryptography focused on devices with limited resources and where processing and power restrictions are very high.

The main objective of this work is to show information about this type of cryptography, and to design an encryptor based on one of the lightweight algorithms and that includes authentication, called TinyJAMBU. This is one of the ten finalists of the National Institute of Standards and Technology (NIST) project that aims to establish new lightweight cryptography standards, and complement those that already exist in conventional cryptography.

The design of the algorithm has been carried out in VHDL language, and using the Xilinx application of ISE Design. In addition to the code itself, verifications have been carried out, in which the correct functioning of the code is verified through simulations.

In addition, two more designs have been carried out, serving as examples of the correct operation of the code. The first of them is a visual demonstration of the correct operation, through the use of a Xilinx FPGA (Field Programmable Gate Array), using the Nexys4DDR board. The second of them has been an example of application through the transmission and reception of data, through the use of a UART.

With all this, it is intended to show the mechanisms of use of cryptography through the design of ciphers and application examples.

Índice

Índice de figuras

Glosario de términos abreviados

- FPGA: Field Programmable Gate Array.
- UART: Universal Asynchronous Receiver-Transmitter.
- AEAD: Authenticated Encryption with Associated Data.
- NIST: *National Institute of Standards and Technology.*
- TFG: Trabajo fin de grado.
- VHDL: VHSIC Hardware Description Language.
- VHSIC: Very High Speed Integrated Circuit.
- AES: Advanced Encryption Standard.
- RSA: Rivest-Shamir-Adleman.
- DES: Data Encryption Standard.
- IOT: internet of things.
- LUT: look-up tables.
- RFID: Radio Frequency Identification.
- GA: Gate Area.
- GE: Gate Equivalent.
- Ram: Ramdon Access Memory.
- Rom: Read Only Memory.
- ASIC: Aplication Specific Integrated Circuit.
- KDF: Key Derivation Function.
- MAC: Message Authentication Code.
- FIPS: *Federal Information Processing Standards*.
- TDEA: Triple Data Encryption Algorithm.
- SHA: Secure Hash Algorithm.
- GCM: Galois Counter Mode.

1. Introducción.

Se va a introducir el trabajo fin de grado (TFG) realizado, el cual, va a tratar sobre la realización en código vhdl del algoritmo de cifrado TinyJAMBU basado en criptografía lightweight.

En este TFG se va a proceder a hablar sobre un tema bastante recurrente en esta nuestra actualidad, se trata del tema de la criptografía, en concreto, se va a buscar información acerca de la criptografía lightweight, una de las modalidades criptográficas que más fuerza está tomando estos últimos años.

El término criptografía, que proviene de las palabras griegas *krypto y graphé*, cuyo significado era el de escritura secreta, tiene varios objetivos hoy en día, no solo el de ocultar la información ante ojos no deseados. Entre sus objetivos más destacados se encuentran, la protección contra robos de información, la capacidad de dar autenticidad a dicha información, la prohibición de incluir información falsa externa o la capacidad para tener confidencialidad a la hora de comunicarse, entre otros factores.

Sin duda alguna, la criptografía se encuentra presente en multitud de aplicaciones cotidianas casi sin darnos cuenta, cada mensaje, cada vídeo que se ve, cada gmail que se recibe, tiene detrás una criptografía que prácticamente nadie tiene en cuenta. Sin embargo, juega un papel muy importante a la hora de proporcionar la seguridad necesaria sobre los datos sensibles de los usuarios.

Este proyecto de criptografía lightweight es importante, por la creciente importancia que está recibiendo este tipo de criptografía, ya que existen una serie de dispositivos a los cuales no se les puede aplicar la criptografía convencional, ya que estos apenas poseen memoria, ni cpu, más adelante, se concreta que tipo de dispositivos se benefician de dicha criptografía.

1.1. Objetivos.

Como se ha dicho previamente, este trabajo está realizado en base al algoritmo de cifrado TinyJAMBU del concurso del *National Institute of Standards and Technology* (NIST). En dicho concurso, todavía sin finalizar, TinyJAMBU se encuentra entre los proyectos finalistas.

El objetivo principal del trabajo es la demostración del funcionamiento de dicho código de criptografía lightweight, en el lenguaje VHSIC Hardware Description Language (VHDL), mediante la aplicación de ISE DESIGN de Xilink. Además se realizarán una serie de ejemplos en los que poner a prueba dicho código utilizando una Field Programmable Gate Array (FPGA), en concreto, la Nexys4DDR.

Hay que destacar también la falta de información que hay acerca de este tipo de criptografía, ya que aunque su crecimiento esta última década sea notable, sigue siendo un mundo mucho más desconocido que la criptografía convencional.

1.2. División del contenido.

Para la realización de dicho trabajo se ha dividido en varias partes, concretamente cuatro partes. La primera de ellas consta del estado del arte, en la que se puede ver, primeramente, de que trata la criptografía, así como algunos ejemplos de esta, para introducir el algoritmo TinyJAMBU y el proyecto del NIST, así como información extra sobre Authenticated Encryption with Associated Data (AEADs), que se trata de una manera de encriptar, la cual, asegura simultáneamente confidencialidad y autenticidad, y FPGAs.

La segunda parte, donde se lleva a cabo el objetivo principal de este proyecto, es decir, se lleva a cabo el desarrollo del código en lenguaje VHDL del algortimo TinyJAMBU, así como la explicación de la descripción del método de cifrado/descifrado, y sus correspondientes pruebas para verificar un correcto funcionamiento.

La tercera parte, se centra en la primera de las demostraciones, en este caso, la más simple, que consta de una demostración visual utilizando la FPGA de Nexys4DDR, en la que además de la propia demostración visual, y una serie de ejemplos de prueba, también se prueba el correcto funcionamiento del código realizado.

En la cuarta y última parte, que es la segunda de las demostraciones, se presenta un proyecto de transmisión y recepción de datos mediante una Universal Asynchronous Receiver-Transmitter (UART), en el que dicha transmisión y recepción se realiza con el mensaje totalmente cifrado. Se trata de un ejemplo de aplicación real. Todo este proyecto ha sido realizado con la aplicación Xilinx y su simulador de prueba, mediante descripciones de código en lenguaje VHDL.

1.3. Aprendizaje.

En este apartado se pretende destacar tanto, todo lo aprendido previamente en la asignatura diseño digital avanzado, impartida por Carlos Jesús, así como todo lo aprendido y perfeccionado en este proyecto, ya que a lo largo de los meses, el dominio adquirido acerca del lenguaje VHDL, así como la información adquirida acerca de los cifradores, ha sido fundamental para la realización de este proyecto.

2. Estado del arte.

2.1. ¿Qué es la criptografía?

2.1.1. Introducción.

La criptografía, tradicionalmente, se define como una de las partes de la criptología, la cual, se encarga de proporcionar confidencialidad, integridad, autenticación y no repudio en la transmisión de información [1]. Para el caso de la confidencialidad, las técnicas de cifrado de ciertos mensajes los convierten en otros totalmente ininteligibles para todo aquel que no tenga las claves adecuadas. Estas técnicas se han utilizado a lo largo de la historia tanto para el arte, la ciencia, o más recientemente, para la tecnología. Cabe destacar uno de los sucesos más importantes de la historia, que probablemente adelantara el fin de la segunda guerra mundial, en parte provocado por Alan Turing, y su descubrimiento del proceso de descifrado de la máquina enigma usada por los alemanes.

En la actualidad, debido al desarrollo de la informática, y al aumento de las comunicaciones digitales, la criptografía tiene gran importancia, ya que todas las comunicaciones deben ser lo más seguras posibles, para no poner en riesgo la confidencialidad y la autenticidad del mensaje transmitido.

Es aquí, donde gracias a la evolución dada en el campo de las matemáticas y la informática, la criptografía ha conseguido avanzar muchísimo en muy poco tiempo.

Para este trabajo, se ha diseñado en lenguaje VHDL el algoritmo de cifrado y descifrado con autenticación TinyJAMBU.

2.1.2. ¿Qué es un cifrado?

El **cifrado** es un método de protección de datos, el cual, consiste en alterarlos hasta hacerlos ilegibles para cualquier persona no autorizada. Los datos pasan de ser *texto sin cifrar, o texto plano,* a ser *texto cifrado* por medio del empleo de un *algoritmo*. Quien quiera acceder a los datos cifrados para obtener el texto plano (descifrado) debe conocer tanto el algoritmo utilizado como la clave adecuada para descifrarlos.

El cifrado cubre uno de los aspectos de la criptografía *[2]***.** Se trata de un proceso que utiliza cálculos y algoritmos para codificar texto plano de forma eficiente y compleja y que además sea muy difícil de descifrar sin conocer las claves correctas.

El cifrado funciona pasando los **datos originales** (o texto plano) por un **algoritmo** (o cifrador) que los convierte en **texto cifrado**. El nuevo texto resulta ilegible si no se utiliza la clave de descifrado adecuada para descodificarlo.

Los algoritmos de cifrado emplean **claves criptográficas** (cadenas de caracteres de un número determinado de bits) para transformar los datos en un sinsentido aparentemente indescifrable. Los algoritmos más utilizados en la actualidad descomponen los datos de texto plano en grupos llamados bloques y luego cifran cada bloque como una unidad. Por este motivo se los denomina **cifradores de bloques, destacar que el TinyJAMBU pertenece a este grupo de cifradores de bloques**.

2.1.3. Tipos de cifrado.

Destacan entre los procesos de cifrado, dos grupos, los de cifrado simétrico, y los de cifrado asimétrico.

Por una parte, se encuentra el **cifrado asimétrico**, el cual, utiliza dos claves, una pública para cifrar los datos, y otra privada para descifrarlos, de esta manera, resulta ser un método mucho más complejo, pero a la par, más

seguro**, y que no necesita un mecanismo para compartir claves.** Para el correcto funcionamiento de este tipo de cifrado, la clave pública puede ser conocida por cualquier persona, ya que es indispensable para poder cifrar el texto plano, sin embargo, solo las personas conocedoras de la clave privada podrán descifrar los datos, y obtener su respectivo texto plano. A destacar también, que es un método **que consume muchos recursos.**

Por otra parte, el **cifrado simétrico** utiliza la misma clave tanto para cifrar los datos como para descifrarlos. Tanto la persona que cifra la información, como la que la descifra posteriormente deben conocer la clave previamente, y solo ellos deben conocer la clave para asegurar la seguridad del proceso. Por eso **el cifrado simétrico también se conoce como *cifrado de clave privada*:** si la clave se hace pública, todo el proceso deja de tener validez, ya que pierde toda seguridad. Los algoritmos de cifrado simétrico consumen muchos menos recursos que los de cifrado asimétrico, aunque tienen el problema de la distribución de claves. Es el que se ha utilizado en este caso, para implementar el funcionamiento del algoritmo del TinyJAMBU.

2.1.4. Ejemplos de métodos.

Se procede a mostrar una serie de ejemplos de los métodos de cifrado más usados hoy en día:

- **Triple DES.**

El sistema simétrico Triple Data Encryption Standard (DES) utiliza tres claves de 56 bits. Los datos se cifran, se descifran y se vuelven a cifrar. A la hora de recuperar la información, Triple DES repite el proceso a la inversa: descifra, cifra y vuelve a descifrar. Como se puede suponer, es mucho más seguro que utilizar una sola clave de 56 bits.

Sin embargo, este método, con el paso del tiempo, los avances en cuanto a tecnología, y nuevos algoritmos mucho más largos, han acabado por dejar obsoleto a este cifrador.

- **AES.**

Advanced Encryption Standard (AES) es un sistema simétrico de cifrado de bloques que suelen utilizar empresas, instituciones financieras y gobiernos, es uno de los métodos simétricos más utilizados a nivel mundial, porque es el estándar seleccionado por el NIST (*National Institute of Standards and Technology*). Cifra los datos en bloques de 128 bits y dispone de variantes con claves de distintos tamaños. Además, cada clave necesita distintas rondas de cifrado (el proceso que codifica y descodifica los datos) para lograr mayor seguridad:

- AES-128: una clave de 128 bits en 10 rondas.
- AES-192: una clave de 192 bits en 12 rondas.
- AES-256: una clave de 256 bits en 14 rondas.

- **RSA.**

Se trata del estándar Rivest-Shamir-Adleman (RSA), uno de los métodos de cifrado asimétricos más utilizados. Suele utilizarse para firmas digitales y para enviar datos por correo electrónico, chats, navegadores seguros y VPN (redes privadas virtuales). El algoritmo RSA utiliza números primos para generar claves de muchos bits. 1024 bits es lo más habitual, pero pueden extenderse hasta los 2048 bits, por lo que el número de recursos necesarios para implementar este método de cifrado es mucho más alto que en los métodos de cifrado simétrico.

Por su tamaño de clave, utilizar RSA en cifrados de muchos datos puede llevar bastante tiempo, por lo que suele utilizarse en configuraciones de cifrado híbridas: se genera una clave y se comparte utilizando RSA pero los datos se cifran mediante un sistema simétrico, haciendo que la implementación del método sea mucho más eficiente, sin dejar de asegurar una correcta seguridad de los datos.

2.1.5. Introducción de la criptografía lightweight.

Para empezar, destacar que los métodos de cifrado tradicionales no sirven de mucho para ciertos dispositivos, como por ejemplo, los pertenecientes al mundo IoT (internet of things), entre los que destacan las etiquetas RFID, las redes de sensores inalámbricos, los controladores industriales y los dispositivos embebidos.

Estos dispositivos utilizan microcontroladores de 8, 16 o 32 bits, y en casos de aplicaciones de muy bajo costo pueden llegar a utilizarse microcontroladores de 4 bits. Sus conjuntos de instrucciones son también muy reducidos debido a las limitaciones propias de la RAM. Algunos, como las etiquetas RFID, no tienen ni batería y se alimentan de la energía electromagnética transmitida por el propio lector al acercarlo.

Es aquí donde entra la necesidad de una criptografía que utilice menos recursos que los métodos convencionales, ya que los dispositivos previamente nombrados no podrían hacer funcionar los métodos de cifrado convencionales.

De esta necesidad, entra en escena el NIST (*National Institute of Standards and Technology*), en la búsqueda de un método estándar de cifrado lightweight para estos dispositivos de recursos limitados.

2.2. Concurso NIST sobre criptografía lightweight.

2.2.1. Introducción.

Este proyecto desarrollado por el NIST tiene su necesidad en el aumento de pequeños dispositivos informáticos, en muchos casos sustituyendo a ordenadores con gran capacidad de cómputo. Dicho cambio trae consigo nuevos problemas respecto a la seguridad y privacidad, ya que al ser dispositivos con recursos limitados, hay bastante limitación al aplicar los estándares de criptografía convencional.

Aquí es donde entra en juego la criptografía *lightweight*, la cual trata de implementar eficientemente los objetivos de la criptografía aplicados a dispositivos con recursos limitados.

Debido a esto, en 2016, el NIST inicia un proyecto para estudiar el rendimiento de los estándares en dispositivos con pocos recursos, y para ver si hay necesidad de realizar estándares para criptografía lightweight. Por ello, realiza dos cursos prácticos en Gaithersburg para ver que opinan los demás sobre dicha necesidad de implementar criptografía lightweight.

Finalmente, en 2018, el NIST *[3]* crea un proyecto para crear un estándar de algoritmos *lightweight* a través de un proceso abierto. Destacar que, en el momento de desarrollo de este trabajo, **en dicho concurso el TinyJAMBU se encuentra entre los 10 finalistas.**

2.2.2. Criptografía lightweight.

Se proceden a explicar todos los detalles y características acerca de este tipo de criptografía [4].

2.2.2.1. Dispositivos hacia los que va dirigida.

Dado su tamaño y la cantidad de recursos que utiliza, la criptografía lightweight puede ser implementada en una amplia variedad de dispositivos.

Equipos servidores y de escritorio	Criptografía convencional
Tabletas y smartphones	
Sistemas embebidos	Criptografía ligera
RFID y redes de sensores	

Figura 1. Dispositivos lightweight.

Aquí se puede ver en la figura 1 aquellos dispositivos en los que es más útil esta criptografía, ya que tanto los ordenadores empotrados, como tabletas o móviles rinden muy bien con la criptografía convencional, por ello, solo los sistemas integrados, los dispositivos RFID y las redes de sensores son aquellos dispositivos a los que va dirigida la criptografía lightweight, ya que esta, está diseñada para dispositivos altamente restringidos.

Por ejemplo, los microcontroladores se beneficiarían mucho de esta criptografía ya que podrían usar más recursos para otras aplicaciones y tardarían mucho menos en realizar el proceso de encriptado de la información, ya que microcontroladores muy pequeños de 4 u 8 bits no cuentan con suficientes recursos para satisfacer de manera eficiente los algoritmos de criptografía convencionales.

Lo malo de todo esto es que dicha criptografía lightweight presenta otras limitaciones que se estudiarán más adelante, sin embargo, el objetivo es mostrar la necesidad de esta para ciertos dispositivos.

Por último, cabe destacar también, que hay ciertas aplicaciones para dispositivos del extremo superior de la imagen en las que convendría utilizar

una criptografía que consumiera menos recursos, por ejemplo, en la interacción con sensores que utilicen algoritmos lightweight.

2.2.2.2. Métricas de rendimiento.

A la hora de hacer estos algoritmos de criptografía, hay que buscar un equilibrio entre el rendimiento y el número de recursos usados, dicho desempeño **puede ser expresado en términos de potencia, consumo energético, latencia y rendimiento.**

Por otra parte, los recursos necesarios para una implementación hardware son: gate area, gate equivalents, o logic blocks. En cambio, para el software se utilizan registros, memoria Ramdom Access Memory (RAM) y memoria Read Only Memory (ROM). El número de estos recursos que se usa es lo que marca el coste del dispositivo, a más memoria, por ejemplo, mayor será el coste de producción del dispositivo.

