Instituto Politécnico Nacional

ESCUELA SUPERIOR DE INGENIERÍA MECÁNICA Y ELÉCTRICA
UNIDAD CULHUACAN

"DISEÑO DE UN MÓDULO UART EN FPGA UTILIZANDO RTL PARA LA IMPLEMENTACIÓN DE TÉCNICAS DE CIFRADO Y DESCIFRADO"

Que para obtener el grado de

"Maestro en Ingeniería En Seguridad y Tecnologías de la Información"

Presenta:

Christian Antonio Colin Cejudo

Directores:

Dr. Aldo Hernández Suarez

Dr. Gonzalo Issac Duchén-Sánchez

Mes 2025

Instituto Politécnico Nacional

ESCUELA SUPERIOR DE INGENIERÍA MECÁNICA Y ELÉCTRICA

UNIDAD CULHUACAN

"DISEÑO DE UN MÓDULO UART EN FPGA UTILIZANDO RTL PARA LA IMPLEMENTACIÓN DE TÉCNICAS DE CIFRADO Y DESCIFRADO"

Que para obtener el grado de

"Maestro en Ingeniería En Seguridad y Tecnologías de la Información"

Presenta:

Christian Antonio Colin Cejudo

Directores:

Dr. Aldo Hernández Suarez	Dr. Gonzalo Issac Duchén-Sánchez

Presidente del Jurado	**Profesor titular**
TBD	TBD

DEDICATORIA

dedico este trabajo a y aa ....

porque ...

bla bla bla bla

bla bla

AGRADECIMIENTOS

Al CONACYT por la beca-crédito otorgada para la realización de mis estudios.

Así mismo agradezco a mis maestros, compañeros y a toda aquella persona que de alguna manera contribuyó al término de mis estudios de maestría.

Contenido

Resumen

Resumen:

La creciente demanda de seguridad informática robusta y eficiente ha impulsado la adopción de hardware especializado para operaciones criptográficas. Ante el aumento de las amenazas cibernéticas y la necesidad de procesar grandes volúmenes de datos en tiempo real, las soluciones criptográficas basadas en hardware ofrecen ventajas significativas en términos de rendimiento, resistencia a ataques y almacenamiento seguro de claves criptográficas.

Esta tesis presenta la implementación de un sistema de comunicación segura utilizando el protocolo UART (Universal Asynchronous Receiver-Transmitter) como base para un diseño a nivel de transferencia de registros (RTL) sobre una plataforma FPGA. El protocolo base fue modificado para introducir una capa adicional de seguridad a nivel de hardware. Además, se integraron técnicas criptográficas específicamente cifrado y descifrado en el diseño con el objetivo de mejorar la protección e integridad de los datos durante la transmisión. Los resultados demuestran la viabilidad de incorporar mecanismos criptográficos directamente en el hardware de comunicación, proporcionando una solución escalable y eficiente para sistemas embebidos seguros.

Palabras Clave: Hardware criptográáfico, Seguridad informática, Cifrado, Descifrado, Claves criptográficas, Firmas digitales, Autenticación, Amenazas cibernéticas, Procesamiento de datos, Protocolos de comunicación, UART (Universal Asynchronous Receiver-Transmitter),

FPGA (Field-Programmable Gate Array), Diseño RTL (Register Transfer Level), Capa de seguridad en hardware, Técnicas criptográficas, Infraestructuras críticas.

Abstract

Abstract: The increasing demand for robust and efficient information security has led to the growing adoption of specialized hardware for cryptographic operations. In response to the rise in cyber threats and the need to process large volumes of data in real time, hardware-based cryptographic solutions offer significant advantages in terms of performance, resistance to attacks, and secure storage of cryptographic keys.

This thesis presents the implementation of a secure communication system using the UART (Universal Asynchronous Receiver-Transmitter) protocol as the foundation for a Register Transfer Level (RTL) design on an FPGA platform. The base protocol was modified to introduce an additional hardware-level security layer. Furthermore, cryptographic techniques—specifically encryption and decryption—were integrated into the design to enhance data protection and integrity during transmission. The results demonstrate the feasibility of embedding cryptographic mechanisms directly into communication hardware, providing a scalable and efficient solution for secure embedded systems.

Palabras Clave: Cryptographic hardware, Information security, Encryption, Decryption, Cryptographic keys, Digital signatures, Authentication, Cyber threats, Data processing, Communication protocols, UART (Universal Asynchronous Receiver-Transmitter), FPGA (Field-Programmable Gate Array), RTL design (Register Transfer Level), Hardware security layer, Cryptographic techniques, Critical infrastructure.

Índice de figuras

Índice de figuras

Introducción

En la actualidad, la seguridad de la información se ha convertido en un pilar fundamental para el desarrollo de sistemas digitales confiables. La creciente sofisticación de las amenazas cibernéticas, sumada al incremento exponencial en la generación y transmisión de datos, exige soluciones cada vez más robustas y eficientes. En este contexto, el uso de hardware especializado para operaciones criptográficas ha emergido como una alternativa eficaz frente a las limitaciones del procesamiento criptográfico basado exclusivamente en software. El hardware criptográfico ofrece ventajas significativas, como mayor rendimiento, menor latencia, menor consumo energético y una mayor resistencia a ataques físicos y lógicos. Estas características lo convierten en una solución ideal para sistemas embebidos, dispositivos IoT, aplicaciones industriales y entornos donde la seguridad y la eficiencia operativa son prioritarias. Los arreglos de compuertas programables en campo, conocidos como FPGAs (Field Programmable Gate Arrays), se han convertido en una de las tecnologías más versátiles para el diseño de sistemas electrónicos. Estos dispositivos ofrecen una solución rentable especialmente para producciones de bajo volumen, ya que el costo inicial para obtener un prototipo es considerablemente menor que el de los circuitos integrados de aplicación específica, conocidos como ASIC (Application-Specific Integrated Circuit). Además, una ventaja significativa es su capacidad de reconfiguración durante la operación, lo cual permite que un solo dispositivo pueda desempeñar diversas funciones definidas previamente, optimizando espacio y reduciendo costos. Hoy

en día, los sistemas electrónicos están presentes en casi todos los aspectos de la vida cotidiana, desde productos de consumo hasta sistemas de control industrial, aplicaciones automotrices, de seguridad, y muchas más. La tendencia actual en diseño electrónico se caracteriza por la creciente complejidad de sus componentes, lo que demanda soluciones que sean fáciles de usar, versátiles, de bajo consumo y que lleguen rápidamente al mercado. La tecnología moderna en circuitos integrados hace posible integrar estos sistemas complejos en dimensiones muy reducidas; a estos se les denomina sistemas embebidos o "System on Chip"(SoC, por sus siglas en inglés). Dichos sistemas, diseñados para cumplir funciones específicas, combinan hardware y software creados específicamente para cada tarea. En este contexto, las FPGAs, gracias a su capacidad de reconfiguración, son herramientas valiosas para el desarrollo de prototipos o series pequeñas a costos accesibles. Entre los protocolos de comunicación más utilizados en sistemas embebidos se encuentra el UART (Universal Asynchronous Receiver-Transmitter), gracias a su simplicidad, bajo costo de implementación y amplia compatibilidad. Sin embargo, dicho protocolo carece de mecanismos nativos de seguridad, lo que lo hace vulnerable a interceptaciones, manipulaciones y accesos no autorizados. En este trabajo de tesis se propone el diseño e implementación de una capa de seguridad criptográfica sobre el protocolo UART, utilizando un enfoque a nivel de transferencia de registros (RTL) en una plataforma FPGA implementando un lenguaje de descripción de hardware por sus siglas en Ingles (HDL) utilizado para modelar y diseñar circuitos digitales. La modificación del protocolo permite no solo asegurar la transmisión de datos, sino también integrar técnicas de cifrado y descifrado directamente en el hardware. Esta solución busca demostrar la viabilidad de incorporar seguridad criptográfica eficiente en sistemas de comunicación embebidos, sin comprometer el rendimiento ni la escalabilidad.

Definición del problema

En la actualidad, la transmisión de datos en sistemas embebidos se realiza, en gran medida, a través de protocolos de comunicación simples y eficientes como el UART (Universal Asynchronous Receiver-Transmitter). Sin embargo, uno de los principales inconvenientes de este protocolo es la ausencia de mecanismos nativos de seguridad, lo que lo expone a una variedad de ataques como la intercepción de datos (sniffing), la manipulación de tramas, la inyección de comandos maliciosos y la suplantación de dispositivos. Este problema se agrava en aplicaciones donde el UART se emplea en contextos críticos, tales como sistemas industriales, redes de sensores, dispositivos médicos o entornos IoT, en los que la confidencialidad y la integridad de los datos son fundamentales. Aunque existen soluciones basadas en software para implementar cifrado, estas suelen generar una sobrecarga computacional considerable en microcontroladores de recursos limitados, afectando el rendimiento y la eficiencia del sistema. Por tanto, se identifica la necesidad de una solución que permita asegurar la transmisión de datos a través del protocolo UART sin comprometer el rendimiento, preferentemente mediante una implementación en hardware que garantice mayor robustez y eficiencia.

Justificación

El avance acelerado de la tecnología ha incrementado la interconexión entre dispositivos, especialmente en sistemas embebidos e infraestructura crítica. Esta conectividad expone a los sistemas a múltiples vectores de ataque que pueden comprometer la integridad, confidencialidad y disponibilidad de la información. Dado que UART es un protocolo ampliamente utilizado en microcontroladores, sensores y módulos de comunicación, fortalecer su seguridad representa una mejora significativa en la protección de sistemas donde el reemplazo por protocolos más complejos no es viable por razones de costo o recursos. El uso de FPGAs como plataforma de implementación permite un diseño flexible, reconfigurable y altamente optimizado. Además, al integrar funciones criptográficas directamente en el hardware, se reduce la dependencia del software y se incrementa la resistencia a ataques de canal lateral, lo que incrementa el nivel de seguridad general del sistema. Esta tesis, por tanto, contribuye con una propuesta de solución viable, eficiente y escalable para la seguridad de comunicaciones en sistemas embebidos, atendiendo a una necesidad real en la industria tecnológica actual.

Objetivo

Objetivo General

Diseñar e implementar una capa de seguridad criptográfica en el protocolo UART mediante un diseño a nivel RTL en FPGA, que integre técnicas de cifrado y descifrado para mejorar la seguridad en la transmisión de datos en sistemas embebidos.

Objetivos Específicos

- Analizar las vulnerabilidades del protocolo UART en contextos de comunicación embebida.
- Diseñar un módulo RTL que modifique el protocolo UART para incorporar una capa de seguridad.
- Integrar algoritmos de cifrado y descifrado simétrico en el diseño propuesto.
- Implementar el sistema en una FPGA para validar su funcionamiento y rendimiento.
- Evaluar la eficiencia y robustez del diseño frente a posibles ataques o vulnerabilidades.

Enfoque en Seguridad de la información

Antecedentes

En investigaciones previas se han propuesto mecanismos de seguridad sobre UART, pero en su mayoría dependen de software, lo que limita la eficiencia y vulnerabilidad ante ataques. Otros estudios han demostrado que la implementación de algoritmos criptográficos en FPGA, como AES en modo CTR, es factible y efectiva para mejorar la seguridad sin afectar el rendimiento significativamente.

CAPÍTULO 1

Marco Teórico Contextual

1.1. Marco Teórico Contextual

Para contextualizar adecuadamente el presente trabajo de tesis dentro del marco teórico, es necesario abordar el origen del algoritmo criptográfico estándar César, así como su evolución y aplicaciones a lo largo del tiempo. Asimismo, se explorarán las condiciones en las que surgieron los servicios y mecanismos de seguridad informática, y las formas en que estos han sido implementados en distintos entornos. Este análisis permite delimitar la problemática que se busca atender mediante la propuesta de implementación en hardware presentada en esta investigación. En consecuencia, se realiza una revisión del estado del arte en la disciplina, con especial énfasis en los sistemas embebidos que incorporan mecanismos de seguridad.

A través de esta revisión se busca comprender las bases que fundamentan el diseño de soluciones criptográficas en plataformas reconfigurables, como las FPGAs, y su relevancia en escenarios donde la protección de la información es crítica. Este marco teórico sienta las bases para el desarrollo metodológico y experimental abordado en los siguientes capítulos.

1.2. Fundamentos de la Criptografía clasica

La criptografía es la ciencia y arte de proteger la información mediante técnicas que transforman datos legibles (texto plano) en datos ininteligibles (texto cifrado), de modo que sólo las personas autorizadas puedan acceder a su contenido (Stallings, 2017). A lo largo de la historia, la criptografía ha evolucionado desde métodos simples de sustitución y transposición hasta complejos algoritmos basados en teoría matemática, álgebra moderna y teoría de números.

Los métodos clásicos, aunque hoy son inseguros frente a los ataques modernos, permiten comprender los conceptos fundamentales sobre el uso de claves, cifrado simétrico y análisis criptográfico. Uno de los algoritmos más representativos de la criptografía clásica es el cifrado César, también conocido como cifrado por desplazamiento.