Potencia y energía consumida son dos métricas sumamente importantes debido a la naturaleza de muchos dispositivos restringidos, de ahí que haya que tener en cuenta otros factores aparte del algoritmo para realizar un consumo energético adecuado.

Otras dos características importantes a tener en cuenta para estos algoritmos son la latencia y el rendimiento, la primera para realizar la operación de cifrado a una velocidad adecuada, y el segundo para optimizar la producción de nuevas salidas, tampoco puede ser excesivamente alto dicho rendimiento, ya que hay que tener en cuenta la limitación de recursos en la que se encuentran los dispositivos que utilizan esta criptografía.

- **Métricas específicas de hardware.**

Los requisitos de recursos para plataformas de hardware generalmente se describen **en términos de gate area (Ga)**. El área de una implementación depende de la tecnología usada, entre otras cosas, y se expresa en µm^2.

Por otra parte, este área puede ser establecida en términos de logic blocks (bloques lógicos) para FPGAs (field-programmable gate arrays) o en términos

de GEs (Gate equivalents) en implementaciones Aplication Specific Integrated Circuit (ASIC).

En FPGA, los logic blocks están compuestos por look-up tables (LUTs), flip-flops y multiplexores, cuyos valores dependen de la familia FPGA usada.

En cambio, para los ASIC, un GE es equivalente al área requerida por la puerta NAND de dos entradas, y su área se calcula dividiendo el área en μm^2 por el área de la puerta NAND. Cabe destacar también, que el número de GEs necesarios en una implementación hardware depende mucho de la tecnología para la que van a ser usados.

De suma importancia todo esto, ya que para la mayoría de implementaciones de recursos limitados, es muy importante utilizar un área lo menor posible, para asegurar un menor consumo de energía.

- **Métricas específicas de software.**

Los requisitos de recursos de las aplicaciones de software, pueden ser medidos mediante **el número de registros,** así como el número de bytes que necesitan la **RAM y la ROM.**

Se tiene que tener en cuenta que, al usar funciones con menor número de registros, estás se sobrecargan más fácilmente, ya que un menor número de variables podrán ser escritas en la pila antes de que se sobrescriban los registros, pudiendo perder así algún valor en caso de no tener esto en cuenta.

Por otra parte, la ROM es utilizada para almacenar el código del programa, que también debe ser lo más eficiente posible, mientras que la RAM se encarga de almacenar valores intermedios que se utilizarán en sus correspondientes cálculos.

2.2.2.3. Inicios de la criptografía lightweight y sus primeros diseños.

En esta última década, este tipo de criptografía ha alcanzado cierta relevancia, y se han propuesto bastantes cosas acerca de ella, como por ejemplo, funciones hash, códigos de autenticación de mensajes, cifrados de bloques, entre otros, con la intención de ofrecer ventajas de rendimiento sobre los estándares criptográficos convencionales. También hay que destacar, que este tipo de criptografía no va dirigida a una amplia gama de aplicaciones, ya que solo para ciertas aplicaciones ofrecen ventajas respecto a la criptografía convencional. El objetivo primordial de este tipo de criptografía es buscar un correcto equilibrio entre seguridad, rendimiento y requisitos necesarios.

2.2.2.3.1. Inicios de los cifradores de bloque lightweight.

En estos últimos años se han propuesto una serie de bloques ligeros para lograr ventajas de rendimiento sobre el AES (*Advanced Encryption Standard*). Generalmente, estos bloques funcionan simplificando los bloques convencionales, por ejemplo, DESL es una variante del DES donde la *round function* utiliza una sola S-box en lugar de ocho, además de llevarse a cabo una optimización de la implementación hardware.

Sin embargo, también existen algunos bloques que fueron creados desde cero, entre los que destaca **PRESENT**, que fue uno de los primeros que se propuso para entornos de hardware restringidos. En la figura 2 se puede ver el desarrollo de su algoritmo, compuesto por 31 rondas, en la que destacan 3 procesos: El primero, combinacional, formado por la clave y el texto plano, el segundo, formado por un proceso de sustitución llamado *sBoxLayer,* y el tercero, que es simplemente un proceso de permutación.

generateRoundKeys()
for $i = 1$ to 31 **do**
  addRoundKey(STATE,K_i)
  sBoxLayer(STATE)
  pLayer(STATE)
end for
addRoundKey(STATE,K_{32})

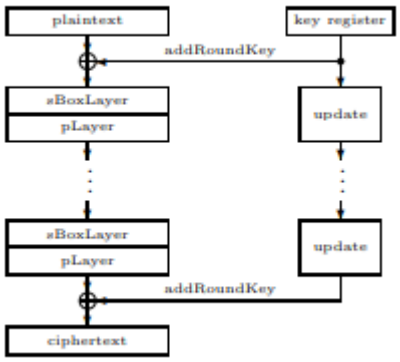

Figura 2. Descripción del algoritmo de PRESENT [5].

Los beneficios de rendimiento de los cifrados de bloque lightweight sobre los cifrados de bloque convencionales se consiguen utilizando opciones de diseño lightweight, entre las que destacan:

- **Tamaños de bloques más pequeños:** Se utilizan bloques más pequeños que el AES (64 u 80, en vez de 128). Esto provoca la limitación del número de bloques de texto plano que podrán ser encriptados.

- **Tamaños de clave más pequeños:** Suelen utilizar tamaños de claves más pequeños para asegurar más eficiencia, teniendo el NIST su mínimo en 112 bits.

- **Rondas más simples:** Se utilizan componentes y operaciones más simples que en un bloque convencional, tanto para simplificar su dificultad, como para ahorrar área. Debido a esto, se deben realizar más rondas para garantizar su seguridad.

- **Programaciones de claves más simples:** Dado que programar una clave compleja requiere bastantes recursos, se prefiere optar por la generación de claves mucho más simples, aunque expuestas a una mayor facilidad de ataque, sin embargo, se están creando técnicas como la función de derivación Key Derivation Function (KDF) que minimizan dichos ataques.

- **Implementaciones mínimas:** Solo se implementan las funciones necesarias que un cifrado puede requerir, utilizando el menor número de recursos posibles y pudiendo llegar a necesitar un dispositivo de soporte diferente para realizar la operación de cifrado, o la de descifrado.

2.2.2.3.2. Funciones hash lightweight.

Las funciones Hash convencionales no son nada útiles para los dispositivos que cuentan con un número restringido de recursos, ya que estas utilizan unos estados internos demasiado grandes, lo que conlleva también a un gran consumo de energía, algo que es inviable para dispositivos con una fuente de energía bastante pequeña. Para ello, se iniciaron ciertos proyectos para conseguir minimizar los gastos de recursos de estas funciones, naciendo así funciones hash lightweight, entre las que destacan los proyectos de **PHOTON, Quark o SPONGENT.**

Estas funciones se diferencian de las convencionales en lo siguiente:

- **Tamaños de salida y estados internos más pequeños:** Se requieren tamaños grandes a la hora de necesitar una mayor resistencia a la colisión para estas funciones, sin embargo, para aplicaciones donde dicha resistencia no sea tan importante, se pueden usar tamaños menores para la salida, y para el estado interno.

- **Tamaño de mensajes más pequeños:** Al ser utilizados para ciertas aplicaciones, los tamaños de entrada pueden ser bastante más pequeños, siendo en los convencionales de alrededor de 264 bits, mientras que estas funciones lightweight tienen un valor máximo de 256 bits. Utilizan menos bits, pero para ello cuentan con una mayor optimización para mensajes más cortos.

2.2.2.3.3. Códigos de autenticación de mensajes lightweight.

Para verificar la autenticidad e integridad del mensaje se necesita una etiqueta, la cual es generada por un código de autenticación de mensajes (*MAC* en inglés) a partir de un mensaje y una clave secreta, suelen ser de 64 bits dichas etiquetas en aplicaciones típicas, sin embargo, existen ciertas aplicaciones que aceptan etiquetas más cortas, además de aceptar ocasionalmente mensajes no auténticos sin reducir drásticamente su seguridad. Entre los algoritmos MAC lightweight destacan **Chaskey, TuLP y LightMAC.**

2.2.2.3.4. Cifrados de flujo lightweight.

Estos cifrados han cogido bastante fuerza en el mundo lightweight, tanto es así, que en 2008, se dieron a conocer los tres finalistas de la competición que lanzó eSTREAM para identificar un cifrado de flujo para aplicaciones hardware de recursos restringidos, fueron los siguientes:

- **Grain**, el cual proporciona flexibilidad de implementación y también tiene una versión que admite autentificación.

- **Trivium**, que parte con la desventaja de admitir solamente claves de 80 bits.

- **Mickey**, que es el menos analizado de los tres, y por ende, también es el que proporciona menor flexibilidad de implementación.

2.2.2.4. Medidas lightweight aprobadas por el NIST en entornos restringidos.

A continuación se describe el rendimiento de los estándares criptográficos aprobados por el NIST en entornos con recursos limitados.

- **Cifrado de bloques.**

Hay dos algoritmos de cifrado de datos aprobados por el NIST: **el AES y el TDEA (Triple Data Encryption Algorithm).** El AES incluye tres variantes, AES-128, AES-196 y AES-256, que admiten tamaños de clave de 128, 196 y 256 bits cada una como indica su nombre, sin embargo, todas tienen un tamaño de bloque de 128 bits. Para el modelo lightweight lo óptimo es partir del AES-128, ya que el número de rondas y el tamaño de clave son menores, y por ende, requiere menos recursos. Las implementaciones de AES-128 requieren desde 2090 GE a 2400 GE y suelen ser implementadas para aplicaciones de software, también destaca su buen rendimiento en algunos microcontroladores de 8 bits.

- **Funciones hash.**

Existen dos FIPS (*Federal Information Processing Standards*) donde se especifican las funciones hash aprobadas, ellos son, Fips 180-4, el cual especifica **SHA-1** (Secure Hash Algorithm) y la familia **SHA2,** y FIPS 202, el cual especifica la familia **SHA-3.** Sin embargo, ninguna de estas funciones hash es adecuada para entornos restringidos, debido a su tamaño, no compatible con la RAM proporcionada.

- **Algoritmos de cifrado autenticados y MACs**

Estos algoritmos proporcionan ventajas tanto en el rendimiento, como en el número de recursos necesarios, ya que simultáneamente proporcionan protección de confidencialidad e integridad. El NIST aprueba el CCM, el GCM (Galois Counter Mode) y algunos MACs independientes, como el CMAC.

2.2.2.5. Estándares de criptografía lightweight.

El primero es **ISO/IEC 29192**, el cual es un estándar que se divide en seis partes. La primera incluye información general, la segunda especifica los cifrados de bloques de **PRESENT, CLEFIA,** y desde 2014 los cifrados de bloque de **SIMON y SPECK,** la tercera parte especifica los cifrados de flujo de **Enocoro y Trivium,** la cuarta especifica tres técnicas asimétricas, además de un esquema de autenticación basado en la curva elíptica incluido en 2014, cuyo nombre es **ELLI,** la quinta parte incluye tres funciones HASH: **PHOTON, SPONGENT y LesamntaLW,** y por último la parte seis está dedicada a los MACs y se encuentra en desarrollo.

Otro estándar es **ISO/IEC 29167,** el cual, proporciona seguridad para comunicaciones de interfaz aéreas RFID. Su primera parte describe la arquitectura la seguridad, las características y los requisitos para los servicios de seguridad para dispositivos RFID. El resto de documentos se encuentran en elaboración.

2.2.3. Proyecto de criptografía lightweight del NIST.

El NIST se encarga de desarrollar estándares utilizando varios enfoques diferentes, ya que crea estándares para varias utilidades.

En este caso, está creando una cartera de algoritmos lightweight que podrán ser usados de manera limitada, debido a la velocidad con la que se está moviendo el panorama de la criptografía lightweight, es decir, que no pueden garantizar un uso a largo plazo de algún algoritmo, ya que durante el proceso de estandarización podría llegar a quedar obsoleto. Estos algoritmos no estarán destinados para un uso general, por lo tanto, cada uno tendrá sus restricciones de uso.

2.2.3.1. Alcance.

El alcance de este proyecto incluye todas las primitivas criptográficas y modos que se necesitan en entornos restringidos. Aunque, el enfoque principal va **hacia los cifrados de bloques, las funciones hash, los códigos de autenticación de mensajes, los cifrados de flujos, las permutaciones criptográficas,** entre otras.

En caso de necesitar seguridad a largo plazo, estos algoritmos deben ser fácilmente reemplazables por algoritmos con seguridad post-cuántica.

El alcance del proyecto también incluye las claves públicas, aunque no estén dentro del enfoque inicial, ya que para incluirlas se deben cumplir ciertos requisitos.

2.2.3.2. Consideraciones del diseño.

Aun siendo cada diseño diferente según el uso dado, hay ciertas características presentes en muchos de ellos:

- **Fortaleza de seguridad:** Debe tener una fortaleza de seguridad mínima, de al menos unos 112 bits.

- **Flexibilidad:** Se buscan algoritmos que puedan ser implementados en múltiples plataforma, de ahí que se busquen algoritmos ajustables, es decir, que sean parametrizables para poder cumplir con diferentes aplicaciones solo cambiando algún valor como el tamaño de la clave o el tamaño del estado.
- **Baja sobrecarga para múltiples funciones:** Se prefiere que múltiples funciones (cifrado y descifrado, por ejemplo) compartan el mismo núcleo, para un menor consumo de recursos.

- **Expansión del texto cifrado:** Se busca un tamaño de texto cifrado no mucho más grande que el texto plano para evitar un mayor consumo del almacenamiento.

- **Canal lateral y ataques de fallos:** Debido a que las implementaciones pueden filtrar cierta información confidencial, se suele recurrir a los ataques de fallos, que se encargan de recuperar la información extraída de los ataques de canal lateral, introduciendo errores en los cálculos. Por eso, se prefieren algoritmos que sean fáciles de proteger contra estos ataques.

- **Limites en el número de pares de plaintext-ciphertext:** Se debe utilizar un límite para estos textos, ya que al contar con un número de recursos limitados, el diseñador del algoritmo no puede utilizar cualquier texto sin cifrar, o cifrado.

- **Ataques de clave relacionada:** Estos ataques permiten descubrir información sobre una clave, utilizando operaciones con claves desconocidas que guardan una relación conocida entre sí. Para ello, se pretende utilizar algoritmos que ofrezcan resistencia a estos ataques, sobre todo en ciertas aplicaciones.

Puede darse el caso en el que no sea posible satisfacer todas las consideraciones previamente mostradas, sobre todo por excederse en el número de recursos tan limitado que se tienen para una determinada aplicación. Sin embargo, cada algoritmo seleccionado para el proyecto a su manera debe garantizar una seguridad mínima, sobre todo, en los ataques de recuperación de claves.

2.2.3.3. Perfiles.

El NIST se encarga de evaluar y recomendar diferentes algoritmos para diferentes perfiles, cada uno con diferentes características. También hay que tener en cuenta que algunos de estos algoritmos han sido diseñados para cumplir varios objetivos según sus características.

Estos perfiles se diseñaran para seleccionar el dispositivo y la aplicación adecuada, para ello, se hizo la siguiente tabla.

Physical characteristics	Performance characteristics	Security characteristics
Area (in GEs, logic blocks, or mm^2) Memory (RAM/ROM) Implementation type (hardware, software, or both) Energy (J)	Latency (in clock cycles or time period) Throughput (cycles per byte) Power (W)	Minimum security strength (bits) Attack models (e.g., related key, multi-key) Side channel resistance requirements

Figura 3. Características de perfiles lightweight.

De las características encontradas en la figura 3, el NIST realiza sus diferentes perfiles, cada uno dirigido hacia diferentes dispositivos en función del número de recursos y la seguridad necesaria para la aplicación pertinente de dicho dispositivo.

2.2.3.4. Evaluación del proceso.

El NIST lleva a cabo el siguiente proceso:

- **1º:** El NIST solicita respuestas para la lista de preguntas sobre las necesidades actuales y futuras de la criptografía lightweight.
- **2º:** El NIST crea perfiles en base a las respuestas obtenidas por la comunidad, y deja un periodo abierto de 30 días para aceptar comentarios sobre si cumple o no los estándares de la actualidad.

- **3º:** El NIST abre una convocatoria para presentar las funciones lightweight obtenidas una vez creado el perfil, se buscarán también presentaciones que generen soluciones para estos perfiles.
- **4º:** El NIST llevará a cabo talleres sobre criptografía lightweight para debatir acerca de las necesidades de la industria, planes de normalización, perfiles, entre otras cosas.

Este proyecto actualmente sigue sin tener una fecha de finalización específica, ya que dicha criptografía está siendo motivo de cambio constante.

2.3. Cifradores AEAD.

Este tipo de cifrados cuentan con el añadido de un dato asociado, el cual, no se necesita cifrar, pero si forma parte del mensaje sobre el que generar la firma para su autenticación.

Dicho dato asociado también puede usarse para referirse a alguna identificación asociada al texto cifrado, por ejemplo, en caso de un historial médico, el dato asociado podría ser el id del usuario, asegurando así que ese texto cifrado solo podría pertenecer a ese usuario.