1.3. El Cifrado César: Origen Histórico

El cifrado César recibe su nombre de Julio César, quien lo empleaba para cifrar mensajes militares. Según registros históricos, César utilizaba un desplazamiento de tres posiciones para codificar sus mensajes (Kahn, 1996). Aunque su simplicidad hoy lo convierte en un cifrado trivial, en su época ofrecía una protección básica frente a lectores no entrenados o no alfabetizados.

Este cifrado pertenece a la familia de los cifrados monoalfabéticos por sustitución, donde cada letra del alfabeto es reemplazada por otra, según una regla fija determinada por una clave de desplazamiento.

1.4. Funcionamiento del Algoritmo

Desde un punto de vista matemátoco, el cifrado César puede representarse mediante la siguiente función:

$$C(x) = (x + k) mod n$$

Donde:

- $C(x)$ es el carácter cifrado.
- x es el índice del carácter en el alfabeto $A = 0, B = 1, ..., Z = 25$
- k es la clave de desplazamiento.
- n el es número total de caracteres del alfabeto, usualmente $n = 26$ en el alfabeto latino.

El descifrado consiste en aplicar la operación inversa

$$P(x) = (x - k) mod n$$

Por ejemplo, si se cifra la palabra ***"HOLA"*** con un desplazamiento de $k = 3$ se obtiene ***"KROD"*** . Para descifrar, se aplica el desplazamiento inverso.

1.5. Seguridad y Vulnerabilidades

El cifrado César es vulnerable a varios tipos de ataques criptográficos:

- Ataque por fuerza bruta: Dado que existen solamente 25 posibles claves (excluyendo el desplazamiento nulo), un atacante puede probar todas las combinaciones en poco tiempo.
- Análisis de frecuencia: Cada idioma tiene una distribución característica de letras. En español, por las letras "E", "A" y "O" son las más comunes. Si se cifra un texto suficientemente largo, estas frecuencias se preservan, permitiendo identificar el desplazamiento utilizado.

Estas debilidades lo hacen inapropiado para cualquier uso moderno que requiera confidencialidad real, pero útil como herramienta de desarrollo.

1.6. Utilidad en el Diseño de Sistemas Criptográficos

A pesar de sus limitaciones, el cifrado César es frecuentemente utilizado en el diseño de sistemas criptográficos hardware/software con fines educativos y experimentales. Su estructura simple lo convierte en una opción ideal para:

- Introducir técnicas de procesamiento secuencial de datos.
- Implementar máquinas de estados finitos para codificación y decodificación.
- Evaluar el consumo de recursos lógicos y tiempos de propagación en plataformas como FPGAs.
- Comparar el comportamiento de cifrados más complejos respecto a algoritmos básicos en entornos de bajo nivel (VHDL, Verilog).

En proyectos de diseño digital con arquitectura RTL (Register Transfer Level), el cifrado César se utiliza como caso de estudio para optimizar operaciones de desplazamiento modular, codificación de caracteres ASCII y generación de claves.

CAPÍTULO 2

Antecedentes System Verilog

2.1. Evolución del diseño digital y el surgimiento de SystemVerilog

El desarrollo del diseño digital ha experimentado transformaciones significativas desde sus inicios, comenzando con el prototipado físico mediante placas de pruebas (protoboards) hasta llegar a sofisticadas plataformas de simulación y modelado asistido por computadora. Herramientas CAD (Computer-Aided Design) permitieron simular circuitos electrónicos con mayor precisión y eficiencia, aunque su uso inicial requería superar ciertas barreras de complejidad técnica.

Estas herramientas facilitaron la implementación de esquemas eléctricos y permitieron definir señales de entrada coherentes con la lógica del sistema, generando salidas fácilmente interpretables. Simuladores como SPICE se convirtieron en referentes dentro del ámbito académico e industrial, al ofrecer una modelación precisa del comportamiento eléctrico. Una vez validados los resultados, se procedía al diseño del circuito físico mediante herramientas espe-

cializadas en tarjetas de circuito impreso (PCB), como el entorno OrCAD, particularmente en su módulo PCB Layout.

Durante esta etapa, el Laboratorio de Computación Adaptable comenzó a implementar modelos de circuitos neuronales electrónicos mediante estas plataformas (Padrón, 1997). Posteriormente, ante el incremento en la complejidad de los modelos de redes neuronales, se incorporaron herramientas como MATLAB y su entorno gráfico SIMULINK, permitiendo una representación matemática directa de sistemas dinámicos. Esta evolución posibilitó una interpretación más eficaz de los resultados, tanto cuantitativa como cualitativamente (Padrón et al., 2000, pp. 338–349).

De forma paralela, la industria del hardware reconfigurable, especialmente a través de las tarjetas FPGA manufacturadas por Xilinx, permitió abordar problemas específicos mediante interfaces programables. Cada tarjeta está diseñada para ofrecer flexibilidad al usuario, quien puede programar sus componentes en lenguajes de descripción de hardware como Verilog o VHDL, e integrar diversas interfaces en función de la disponibilidad física o lógica del sistema. Esta versatilidad convirtió a los kits de desarrollo en herramientas fundamentales para el diseño de soluciones a medida, especialmente en contextos de prototipado rápido y validación funcional.

Conforme se incrementaron las necesidades de diseño a nivel de sistemas y la verificación se volvió una etapa crítica, surgió SystemVerilog como una extensión y evolución del lenguaje Verilog. SystemVerilog no solo incorporó mejoras en la descripción estructural y comportamental del hardware, sino que integró capacidades orientadas a la verificación funcional, programación orientada a objetos y modelado de sistemas complejos. Esta evolución respondió a la necesidad de unificar el flujo de diseño y verificación bajo un mismo lenguaje, facilitando la implementación de testbenches avanzados, interfaces reutilizables y entornos de verificación compatibles con metodologías modernas como UVM (Universal Verification Methodology).

En este sentido, el uso de tarjetas FPGA programadas con SystemVerilog permite combinar la precisión del diseño RTL con capacidades avanzadas de verificación, lo cual resulta esencial en entornos de desarrollo donde se busca garantizar funcionalidad, rendimiento y confiabilidad

desde las etapas tempranas del proyecto. La historia y evolución de este lenguaje refleja una tendencia hacia la consolidación de herramientas que integren diseño, simulación, verificación y síntesis dentro de un mismo entorno de desarrollo.

Cronología relevante del desarrollo de SystemVerilog

- 1984: Se introduce el lenguaje Verilog como herramienta para descripción de hardware por Gateway Design Automation, posteriormente adquirido por Cadence (IEEE, 2008).
- 1990s: Verilog se convierte en uno de los estándares principales para diseño digital, ampliamente utilizado en la industria electrónica.
- 1999: Se inicia el desarrollo de SystemVerilog como una extensión de Verilog para abordar las limitaciones en verificación y modelado (Accellera Systems Initiative, 2002).
- 2002: SystemVerilog es adoptado formalmente por Accellera como estándar para diseño y verificación.
- 2005: SystemVerilog es estandarizado por IEEE como el estándar IEEE 1800-2005, consolidando su uso en la industria (IEEE, 2005).
- 2012: Se publica la revisión IEEE 1800-2012, incorporando mejoras en síntesis y verificación.
- 2017: Nueva revisión IEEE 1800-2017 con ampliaciones en características para diseño y verificación de sistemas complejos.

2.2. Lenguajes de Descripción de Hardware (HDL) y su Aplicación en el Diseño Digital

Los lenguajes de descripción de hardware (HDL, por sus siglas en inglés) surgieron como respuesta a la necesidad de los diseñadores digitales de contar con herramientas formales que permitieran especificar, modelar y verificar sistemas digitales de forma estructurada y

en distintos niveles de abstracción. Estos lenguajes no solo facilitan la comunicación entre diseñadores, sino que también permiten una interacción fluida entre las herramientas de diseño asistido por computadora (CAD) y los propios modelos digitales (Terés et al., 1998).

Los entornos de desarrollo basados en HDL integran herramientas de compilación, simulación y síntesis, lo que permite validar el comportamiento funcional de un diseño antes de su implementación física. En la actualidad, los lenguajes HDL más utilizados son VHDL y Verilog, ambos estandarizados por el IEEE (Institute of Electrical and Electronics Engineers), y ampliamente adoptados en la industria del diseño digital.

VHDL: Descripción, Aplicación y Estructura VHDL (VHSIC Hardware Description Language) fue desarrollado originalmente para documentar y verificar circuitos integrados de alta velocidad dentro del programa VHSIC del Departamento de Defensa de los EE.UU. Su sintaxis y semántica están basadas en el lenguaje ADA, lo cual le proporciona una estructura sólida, orientada a sistemas críticos y de tiempo real.

VHDL permite modelar un sistema digital a través de diferentes niveles de descripción: comportamiento, transferencia de registros (RTL) y nivel lógico estructural. Esto permite abarcar prácticamente todo el ciclo de desarrollo digital, desde las especificaciones funcionales hasta la generación del prototipo, excluyendo únicamente el trazado físico o layout.

La correcta elección del nivel de abstracción en la descripción depende del objetivo de diseño. Por ejemplo, si se busca generar una implementación física mediante herramientas de síntesis, es preferible utilizar el nivel RTL. En cambio, para validar la funcionalidad de algoritmos complejos mediante simulación, el nivel algorítmico es más adecuado (Baena, 2010).

2.3. SystemVerilog: Lenguaje de Descripción y Verificación de Hardware

SystemVerilog es un lenguaje de descripción de hardware (HDL) y verificación funcional desarrollado como una extensión del lenguaje Verilog, con el objetivo de unificar el modelado estructural, la verificación orientada a objetos y la abstracción de sistemas en un solo entorno.

Su aparición responde a la necesidad creciente de los diseñadores digitales de contar con herramientas capaces de describir, validar y verificar sistemas digitales complejos, como SoCs (System-on-Chip), en múltiples niveles de abstracción.

A diferencia de VHDL, SystemVerilog incorpora elementos sintácticos y semánticos tanto de lenguajes de descripción como de programación (inspirado en C y C++), lo que lo convierte en un lenguaje híbrido y altamente expresivo. Fue estandarizado por el IEEE como el estándar IEEE 1800, inicialmente en 2005, y ha sido adoptado ampliamente en la industria por su compatibilidad con flujos de diseño RTL y metodologías de verificación como UVM (Universal Verification Methodology).

Enfoque de Verificación en SystemVerilog Una de las mayores fortalezas de SystemVerilog es su orientación a la verificación, lo que lo diferencia fundamentalmente de lenguajes como VHDL o Verilog puro.

2.4. Comparativa entre Lenguajes de Descripción de Hardware y Desarrollo de Software

Desde sus orígenes, las distintas herramientas y entornos de diseño asistido por computadora (CAD) han empleado lenguajes y formatos específicos para representar y gestionar los diversos elementos involucrados en el diseño electrónico. Algunos de estos lenguajes consistían en notaciones explícitas utilizadas por el usuario para describir entradas a determinadas herramientas, mientras que otros correspondían a formatos intermedios internos, optimizados para el procesamiento automatizado dentro del entorno CAD y generalmente ocultos al diseñador.

Sin embargo, estas descripciones se encontraban limitadas al ecosistema particular de cada herramienta, sin ofrecer portabilidad ni estandarización. La aparición de los lenguajes de descripción de hardware (HDL, por sus siglas en inglés) supuso un avance significativo al introducir un mayor grado de estandarización y al incorporar principios propios de la ingeniería de software en la especificación y modelado de hardware digital.

Desde el punto de vista sintáctico, los HDL presentan similitudes con los lenguajes de

programación de alto nivel (HLL), lo cual facilita su aprendizaje por parte de ingenieros con experiencia previa en desarrollo de software. Por ejemplo, Verilog comparte muchas características con el lenguaje C, mientras que VHDL hereda su estructura y semántica del lenguaje ADA. Estas semejanzas, si bien pueden acelerar la adopción de HDL, pueden también inducir errores conceptuales si se emplean estrategias propias del software en contextos donde se requiere una representación precisa del comportamiento físico del hardware.

Es fundamental que, cuando un modelo HDL esté destinado a la síntesis e implementación física (por ejemplo, en FPGAs o ASICs), el diseñador adopte una mentalidad orientada al hardware, describiendo la lógica con un enfoque estructural o de transferencia de registros (RTL), y no meramente algorítmico. Por otro lado, el uso de estructuras típicas del software puede ser apropiado cuando el objetivo sea únicamente la simulación funcional del sistema, permitiendo optimizar el rendimiento en las etapas de verificación y análisis temporal.

En resumen, los HDL son lenguajes formales de alto nivel, diseñados específicamente para representar circuitos electrónicos a diversos niveles de abstracción —desde compuertas lógicas básicas hasta sistemas digitales completos— y permiten un modelado flexible, estructurado y preciso del hardware, aprovechando conceptos tanto del diseño electrónico como del desarrollo de software, tal y como se muestran en la Figura (2.1).