Entre las diferentes ventajas que se obtienen usando AEAD *[6] [7]*, destacan:

- Realiza en una sola pasada de manera simultánea lo necesario para **garantizar seguridad y autenticidad,** ya que no necesita dos procesos para ello.

- La capacidad de poder cambiar el mensaje a enviar, sin cambiar el texto no cifrado, es decir, que cambiando por ejemplo, dicho dato asociado, se puede cambiar el mensaje completo que se va a enviar, consiguiendo así que **un texto no cifrado no corresponda solamente a un determinado mensaje, puesto que el dato asociado en este caso, puede variar.**

- **El ahorro en código,** cosa bastante importante para dispositivos con recursos limitados, como es el caso de este proyecto.

- La posibilidad de ir encriptando bloques sobre la marcha, sin conocer el siguiente bloque de texto sin cifrar, provocando así una mayor velocidad, y una menor latencia.

- Proporciona **seguridad incluso si se repite el nonce, o incluso sin nonce.**

- Muy adecuado para aplicaciones lightweight, donde almacenar contadores o generar el número aleatorio, puede ser difícil de implementar.

El algoritmo del TinyJAMBU es un cifrador AEAD, porque además de ser un cifrador con autenticación, también tiene la posibilidad de añadirle un dato asociado.

2.4. Tecnología y diseño.

En este caso, la tecnología que se va a utilizar es la FPGA, en concreto un dispositivo Artix 7 insertado en la placa Nexys4 DDR de Digilent. El diseño del proyecto está realizado en la aplicación ISE Design de Xilinx, en el lenguaje de descripción VHDL.

Se procede a explicar que es una FPGA *[8]*.

Una **FPGA o Field Programable Gate Array** se trata de un circuito integrado, como puede ser cualquier chip, que está pensado para entregarse al cliente sin configurarlo para que luego cada uno **lo personalice y programe según la tarea que necesite que haga ese dispositivo**. Básicamente, son circuitos de hardware que el usuario puede configurar para que realicen cualquier funcionalidad. Es como si fabricaran un procesador y luego lo configuraran en función de lo que se necesite, en vez de enfocarlo a una tarea o un uso

específico durante el proceso de producción, esto le ofrece una flexibilidad a las FPGAs que no se ve en ningún otro producto.

Estas FPGAs normalmente contienen diferentes bloques de procesamiento en su interior, así como diferentes conectores que se utilizan para configurar cada uno de los bloques de lógica, pudiendo hacer tanto operaciones sencillas como algunas puertas lógicas, como operaciones de un calibre mucho mayor. También en su interior, se puede llegar a encontrar memoria, aunque esto suele verse solo en los módulos de lógica más complejos y avanzados.

Estos pequeños chips empezaron como **dispositivos de memoria de solo lectura (ROM) programables** y procesadores de lógica, los cuales se podían programar en fábrica, pero tenían una gran limitación, la cual es, que esos módulos de lógica **estaban ya fijados a determinadas puertas lógicas** y no se podían luego personalizar físicamente, solo pudiendo personalizarse **mediante la programación**. Años más tarde llegaría la compañía Altera, la cual, fundada en 1983 crearía el primer FPGA que permitiría su reconfiguración, borrando la configuración anterior del dispositivo y permitiendo su reutilización.

El primer FPGA que permitiría **reprogramar sus puertas lógicas y las conexiones entre ellas sería el XC2064**. Este chip creado por los fundadores de la compañía Xilinx llegaría en 1985 con 64 bloques de lógica y significaría un gran paso en el campo. En los años posteriores, los FPGA se empezaron a ver **en más campos** aparte del de las telecomunicaciones y redes, sobre todo a destacar, la década de los 90, donde se empezaron a usar en diferentes industrias, como la automoción. Finalmente, con el paso de los años, han ido evolucionando, dando un gran **salto en complejidad y prestaciones** hasta llegar a la actualidad, donde existen FPGAs con **miles de millones de transistores**.

Destacar, que a raíz de lo previamente dicho, lo que se ha hecho ha sido reprogramar la FPGA que se puede ver en la figura 4, para que esta realizara la tarea que se quería, utilizando así el código realizado para simular el código propuesto por TinyJAMBU en el concurso del NIST. Todo este proceso ha sido realizado en el lenguaje VHDL *[9]*, el cual, es uno de los lenguajes de

programación especial, o también conocido, como lenguaje de descripción de hardware. Apoyado por el entorno de desarrollo de Xilink ISE Design, que es específico en el diseño de sistemas que van a implementarse en una FPGA de Xilinx.

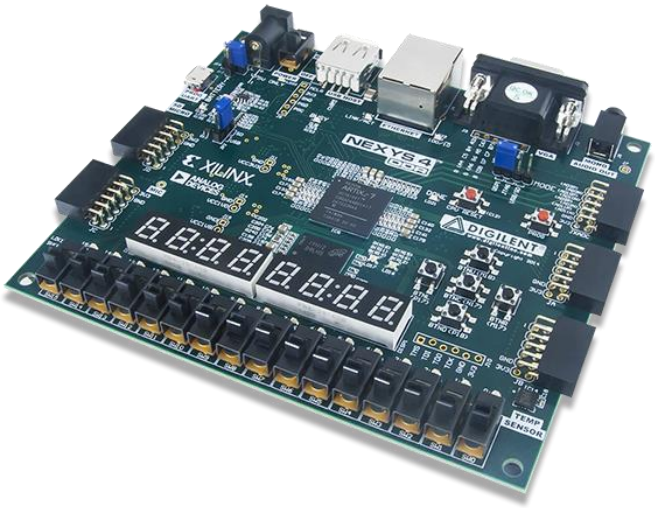

Figura 4. FPGA Nexys4DDR [10].

3. TinyJAMBU.

3.1. Descripción.

En este apartado, se van a exponer algunas de las características principales del método de cifrado y descifrado de la propuesta TinyJAMBU, así como, la descripción principal de su algoritmo [11].

Este algoritmo proviene de JAMBU, un modo de encriptación creado para el concurso del CAESAR, el cual, fue seleccionado para la tercera ronda de la competición. El algoritmo JAMBU fue previamente presentado al NIST en 2015, sin embargo, se decidió hacer todavía más liviano para poder participar en dicho concurso del NIST, pasando a llamarse TinyJAMBU.

Este modo de encriptación está basado en una permutación con clave, y a diferencia del modo JAMBU, tiene un tamaño menor, tanto del estado, como del bloque de mensajes. Destacar que dicha permutación con clave, puede hacerse con **una clave de 128, 192 o 256 bits.**

Pese a que la clave si puede variar, dentro del algoritmo se encuentra un estado fijo de 128 bits, del cual se acabarán sacando todas las salidas de dicho proceso.

Una vez dentro del cifrador, se encuentran las siguientes **entradas y salidas:**

- **Key:** Señal de entrada, que como su nombre indica, se trata de la clave, la cual puede ser de 128, 192 o 256 bits, como se ha dicho previamente, y debe conocerse antes de poner en funcionamiento dicho código, ya que es fundamental para proceder a un correcto funcionamiento, puesto que las permutaciones de la señal de estado con la clave son el proceso más repetido a lo largo del código. Destacar que al usar tamaños de claves diferentes, solo cambian ligeramente la cantidad de algunas permutaciones, por ejemplo, utilizando la clave de 128 bits, hay partes del código con 1024 permutaciones, sin embargo, con 192 bits de clave, estas permutaciones pasan a ser 1152, y con 256 bits de clave, serían 1280 permutaciones.

- **Nonce:** Señal de entrada, que también forma parte de la inicialización, la cual, es combinada con el registro de estado mediante puertas lógicas XOR. Siempre está formada por 96 bits y su implementación en el código se hace de 32 en 32 bits.

- **Associated Data:** Señal de entrada, que viene separada de 32 en 32 bits, la cual también se combina con el estado mediante puertas XOR. Destacar que, la última carga de datos de esta, puede no ser de 32 bits, en ese caso, solo se mezclarían los bits correspondientes. Esta señal no puede superar los 2^50 bits.

- **Plaintext:** Texto plano, o texto a encriptar, el cual se trata como entrada durante el proceso de encriptado, y como salida en el proceso de desencriptado, se transmite de forma muy parecida al Associated Data, es decir, de 32 en 32 bits, pudiendo no tener 32 bits en su última carga.

- **Ciphertext:** Texto cifrado, que de manera inversa al plaintext, funciona como salida durante el proceso de encriptado, y como entrada en el proceso de desencriptado. Igualmente se transmite de 32 en 32 bits, pudiendo tener menos bits en su última transmisión.

Tanto ciphertext, como plaintext, tampoco pueden superar los 2^50 bits, y ambos deben tener el mismo número de bits.

- **Tag:** Se trata de la etiqueta que da autenticidad al resultado obtenido, en caso de estar encriptando se trataría como una señal de salida, y en caso de estar desencriptando, se trataría como una señal de entrada, la cual sería comparada con el tag final obtenido en dicho proceso de desencriptado, dando dicha comparación la verificación de que el proceso ha funcionado correctamente. Esta señal consta siempre de 64 bits.

Partiendo de estas entradas y salidas, y del registro de estado, **el proceso está formado por los siguientes pasos:**

- **Inicialización.**

En este caso, se ponen los 128 bits del estado a ‘0’, se realizan las primeras permutaciones usando la clave y el nonce.

- **Uso del associated data.**

Ahora, en este proceso, se procede a realizar algunas permutaciones de la clave, y posteriormente, a utilizar una serie de frame bits, los cuales tienen un valor concreto para cada parte del código, en este caso su valor es de 3 (en binario “011 “). Estos frame bits se mezclan con el estado mediante puertas XOR. Por último, utilizando también puertas XOR, se combinan los 32 bits correspondientes del associated data, o menos de 32 en caso de tener un último dato menor de 32 bits, con la señal de estado, actualizando de esta manera su valor.

- **Encriptado.**

En este apartado tienen lugar 4 sucesos, el primero, es la combinación de los frame bits (con valor 5, o “101”) con el estado mediante puertas XOR, el segundo, la actualización de la señal de estado mediante las permutaciones con la clave, el tercero, una nueva actualización de la señal de estado con la combinación de esta con el plaintext, y por último, otra combinación de la señal de estado con el plaintext, esta vez, con el objetivo de obtener los bits correspondientes del ciphertext.

- **Desencriptado.**

En esta parte del proceso, también aparecen 4 procedimientos, siendo el primero y el segundo, los mismos que en el proceso de encriptado, la diferencia en este apartado, recae sobre los dos últimos. Como tercer paso, se tiene ahora la combinación de la señal de estado con el ciphertext mediante puertas XOR, para la obtención del plaintext, y como último paso, se encuentra la actualización de la señal de estado mediante la combinación de esta y los recién obtenidos bits del plaintext.

Destacar para los procesos del associated data, plaintext y ciphertext que, en caso de tener una última transmisión inferior a 32 bits, también se debe

actualizar el estado con la combinación de este con el número de bytes de dicho último dato, pudiendo ser un valor de 1,2 o 3 (en binario “01”, “10” o “11”).

- **Finalización.**

Este apartado solo se daría en el caso de estar encriptando. Sigue el siguiente orden: primero, realiza la actualización de la señal de estado con la combinación de los frame bits (7 en este caso, o “111”), luego actualiza el estado con las permutaciones de la clave, y por último, genera los 32 bits menos significativos del tag. Una vez generado estos bits del tag. Se vuelve a repetir el proceso, con la diferencia de que se llevan a cabo menos permutaciones de la clave, obteniendo finalmente, los 32 bits más significativos del tag.

- **Verificación.**

Este apartado solo es aplicable en el proceso de desencriptado. Funciona de igual manera al apartado anterior, sin embargo, en esta ocasión, una vez obtenidos los 64 bits del tag, se deben comparar con los 64 bits del tag previamente recibidos, si resultan ser iguales, se aceptaría el mensaje, verificando así, su autenticidad.

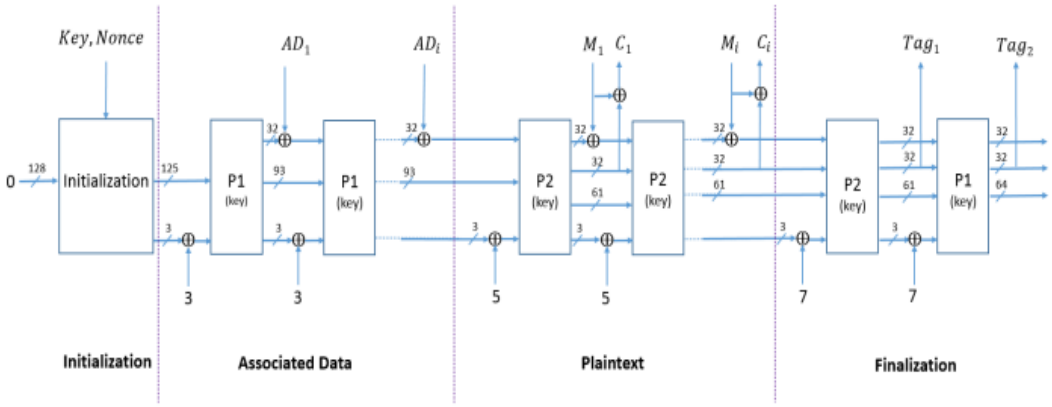

Figura 5. Diferentes procesos del código TinyJAMBU durante el encriptado.

La figura 5 es una explicación visual de lo explicado previamente, donde se pueden ver claramente los diferentes procesos del proceso de encriptado, así

como la numerosa cantidad de veces que se permuta la clave, y los diferentes usos de los frame bits, que se pueden ver en la parte inferior.

Dicha descripción se usa de forma genérica, pero hay ciertas partes del algoritmo que se actualizaron a lo largo del concurso, dando paso a dos versiones del código TinyJAMBU.

3.2. Nueva versión.

En este apartado, se tiene que destacar que TinyJAMBU tiene dos versiones, la primera, que fue la presentada primeramente en el concurso del NIST, y la segunda, la cual fue implementada una vez estaban dentro del concurso *[12]*.

Entre ambas, prácticamente no hay diferencia, toda la descripción previamente explicada es válida para ambas versiones.

Lo único que cambia entre versiones son las permutaciones de la clave de ciertas partes del código. Para ser exactos, las permutaciones realizadas en el procesamiento del nonce y del associated data. En la primera versión, en esta parte del código se realizaban 384 permutaciones, siendo así 3 vueltas de 128 permutaciones, pero debido a varias comprobaciones ante diversos ataques, decidieron aumentar estas permutaciones hasta 640 (5 vueltas de 128 permutaciones), provocando así un aumento de la seguridad del código frente a estos ataques. Bien es cierto, que también aumenta el número de recursos, pero este aumento no es significativo en comparación del significativo aumento de la seguridad.

3.3. Implementación del diseño en VHDL.

Se procede en este apartado a explicar la implementación en VHDL del diseño del código de TinyJAMBU.

Para dicha implementación, se ha decidido realizarla mediante una máquina de estados, es decir, que el código sigue un cierto proceso secuencial, en función de las entradas y salidas, las cuales se proceden a explicar.

Lo primero que se va a explicar son las entradas y salidas que se ha utilizado en el código y su finalidad.

3.3.1. Entradas y salidas.

El código con las entradas y salidas quedaría de la siguiente manera:

```
entity TinyJambu is
    Generic( velocidad : integer:= 32;
             vueltas : integer:= 3 );
    Port ( Clk : in STD_LOGIC;
           Datos_In : in STD_LOGIC_VECTOR(31 downto 0);
           Start : in STD_LOGIC;
           Reset : in STD_LOGIC;
           Mode_ED : in STD_LOGIC;
           Mode : in STD_LOGIC_VECTOR(1 downto 0);
           Answ_Key : in STD_LOGIC;
           Answ_Nonce : in STD_LOGIC;
           Answ_Tag : in STD_LOGIC;
           Answ_Ad: in STD_LOGIC;
           Ad_Fin : in STD_LOGIC;
           Ad_Bytes: in STD_LOGIC_VECTOR(1 downto 0);
           Answ_DatosIn: in STD_LOGIC;
           DatosIn_Fin : in STD_LOGIC;
           DatosIn_Bytes: in STD_LOGIC_VECTOR(1 downto 0);

           Tag_Ready1 : in STD_LOGIC;
           Tag_Ready2 : in STD_LOGIC;
           DatosOut : out  STD_LOGIC_VECTOR(31 downto 0);
           Req_Ad : out STD_LOGIC;
           Req_DatosIn : out STD_LOGIC;
           Req_Key : out STD_LOGIC;
           Req_Nonce : out STD_LOGIC;
```

```
            Req_Tag : out STD_LOGIC;
            Fin_DatosIn : out STD_LOGIC;
            Fin_Tag1 : out STD_LOGIC;
            Fin_Tag2 : out STD_LOGIC;
            Mensaje_Correcto : out STD_LOGIC);

end TinyJambu;
```

Para empezar, se explica el uso de las **constantes genéricas:**

- **Velocidad:** Valor utilizado para referirnos al número de bits que se permutan en cada una de las permutaciones al utilizar la clave, como se ve más adelante, se le dio una constante a este valor para cambiarlo en cada parte del código con solo un número, en este caso, los valores disponibles son 1, 2, 4, 8, 16 y 32.

- **Vueltas:** Número de vueltas que se deben repetir las permutaciones arriba mencionadas, en caso de que sean 1024 permutaciones, serían 8 vueltas de 128 cada una. Este valor es usado para variar las vueltas de algunas partes del código, ya que no todas pueden llegar a cambiar, en este caso este valor puede ser de 3 o de 5, ya que TinyJAMBU sacó dos versiones de su código, una en la que ciertas partes del código permutaban 384 veces, siendo el equivalente a 3 vueltas y otras en las que permutaban 640 veces, siendo el equivalente a 5 vueltas.

Se procede ahora a la explicación del uso de las **entradas y salidas:**

Entradas:

- **Clk:** Reloj del diseño.

- **Datos_In:** Valor de entrada utilizado para cargar la clave, el nonce, el associated data, el plaintext, el ciphertext, y el tag, aunque estos dos últimos solo se cargarían si se está descifrando, igualmente el plaintext solo se cargaría en caso de estar encriptando, el resto podrían darse en

ambos casos. Se introduce de 32 en 32 bits, cambiando de valor cuando sea necesario.

- **Start:** Valor de entrada usado para comenzar a funcionar el proceso.

- **Reset:** Valor de entrada usado para cancelar el proceso, esté donde esté.

- **Mode_ED:** Valor de entrada usado para saber si se está encriptando o desencriptando, cuando vale '0', se está encriptando y cuando vale '1' se encuentra desencriptando.

- **Mode:** Valor de entrada usado para saber qué modo de operación se quiere utilizar. Tiene 4 posibles valores, el "00" es para el modo de funcionamiento sin associated data y sin plaintext/ciphertext, solo con la generación del tag, el segundo modo, el "01" es para el funcionamiento solamente con associated data, pero sin plaintext/ciphertext, el tercer modo, asociado a "10" consta del funcionamiento sin associated data, pero si con plaintext/ciphertext y el último modo, el "11" está relacionado al funcionamiento tanto con associated data, como con plaintext/ciphertext.

- **Answ_Key:** Valor de entrada para poner el funcionamiento el código cuando se encuentra a la espera de recibir la clave, es decir, su puesta a 1 significa que en la señal de entrada Datos_in ya está introducido el siguiente valor de 32 bits de la clave, por ende, el código puede continuar.

- **Answ_Nonce:** Señal de entrada, cuyo funcionamiento es igual al de la señal answ_key, con la diferencia de que en esta ocasión, los 32 bits pertenecen al nonce.

- **Answ_Tag:** Señal de entrada, cuyo funcionamiento es igual al de la señal answ_key, con la diferencia de que en esta ocasión, los 32 bits

pertenecen al tag. Esta entrada solo se utiliza en el proceso de desencriptado, para pedir el tag correcto, con el que luego se comparará el tag obtenido.

- **Answ_Ad:** Señal de entrada, cuyo funcionamiento es igual al de la señal answ_key, con la diferencia de que en esta ocasión, los 32 bits pertenecen al associated data.

- **Ad_Fin:** Valor de entrada utilizado para saber cuál es el último valor de entrada del associated data.

- **Ad_Bytes:** Valor de entrada que muestra el número de bytes útiles de la última carga de datos de 32 bits del associated data mediante la entrada Datos_in. Por ejemplo, en caso de tener un último valor del associated data de “00AC345C” se utilizarían 3 bytes útiles, siendo el valor de AD_Bytes igual a “11”. Hay que destacar que como los valores del associated data se van cargando de los 32 menos significativos a los 32 más significativos, el último mensaje en caso de no tener 32 bits justos, se le añaden ceros por la izquierda hasta completar los 32 bits.

- **Answ_DatosIn:** Valor de entrada que pone el funcionamiento el código cuando se recibe el valor correspondiente de 32 bits del plaintext/ciphertext en la señal de entrada Datos_in.

- **DatosIn_Fin:** Valor de entrada que avisa de la última carga de datos del plaintext/ciphertext.

- **DatosIn_Bytes:** Valor de entrada con un funcionamiento igual al de la señal de entrada ad_bytes, pero respecto a la carga de datos del plaintext/ciphertext.

- **Tag_Ready1:** Entrada utilizada para dar continuidad al código una vez están listos los 32 bits menos significativos del tag. Se creó esta entrada para poder parar el código una vez obtenido por completo el

ciphertext/plaintext y así poder mostrar los últimos 32 bits del plaintext/ciphertext sin que el código continuara.

- **Tag_Ready2:** Entrada utilizada para dar continuidad al código una vez están listos los 32 bits más significativos del tag. Se creó esta entrada para poder parar el código una vez obtenido los 32 bits menos significativos del tag sin que el código continuara.

Salidas.

- **Datos_Out:** Valor de salida de 32 bits, en el que se cargan los valores obtenidos durante el proceso, en caso de estar encriptando, se cargarían los valores del ciphertext, y posteriormente los del tag, y en caso de estar desencriptando, se cargarían los del plaintext, y posteriormente los del tag.

- **Req_Ad:** Valor de salida que muestra cuando el código necesita la carga de datos del associated data para poder seguir con el código.

- **Req_DatosIn:** Valor de salida que, en este caso, pide la carga de datos del plaintext/ciphertext para seguir con el resto del código.

- **Req_Key:** Valor de salida, el cual, pide la carga de datos de la clave para seguir con el resto del código.

- **Req_Nonce:** Valor de salida, el cual, pide la carga de datos del nonce para seguir con el resto del código.

- **Req_Tag:** Valor de salida, el cual, pide la carga de datos del tag para seguir con el resto del código. Esta salida, como previamente pasaba con la entrada answ_tag, solo se utilizará en el proceso de desencriptado.

- **Fin_Datosin:** Es una salida utilizada para saber en qué momento se ha terminado el proceso de encriptado/desencriptado, se pone a uno justo después de cargar en datos_out los últimos 32 bits del plaintext/ciphertext, en función de si se encuentra encriptando o desencriptando.

- **Fin_tag1:** En este caso, esta salida es utilizada en el momento en que se han obtenido los 32 bits menos significativos del tag, poniéndose a uno cuando en la salida datos_out esté cargado dicho valor del tag.

- **Fin_tag2:** En este caso, esta salida es utilizada en el momento en que se han obtenido los 32 bits más significativos del tag, poniéndose a uno cuando en la salida datos_out esté cargado dicho valor del tag.

- **Mensaje_Correcto:** Valor de salida únicamente utilizado durante el proceso de desencriptado, cuya puesta a uno significa que el tag generado, y el tag recibido mediante la entrada datos_in son iguales, verificando así un correcto funcionamiento de dicho proceso de desencriptado.

Una vez explicadas todas las entradas y salidas del código, se procede a explicar el funcionamiento del diseño.

3.3.2. Permutación de la clave.

Para empezar, se tendría la acción que más se repite a lo largo del proceso de encriptado/desencriptado, que se trata de la permutación de la clave. Dicha clave se va introduciendo mediante una serie de operaciones con puertas lógicas, a un registro de estado de 128 bits. Esta señal, va a ser muy importante para el resto del código, ya que es de donde se sacan el resto de salidas, como el tag, o el texto cifrado. Estas permutaciones van ocurriendo introduciendo un bit, el cual es resultado de las operaciones realizadas entre un bit diferente de la clave, del menos significativo al más significativo, junto a

diferentes puertas lógicas, esto da lugar a un bit resultante, que es el que se va introduciendo en la señal de estado. Destacar que tras 128 permutaciones, se ha cumplido una vuelta, y se volvería a introducir el bit menos significativo de la clave.

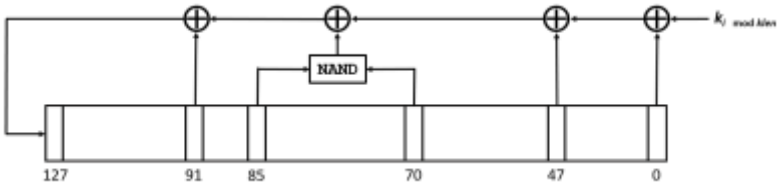

Figura 6. Permutaciones de la clave.

En la figura 6 se puede ver la explicación del TinyJAMBU, en la que se puede observar lo previamente dicho, que sería la explicación de dichas permutaciones, así como las puertas lógicas implicadas, que serían, una puerta nand entre los bits 85 y 70, y luego, una puerta Xor entre el resultado de la puerta nand, los bits 91,47 y 0, y el bit correspondiente de la clave. Destacar que a lo largo del código, el número de permutaciones varía.

Para poder implementarlo en VHDL, simplemente se tuvo que hacer una adaptación del código, introduciendo la constante velocidad, previamente explicada, para así poder hacer la permutaciones de 1 hasta 32 bits.

```
for l in 0 to (128 - (velocidad+1) ) loop
  state(l)<= state(l+velocidad);
end loop;
for m in 0 to (velocidad - 1) loop
  state((128 - velocidad)+m)<= clave(0+m) xor state(0+m) xor state(47+m) xor (state(70+m) nand state(85+m)) xor state(91+m)
end loop;
clave((127 - velocidad) downto 0)<=clave(127 downto velocidad);
clave(127 downto (128 - velocidad))<=clave((velocidad - 1) downto 0);
i<= i+1;
if i=(128/velocidad -1) then
  j<= j+1;
  i<= 0;
  if j=7 then
    estado<=s5;
    j<=0;
    i<=0;
  end if;
end if;
```

Figura 7. Código de permutaciones de la clave.

Como se puede ver en la figura 7, la implementación está formada por varios bucles for, los cuales llevan a cabo el mismo proceso una y otra vez, solamente aumentando el valor correspondiente, el primero va desplazando la señal state, que es la que tiene que permutar y el segundo, va introduciendo por los bits más significativos los nuevos valores de la señal state, que es el resultado de la combinación de puertas lógicas previamente visto.

Por último, también se desplaza la clave, para que en el siguiente ciclo de reloj, introduzca en la señal state los valores pertinentes de la clave.

En la señal clave, debe estar guardada la clave del proceso de encriptado/desencriptado, proceso que se ha debido hacer al inicio del código.

3.3.2.1. Análisis de recursos para diferentes velocidades.

Destacar que todo está hecho en función de la constante velocidad, como se puede ver en la figura 7, la cual, marca cuantos ciclos de reloj son necesarios para realizar todas las permutaciones de la clave.

Por ejemplo, si se necesitaran 1024 permutaciones, que serían 8 vueltas de 128 permutaciones, a velocidad 1, tardaría 1024 ciclos de reloj, en cambio, si se eligiese velocidad 32, tardaría solamente 32 ciclos de reloj.

Para ver realmente que velocidad es la más eficiente, se procede a hacer diferentes pruebas de síntesis.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	570	126800	0%
Number of Slice LUTs	1070	63400	1%
Number of fully used LUT-FF pairs	569	1071	53%
Number of bonded IOBs	92	210	43%
Number of BUFG/BUFGCTRLs	1	32	3%

Figura 8. Recursos para velocidad 1.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	569	126800	0%
Number of Slice LUTs	1066	63400	1%
Number of fully used LUT-FF pairs	568	1067	53%
Number of bonded IOBs	92	210	43%
Number of BUFG/BUFGCTRLs	1	32	3%

Figura 9. Recursos para velocidad 2.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	571	126800	0%
Number of Slice LUTs	1032	63400	1%
Number of fully used LUT-FF pairs	567	1036	54%
Number of bonded IOBs	92	210	43%
Number of BUFG/BUFGCTRLs	1	32	3%

Figura 10. Recursos para velocidad 4.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	570	126800	0%
Number of Slice LUTs	1037	63400	1%
Number of fully used LUT-FF pairs	566	1041	54%
Number of bonded IOBs	92	210	43%
Number of BUFG/BUFGCTRLs	1	32	3%

Figura 11. Recursos para velocidad 8.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	575	126800	0%
Number of Slice LUTs	1071	63400	1%
Number of fully used LUT-FF pairs	566	1080	52%
Number of bonded IOBs	92	210	43%
Number of BUFG/BUFGCTRLs	1	32	3%

Figura 12. Recursos para velocidad 16.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	573	126800	0%
Number of Slice LUTs	1095	63400	1%
Number of fully used LUT-FF pairs	565	1103	51%
Number of bonded IOBs	92	210	43%
Number of BUFG/BUFGCTRLs	1	32	3%

Figura 13. Recursos para velocidad 32.

Como se puede ver en las diferentes figuras 8, 9, 10, 11, 12 y 13 la diferencia de recursos entre las distintas velocidades es muy pequeña, teniendo como único cambio significativo al número de LUTs, que fluctúa en torno al 5%, un cambio insignificante, por lo que si se quiere ser más eficiente, se debería optar por el que tarde menos ciclos de reloj, que en este caso, sería el de velocidad 32. A continuación, se puede ver más detenidamente.

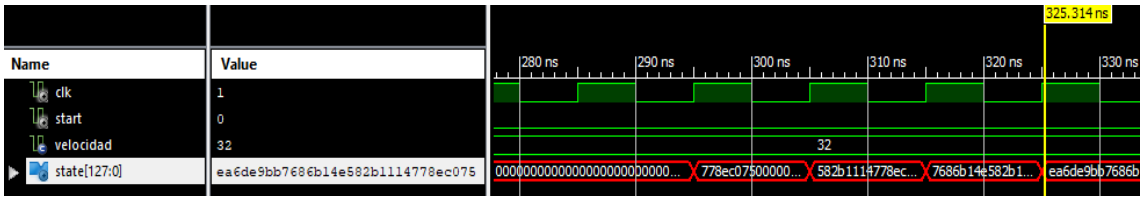

Figura 14. Ejemplo para velocidad 32.

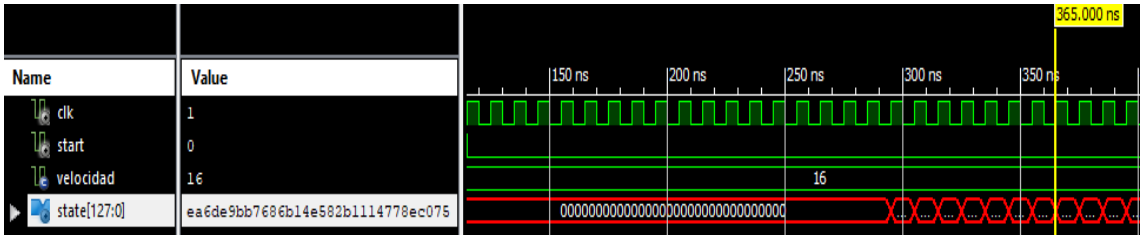

Figura 15. Ejemplo para velocidad 16.

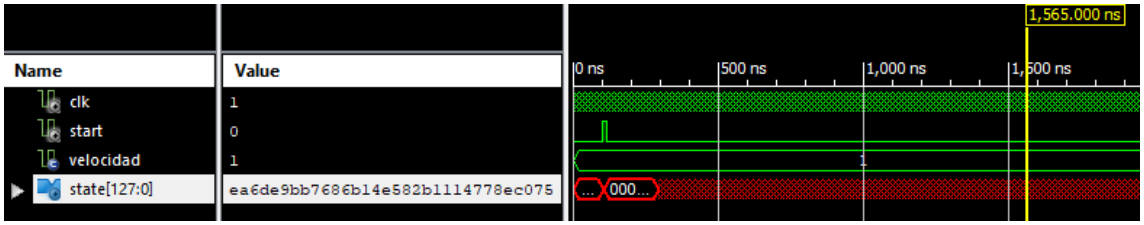

Figura 16. Ejemplo para velocidad 1.

En la figura 14 se puede ver un ejemplo de los ciclos de reloj que se tardan en realizar 128 permutaciones de la señal state, que corresponde a la señal de 128 bits de estado, para un mensaje cualquiera. Como se puede ver, con un reloj de 100 Mhz (frecuencia = 10 ns) tardaría 4 ciclos de reloj, o 40 ns, todo esto para un valor de velocidad igual a 32.

Por otra parte, se obtiene la figura 15, que comprueba el tiempo de 128 permutaciones para una velocidad 16. En este caso se tienen 8 ciclos de reloj, que para una frecuencia de 10 ns implica 80 ns, como era de esperar el doble que con velocidad 32.

Por último, destacar el tiempo que se tarda para una velocidad 1, que como se puede ver en la figura 16, se trata de unos 1280 ns, o en su defecto, de 128 ciclos de reloj, que como previamente se ha visto, es el valor correspondiente.

Con estas tres figuras, está más que clara la afirmación de que la velocidad 32 es la más eficiente, puesto que los ciclos de reloj son significativamente menores, a diferencia de los recursos necesarios.

3.3.3. Entrada de datos.

Se procede ahora a explicar cómo funciona la petición de datos, su temporización, y el funcionamiento de las señales de control.

En este caso, se necesita la entrada de datos antes de propiamente empezar el funcionamiento, ya que lo primero de todo es comunicar cual será la clave y el nonce, en caso de estar encriptando. En caso de estar desencriptando, habría que sumar a lo anterior los 64 bits del tag.

Para ello, como se ha explicado anteriormente, se tienen las entradas answ y las salidas req. Estas salidas, como por ejemplo, la llamada req_key, son las responsables de indicar cuál es el valor de entrada necesario, en este caso, si req_key se pusiera a '1' indicaría que se requiere de la entrada de la clave, y sus correspondientes bits (128, 192 o 256). Por otra parte, las entradas answ se encargan de indicar que ahora mismo se puede realizar la carga de los datos correspondientes. Dicha carga de datos se realiza de 32 en 32 bits mediante la entrada Datos_In.

```
req_key<='1';
  if answ_key='1' then
    clave(32*k+31 downto 32*k)<= Datos_In(31 downto 0);
    k<=k+1;
    req_key<='0';
    if k=3 then
      estado<=s2;
      k<=0;
    end if;
  end if;
```

Figura 17. Código de petición de datos.