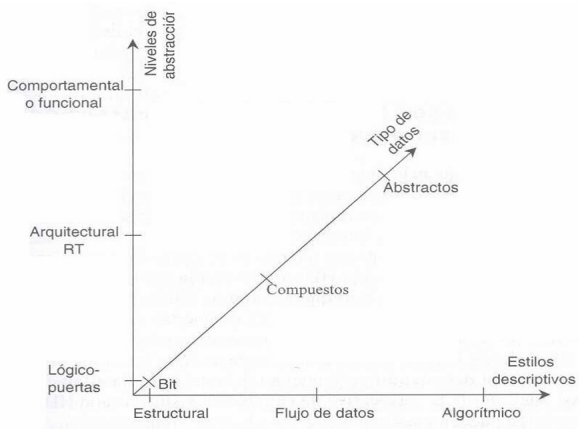


Imagen 2.1: Niveles de abstracción/precisión y estilos de modelado VHDL.

Los lenguajes de descripción de hardware (HDL) fueron desarrollados inicialmente con el propósito de modelar el comportamiento funcional de los componentes electrónicos, permitiendo su simulación previa a la implementación física. No obstante, también son ampliamente utilizados para la descripción estructural de circuitos digitales, facilitando su posterior síntesis y verificación mediante procesos de simulación validados (2.2).

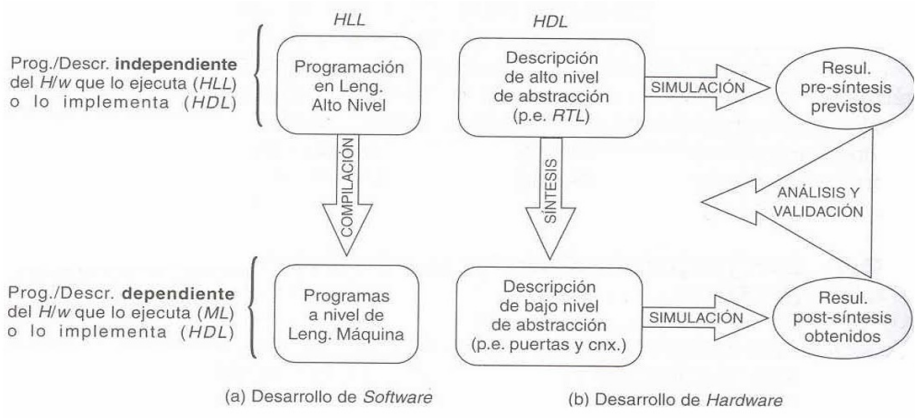


Imagen 2.2: Desarrollo en software versus hardware: (a) niveles de abstracción y lenguajes de alto nivel y (b) esquema básico del diseño descendente con HDL. .

A partir de las descripciones iniciales, los enfoques de diseño descendente (top-down) permiten aplicar procesos progresivos de síntesis que conducen gradualmente a niveles más detallados de implementación, hasta alcanzar una descripción física específica y dependiente de la tecnología utilizada. La validación en cada etapa del diseño se lleva a cabo mediante simulaciones y análisis funcionales, permitiendo iteraciones sucesivas de verificación y corrección hasta lograr un comportamiento conforme a los requisitos especificados (2.2(b)).

2.5. Descripción de la tarjeta de desarollo Altera DE2-115

La tarjeta Altera DE2-115, desarrollada por Terasic Technologies, es una plataforma de prototipado basada en la FPGA Cyclone IV EP4CE115F29C7N de Altera (ahora Intel), orien-

tada a entornos educativos, de investigación y desarrollo de sistemas digitales avanzados. Esta tarjeta ofrece una arquitectura versátil que permite implementar desde diseños lógicos básicos hasta sistemas digitales complejos, incluyendo sistemas embebidos, controladores personalizados, procesamiento de señales digitales (DSP) y prototipos de SoC (System-on-Chip).

Entre sus características principales se destaca la presencia de 114,480 elementos lógicos (LEs), 3.888 Kbits de memoria RAM embebida, y 528 pines de entrada/salida de propósito general (GPIO), lo que proporciona una gran flexibilidad para interactuar con múltiples dispositivos periféricos. La tarjeta también incluye memorias externas como SDRAM de 128 MB, SRAM de 2 MB, y Flash NOR de 2 MB, además de soporte para almacenamiento mediante tarjeta SD.

La DE2-115 incorpora interfaces esenciales para el desarrollo de sistemas interactivos, tales como puertos USB, Ethernet Gigabit, salida VGA, entradas y salidas de audio, además de un conjunto de dispositivos de entrada/salida integrados como interruptores, botones, LEDs y displays de siete segmentos. Adicionalmente, cuenta con un oscilador de cristal de 50 MHz y conectores de expansión compatibles con módulos adicionales.

Esta plataforma es ampliamente utilizada en laboratorios académicos y en entornos de investigación aplicada debido a su compatibilidad con herramientas de desarrollo como Quartus Prime, facilitando la implementación de diseños en lenguajes HDL como VHDL y SystemVerilog. Su capacidad para realizar simulación, verificación y síntesis de diseños en una sola herramienta la hace ideal para proyectos de educación superior en ingeniería electrónica, mecatrónica y sistemas digitales.

Características técnicas destacadas:

FPGA: Altera Cyclone IV EP4CE115F29C7N con 114,480 elementos lógicos (LEs)

- Memoria:
  - 2 MB SRAM
  - 128 MB SDRAM
  - 2 MB Flash NOR

    - Tarjeta SD para almacenamiento externo

- Interfaces de usuario:
    - 18 interruptores (switches) y 18 botones pulsadores
    - 18 LEDs rojos y 9 LEDs verdes
    - 8 displays de 7 segmentos
- Conectividad:
    - Puertos USB tipo A y B
    - Puerto Ethernet 10/100/1000 Mbps (Gigabit)
    - Entrada/salida VGA
    - Conectores de expansión GPIO de 40 pines
- Audio y Video:
    - Entrada y salida de audio (jack de 3.5 mm)
    - Entrada de señal VGA (con ADC) y salida VGA
- Reloj: oscilador de cristal de 50 MHz
- Programación y depuración: Soporte JTAG, compatible con Quartus II

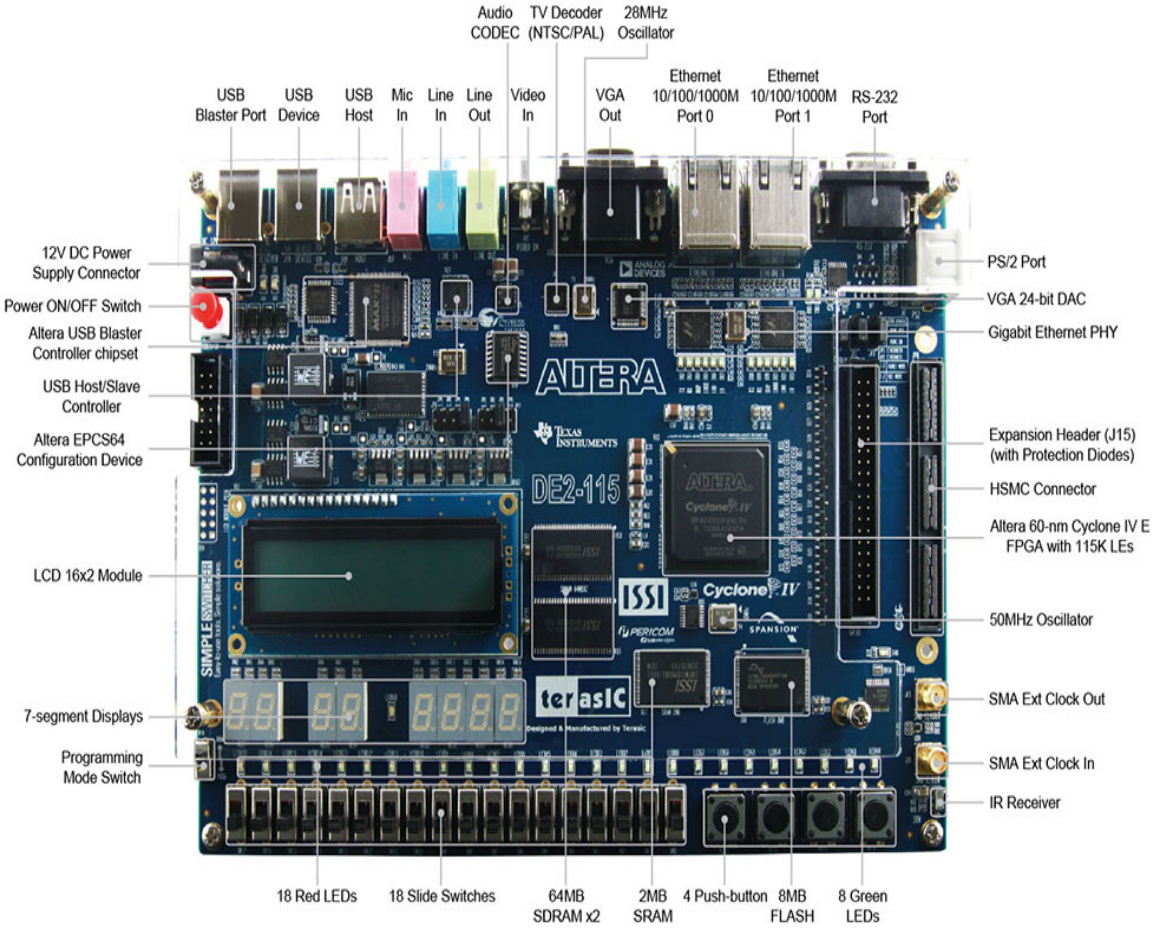


Imagen 2.3: Tarjeta de desarollo Altera DE2-115.

CAPÍTULO 3

Diseño de Plataforma para Transmisión Serial de Datos

3.1. Plataforma de Comunicación Serial para el Cifrado y Descifrado

La seguridad de la información, especialmente en lo que respecta a la confidencialidad, representa una de las principales preocupaciones en el ámbito de las comunicaciones. Para mitigar los riesgos de acceso no autorizado, es común la implementación de algoritmos criptográficos que aseguren que solo el receptor legítimo pueda interpretar los datos transmitidos.

Uno de los principales objetivos de este trabajo es desarrollar una plataforma basada en FPGA que permita integrar mecanismos de codificación de flujo para proteger la transmisión de datos entre dos dispositivos que se comunican mediante el protocolo UART. En lugar de utilizar UART en su forma tradicional, este protocolo es modificado y ampliado para incorporar una capa de seguridad directamente a nivel de hardware, mejorando así la confidencialidad de la información transmitida.

Para ello, se diseñó una plataforma base que incluye los módulos necesarios para enviar y

recibir datos, realizar la codificación y decodificación de los mensajes, y establecer la comunicación segura entre ambos extremos. Esta arquitectura ha sido planteada de forma flexible, permitiendo que el proceso de codificación se defina de manera genérica. De este modo, es posible integrar diferentes algoritmos criptográficos sin necesidad de rediseñar por completo el sistema, lo que facilita su adaptación a distintos niveles de seguridad o aplicaciones específicas.

Un aspecto fundamental para el desempeño de un codificador de flujo es que su velocidad de operación no afecte la eficiencia de la comunicación. Por esta razón, la plataforma está implementada en una FPGA, lo que permite ampliar y modificar el protocolo de comunicación para incluir una capa de seguridad directamente a nivel hardware. Así, el proceso de cifrado se realiza en tiempo real, garantizando que la protección de los datos no genere retrasos ni impacte negativamente en la velocidad de transmisión.

Se realiza la prueba y validación del desempeño de la plataforma mediante la comparación de su funcionamiento al implementar un protocolo personalizado de comunicacion para incrementar el grado de seguridad.

3.2. Protocolo Seguro para Comunicación Serial

Como parte del desarrollo de esta tesis, se aborda el diseño de un sistema de comunicación seguro, iniciando con el análisis de las características fundamentales de los sistemas de comunicación digital y los mecanismos para garantizar que la información transmitida sea recibida y correctamente interpretada únicamente por el destinatario autorizado. Durante este proceso analítico se identifican múltiples criterios que definen las propiedades esenciales para un algoritmo criptográfico eficiente, los cuales serán abordados detalladamente en este capítulo.

En el proceso de establecer los parámetros para una comunicación segura, se busca lograr este objetivo minimizando el impacto sobre los sistemas de comunicación existentes. Cualquier mecanismo implementado para garantizar la seguridad debe ser transparente para el usuario y operar en tiempo real, sin afectar las características esenciales de la comunicación, tales como la calidad y la velocidad de transmisión.

Actualmente, la mayoría de los sistemas de comunicación emplean transmisión de datos serial, generalmente en modo full-duplex, para el intercambio de información. En la búsqueda de sistemas de comunicación seguros, funcionales y eficientes, es necesario analizar y evaluar en tiempo real la ejecución de un algoritmo criptográfico estándar. Posteriormente, será posible implementar y evaluar otros algoritmos conforme a las necesidades del sistema.

A partir de los lineamientos previamente establecidos, se definen las características fundamentales para el diseño de la plataforma de comunicación criptográfica, las cuales son:

a) Integración transparente: El sistema de cifrado debe ser incorporado en la línea de comunicación sin necesidad de modificar el equipo de comunicaciones existente, manteniendo así la compatibilidad con la infraestructura actual.

b) Compatibilidad con comunicación serial full-duplex asíncrona: La plataforma debe ser capaz de manejar flujos de datos bidireccionales simultáneos (full-duplex) en modo asíncrono, soportando múltiples velocidades de transmisión comúnmente utilizadas en entornos industriales o embebidos.

c) Flexibilidad criptográfica: Debe ofrecer la posibilidad de cambiar el algoritmo de cifrado de manera sencilla, permitiendo al usuario seleccionar entre distintas opciones según los requerimientos de seguridad del sistema.

d) Procesamiento en tiempo real: Las operaciones de cifrado y descifrado deben ejecutarse de forma continua y en tiempo real, sin interrupciones en el flujo de información, garantizando la integridad y la eficiencia de la comunicación.

Para alcanzar el primer objetivo —la integración transparente del sistema de seguridad— se diseñaron módulos hardware que pueden ser instalados en ambos extremos de la línea de comunicación. Estos dispositivos pueden colocarse directamente en los puntos finales del canal de transmisión, o bien actuar como una interfaz entre el usuario y el equipo de comunicación. La ubicación óptima dependerá del contexto de uso, de las características del equipo involucrado y del medio físico de transmisión utilizado.

La Figura (3.1) muestra un diagrama de bloques que ilustra cómo puede integrarse el sistema de seguridad dentro de una arquitectura de comunicación serial, permitiendo el cifrado de los datos sin alterar la funcionalidad básica del canal.

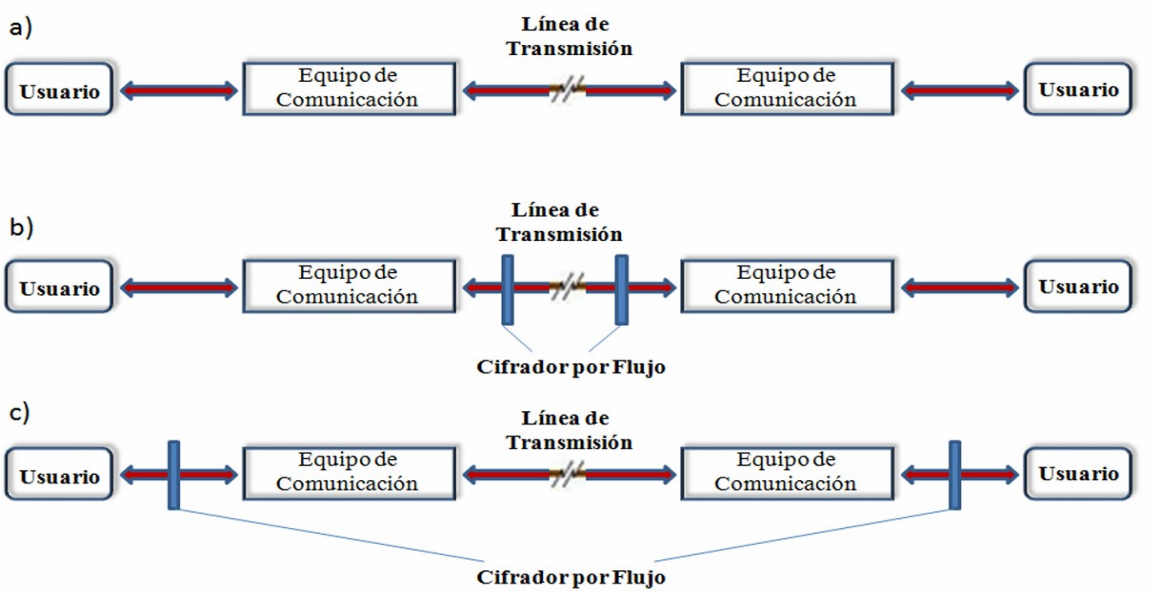


Imagen 3.1: Diagrama de bloques de un sistema de comunicación simplificado. a) Sistema Inseguro. b) Sistema con el cifrado en los extremos de la línea de la transmisión. c) Sistema con el cifrado como interfaz entre el usuario y el equipo de comunicación.

El mecanismo encargado del cifrado debe incluir, en términos generales, un módulo central de control responsable de coordinar todo el proceso de cifrado y descifrado. Este módulo gestiona el flujo de datos entre dos interfaces de comunicación serial, actuando como intermediario entre el transmisor y el receptor. Su funcionamiento se basa en controlar la entrada y salida de datos cifrados y descifrados de acuerdo con la lógica definida por el algoritmo criptográfico. La Figura (3.2) ilustra la estructura general del sistema, destacando el papel del controlador central dentro del esquema de comunicación segura.

Imagen 3.2: Diagrama de bloques del dispositivo de cifrado y Descifrado.

Con el objetivo de garantizar que el proceso de cifrado y descifrado se ejecute de forma continua, sin interrumpir ni degradar el flujo original de información, se opta por el uso de algoritmos de cifrado por flujo. Este tipo de algoritmos permite el procesamiento de los datos en tiempo real, bit a bit o byte a byte, lo cual es especialmente adecuado para sistemas de comunicación serial donde la información se transmite de manera secuencial.

3.3. Diseño de la Plataforma Criptográfica

Para llevar a cabo el diseño de la plataforma criptográfica, se parte de las especificaciones definidas en la sección anterior. El método comúnmente empleado para el intercambio de información es la transmisión de datos en serie. Por ello, se implementan dos puertos "Universal Asynchronous Receiver/Transmitter" (UART), los cuales permiten la conexión de la plataforma con el medio de comunicación: uno destinado a recibir y enviar datos en su formato original hacia y desde el usuario, y otro encargado de manejar la información cifrada a través del canal de transmisión. Esta configuración facilita la incorporación del proceso de cifrado en cualquier sistema de comunicación digital que utilice transmisión serial full-dúplex asincrónica, requiriendo únicamente el desarrollo de una interfaz adecuada según el punto de integración dentro del sistema.

Por otro lado para poner la plataforma criptográfica en ejecución se decide utilizar un FPGA, aprovechando sus características, incluyendo su operación de alta velocidad, bajo costo, facilidad de empleo y reconfiguración. En trabajo se decidió utilizar una tarjeta de desarrollo con un EP4CE115integrado, de la familia a Cyclone® IVSpartan. Este FPGA es de tamaño

medio, con el equivalente a 114,480 mil compuertas lógicas. El integrado tiene gran capacidad de poder integrar la plataforma criptográfica junto con cualquier bloque que se quiera cifrar y evaluar. Esta plataforma de desarrollo funciona a 50 [MHz], que permite configurar el UART para funcionar a una velocidad de 3.125 [Mbps], que es bastante para funcionar en tiempo real como la mayor parte de los sistemas de comunicación.

Ahora bien para probar la operación total del cifrador, que podría ser integrado en una línea de comunicación, primero se desarrolló el esquema de cifrado que se muestra en la Figura (3.3). Esta implementación consiste de un módulo con una máquina de estados que controle al dispositivo, maneje la operación de los UART y que lleve a cabo los procesos de cifrado y descifrado.

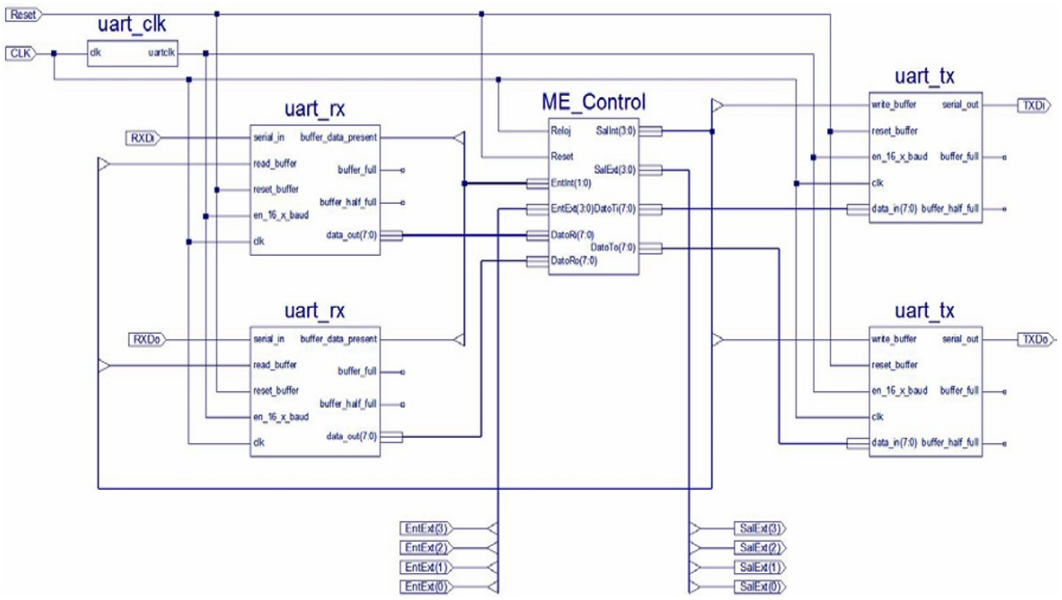


Imagen 3.3: Diseño de la Plataforma integrando el proceso de cifrado y el bloque de control.

Este dispositivo se emplea para validar el funcionamiento de la plataforma criptográfica, permitiendo la ejecución del proceso de cifrado dentro de un sistema de comunicación. Durante las pruebas experimentales, se utilizaron dos de estos dispositivos conectados entre dos computadoras que se comunicaban mediante un puerto serial RS-232. Los resultados obtenidos

a partir de estas pruebas se detallan en el capítulo ####, correspondiente a los resultados.

Con este diseño se logra verificar el funcionamiento general de la plataforma criptográfica; sin embargo, no se satisfacen completamente los requisitos establecidos, particularmente en lo referente a la flexibilidad para cambiar el algoritmo de cifrado. Esto se debe a que el módulo criptográfico está integrado dentro de la máquina de estados del bloque de control, lo que implica que incluso un cambio simple en el algoritmo requiere modificaciones estructurales en todo el sistema.

Para solucionar esta limitación, se propone una reestructuración del diseño, separando el elemento de seguridad del bloque de control. Así, los procesos de cifrado y descifrado se implementan en un módulo independiente, permitiendo que el bloque de control se limite exclusivamente a gestionar la comunicación a través de los UARTs, una vez que los datos han sido procesados por el bloque criptográfico.

Se tiene especial cuidado en proporcionar al bloque de cifrado/descifrado una interfaz genérica que le permita interactuar con el bloque de control sin depender del algoritmo utilizado. Para ello, se define un protocolo simple de handshake que permite al bloque de cifrado/descifrado recibir una señal cuando hay datos listos para ser procesados, y notificar al bloque de control una vez finalizado el procesamiento. Este mecanismo asegura que no se pierda información, evitando que se envíen nuevos datos al cifrador antes de que haya finalizado el procesamiento de los datos actuales.

Si bien el alcance de esta tesis se limita a la implementación del algoritmo de cifrado CESAR dentro de la plataforma de comunicación, el diseño apunta a una arquitectura escalable que permita integrar distintos algoritmos criptográficos. De esta manera, cuando se requiera implementar un nuevo dispositivo de cifrado/descifrado, bastará con desarrollar el bloque correspondiente al nuevo algoritmo y conectarlo a la plataforma para obtener un sistema completamente funcional. El dispositivo resultante puede observarse en la Figura (3.4).

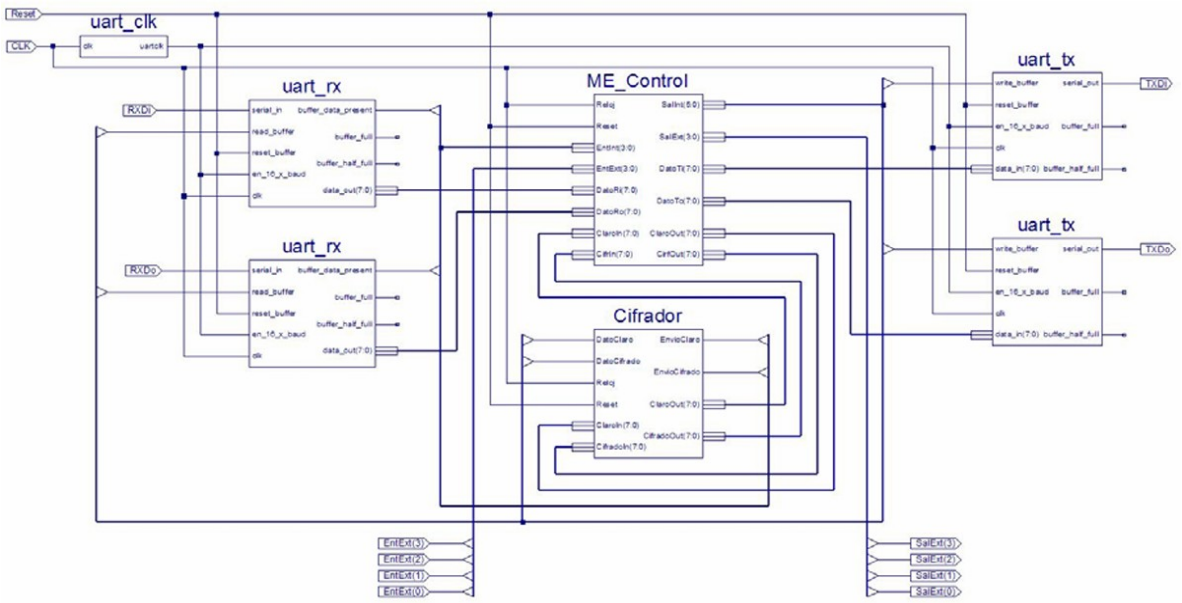


Imagen 3.4: Diseño de la Plataforma Criptográfica incorporando el proceso de cifrado/descifrado como un bloque independiente.