Como se puede ver en la figura 17, se trata de una parte del código en la que se puede ver con mucha claridad el funcionamiento del mismo, siendo la entrada answ_key la que hace funcionar el código durante un clico de reloj, haciendo que se guarde en la señal clave los 32 bits menos significativos de dicha clave, pero todo esto no podría funcionar si previamente no se le pide mediante la señal de salida req_key que se necesita dicha clave.

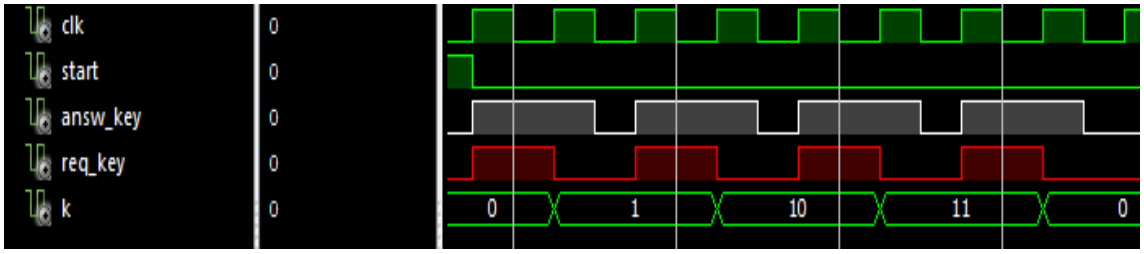

Figura 18. Señales de control I.

En la figura 18, se puede ver la temporización, y se ve, como al ponerse a ‘1’ req_key, acto seguido lo hace también answ_key, debido a que en el test bench se ha indicado de esa manera. Una vez puesto answ_key a ‘1’ se está indicándo que empiece a funcionar el código, acto que se repite las veces pertinentes, en este caso, serían 4 veces, ya que al introducir los datos de 32 en 32 bits, se necesita 4 ciclos para introducir los 128 bits de la clave.

Para el resto del código, sería un funcionamiento prácticamente igual, en el que según en qué parte el algoritmo se encuentre, se introducen unos datos u otros, y se utilizan unas entradas answ y salidas req, u otras.

De esta manera, se consigue solucionar una de las tareas más tediosas del código, de una manera sencilla y eficaz, ya que funciona instantáneamente sin perder muchos ciclos de reloj.

Ahora bien, para las señales del associated data, y del plaintext/ciphertext, hacen falta otras dos señales de control extra, que en este caso se llaman ad_fin o datosin_fin, y ad_bytes o datosin_bytes. Estas señales se han utilizado para marcar la última entrada de datos de dichas señales, pudiendo ser de 32 bits, o menos, en caso de que la señal no tenga un número de bits múltiplo de 32. Por ejemplo, la puesta a ‘1’ de ad_fin significa se han enviado los últimos bits de dicha señal, y la señal ad_bytes se encarga de mostrar de cuantos bits será dicha transmisión, de la siguiente manera, si ad_bytes vale “00” serán 32 bits de transmisión, si tiene el valor “01”, serán 8 bits, si es igual a “10” serán 16 bits, y por último, si tiene el valor “11” serán 24 bits.

Destacar que las señales de nonce y key no tienen una señal que marque su final, ya que a diferencia del associated data, y del plaintext/ciphertext, estas tienen un tamaño fijo, y no variable.

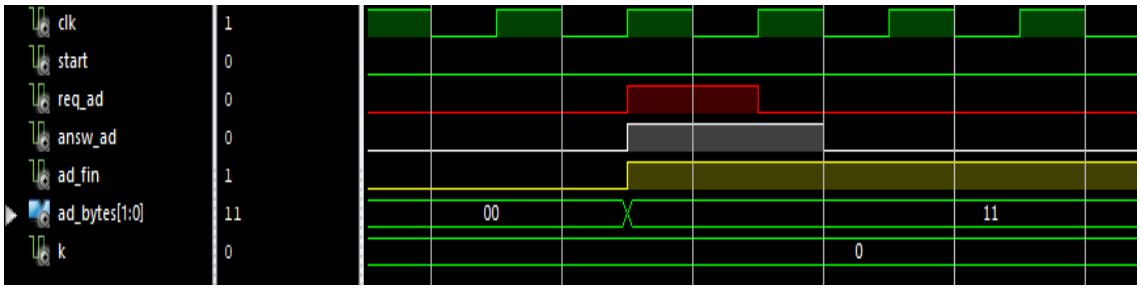

Figura 19. Señales de control II.

En la figura 19 se puede ver la temporización de las señales, siendo otra vez, la señal req_ad la que marca la necesidad de nuevos datos del associated data, y la señal answ_ad la que pone en funcionamiento en código, sin embargo, en esta ocasión también se pone a '1' la señal ad_fin, marcando la última transmisión del associated data, siguiendo así el código con el siguiente proceso, y además, de esta última transmisión se puede saber que se trata de 24 bits, y no de 32, ya que la señal ad_bytes adquiere el valor "11".

A continuación, se pasa a explicar los diferentes modos de operación que están disponibles en esta implementación del código del TinyJAMBU.

3.4. Modos de operación.

Los modos de operación se pueden dividir en función de las siguientes dos entradas, mode_ed (entrada de un bit) y mode (entrada de dos bits), la primera se encarga de decidir si el algoritmo se encuentra encriptando o desencriptando, y la segunda se encarga de decir en qué modo de funcionamiento se encuentra el algoritmo.

Ahora bien, por lo tanto, en total habría 6 modos de funcionamiento diferentes, ya que los dos modos que se tienen sin plaintext/ciphertext no se ven afectados por el valor de la entrada mode_ed.

Los diferentes valores de estas entradas cambian significativamente el resto del código, como es lógico, dicho código sin el proceso del associated data, o sin el proceso de encriptado/desencriptado, es mucho más rápido, y consume menos recursos. Aunque cómo se puede imaginar, este código está hecho para funcionar con todas las entradas disponibles, pero admite esas variantes, las cuales, dotan al código de un número de aplicaciones mayor, así como mayor versatilidad.

Como se ha explicado anteriormente, la entrada mode_ed marca si el algoritmo se encuentra encriptando, o desencriptando, siendo '0' encriptando y '1' desencriptando.

En el caso de la entrada mode, se tienen cuatro posibles opciones:

- "00" = Sin plaintext/ciphertex, y sin associated data
- "01" = Sin plaintext/ciphertext, pero sí con associated data
- "10" = Con plaintext/ciphertext, pero sin associated data
- "11" = Con plaintext/ciphertext y con associated data.

```
if mode="00" then
  estado<=s21;
elsif mode="10" then
  if mode_ED='0' then
    estado<=s17;
  else
    estado<=s12;
  end if;
else
  estado<=s8;
end if;
```

Figura 20. Código de elección del modo de operación.

En la figura 20 se puede ver una parte del código en la que en función del valor de la señal de entrada **mode**, procede a continuar la máquina de estados en un estado u otro.

Esto, al ser realizado mediante una máquina de estados, no tuvo mucha complicación, ya que solo hubo que arreglar el código para que en función del modo en que el algoritmo se encuentre, pase por unas determinadas partes del código. Como se puede imaginar, cuantas más cosas tiene el código, más recursos consume, y más ciclos de reloj se necesitan para procesarlo, por eso, el modo “00” es el más rápido de todos, si bien es cierto, que al no tener plaintext, no se encriptará nada.

Una vez conocido los diferentes modos de funcionamiento se procede a realizar varias verificaciones con dichos modos, para asegurar así, que dicho código funciona correctamente.

3.5. Verificaciones.

Aquí se va a exponer algunas simulaciones realizadas, para poner en valor todo lo previamente explicado, estas simulaciones están basadas en algunas implementaciones realizadas por Carlos Jesús, basadas en códigos que se pudieron encontrar en la página de Github *[13]*, las cuales se han utilizado de ejemplo para garantizar el funcionamiento de los diferentes modos de funcionamiento.

Todas estas verificaciones son llevadas a cabo mediante test bench del código, las cuales, son realizadas con la ayuda de la aplicación de Xilink junto a su respectivo simulador.

Se va a garantizar el funcionamiento del código mediante las 5 verificaciones siguientes:

- **1º Verificación:**

La primera de todas, constará de la versión más sencilla, que como se ha dicho anteriormente, es la equivalente al “00” de la entrada mode, en este caso, el algoritmo se encuentra tanto sin associated data, como sin plaintext/ciphertext, por ende, lo único que tiene que generar es la señal de salida del tag, y solo

contará con las entradas de la clave y el nonce, ambas imprescindibles en cualquier modo.

```
#### Msg   1
key      = C45C979BA3D56832F7BC492BB9CCFD18
npub     = B0ABE58260ABB1D7B5AA3A1B
ad       =
pt       =
ct       =
tag      = 376DAB83BE4F0B7E
```

Figura 21. Primer mensaje de verificación.

Como se puede ver en la figura 21, se trata del primer mensaje de verificación. Destacar que estas implementaciones están basadas en la primera versión de TinyJAMBU, es decir, que algunas partes solo permutan 384 veces, como el funcionamiento del código es el mismo en ambas versiones, demostrando un correcto funcionamiento de la primera versión, también se está demostrando para la segunda versión.

Se realizó la simulación con estos datos, y se obtuvo la figura 22.

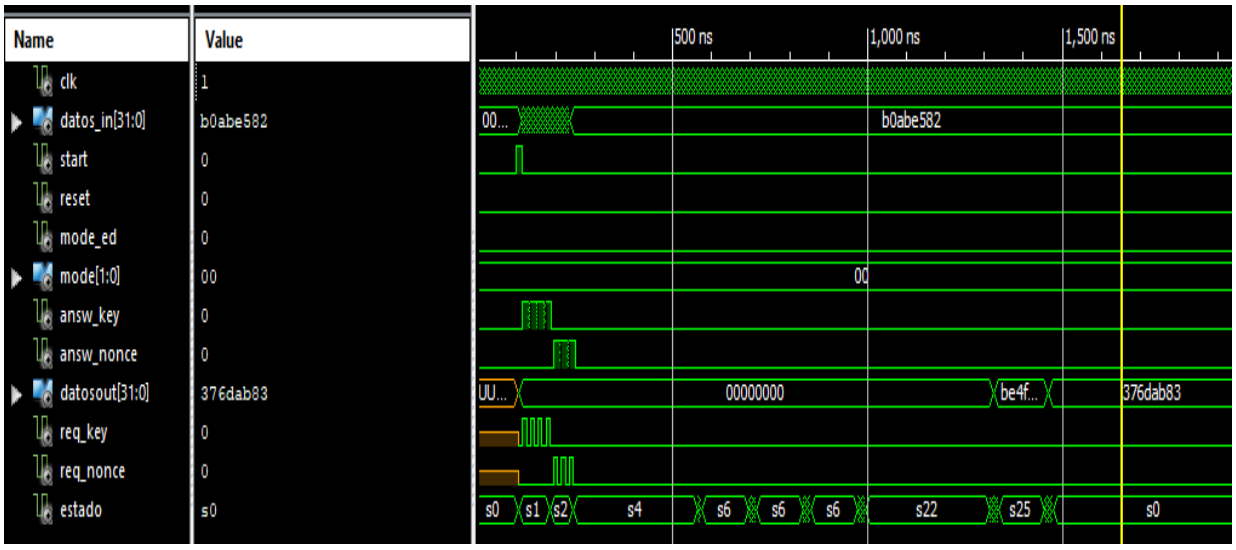

Figura 22. Simulación primer mensaje.

De dicha figura 22 se puede obtener bastante información, por ejemplo, con las señales req_key, req_nonce, answ_key y answ_nonce se puede ver cómo va

pasando por los distintos procesos del código. La señal de estado también muestra dicho avance, y se puede ver cómo pasa rápidamente del estado S6 al S22, saltándose así gran parte del código, puesto que no tiene sentido pasar por el proceso del associated data, o del plaintext/ciphertex, ya que no se dispone de estos.

Por último, recalcar que en la señal de salida de datos, se puede ver el valor del tag, que es la única señal de salida que se tiene, en este caso como último valor se tiene en formato hexadecimal “376dab83”, y se puede comprobar que se trata de los 32 bits más significativos del tag, dando lugar a un correcto funcionamiento del código.

Este modo de funcionamiento tarda en torno 1400 ns, o 140 ciclos de reloj, en realizar todo el proceso, desde que se pone a ‘1’ la señal de start, y puesto que no tiene ningún valor que varíe en tamaño, se trata de un valor fijo.

Se procede ahora a realizar la segunda verificación con el siguiente modo de funcionamiento.

- **2º Verificación:**

En este caso se procede a realizar la simulación con el segundo modo de funcionamiento, que es el de valor “01” de la entrada mode. Como se puede ver, el algoritmo se encuentra también sin plaintext/ciphertext, pero si con associated data, dando lugar a un mayor consumo de recursos, y a un mayor tiempo de procesamiento del código. Se procede ahora, a simular el mensaje que aparece en la figura 23.

```
#### Msg  81
key     = C2FFD690EC4A5È3C634F1E55D2AA9B7C
npub    = 156C4BB9CAF92F165A0F7516
ad      = 66D69BF378F2CF9CEA
pt      =
ct      =
tag     = AEF4D42BCDC71007
```

Figura 23. Segundo mensaje de verificación.

Del mensaje de la figura 23, se obtiene la siguiente simulación.

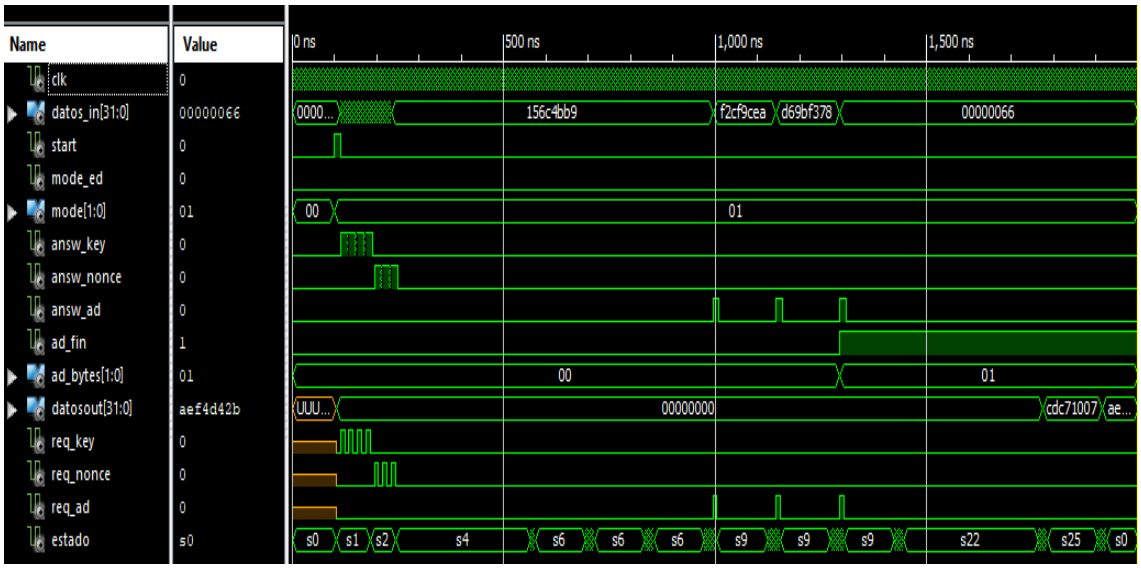

Figura 24. Simulación del segundo mensaje.

Como se puede ver en la simulación de la figura 24, el tiempo total para procesar el código ahora aumenta siendo en este caso de unos 1800 ns, o 180 ciclos de reloj, desde que la señal start se pone a '1'. La diferencia con el modo anterior, es que el tamaño de este no es fijo, ya que el valor del associated puede variar hasta los 2^50 bits, sin embargo, como se puede ver el tiempo de diferencia que hay entre dos puestas a '1' de la señal req_ad, se puede saber el tiempo según su tamaño. Por lo tanto, según la figura 24, se puede ver una diferencia de 150 ns, o 15 ciclos de reloj para cada entrada de 32 bits del associated data.

Como en la primera simulación, se puede ver las diferentes señales de temporización, que se encargan de hacer funcionar el código, en este caso, se incluye la señal req_ad y answ_ad, puesto que ahora sí se tiene associated data, destacar también, que en esta ocasión se transcurre por más estados que en la simulación anterior.

Se sigue de esta manera con la siguiente verificación del código.

- **3º Verificación:**

En esta verificación se desea asegurar un correcto funcionamiento del modo de funcionamiento con plaintext/ciphertext, pero sin associated data, en este caso, si se puede estar tanto encriptando como desencriptando, siendo en ambos resultados muy parecidos. Para realizar la simulación se parte del siguiente mensaje de ejemplo de la figura 25.

```
#### Msg   5
key     = 364BE15D4592E38C21D365B9A1E3C68C
npub    = F4387308C640E16E1E531EC3
ad      =
pt      = 4BF6839D854294AA69036467381F864E8CF30C
ct      = 1C2DC77B0705E657773A380937F56F1CF447E6
tag     = 6F09643481320F23
```

Figura 25. Tercer mensaje de verificación.

Una vez introducido los datos de la figura 25 en el test bench, se obtiene como resultado la siguiente simulación.