3.4. Aspectos Clave del UART

La unidad UART se encarga de controlar los puertos y dispositivos serie destinados a la comunicación de datos. En la tarjeta de desarrollo Altera DE2-115 se encuentra implementado un puerto RS-232; sin embargo, este no fue utilizado en el presente diseño. En su lugar, se optó por la personalización de los pines de comunicación a través del Expansion Header (J15), con el propósito de facilitar la integración práctica del sistema, tal como se ilustra en la Figura (2.3). Se desarrollaron los bloques necesarios para generar el módulo encargado de implementar una UART personalizada en lenguaje Verilog, destinada a los pines correspondientes del Expansion Header.

En el contexto del diseño digital mediante RTL sobre FPGA, la Unidad de Recepción-Transmisión Asíncrona (UART) cumple funciones esenciales, entre las que destacan la gestión de interrupciones provenientes de los dispositivos conectados al puerto serie y la conversión de datos entre los formatos paralelo y serie. En este proceso, los datos en formato paralelo provenientes del bus del sistema son transformados a formato serie para su transmisión a través de la interfaz de comunicación, mientras que los datos recibidos en formato serie son convertidos nuevamente a formato paralelo para su procesamiento interno.

3.4.1. Propiedades Principales del UART

Para fines demostrativos del proceso de cifrado y con el objetivo de verificar la correcta transmisión de la información, se implementó la comunicación estándar hacia computadoras personales. Es importante señalar que no es posible establecer una comunicación directa entre la tarjeta de desarrollo Altera DE2-115 y un ordenador, debido a que uno de los objetivos de este trabajo consistió en modificar el protocolo UART para incorporar una capa adicional de seguridad a nivel de hardware. En este sentido, si la tarjeta se conecta directamente a un ordenador, este únicamente dispondrá de la comunicación bajo el estándar UART original, lo que imposibilita una transmisión exitosa entre ambas plataformas (PC y FPGA). Para solventar esta limitación, se empleó una segunda FPGA del mismo modelo (Altera DE2-115),

en la cual se implementó la misma arquitectura. De esta forma, la comunicación entre ambas FPGA resulta exitosa y adecuada, considerando que en una aplicación real no se requerirá del uso de ordenadores para el monitoreo de la información.

3.5. Principio de Operación de la UART

En investigaciones previas se han propuesto mecanismos de seguridad sobre UART, pero en su mayoría dependen de software, lo que limita la eficiencia y vulnerabilidad ante ataques. Otros estudios han demostrado que la implementación de algoritmos criptográficos en FPGA, como AES en modo CTR, es factible y efectiva para mejorar la seguridad sin afectar el rendimiento significativamente.

3.5.1. Mecanismo de transmisión y recepción UART

Un UART, debido a la forma en que están construidos el transmisor y el receptor, no opera de manera sincronizada; sin embargo, ambos emplean un tiempo de referencia con cierta tolerancia que permite la transferencia serial de cada byte de datos. La transmisión se realiza de forma serial, enviando primero el bit menos significativo (LSB) a una razón de bits determinada (tasa en baud), la cual es conocida tanto por el transmisor como por el receptor. De esta manera, el transmisor puede iniciar el envío de datos en cualquier momento, mientras que el receptor requiere un mecanismo que le permita identificar cuándo ha sido transmitido el primer bit (LSB). Este proceso se logra mediante el envío de una señal de inicio (start bit), activada en nivel bajo durante la duración de un bit, como se muestra en la Figura (3.5).

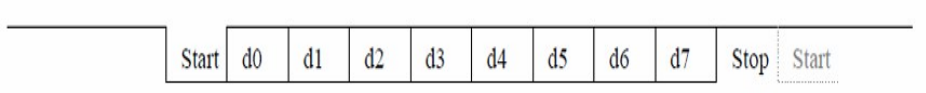


Imagen 3.5: Esquema de tiempo para la transmisión de un byte.

El receptor utiliza el flanco de bajada del bit de inicio para activar un circuito de temporización interno. Este tiempo de referencia permite muestrear la señal de entrada serial aproximadamente a la mitad de la duración de cada bit de datos, momento en el que se espera que el valor del bit sea estable. Una vez que se ha leído el último bit de datos (MSB), el receptor verifica que el bit de parada esté en nivel alto, como se espera, lo cual confirma que la operación de recepción se realizó correctamente Figura (3.6).

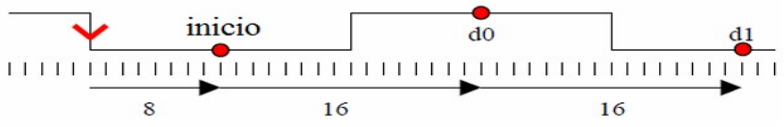


Imagen 3.6: Esquema temporal para la recepción de datos.

El receptor se vuelve a sincronizar utilizando el flanco de bajada de cada bit de inicio mediante su circuito temporal interno. Para que la transmisión y recepción funcionen correctamente, el tiempo de ambos sistemas solo necesita coincidir dentro de una tolerancia de aproximadamente la mitad del período de un bit en cada 10 períodos de bit. Esta precisión del 5 % es difícil de alcanzar con sistemas digitales convencionales.

Al igual que muchas implementaciones UART, estos macros esperan un tiempo de referencia proporcionado mediante una señal habilitadora, denominada "en_16_x_baud", que se aplica a razón de una vez por cada 16 bits transmitidos.

3.5.2. Criterios para la terminación de la transmisión UART

En condiciones normales, una línea serial se mantiene activa en nivel alto, de modo que un nuevo bit de inicio se identifica mediante un flanco de bajada. Sin embargo, bajo una condición de fallo, el transmisor puede mantener continuamente la línea en nivel bajo (por ejemplo, por falta de alimentación). En este caso, el receptor interpretará el nivel bajo como un bit de inicio seguido de bits de datos en cero. Como consecuencia, el bit de parada no será válido y la trama incorrecta será descartada, Figura (3.7).

El receptor permanecerá a la espera hasta que la línea vuelva a nivel alto y únicamente se

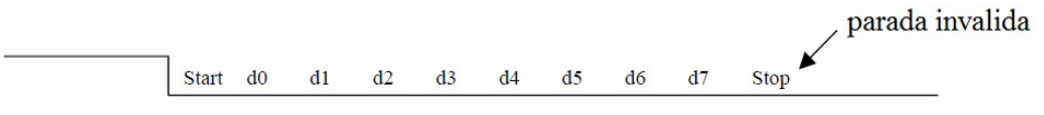


Imagen 3.7: Esquema temporal de una parada invalida.

volverá a sincronizar con el siguiente flanco de bajada correspondiente al bit de inicio Figura 3.8).

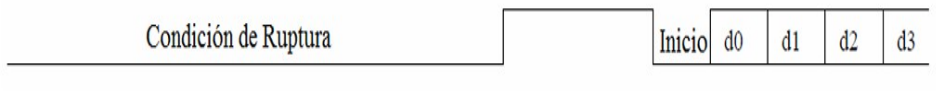


Imagen 3.8: Esquema temporal de una operacion de ruptura.

El macro del transmisor, de forma natural, no realizará ninguna transmisión bajo una condición de fallo. En cambio, el macro del receptor reconoce esta situación y continúa operando según lo descrito previamente.

3.5.3. Tasa de baudios de la UART

A partir de los macros de transmisión y recepción se generan los tiempos de la señal de referencia "en_16_x_baud". Como indica su nombre, esta señal debe aplicarse al macro de la tasa de transmisión, que opera a 16 veces la velocidad de bits deseada Figura (3.9).

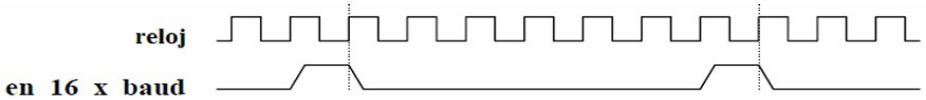


Imagen 3.9: Esquema temporal en BAUD's.

De esta forma, la señal se utiliza como un reloj habilitador dentro de los macros. Debe ser generada a partir de un reloj sincronizado y tener una duración de pulso equivalente a un solo ciclo de reloj (lo que requiere, al menos, una frecuencia de reloj máxima dividida entre 16).

3.5.4. Implementación RTL del Bloque UART_Tx y UART_rx

El UART receptor está formado por un archivo en Verilog. El nombre del archivo es ”receiver.sv” que esta implementando en combinación con los modulos de ”binary_to_7seg.sv” y ”transmitter.sv”, los modulos tanto de receiver y transmitter cuentan con su capa de cifrado y descifrado, incluyendo las variantes de rotación de bits y la selecció de rango de esta misma.

3.5.5. Metodología de Verificación del UART mediante Testbench

Debido a las capacidades de prototipado que proporciona la herramienta Quartus, resulta posible realizar una validación exhaustiva del protocolo UART mediante el uso de un testbench. Esta metodología permite comprobar de manera precisa el cumplimiento de los tiempos, secuencias de transmisión y recepción, así como la integridad funcional del sistema previo a su síntesis e implementación en el dispositivo FPGA Figura (3.10).

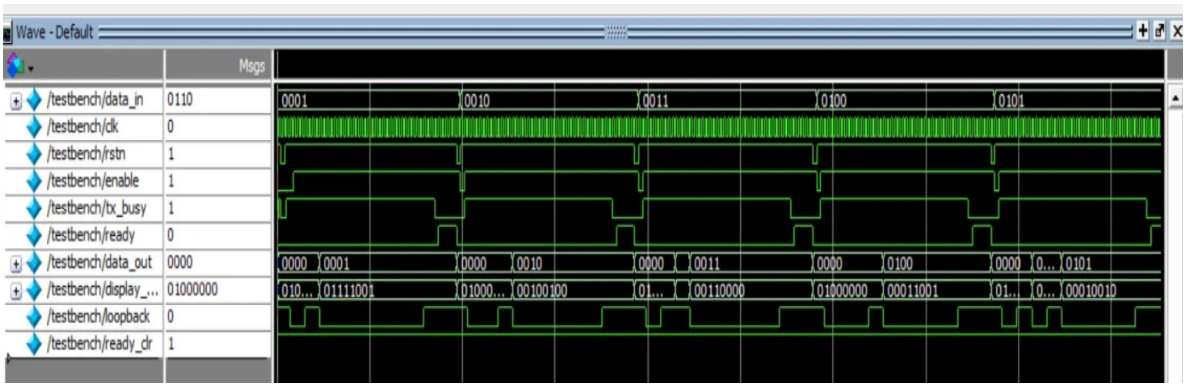
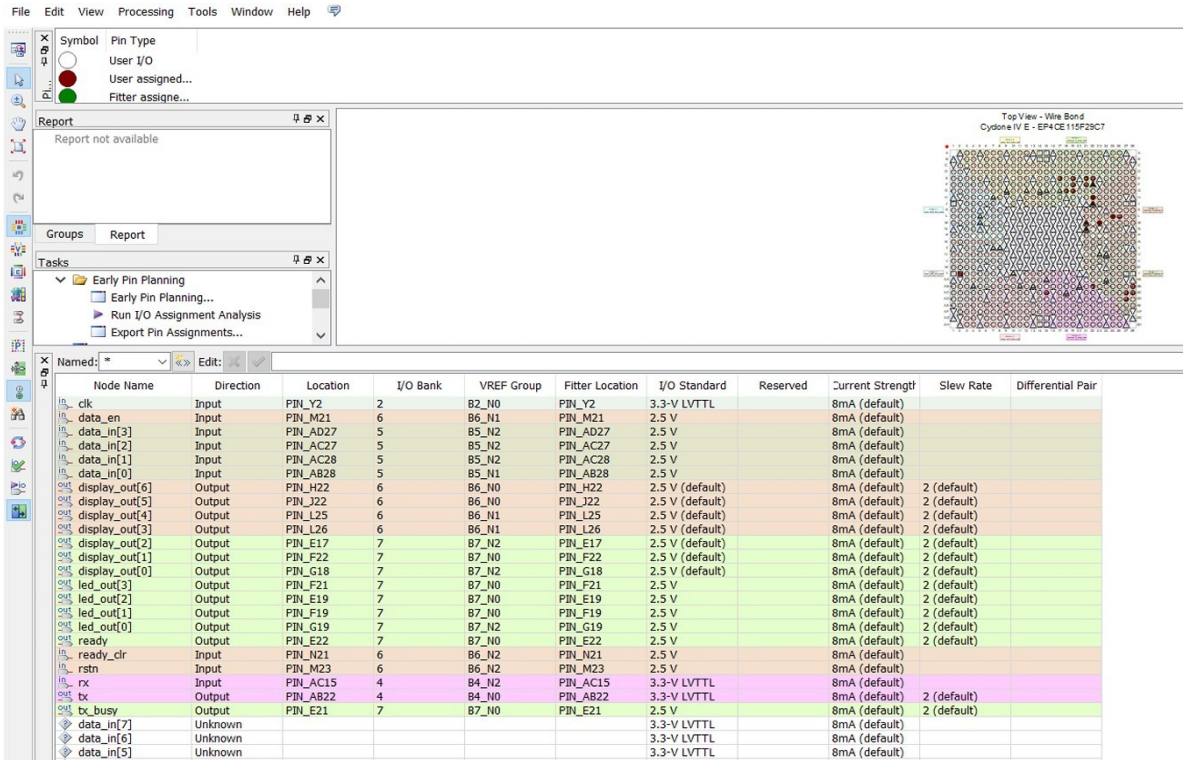


Imagen 3.10: Salida del testbench para la validación del protocolo UART.

3.5.6. Configuración de las Interfaces Seriales con las TarjetasAltera DE2–115

En la Figura (3.11) se ilustra la definición de los pines destinados a la comunicación serial, la cual fue realizada dentro del software Quartus. Este procedimiento permite establecer de manera explícita las asignaciones físicas entre los pines de la FPGA y las señales lógicas del diseño, asegurando así la correcta implementación del sistema en el dispositivo reconfigurable.



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair
clk	Input	PIN_Y2	2	B2_N0	PIN_Y2	3.3-V LVTTL		8mA (default)		
data_en	Input	PIN_M21	6	B6_N1	PIN_M21	2.5 V		8mA (default)		
data_in[3]	Input	PIN_AD27	5	B5_N2	PIN_AD27	2.5 V		8mA (default)		
data_in[2]	Input	PIN_AC27	5	B5_N2	PIN_AC27	2.5 V		8mA (default)		
data_in[1]	Input	PIN_AC28	5	B5_N2	PIN_AC28	2.5 V		8mA (default)		
data_in[0]	Input	PIN_AB28	5	B5_N1	PIN_AB28	2.5 V		8mA (default)		
display_out[6]	Output	PIN_H22	6	B6_N0	PIN_H22	2.5 V (default)		8mA (default)	2 (default)	
display_out[5]	Output	PIN_J22	6	B6_N0	PIN_J22	2.5 V (default)		8mA (default)	2 (default)	
display_out[4]	Output	PIN_L25	6	B6_N1	PIN_L25	2.5 V (default)		8mA (default)	2 (default)	
display_out[3]	Output	PIN_L26	6	B6_N1	PIN_L26	2.5 V (default)		8mA (default)	2 (default)	
display_out[2]	Output	PIN_E17	7	B7_N2	PIN_E17	2.5 V (default)		8mA (default)	2 (default)	
display_out[1]	Output	PIN_F22	7	B7_N0	PIN_F22	2.5 V (default)		8mA (default)	2 (default)	
display_out[0]	Output	PIN_G18	7	B7_N2	PIN_G18	2.5 V (default)		8mA (default)	2 (default)	
led_out[3]	Output	PIN_F21	7	B7_N0	PIN_F21	2.5 V		8mA (default)	2 (default)	
led_out[2]	Output	PIN_E19	7	B7_N0	PIN_E19	2.5 V		8mA (default)	2 (default)	
led_out[1]	Output	PIN_F19	7	B7_N0	PIN_F19	2.5 V		8mA (default)	2 (default)	
led_out[0]	Output	PIN_G19	7	B7_N2	PIN_G19	2.5 V		8mA (default)	2 (default)	
ready	Output	PIN_E22	7	B7_N0	PIN_E22	2.5 V		8mA (default)	2 (default)	
ready_clr	Input	PIN_N21	6	B6_N2	PIN_N21	2.5 V		8mA (default)		
rstn	Input	PIN_M23	6	B6_N2	PIN_M23	2.5 V		8mA (default)		
rx	Input	PIN_AC15	4	B4_N2	PIN_AC15	3.3-V LVTTL		8mA (default)		
tx	Output	PIN_AB22	4	B4_N0	PIN_AB22	3.3-V LVTTL		8mA (default)	2 (default)	
tx_busy	Output	PIN_E21	7	B7_N0	PIN_E21	2.5 V		8mA (default)	2 (default)	
data_in[7]	Unknown					3.3-V LVTTL		8mA (default)		
data_in[6]	Unknown					3.3-V LVTTL		8mA (default)		
data_in[5]	Unknown					3.3-V LVTTL		8mA (default)		

Imagen 3.11: Seleccion de pines mediante el Software Quartus.

En la Figura (3.12) se observa la conexión de los puertos definidos por el usuario, lo cual pone de manifiesto la flexibilidad inherente a las FPGA. Esta característica permite que los pines de entrada y salida no estén restringidos a conectores predeterminados, sino que puedan ser asignados libremente por el diseñador. De esta manera, además de optimizar la arquitectura del sistema, se incrementa la dificultad para que un agente externo pueda identificar los pines específicos donde se lleva a cabo la comunicación (Brown & Vranesic, 2013).

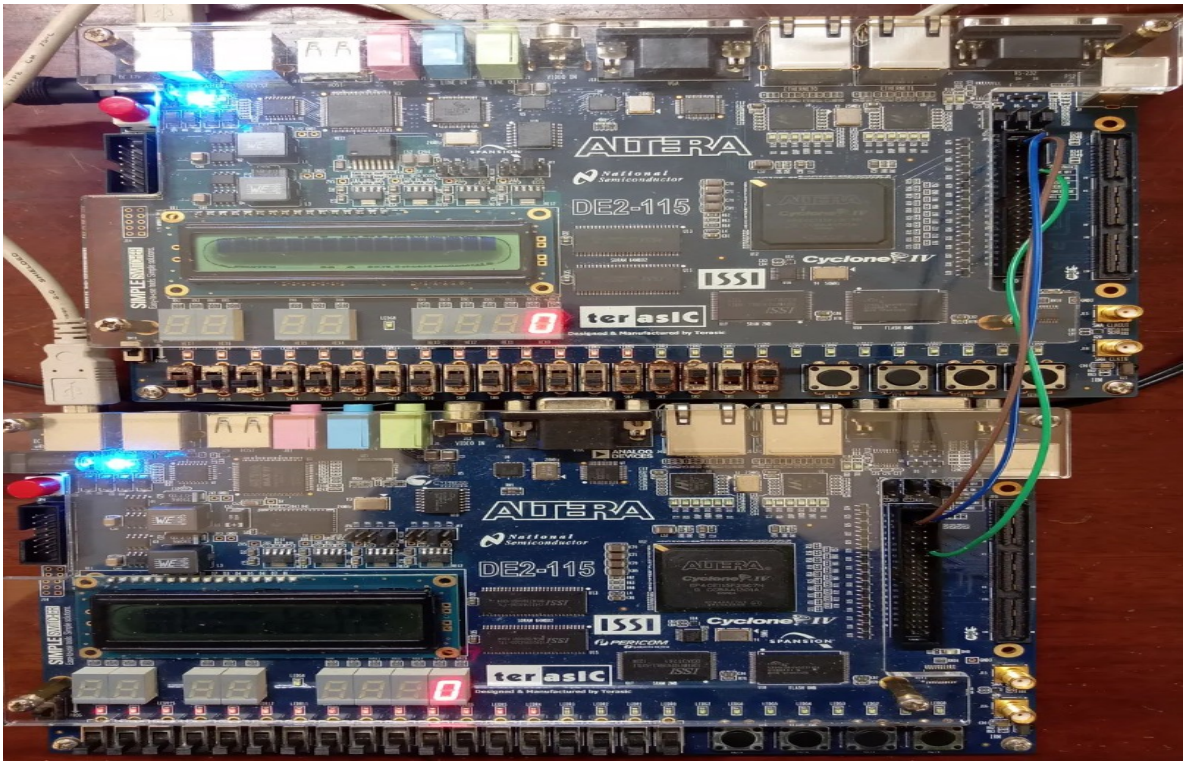


Imagen 3.12: Conexion de las dos plataformas Altera DE2–115 mediante la selección de pines del modulo Jtag .

A continuación, en la Figura (3.13), se presenta el esquema de las conexiones empleadas para la validación del sistema. Con el propósito de verificar el correcto funcionamiento de los módulos implementados, se estableció una comunicación entre dos ordenadores conectados en los extremos, lo que permitió comprobar la transmisión y recepción adecuada de los datos.

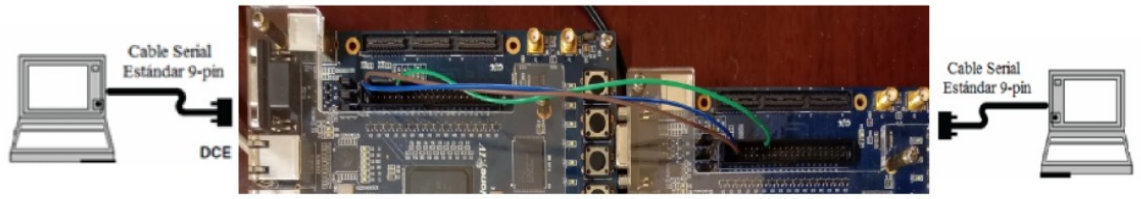


Imagen 3.13: Esquema de conexiones para la plataforma de cifrado y su validación.

CAPÍTULO 4

RESULTADOS

4.1. Resultados

En este último capítulo se presentan los resultados obtenidos durante el desarrollo de las implementaciones propuestas, tanto en relación con los objetivos particulares como con el objetivo general de la presente tesis. Algunos de estos resultados se contrastan con los publicados en los estándares correspondientes, tanto para el algoritmo de cifrado seleccionado y programado, como para el modo de operación implementado. Asimismo, se incluyen resultados derivados de la aplicación desarrollada para la integración de los mecanismos de seguridad, específicamente aquellos vinculados a las pruebas realizadas en la plataforma criptográfica destinada a la comunicación serial segura. Finalmente, para la consulta de los códigos fuente y ciertos detalles específicos de la programación, el lector puede remitirse a los anexos ubicados al final de este capítulo.

Con el fin de validar el mecanismo de ofuscación propuesto como una mejora al cifrado César tradicional, se llevaron a cabo una serie de experimentos tanto en el ámbito de software

como en el de hardware. El objetivo principal fue demostrar que la integración de una capa adicional de ofuscación no solo incrementa la complejidad del proceso de descifrado del mensaje encriptado, sino que también preserva la equivalencia funcional entre el prototipo en software y su implementación en hardware.

La primera etapa de la experimentación consistió en el desarrollo de un script en Python que implementaba el cifrado César combinado con la técnica de ofuscación propuesta. Este entorno permitió un prototipado rápido y la verificación de la corrección funcional. Se utilizaron diversos conjuntos de datos de entrada, incluyendo cadenas alfanuméricas y secuencias binarias, los cuales fueron sometidos a pruebas de cifrado y descifrado. Posteriormente, se aplicaron medidas estadísticas para evaluar la distribución de los caracteres del texto cifrado Figura (4.1, con el objetivo de confirmar que la ofuscación lograba una reducción en los patrones de frecuencia predecibles, típicamente explotables en los cifrados clásicos por sustitución. Los gráficos resultantes evidenciaron un aplanamiento notable en las distribuciones de frecuencia de los caracteres, lo que indica que el cifrado modificado mitiga de manera efectiva una de las principales debilidades del esquema César tradicional.

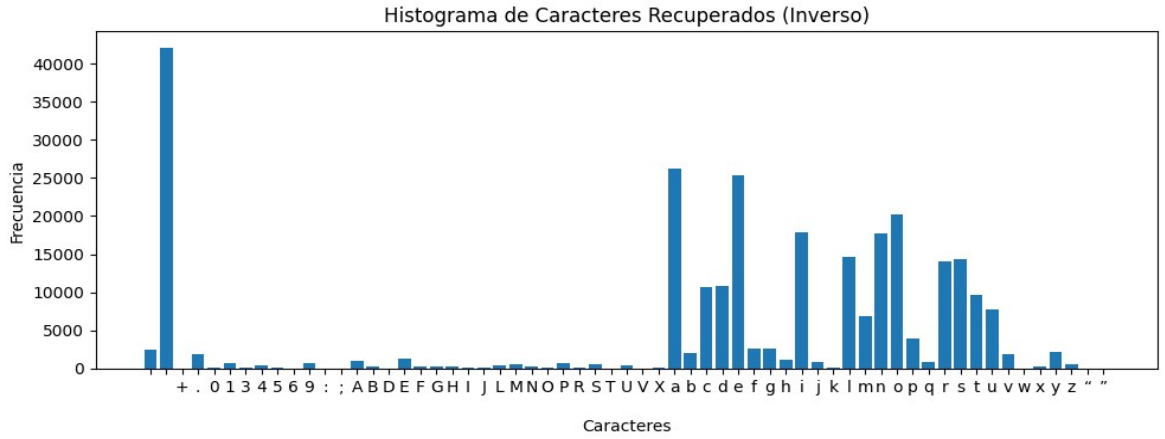


Imagen 4.1: Distribución de texto simple en español.