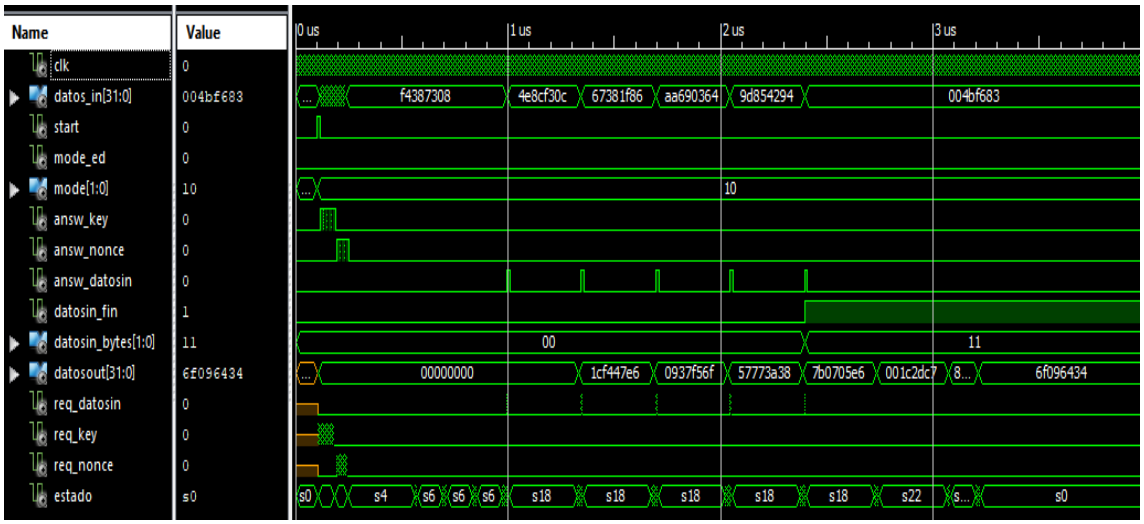

Figura 26. Simulación del tercer mensaje.

En esta ocasión, como se puede ver en la figura 26, se tienen datos de entrada a cifrar, que serían equivalentes al plaintext, debido a que esta parte del

proceso es más tediosa que la del associated data, se tarda hasta aproximadamente 3150 ns en realizar el código completo. En la figura 26, también se puede ver la diferencia de tiempo entre dos puestas a '1' de la señal req_datosin, calculando así el tiempo promedio que tarda en hacer un ciclo de encriptado de datos para el plaintext, que debe ser prácticamente igual al tiempo para su desencriptado. Por lo tanto, como indica la figura 26, se estaría hablando de unos 360 ns, que equivaldrían a 36 ciclos de reloj.

Sin demora, mediante la señal de salida datos_out se pueden ver también los diferentes valores del texto cifrado, los cuales coinciden con el mensaje de ejemplo, dando como resultado la verificación del correcto funcionamiento del código. Recalcar también que en esta simulación se ha encriptado una serie de datos, pero también se podría haber desencriptado cambiando el valor de la entrada mode_ed de '0' a '1'.

Se continúa ya con las dos últimas verificaciones, que son realmente las que más importan, puesto que van a contar con todos los datos posibles.

- **4º Verificación:**

En esta verificación ya se pueden ver todos los datos posibles, es decir, se cuenta con la clave y el nonce, como en todas las verificaciones, y también con el associated data, y el plaintext/ciphertext, por lo tanto, se estaría hablando del valor "11" de la entrada mode. Ahora pues, se va a proceder a hacer dos verificaciones con el mismo mensaje, en una se hará encriptando e introduciendo como datos de entrada los del plaintext o texto a cifrar, y en la otra se desencriptará dicho mensaje, teniendo como datos del entrada los del ciphertext, o texto cifrado.

Se realiza pues la simulación del siguiente mensaje de la figura 27.

```
#### Msg   74
key      = D66CF0D25C794F3F020CE04B88713F8A
npub     = 8E46B6708591323D57137D56
ad       = 66617E779364475048F503CB25ECDAA92E017B
pt       = 122349BAC332AD892FC5897ECACDE8F2
ct       = 4583018674EB56413A7C7770D7C00ED0
tag      = D2673C823046D638
```

Figura 27. Cuarto mensaje de verificación.

Se procede a realizar la simulación mediante el uso de un test bench, obteniendo lo siguiente.

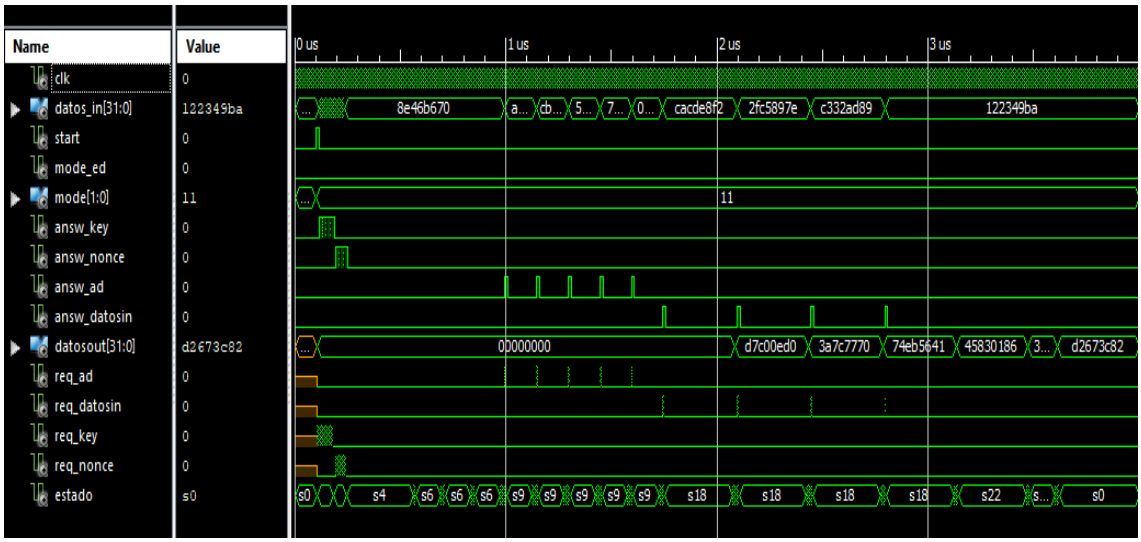

Figura 28. Simulación del proceso de encriptado del cuarto mensaje.

En la simulación de la figura 28 se destaca lo siguiente:

-Valor ‘0’ de la entrada mode_ed, la cual indica que se está encriptando.

-Mayor tiempo de realización del código, en este caso solo 3500 ns, debido a que se tienen 5 entrada de datos por parte del associated data, y 4 por parte del plaintext.

-Un correcto funcionamiento, puesto que los valores obtenidos a la salida concuerdan con los del mensaje de ejemplo.

Por último, se procede a realizar la última simulación, que es el mismo mensaje, pero cambiando ahora el valor de la entrada mode_ed, para proceder a desencriptar.

- **5º Verificación:**

Se realiza la misma simulación del mensaje de la figura, llevando a cabo ahora el desencriptado, y obteniendo la siguiente simulación.

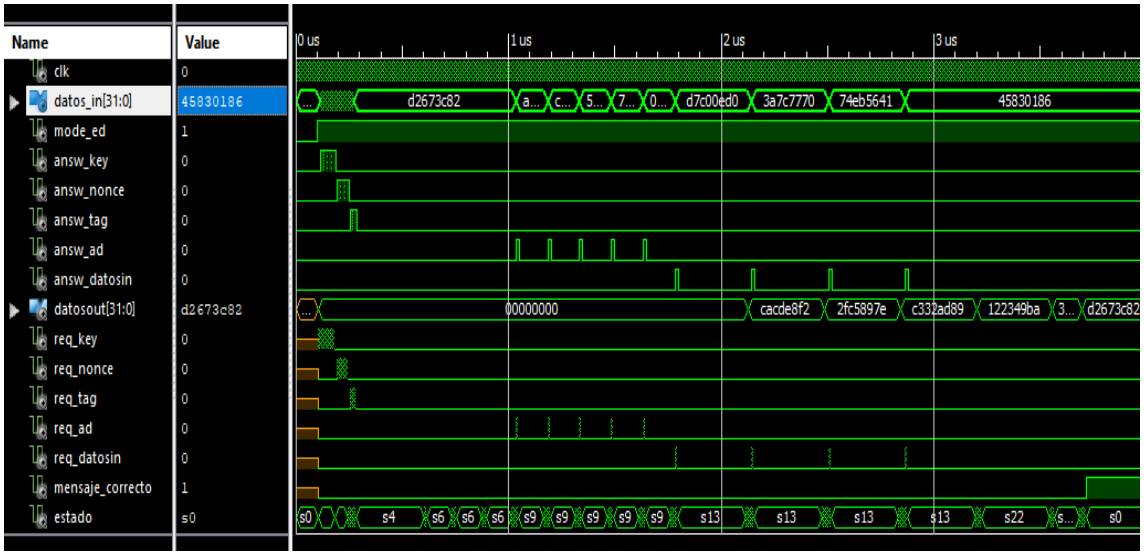

Figura 29. Simulación del proceso de desencriptado del cuarto mensaje.

De la simulación de la figura 29 se obtiene la siguiente información:

-Lo primero que se puede ver es que en esta ocasión se tiene una entrada adicional**,** que se trata del tag o etiqueta, valor que se utiliza al final del código para asegurar que el tag de entrada y el tag que genera el código son los mismos, dando veracidad al correcto funcionamiento del código.

-Recalcar que se cumple lo anteriormente comentado, ya que la entrada mensaje_correcto está a 1, y es la señal encargada de verificar dicha comparación entre tags.

-También se puede apreciar algo más de tiempo para la ejecución del código, siendo ahora el código de 3600 ns, debido a dicha pedida de datos de entrada extra.

Una vez terminada esta simulación, ya se ha terminado de hacer todas las verificaciones, y como se ha podido ver, el código funciona a la perfección, ya que en cada una de las verificaciones se ha obtenido los datos de salida correctos.

3.5.1. Análisis de los ciclos de reloj para cada proceso.

Se obtiene como referencia la figura 30, en la que se puede ver el número de ciclos que se tarda en realizar cada proceso del código, para las dos versiones del mismo, destacar que el tiempo es el necesario para una única realización del proceso, es decir, si se tiene 5 entradas de datos del plaintext, sería la equivalencia de realizar 5 veces el proceso de encriptado. Recalcar también que los cambios de versiones, solo afectan al proceso del nonce, del associated data y al del tag.

Proceso	Versión (1=384 permutaciones ó 2=640)	Tiempo en realizar el proceso (frecuencia=10ns)	Número de ciclos de reloj	Figura en la que puede apreciarse
Clave	1	330 ns	33	31
	2	330 ns	33	
Nonce	1	140 ns	14	32
	2	220 ns	22	33
Associated data	1	150 ns	15	34
	2	230 ns	23	35
Plaintext/ciphertext	1	360 ns	36	36
	2	360 ns	36	
Tag	1	490 ns	49	37
	2	570 ns	57	38

Figura 30. Tabla sobre los ciclos de reloj de cada proceso.

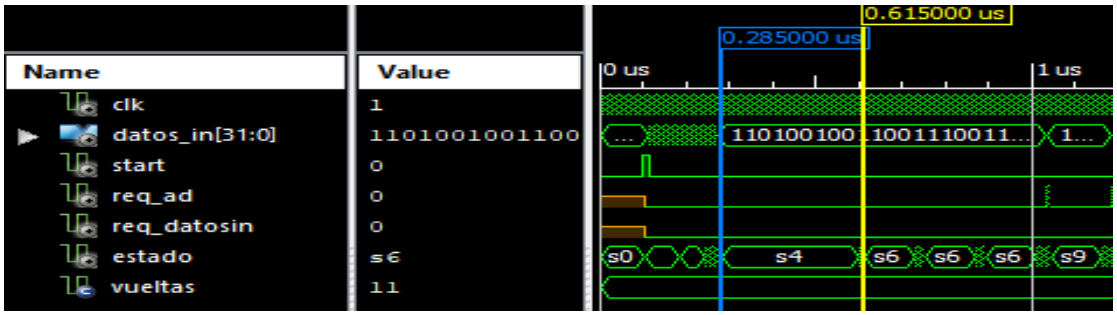

Figura 31. Tiempo del proceso de la clave para ambas versiones.

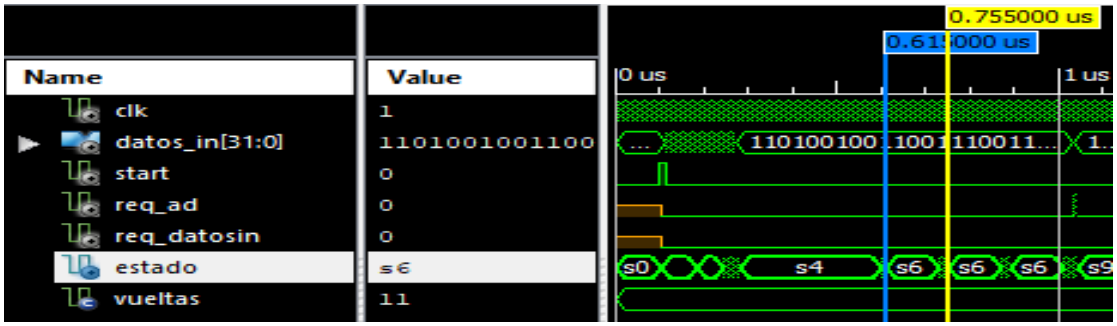

Figura 32. Tiempo del proceso del nonce para la primera versión.

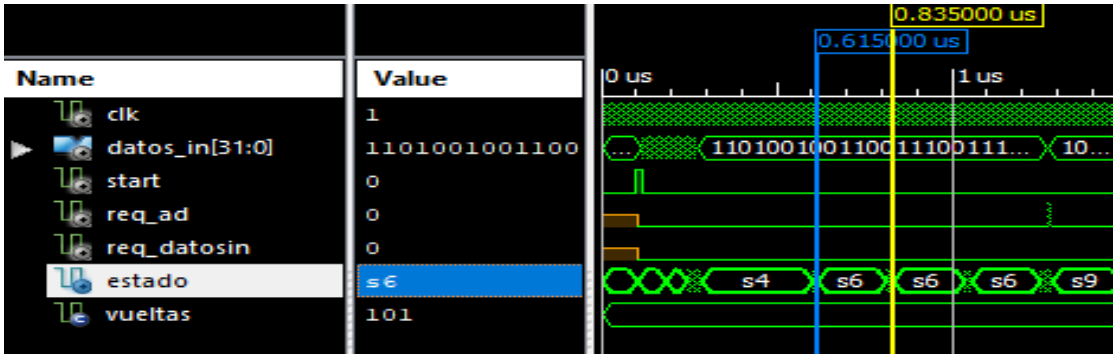

Figura 33. Tiempo del proceso del nonce para la segunda versión.

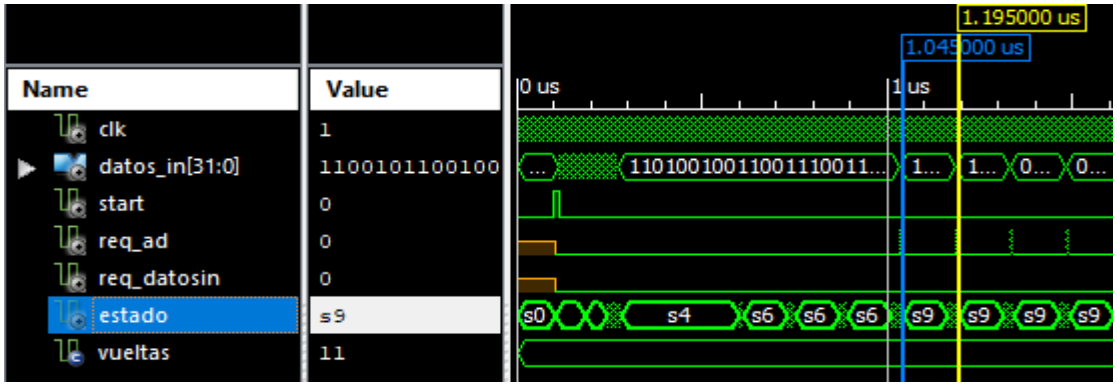

Figura 34. Tiempo del proceso del associated data para la primera versión.

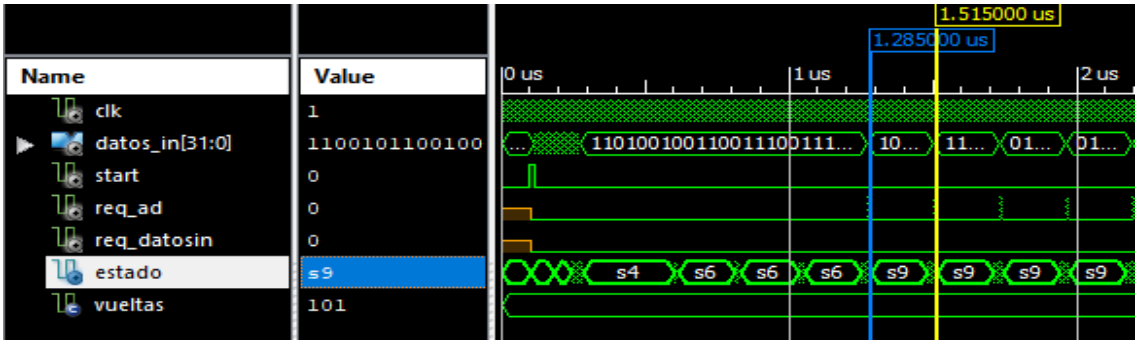

Figura 35. Tiempo del proceso del associated data para la segunda versión.

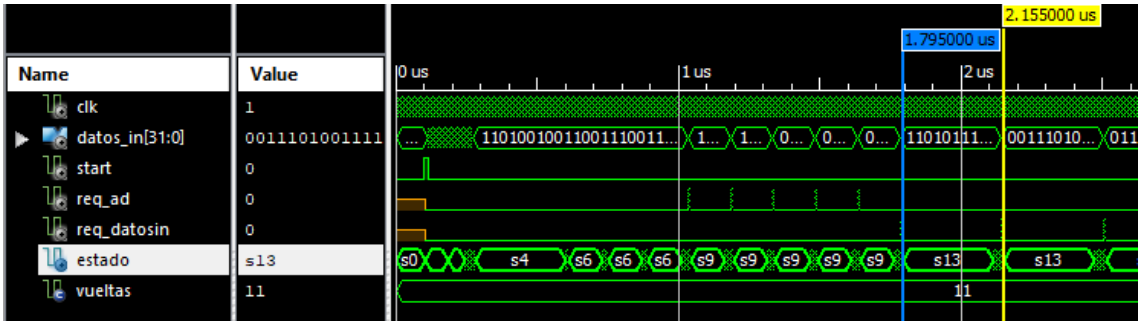

Figura 36. Tiempo del proceso del plaintex/ciphertext para ambas versiones.

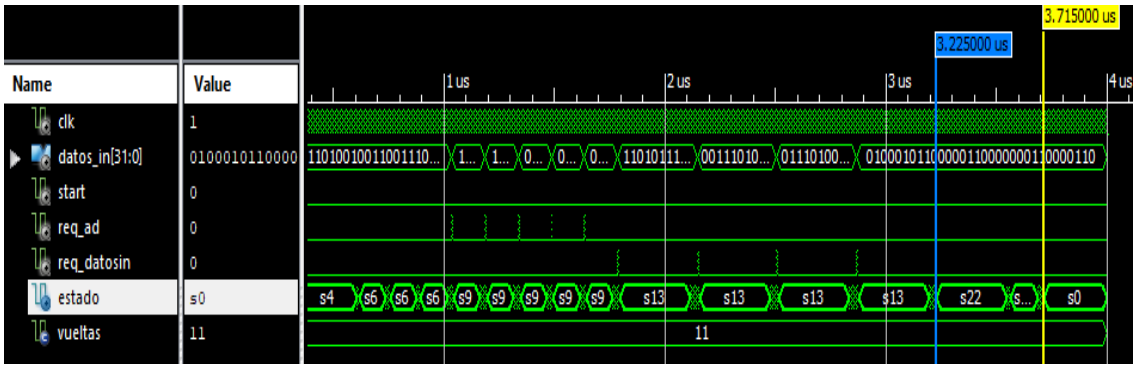

Figura 37. Tiempo del proceso del tag para la primera versión.

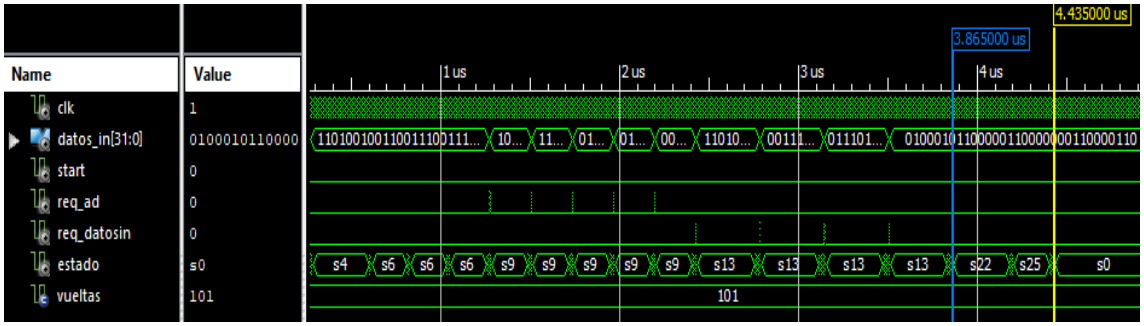

Figura 38. Tiempo del proceso del tag para la segunda versión.

Una vez vistas todas las figuras, se puede saber perfectamente el número de ciclos transcurrido para cada parte del código, pudiendo saber de esta manera el tiempo que se va a tardar en encriptar/desencriptar cada mensaje.

Una vez demostrado todo el funcionamiento del código, su diseño, y sus diferentes procesos y verificaciones, se proceden a realizar las pruebas prácticas, para ver un funcionamiento del código más visual.

4. Prueba experimental del funcionamiento del algoritmo Tinyjambu.

El primer objetivo, fue realizar el algoritmo del TinyJAMBU en leguaje vhdl, una vez conseguido ese objetivo, se llevó a cabo otro proyecto para hacer un ejemplo experimental y más visual con dicho código, para ello, se volvió a utilizar la aplicación de Xilinx, con la diferencia de que ahora el código del TinyJAMBU forma parte de este proyecto como un componente más.

4.1. Entradas y salidas.

Entradas:

- **Clk:** Reloj del diseño.
- **Reset:** Señal que puede resetear el código, da igual donde se encuentre.
- **Init:** Señal que inicializa el funcionamiento del código.
- **Next_do:** Señal utilizada para continuar con el código, y mostrar por los leds de 7 segmentos el siguiente valor de salida del código TinyJAMBU.
- **Mode:** Señal utilizada para indicar el modo de operación del código TinyJAMBU.
- **Mode_ed:** Señal utilizada para indicar si el algoritmo se encuentra encriptando o desencriptando.
- **Select_prueba:** Señal encargada de seleccionar el mensaje de prueba que se va a mostrar, es decir, el código cuenta con varios mensajes de prueba, y esta señal es la que se encarga de elegir cual es el que va a ser mostrado en los leds de 7 segmentos.

Salidas:

- **Led_mc:** Señal de salida que se pone a '1' cuando así lo hace la señal de mensaje_correcto del código TinyJAMBU, se utiliza para verificar un correcto desencriptado del mensaje seleccionado.
- **An:** Señal de 8 bits utilizada en el componente del visualizador del 7 segmentos, se utilizada para actualizar los valores de los leds uno por uno.
- **Ca:** Señal asociada al segmento A del visualizador de 7 segmentos.
- **Cb:** Señal asociada al segmento B del visualizador de 7 segmentos.
- **Cc:** Señal asociada al segmento C del visualizador de 7 segmentos.
- **Cd:** Señal asociada al segmento D del visualizador de 7 segmentos.
- **Ce:** Señal asociada al segmento E del visualizador de 7 segmentos.
- **Cf:** Señal asociada al segmento F del visualizador de 7 segmentos.
- **Cg:** Señal asociada al segmento G del visualizador de 7 segmentos.

Se procede ahora, a explicar el funcionamiento de dicho proyecto.

4.2. Descripción del código.

Para este código, se volvió a utilizar una máquina de estados para el funcionamiento general del proceso, e igualmente se hizo en formato VHDL. Una de las principales diferencias es el uso de la FPGA nexys 4 DDr para llevar a cabo una demostración visual del funcionamiento del código.

El objetivo primordial del proyecto es mostrar mediante una serie de leds de 7 segmentos los datos de salida que va generando dicho código basado en el algoritmo TinyJAMBU, es decir, en caso de estar en un modo con todos los datos y queriendo realizar el método de cifrado, en estos leds de 7 segmentos

deberían mostrarse de 32 en 32 bits (8 dígitos de 7 segmentos, al ser hexadecimal) los valores encriptados, así como la generación posterior del tag.

Las diferentes entradas del código son proporcionadas como constantes dentro del propio código, por lo tanto, tanto clave como nonce, associated data, y plaintext/ciphertext son constantes y mediante los switches de la FPGA se decidirá qué valores toman estas entradas, de esto es de lo que se encarga la señal select_prueba, pudiendo tomar de esta manera diferentes mensajes de prueba.

Con estos switches o interruptores también se puede cambiar el modo de funcionamiento en el que el algoritmo se encuentra, ya sea con todos los datos, o por ejemplo, sin associated data.

Destacar también que el diseño del código está realizado de manera que los valores de salida se muestren y el código quede parado, no obstante, mediante uno de los pulsadores de dicha placa se podrá continuar con el resto del código, esto se hizo de esta manera para así poder ver de manera adecuada los diferentes valores de salida, pudiendo ser elegido el tiempo total que tarda en procesar todo el código.

Por otra parte, aclarar que el código sigue sin gastar muchos recursos como se puede ver en la figura 39.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	664	126800	0%
Number of Slice LUTs	1375	63400	2%
Number of fully used LUT-FF pairs	656	1383	47%
Number of bonded IOBs	24	210	11%
Number of BUFG/BUFGCTRLs	1	32	3%

Figura 39. Análisis de recursos del simulador visual.

En general, se trata de un diseño simple y sin muchas complicaciones, el cual facilita de sobremanera una demostración visual, la cual, muestra un correcto funcionamiento del código.

4.3. Imágenes de prueba.

Por último, he aquí en las figuras 40 y 41 una serie de imágenes de ejemplo una vez programado el código en la FPGA.

En la figura 40 se puede ver la placa programada, pero todavía sin funcionar el código, marcando de esa manera todos los segmentos el valor ‘0’, y en la figura 41 se puede ver el primer valor de salida obtenido para el mensaje determinado, que como se puede ver, el código empieza a funcionar una vez pulsado el botón correspondiente.

Según el modo de operación en el que el algoritmo se encuentre, el cual puede ser elegido mediante los switches de la placa, los valores vistos en la placa mediante los 7 segmentos, pueden ser tanto del texto cifrado, como del texto descifrado, también se mostrarían en ambos casos, los valores resultantes del tag, y en el modo descifrado, además, se encendería un led, en caso de obtener un tag correcto.

Para seguir procesando el resto de los valores del texto cifrado/descifrado, se debe pulsar de nuevo el botón, continuando así hasta su fin.

Como se puede ver en ambas figuras, el código de los 7 segmentos se encuentra en hexadecimal, es decir, que en cada actualización de los leds de 7 segmentos, se verían 32 bits diferentes, teniendo aparte un botón de reset para el caso en el que se quiera empezar de nuevo la demostración, pero esta vez, cambiando algo, como por ejemplo el modo de operación.

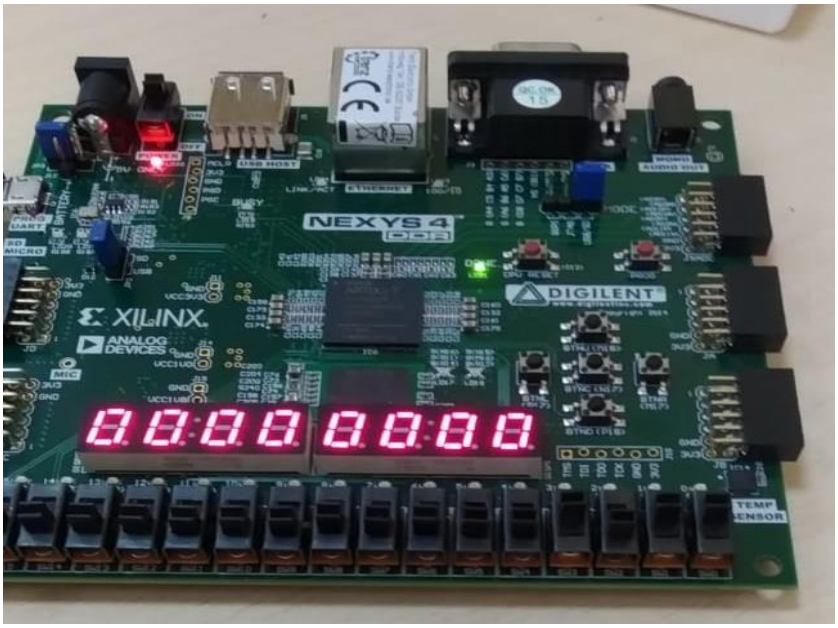

Figura 40. FPGA con código del simulador I.

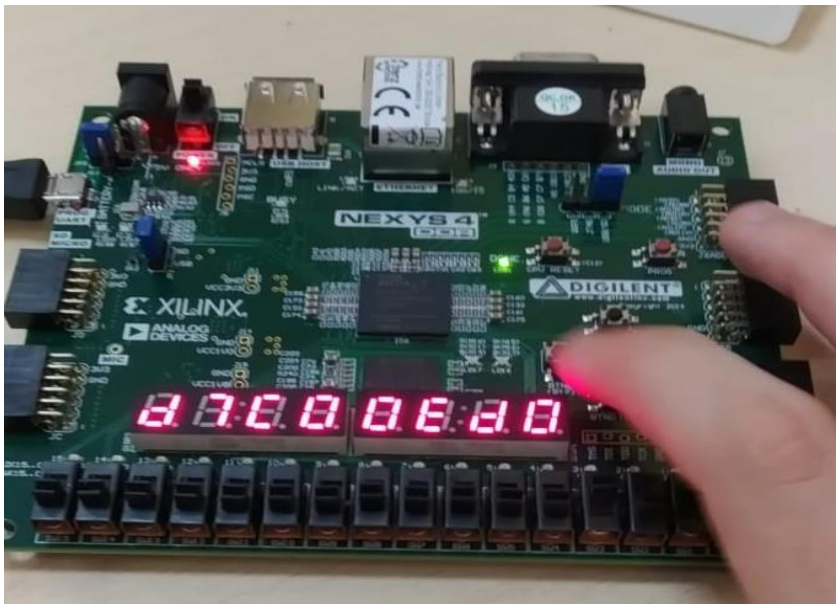

Figura 41. FPGA con código del simulador II.

5. Demostrador.

Como objetivo final de trabajo, se tiene un diseño más complejo que el del apartado anterior, en este caso se trata de un proyecto basado en la transmisión de datos cifrados mediante una uart, y en la recepción de estos datos, para posteriormente descifrarlos.

Como en los casos anteriores se trata de un código realizado mediante una máquina de estados, sucediéndose así de uno en uno los diferentes procesos por los que pasa el código. También destacar que se ha hecho en VHDL y mediante la aplicación de Xilinx.

5.1. Proyecto de transmisión de datos con uart.

Para empezar, lo primero que se va a explicar son las entradas y salidas de este proyecto, así como las constantes utilizadas.

5.1.1. Entradas, salidas y constantes genéricas.

Constantes genéricas:

- **Key:** Constante de 128 bits, que se encuentra también en el proyecto de recepción, utilizada para guardar los 128 bits correspondientes a la clave.
- **Nonce_lsb:** Constante de 64 bits, que igual que la constante key, se encuentra en ambos proyectos, y es utilizada para guardar los 64 bits menos significativos del nonce, puesto que los 32 más significativos se obtienen de manera aleatoria.
- **Bytes_pl:** Constante de 8 bits, utilizada para mostrar el número de bytes que contiene el ciphertext a transmitir.

Entradas:

- **Clk:** Señal de entrada de 1 bit utilizada como reloj del código.
- **Reset:** Señal de entrada de 1 bit utilizada para resetear el códgio en cualquier momento.
- **Addr:** Señal de 8 bits utilizada para guardar en ella la dirección del envío del mensaje.
- **Init:** Señal utilizada para iniciar el funcionamiento del código.

Salidas:

- **Tx:** Señal de salida que utilizada para transmitir los bits que van saliendo por la uart.

5.1.2. Descripción del código.

Este código está formado por los siguientes pasos:

- **1º:** El primer paso es la elección del mensaje que se va a encriptar mediante el método de TinyJAMBU, que en este caso, va a transmitirse de 8 en 8 bits, puesto que es el valor máximo que ofrece la uart utilizada. Para este código, se ha elegido un array formado por 48 filas de 8 bits cada una, dando lugar así a un mensaje total de 384 bits.

La idea principal con estas 48 filas de 8 bits es la de alimentar 16 leds RGB, de ahí que cada 3 filas de 8 bits, sean los valores de un led.

- **2º:** El segundo paso es el diseño de la máquina de estados, separando de manera clara los diferentes procesos, que en este diseño, serían los siguientes: Inicialización, petición de la entrada de datos necesarios para el código TinyJAMBU, realización del proceso de encriptado, envío de los datos de salida obtenidos mediante uart.

- **3º:** Explicación de los diferentes procesos de dicha máquina de estados, en este caso, se empieza por la inicialización, que no es más que la

puesta en marcha del proceso, y la carga de los primeros datos, como serían la clave o el nonce.

- **4º:** Ahora es el turno de la petición de datos, la cual funciona de una manera sencilla, se hace uso de las banderas que vienen incorporadas en el código TinyJAMBU, es decir, una vez la señal req_ad se pone a 1, el código interpreta que hace falta introducir el associated data, y es entonces cuando se procede a introducir de 32 en 32 bits los datos convenientes.

Para facilitar ambos códigos la clave, y los 64 bits menos significativos del nonce son constantes que se han elegido para tanto el proyecto de transmisión de datos, como para el de recepción. Los 32 bits restantes del nonce se escogen de manera aleatoria mediante un contador, dando lugar así a diferentes nonces en cada envío de datos.

Destacar también que el associated data está formado por una serie de señales en el siguiente orden: Primero, Por la señal de dirección addr, utilizada para que el proyecto de recepción verifique la proveniencia de los datos recibidos, segundo, la señal bytes, que muestra el número de bytes totales del mensaje, y por último, también contiene los 32 bits más significativos del nonce, que como se ha comentado en el párrafo anterior tienen un valor aleatorio.

- **5º:** Ahora se llevaría a cabo el proceso de encriptado, el cual facilitaría los datos de salida, que en este momento, se estaría hablando del texto cifrado, y por último, del tag.