Tras la validación exitosa en el dominio de software, el algoritmo fue traducido a una

descripción de hardware adecuada para su implementación en FPGA. Específicamente, se diseñó un bloque RTL (Register Transfer Level) utilizando VHDL con el fin de replicar las operaciones tanto del cifrado César como de la capa de ofuscación. La elección de RTL aseguró que el diseño permaneciera sintetizable y pudiera desplegarse físicamente en la plataforma FPGA Altera DE2-115. Este proceso de traducción requirió especial atención al temporizado, la utilización de recursos y la gestión de la concurrencia, ya que la ejecución en hardware introduce restricciones que no están presentes en la ejecución secuencial del software.

La simulación del bloque RTL se llevó a cabo antes de la síntesis, garantizando que la salida coincidiera con la implementación de referencia en Python en todos los vectores de prueba. El análisis de formas de onda confirmó la correcta alineación de las rutas de datos, la sincronización del reloj y el comportamiento de la lógica de control. Posteriormente, el diseño fue sintetizado, colocado y ruteado sobre la FPGA. Los reportes de utilización de recursos indicaron que la lógica adicional de ofuscación introdujo únicamente un incremento moderado en el consumo de elementos lógicos, manteniéndose ampliamente dentro de la capacidad del dispositivo Cyclone IV.

Para validar aún más la efectividad del diseño en hardware, se recolectaron resultados experimentales directamente de la implementación en la FPGA. Secuencias de texto de entrada fueron encriptadas, procesadas a través del mecanismo de ofuscación y posteriormente desencriptadas en tiempo real. Los textos cifrados resultantes fueron nuevamente sometidos a un análisis estadístico de frecuencias (véase Fig. 4.2). La comparación entre las distribuciones generadas en software y en hardware confirmó la equivalencia funcional: ambas plataformas alcanzaron niveles similares de dispersión e imprevisibilidad de caracteres.

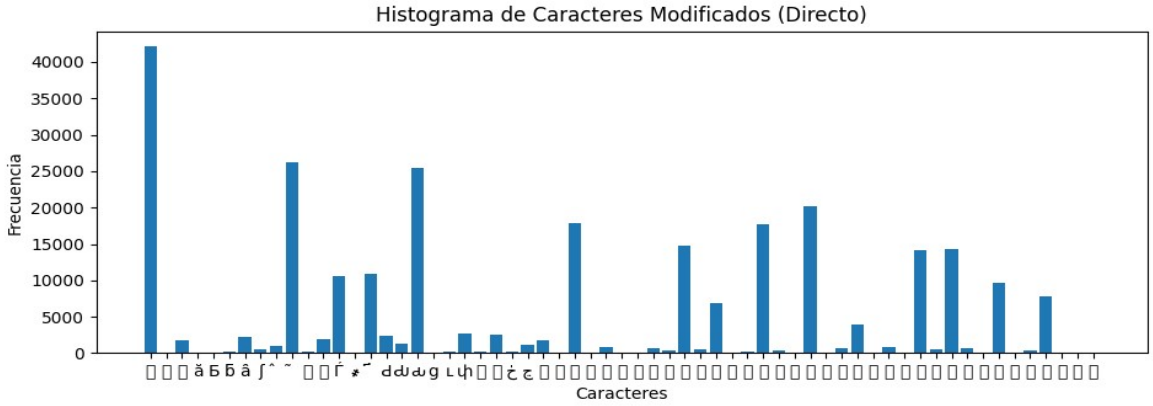


Imagen 4.2: Distribución del texto cifrado producido por la capa de ofuscación implementada en hardware.

Con el propósito de evaluar la robustez del método de ofuscación implementado, se diseñó un escenario controlado en el cual se introdujo un agente externo destinado a interceptar la comunicación (man in the middle) establecida entre las FPGA. En este contexto, se asume que dicho agente ha logrado identificar las líneas físicas responsables de la transmisión de información bajo el protocolo UART. Para llevar a cabo la interceptación, se empleó una tarjeta de desarrollo Arduino Uno (véase Figura 4.3), configurada de acuerdo con los parámetros definidos por dicho protocolo. Adicionalmente, se realizó un barrido sistemático de los baudios estándar con el fin de determinar si el agente externo sería capaz de recuperar información significativa a partir de los datos cifrados y ofuscados.

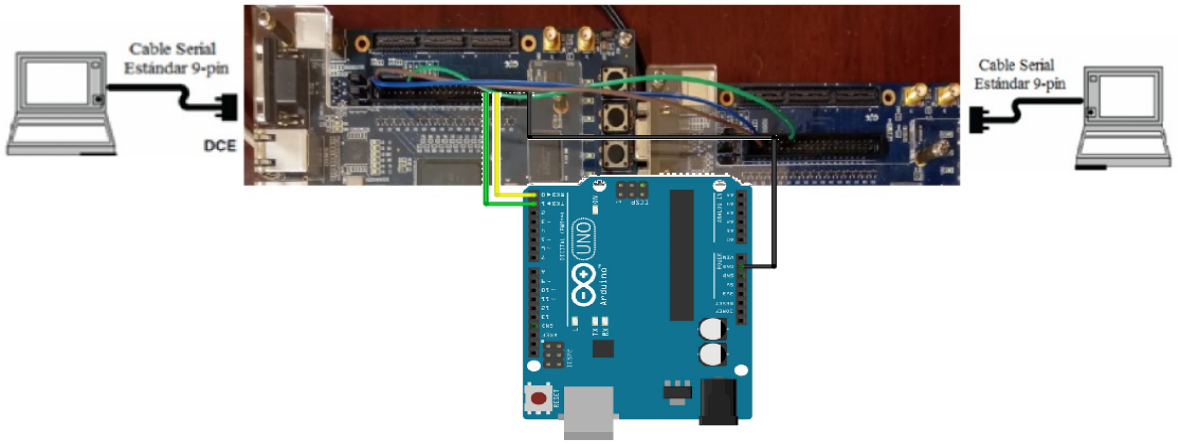


Imagen 4.3: Configuración experimental en la que se observa la conexión entre las FPGA y la tarjeta Arduino Uno utilizada como agente externo para intentar la interceptación de la comunicación UART.

Los resultados experimentales obtenidos permiten confirmar la efectividad del mecanismo de ofuscación implementado en el sistema de comunicación. Durante las pruebas, se recreó un escenario de ataque del tipo man-in-the-middle, en el cual un agente externo intentó interceptar la comunicación establecida entre las FPGA. Para ello, dicho agente fue configurado con los parámetros estándar del protocolo UART y realizó un barrido completo de las velocidades en baudios con el objetivo de sincronizarse con el flujo de datos transmitidos.

Una vez establecida la sincronización, el agente externo logró capturar correctamente las tramas UART generadas por el sistema. Sin embargo, debido al proceso de cifrado César y a la capa de ofuscación por rotación de bits aplicada en hardware, la información recuperada no presentó ninguna estructura interpretable. En otras palabras, aunque el atacante consiguió acceder físicamente a la línea de transmisión y leer los datos, el contenido obtenido carecía por completo de significado semántico o lingüístico.

Este comportamiento se aprecia claramente en la Figura 4.4, donde se muestra la salida obtenida por el dispositivo de escucha. En dicha salida únicamente se observan caracteres aleatorios y sin coherencia, lo cual evidencia que la transformación criptográfica impide la recuperación del mensaje original sin la clave correspondiente. Este resultado es particularmente relevante, ya que demuestra que la ofuscación no solo opera correctamente en condiciones de simulación, sino también bajo condiciones reales de operación y frente a un intento explícito de interceptación.

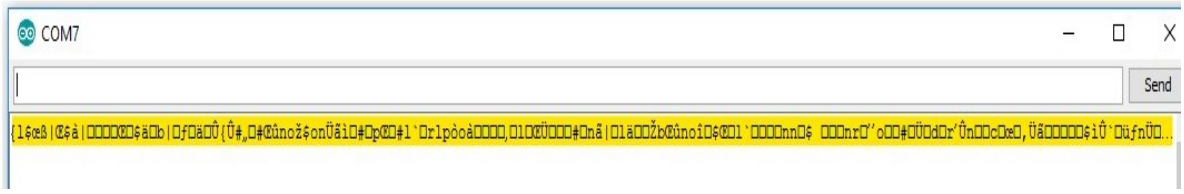


Imagen 4.4: Salida capturada por el agente externo durante el intento de interceptación. Se observa que los datos recuperados carecen de significado, mostrando únicamente caracteres no legibles debido al proceso de cifrado y ofuscación aplicado.

En conjunto, estos hallazgos validan que la arquitectura propuesta proporciona un nivel adicional de protección para el canal UART, mitigando la posibilidad de que un agente no autorizado pueda leer o reconstruir la información transmitida. Por tanto, se confirma que el método de ofuscación cumple su objetivo principal: mantener la confidencialidad del contenido incluso ante un atacante con acceso directo al medio físico de comunicación.

Anexos

Anexo 1. Desarrollo del Programa BitChange en Python para la Evaluacion Experimental de la Ofuscacion

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
from collections import Counter
import csv
import unicodedata

# ---------------------- Funcion para limpiar texto ----------------------
def limpiar_texto(texto):
    # Quitar comas
    texto = texto.replace(",", "")
    # Quitar acentos
    texto = ''.join(c for c in unicodedata.normalize('NFD', texto)
                    if unicodedata.category(c) != 'Mn')
    return texto

# ---------------------- Cifrado Cesar ----------------------
def cifrado_cesar(texto, llave):
```

```
    resultado = []
    for c in texto:
        if c.isalpha():
            base = ord('A') if c.isupper() else ord('a')
            resultado.append(chr((ord(c) - base + llave) % 26 + base))
        else:
            resultado.append(c)
    return ''.join(resultado)

def descifrado_cesar(texto, llave):
    return cifrado_cesar(texto, -llave)

# ———————————————— Binario y rotacion circular ————————————————
def letra_a_binario(c, bits=12):
    return format(ord(c), f'0{bits}b')

def rotar_bloque(bits, inicio, fin, direccion='L', posiciones=1):
    bloque = bits[inicio:fin+1]
    n = len(bloque)
    posiciones = posiciones % n
    if direccion == 'L':
        bloque_rotado = bloque[posiciones:] + bloque[:posiciones]
    elif direccion == 'R':
        bloque_rotado = bloque[-posiciones:] + bloque[:-posiciones]
    else:
        raise ValueError("Direccion debe ser 'L' o 'R'")
    bits[inicio:fin+1] = bloque_rotado
    return bits
```

```
def rotacion_circular_inversa(bits,inicio,fin,direccion='L',posiciones=1):
    direccion_inv = 'R' if direccion=='L' else 'L'
    return rotar_bloque(bits, inicio, fin, direccion_inv, posiciones)

def intercambiar_bits_circular(bin_str,inicio,fin,direccion='L',posiciones=1):
    bits = list(bin_str)
    bits_rev = bits[::-1]  # contar desde LSB
    bits_rev = rotar_bloque(bits_rev, inicio, fin, direccion, posiciones)
    return ''.join(bits_rev[::-1])

def invertir_rotacion_circular(bin_str,inicio,fin,direccion='L',posiciones=1):
    bits = list(bin_str)
    bits_rev = bits[::-1]
    bits_rev=rotacion_circular_inversa(bits_rev,inicio,fin,direccion,posiciones)
    return ''.join(bits_rev[::-1])

def binario_a_caracter(bin_str):
    codigo = int(bin_str, 2)
    try:
        return chr(codigo)
    except ValueError:
        return '?'