- **6º:** Por último, y no menos importante, sería el momento de enviar los datos de salida obtenidos de 8 en 8 bits. El código funciona enviando los datos en el siguiente orden: associated data, texto cifrado o ciphertext, y por último, el tag.

- **7º:** Realización de un test bench de prueba para asegurarnos de que el funcionamiento de la uart es correcto, así como para asegurar que la máquina de estados, funciona también de una manera correcta.

La simulación utilizando dicho test bench se puede ver en la figura 42, viendo así la cantidad de valores diferentes que toma la salida tx_data, que es la señal donde se almacenan los 8 bits a transmitir. También se puede ver la señal tx_ena, que se pone a 1 cada vez que se va a transmitir el mensaje, y para terminar, recalcar también la cantidad de estados que tiene que recorrer la máquina, además del tiempo de funcionamiento, que ronda los 19 ms (19000ns), siendo este proceso mucho más tedioso que el experimento visual visto en el capítulo anterior.

Además, se puede ver el consumo de recursos del proyecto indicados en la figura 43, siendo estos de una cantidad no significativa.

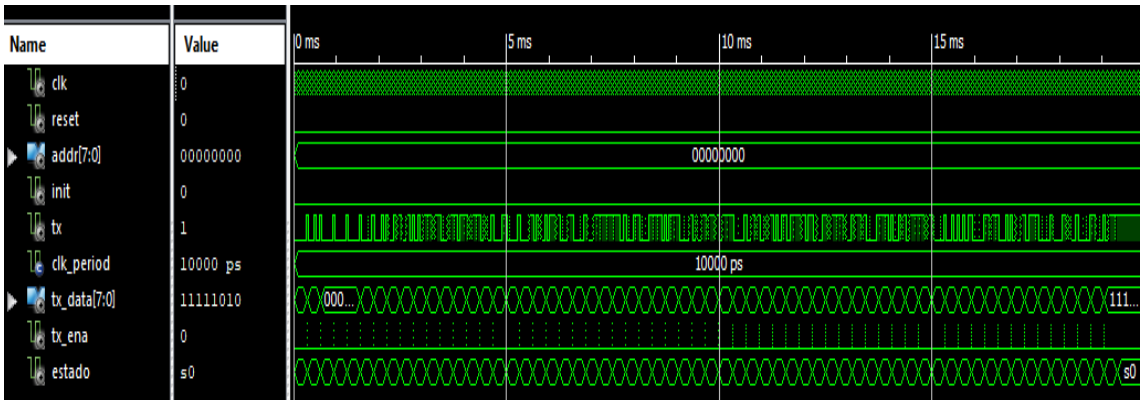

Figura 42. Simulación proyecto de transmisión.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	562	126800	0%
Number of Slice LUTs	897	63400	1%
Number of fully used LUT-FF pairs	522	937	55%
Number of bonded IOBs	12	210	5%
Number of BUFG/BUFGCTRLs	1	32	3%

Figura 43. Análisis de recursos del proyecto de transmisión.

5.2. Proyecto de recepción de datos con uart.

Ahora pues, se va a explicar el funcionamiento del proyecto complementario, el cual, debe ser capaz de recibir los datos, desencriptarlos, y formar el mensaje inicial de 48 filas de 8 bits.

5.2.1. Entradas, salidas y constantes genéricas.

Constantes genéricas

- **key:** Constante con la misma descripción que en el proyecto de transmisión.
- **Nonce_lsb:** Constante con la misma descripción que en el proyecto de transmisión.

Entradas

- **Clk:** Reloj del diseño.
- **Reset:** Funcionamiento idéntico al del proyecto de transmisión.
- **Addr_esclavo:** La contraparte de la señal addr del proyecto de transmisión, es decir, la dirección de envío del proyecto de recepción.
- **Addr_ram:** Señal de 6 bits, utilizada en un primer momento para seleccionar un valor concreto de entre los 48 valores de 8 bits del mensaje desencriptado, pudiendo así compararlo con el mensaje inicial.

- **RX:** Señal de 1 bit, por la que se introducen los valores que se van recibiendo por la uart.

Salidas

- **Led:** Señal de 16 bits, utilizada en un primer momento, para visualizar en los leds de la FPGA, los valores de un array de 8 bits del mensaje encriptado, y del mensaje desencriptado, pudiendo compararlos entre sí.

5.2.2. Descripción del código.

Como en el apartado anterior, se va a proceder a explicar los diferentes procesos que tiene el proyecto para su correcto funcionamiento:

- **1º:** En este punto, lo primero y más importante, es la recepción de datos, puesto que no se tiene nada, hasta que no llegan dichos datos. Por ello, los primeros pasos de la máquina de estados funcionan guardando los diferentes datos que se van recibiendo en sus respectivas señales, las cuales se utilizarán posteriormente en el proceso de desencriptado.

El primer valor en ser recibido, debe ser la señal addr, o señal de dirección, cuyo valor debe ser igual que el addr del proyecto de recepción, en caso de ser iguales, el código seguiría su funcionamiento con total normalidad, no obstante, en caso de ser diferente, significa que ese mensaje no es válido, y el código terminaría ahí, sin recibir más nada, a la espera de la siguiente recepción de mensajes, con un nuevo addr.

Una vez recibido un addr correcto, se pasará a recibir el número de bytes, que en este caso son 48, después los 32 bits más significativos del nonce, posteriormente las 48 filas de 8 bits de texto cifrado, y por último, la señal del tag.

- **2º:** Ahora bien, una vez obtenidos los diferentes mensajes a través de la uart, se procede a poner en funcionamiento el código del TinyJAMBU, introduciéndole cuando este lo requiera las entradas necesarias, que deben estar guardadas en diferentes señales, salvo la clave y los 64 bits

menos significativos del nonce, que son constantes como lo eran en el proyecto de transmisión.

- **3º:** Ya entrado en funcionamiento el proceso del TinyJAMBU, lo único que quedar por hacer es obtener las salidas correspondientes, hasta obtener las 48 filas de 8 bits del mensaje original, y por último, comparando el tag de entrada con el generado, para asegurar de esta manera un correcto funcionamiento.

- **4º:** Realización de un test bench conjunto con el de transmisión, de prueba, para poder comprobar si la señal de recepción de datos funciona correctamente.

Destacar también el número de recursos utilizados para la realización del proyecto, que se pueden apreciar en la figura 44, siendo éstos mayores que los del proceso de transmisión, ya que en esta ocasión hacen falta más señales para guardar los diferentes datos que se transmiten.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	663	126800	0%
Number of Slice LUTs	1283	63400	2%
Number of fully used LUT-FF pairs	636	1310	48%
Number of bonded IOBs	33	210	15%
Number of BUFG/BUFGCTRLs	1	32	3%

Figura 44. Análisis de recursos del proyecto de recepción.

5.3. Proyecto conjunto de transmisión y recepción de datos con uart.

En este apartado, se van a juntar ambos proyectos para comprobar su funcionamiento en conjunto, que es el motivo por el que diseñaron.

Este proceso fue llevado a cabo mediante un test bench, uniendo la salida de la uart del proyecto de transmisión con la entrada de la uart del proyecto de recepción.

```
constant rom_num : rom_type :=(
  "00001111", "00000000", "00000000",
  "00000000", "00001111", "00000000",
  "00000000", "00000000", "00001111",
  "00000001", "00000001", "00000001",
  "00000000", "00000000", "00000000",
  "00001111", "00001111", "00000000",
  "00000000", "00001111", "00001111",
  "00000011", "00000011", "00000011",
  "00000011", "00000000", "00000011",
  "00000011", "00000001", "00001111",
  "00000111", "00001000", "00000100",
  "00000001", "00000100", "00001100",
  "00000011", "00001111", "00000000",
  "00000000", "00000001", "00000000",
  "00000001", "00000000", "00000000",
  "00000000", "00000000", "00000001"
    );
```

Figura 45. Mensaje de prueba a transmitir.

Como se puede ver en la figura 45, este es el mensaje original de 48 filas de 8 bits, y por ende, es el mensaje final que se debería obtener en el proyecto de recepción.

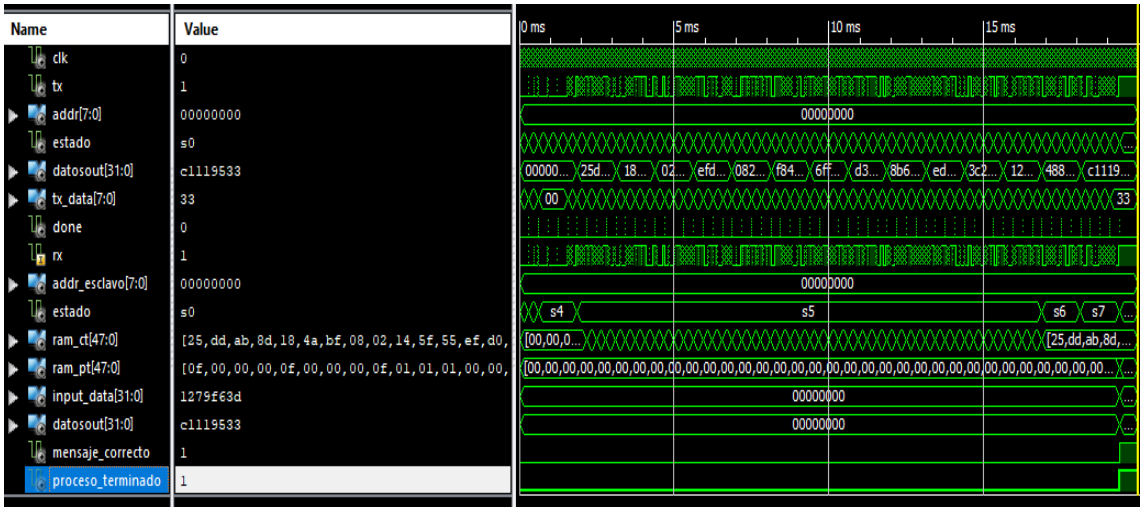

Figura 46. Simulación del proyecto final.

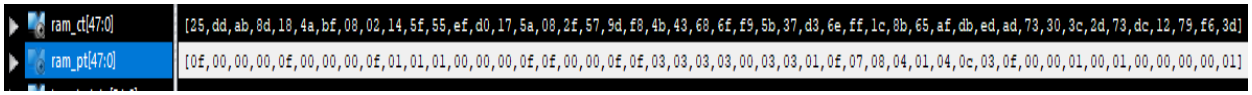

Figura 47. Mensaje de 48 bytes resultante.

Ahora toca el turno de explicar el resultado obtenido en el test bench, el cual, se puede ver en la figura 46, en este caso, se va a explicar señal por señal que se ha obtenido.

Primeramente se ve la señal de tx, que son los valores de transmisión por la uart, después se ve la señal addr, que muestra la dirección que va incluida dentro del mensaje de transmisión, la cual debe coincidir con la señal addr_esclavo, que es la dirección del proyecto de recepción, dándose en este caso la igualdad de ambas.

Por otra parte, se pueden ver las señales de estado de ambos proyectos, siendo la primera del proyecto de transmisión, y la segunda, del proyecto de recepción, como es obvio, muestra los diferentes procesos por los que va pasando la máquina de estados. Recalcar que el proyecto de recepción está gran parte del tiempo en el estado S5, puesto que es el estado en el que debe

recibir el texto cifrado, y debido a la gran cantidad de bits que se deben transmitir por la uart, este proceso se repite en numerosas ocasiones.

En esta ocasión, aparece la señal datos_out, que igualmente a la señal estado, la primera pertenece al proyecto de transmisión, y la segunda al de recepción, su función es la de mostrar los datos de salida obtenidos en el proceso del TinyJAMBU, es decir, el ciphertext/plaintext, y el tag. En el proyecto de transmisión, los datos obtenidos en esta señal son los que se envían mediante la señal tx_data, sin embargo, deben separarse de 8 en 8 bits los datos obtenidos para poder enviarse de manera correcta.

Ahora, en la figura 47, ya se pueden ver con más precisión las 48 filas de 8 bits que se obtienen, los datos guardados en la señal ram_ct corresponden a los valores encriptados, y la señal ram_pt corresponde a estos valores una vez han sido desencriptados, como se puede ver en la comparación entre las figuras 45 y 47, los bits obtenidos en la señal ram_pt coinciden con el mensaje original, generando esto una confianza en el correcto funcionamiento del código, así como una correcta transmisión de datos.

Destacar que la señal mensaje_correcto indica que el tag obtenido en el proyecto de recepción es el mismo al obtenido en el de transmisión, dando otra razón de peso que verifica el correcto funcionamiento del código.

Se terminan de esta manera las demostraciones de uso del algoritmo TinyJAMBU.

6. Conclusiones.

En este apartado, y como conclusión final obtenida, se va a hacer un último resumen de todo el trabajo realizado en este extenso documento, así como todas las habilidades y conocimientos adquiridos, que han sido los siguientes:

- Lo primero de todo, se ha estudiado y obtenido una clara idea del funcionamiento de la criptografía, así como sus tipos y formas de aplicación a lo largo de la historia.

- Se han estudiado la justificación y los principales requerimientos del proyecto lanzado por el NIST acerca de criptografía lightweight, así como la obtención de información sobre este tipo de criptografía y su auge en estos últimos años.

- Se ha recabado información acerca del funcionamiento de los AEADs, y las FPGAs, dos partes importantes de la realización del trabajo.

- Se ha investigado y comprendido el funcionamiento del algoritmo TinyJAMBU, uno de los finalistas del proyecto del NIST sobre criptografía lightweight. Se destaca de este cifrador que es uno de los más limitados en recursos.

- Se ha realizado el diseño en lenguaje VHDL del algoritmo TinyJAMBU, siguiendo además las restricciones de las principales herramientas de síntesis. Se han realizado además simulaciones de prueba del algoritmo, variando los posibles modos de funcionamiento que puede tener el algoritmo, asegurando así un correcto funcionamiento en cualquier escenario posible, todo, utilizando la aplicación de ISE DESIGN de Xilinx.

- El código realizado es parametrizable, de forma que se puede cambiar durante la realización de los desplazamientos, el número de bits que se procesan en cada ciclo de reloj. Esto da lugar a varias versiones en las que cambia el throughput y los recursos consumidos. Todas las

versiones han sido simuladas y se ha hecho una comparación de los recursos de cada una de ellas.

- Se ha realizado un análisis de los recursos utilizados para todas las variaciones de los parámetros y modos de funcionamiento posibles, así como, los distintos ciclos de reloj usados en cada parte del código de implementación del algoritmo TinyJAMBU, consiguiendo así el modo de funcionamiento más eficiente.

- Se ha realizado otro diseño, utilizando el código realizado sobre el algoritmo TinyJambu, para comprobar el funcionamiento experimental de éste, así como, para verificación más visual de dicho algoritmo, apoyado en la placa Nexys4ddr y su visualizador 7 segmentos.

- Se ha realizado un último diseño, en este caso, que sirve como un ejemplo más similar al de una aplicación real, que consiste en la utilización de una UART para poder llevar a cabo una transmisión y recepción de datos, junto a la utilización del algoritmo de TinyJAMBU. La transmisión usa datos cifrados por el TinyJAMBU que son descifrados por otro TinyJAMBU situado en la parte de recepción. Este diseño ha sido verificado mediante simulaciones, comprobando así su correcto funcionamiento.

7. Bibliografía.

[1]: Colaboradores de Wikipedia. Criptografía. En: Wikipedia, la enciclopedia libre, 2022. Disponible en: http://es.wikipedia.org/wiki/Criptografía. Accedido 21/12/2022.

[2]: Derek Dewitt, *Cifrado de datos: ¿en qué consiste?, 2021.* Disponible en: https://www.avast.com/es-es/c-encryption. Accedido 21/12/2022.

[3]: NIST. Lightweight Cryptography Project, disponible en: https://csrc.nist.gov/Projects/Lightweight-Cryptography. Accedido 21/12/2022.

[4]: Kerry McKay, Larry Bassham, Meltem Sönmez Turan, Nicky Mouha, *Report on Lightweight Cryptography*. NISTIR 8114, 2017, disponible en: https://doi.org/10.6028/NIST.IR.8114. Accedido 21/12/2022.

[5]: Juan Carlos Espinosa Ceniceros, *Block Cipher: Present,* 2012. Disponible en: https://juankenny.blogspot.com/2012/10/sc-block-ciphers-present.html. Accedido 21/12/2022.

[6]: Avik Chakraborti, Nilanjan Datta, Ashwin Jha, Mridul Nandi, *Classification os AEADs.* NIST, 2020.

[7]: Tink Cryptography library, *Authenticated Encryption with Associated Data (AEAD),* 2021.

[8]: Javier Romero, *¿Qué es un FPGA y para qué sirve?, 2021.* Disponible en: https://www.geeknetic.es/FPGA/que-es-y-para-que-sirve. Accedido 21/12/2022.

[9]: Marcos Sánchez Élez, *Introducción a la programación en vhdl,* Madrid: Universidad complutense de Madrid, 2014.

[10]: Digilent, Nexys4 DDR FPGA Board Reference Manual, 2016.

[11]: Hongjun Wu, Tao Huang. *The tweak to TinyJAMBU.* NIST, 2021.

[12]: Hongjun Wu, Tao Huang, *TinyJAMBU: A Family of Lightweight Authenticated Encryption Algorithms (Version 2).* NIST, 2021.

[13]: GMUCERG, TinyJAMBU, 2020, disponible en: https://github.com/GMUCERG/TinyJAMBU. Accedido 21/12/2022.