# ---------------------- Procesamiento ----------------------
def procesar_texto(texto,texto_cifrado,inicio,fin,direccion='L',posiciones=1):
    resultados = []
    texto_final = []
```

```
    for c_orig, c_cesar in zip(texto, texto_cifrado):
        bin_orig = letra_a_binario(c_cesar, bits=12)
        bin_mod=intercambiar_bits_circular(bin_orig,inicio\
                                           ,fin,direccion,posiciones)
        char_mod = binario_a_caracter(bin_mod)
        resultados.append((c_orig, c_cesar, bin_orig, bin_mod, char_mod))
        texto_final.append(char_mod)
    return resultados, ''.join(texto_final)

def proceso_inverso(texto_modificado, inicio, fin,\
                    direccion='L', posiciones=1, llave=0):
    resultados_inv = []
    texto_recuperado = []
    for c_mod in texto_modificado:
        bin_mod = letra_a_binario(c_mod, bits=12)
        bin_orig = invertir_rotacion_circular(bin_mod,\
                                inicio, fin, direccion, posiciones)
        c_cesar = binario_a_caracter(bin_orig)
        c_orig = descifrado_cesar(c_cesar, llave)
        resultados_inv.append((c_mod, bin_mod, bin_orig, c_cesar, c_orig))
        texto_recuperado.append(c_orig)
    return resultados_inv, ''.join(texto_recuperado)

# ——————————————— Guardar CSV completo ———————————————
def guardar_csv_completo(resultados_directo, resultados_inverso,\
                nombre_archivo="resultado_binarios_modificados.csv"):
#Combina la informacion del proceso directo y el inverso en un solo CSV.
#Columnas: Caracter original | Caracter despues de Cesar | Binario original |
```

```
# Binario modificado | Caracter modificado | Caracter recuperado

    with open(nombre_archivo, "w", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow([
            "Caracter original", "Caracter despues de Cesar",
            "Binario original", "Binario modificado",
            "Caracter modificado", "Caracter recuperado"
        ])
        for direct, inverso in zip(resultados_directo, resultados_inverso):
            c_orig, c_cesar, bin_orig, bin_mod, char_mod = direct
            _, _, _, _, c_recuperado = inverso
            writer.writerow([c_orig,c_cesar,bin_orig,bin_mod,\
                                        char_mod,c_recuperado])

# ------------------ Histograma ------------------
def histograma_frecuencias(texto, titulo="Histograma"):
    conteo = Counter([c for c in texto if c != '?'])
    ordenados = sorted(conteo.items(), key=lambda x: ord(x[0]))
    if ordenados:
        chars, freqs = zip(*ordenados)
        plt.figure(figsize=(16,6))
        plt.bar(chars, freqs)
        plt.title(titulo)
        plt.xlabel("Caracteres")
        plt.ylabel("Frecuencia")
        plt.show()
    return conteo
```

```
# ———————————————— Programa principal ————————————————
if __name__ == "__main__":
    # Leer archivo
    with open("segunda_guerra_mundial.txt", "r", encoding="utf-8") as f:
        texto = f.read()

    # Limpiar comas y acentos
    texto = limpiar_texto(texto)
    # Parametros
    llave = int(input('Ingrese la llave para el cifrado Cesar: '))
    direccion = input('Ingrese direccion de rotacion circular (L/R): ').upper()
    posiciones = int(input('Ingrese cantidad de posiciones a rotar: '))
    inicio = int(input('Ingrese posicion inicial del bloque a rotar (0-11): '))
    fin = int(input('Ingrese posicion final del bloque a rotar (0-11): '))

    # Proceso directo
    texto_cifrado = cifrado_cesar(texto, llave)
    resultados_directo, texto_modificado = procesar_texto(texto, texto_cifrado,\
                                inicio, fin, direccion, posiciones)

    # Proceso inverso
    resultados_inverso, texto_recuperado = proceso_inverso(texto_modificado,\
                               inicio, fin, direccion, posiciones, llave)

    # Guardar CSV completo
    guardar_csv_completo(resultados_directo, resultados_inverso,\
                                "resultado_binarios_modificados.csv")
```

```
print(Archivo 'resultado_binarios_modificados.csv' generado\
                                con caracter modificado y recuperado.)

# Mostrar ejemplos del proceso inverso
print("\nEjemplos del proceso inverso:")
for i, (direct, inverso) in enumerate(zip(resultados_directo[:15],\
                                                resultados_inverso[:15])):
    c_orig, c_cesar, bin_orig, bin_mod, char_mod = direct
    _, _, _, _, c_recuperado = inverso
    print(f Original: {c_orig} | Despues Cesar: {c_cesar} | \
            Binario modificado: {bin_mod} | Caracter modificado: {char_mod}\
                                        | Recuperado: {c_recuperado})

# Histogramas comparativos
histograma_frecuencias(texto_modificado, Histograma de Caracteres \
                                                Modificados (Directo))
histograma_frecuencias(texto_recuperado, Histograma de Caracteres \
                                                Recuperados (Inverso))
```

Anexo 2. Codigo de modulos en Verilog implementados en FPGA

```
#----------Transmitter.sv-----

module transmitter #(
        parameter CLOCKS_PER_PULSE = 16
)
(
        input logic [7:0] data_in,
        input logic data_en,
        input logic clk,
        input logic rstn,
        output logic tx,
        output logic tx_busy
);
        enum {TX_IDLE, TX_START, TX_DATA, TX_END} state;
```

```
logic [7:0] data = 8'b0;
logic [2:0] c_bits = 3'b0;
logic [$clog2(CLOCKS_PER_PULSE)-1:0] c_clocks = 0;

always_ff @(posedge clk or negedge rstn) begin
        if (!rstn) begin
                c_clocks <= 0;
                c_bits <= 0;
                data <= 0;
        tx <= 1'b1;
                state <= TX_IDLE;
        end else begin
                case (state)
                TX_IDLE: begin
                        if (~data_en) begin
                                state <= TX_START;
                                data <= data_in;
                                c_bits <= 3'b0;
                                c_clocks <= 0;
                        end else tx <= 1'b1;
                end
                TX_START: begin
                        if (c_clocks == CLOCKS_PER_PULSE-1) begin
                                state <= TX_DATA;
                                c_clocks <= 0;
                        end else begin
                                tx <= 1'b0;
                                c_clocks <= c_clocks + 1;
```

```
                end
            end
            TX_DATA: begin
                if (c_clocks == CLOCKS_PER_PULSE-1) begin
                    c_clocks <= 0;
                    if (c_bits == 3'd7) begin
                        state <= TX_END;
                    end else begin
                        c_bits <= c_bits + 1;
                        tx <= data[c_bits];
                    end
                end else begin
                    tx <= data[c_bits];
                    c_clocks <= c_clocks + 1;
                end
            end
            TX_END: begin
                if (c_clocks == CLOCKS_PER_PULSE-1) begin
                    state <= TX_IDLE;
                    c_clocks <= 0;
                end else begin
                    tx <= 1'b1;
                    c_clocks <= c_clocks + 1;
                end
            end
        default: state <= TX_IDLE;
            endcase
    end
```

```
        end
        assign tx_busy = (state != TX_IDLE);

endmodule
```

```
#----------Receiver.sv------
module receiver #(
        parameter CLOCKS_PER_PULSE = 16
)
(
        input logic clk,
        input logic rstn,
        input logic ready_clr,
        input logic rx,
        output logic ready,
        output logic [7:0] data_out
);

        enum {RX_IDLE, RX_START, RX_DATA, RX_END} state;

        logic[2:0] c_bits;
        logic[$clog2(CLOCKS_PER_PULSE)-1:0] c_clocks;

        logic[7:0] temp_data;
        logic rx_sync;

        always_ff @(posedge clk or negedge rstn) begin

                if (!rstn) begin
                        c_clocks <= 0;
                        c_bits <= 0;
                        temp_data <= 8'b0;
```

```
                //data_out <= 8'b0;
                ready <= 0;
                state <= RX_IDLE;

        end else begin
                rx_sync <= rx;  // Synchronize the input ...
                                        signal using a flip-flop

                case (state)
                RX_IDLE : begin
                        if (rx_sync == 0) begin
                                state <= RX_START;
                                c_clocks <= 0;
                        end
                end
                RX_START: begin
                        if (c_clocks == CLOCKS_PER_PULSE/2-1) begin
                                state <= RX_DATA;
                                c_clocks <= 0;
                        end else
                                c_clocks <= c_clocks + 1;
                end
                RX_DATA : begin
                        if (c_clocks == CLOCKS_PER_PULSE-1) begin
                                c_clocks <= 0;
                                temp_data[c_bits] <= rx_sync;
                                if (c_bits == 3'd7) begin
                                        state <= RX_END;
```

```
                                        c_bits <= 0;
                                    end else c_bits <= c_bits + 1;
                                end else c_clocks <= c_clocks + 1;
                        end
                        RX_END : begin
                                if (c_clocks == CLOCKS_PER_PULSE-1) begin
                                        //data_out <= temp_data;
                                        ready <= 1'b1;
                                        state <= RX_IDLE;
                                        c_clocks <= 0;
                                end else c_clocks <= c_clocks + 1;
                        end
                        default: state <= RX_IDLE;
                        endcase
                end
        end
        assign data_out = temp_data;
endmodule
```

```
#----------Binary to 7seg.sv-----


module binary_to_7seg (
        input logic [3:0] data_in,
        output logic [6:0] data_out
);
    // Make a LUT to convert digits to 7 segment output
    // Input - 4 bits, output - 7 bits
    logic [15:0][6:0] lut_7seg;

    // Output is gfedcba
    assign lut_7seg[0] = 7'b0111111;
    assign lut_7seg[1] = 7'b0000110;
    assign lut_7seg[2] = 7'b1011011;
    assign lut_7seg[3] = 7'b1001111;
    assign lut_7seg[4] = 7'b1100110;
    assign lut_7seg[5] = 7'b1101101;
    assign lut_7seg[6] = 7'b1111101;
    assign lut_7seg[7] = 7'b0000111;
    assign lut_7seg[8] = 7'b1111111;
    assign lut_7seg[9] = 7'b1101111;
    assign lut_7seg[15:10] = 7'b0;     // unused

          assign data_out = ~lut_7seg[data_in];

endmodule
```

```
/*
Seven segment display
     a
f         b
     g
e         c
     d
*/
```

```
#----------Ofuscacion.sv-----
// - Cifrado Cesar con clave 5
// - Rotacion a la izquierda de 5 posiciones (bits 0 a 11)


module obfuscation_module (
    input  wire [11:0] data_in,     // Entrada de 12 bits
    output wire [11:0] data_out     // Salida ofuscada
);

    // ------------------------------
    //Cifrado Cesar (clave = 5)
    // ------------------------------
    wire [11:0] caesar_out;
    assign caesar_out = data_in + 12'd5;

    // ------------------------------------------------------------
    // Rotacion a la izquierda de 5 posiciones (bits 0 a 11)
    //    Ejemplo: ROTL12(x,5) = {x[6:0], x[11:7]}
    // ------------------------------------------------------------
    wire [11:0] rotated_out;
    assign rotated_out = {caesar_out[6:0], caesar_out[11:7]};

    // Salida final
    assign data_out = rotated_out;

endmodule
```

#----------Desofuscacion.sv------

```
//  Deshacer rotacion izquierda de 5 bits --> rotacion derecha 5 bits
//  Deshacer Cesar clave 5 --> resta de 5
//=============================================================

module deobfuscation_module (
    input  wire [11:0] data_in,      // Entrada de 12 bits (cifrada + ofuscada)
    output wire [11:0] data_out      // Salida original restaurada
);

    // ------------------------------------------------------------
    // Rotacion a la derecha de 5 posiciones (inverso del ROTL)
    // ROTR12(x,5) = {x[4:0], x[11:5]}
    // ------------------------------------------------------------
    wire [11:0] rotated_back;
    assign rotated_back = {data_in[4:0], data_in[11:5]};

    // ------------------------------------
    // 2) Deshacer el Cesar (clave=5)
    // ------------------------------------
    wire [11:0] original;
    assign original = rotated_back - 12'd5;

    assign data_out = original;

endmodule
```

```
#----------Testbench validacion_cifrado_descifrado.sv-----

`timescale 1ns/1ps

module tb_cipher_validation;

    // Senales del test
    reg  [11:0] data_in;
    wire [11:0] encrypted;
    wire [11:0] decrypted;

    // Instancia del modulo de cifrado (ofuscacion)
    obfuscation_module uut_encrypt (
        .data_in(data_in),
        .data_out(encrypted)
    );

    // Instancia del modulo de descifrado (deofuscacion)
    deobfuscation_module uut_decrypt (
        .data_in(encrypted),
        .data_out(decrypted)
    );

    // Procedimiento de prueba
    initial begin
        $display("==========================================");
        $display("   TESTBENCH: Validacion Cifrado <-> Descifrado");
```

```
        $display("================================================");

        // Pruebas con distintos datos
        apply_test(12'h000);
        apply_test(12'h001);
        apply_test(12'h00A);
        apply_test(12'h123);
        apply_test(12'h555);
        apply_test(12'hAAA);
        apply_test(12'hF0F);
        apply_test(12'hFFF);

        $display("================================================");
        $display("    FIN DE LA SIMULACION");
        $display("================================================");
        $finish;
    end

    // Tarea para aplicar un vector de prueba
    task apply_test(input [11:0] value);
    begin
        data_in = value;
        #10; // Esperar propagacion combinacional

        $display("Entrada Original : %h", data_in);
        $display("Cifrado (Obfusc.): %h", encrypted);
        $display("Descifrado       : %h", decrypted);
```

```
            if (decrypted == data_in)
                $display( " TEST OK Descifrado correcto. ");
            else begin
                $display(" ERROR El descifrado NO coincide. ");
                $stop;
            end
        end
        endtask

endmodule
```

```
#----------Testbench uart.sv-----
module uart #(
        parameter CLOCKS_PER_PULSE = 5208
)
(
        input logic [3:0] data_in,
        input logic data_en,
        input logic clk,
        input logic rstn,
        output logic tx,
        output logic tx_busy,
        input logic ready_clr,
        input logic rx,
        output logic ready,
        output logic [7:0] led_out,
        output logic [6:0] display_out
);
        logic [7:0] data_input;
        logic [7:0] data_output;

        transmitter #(.CLOCKS_PER_PULSE(CLOCKS_PER_PULSE)) uart_tx (
                .data_in(data_input),
                .data_en(data_en),
                .clk(clk),
                .rstn(rstn),
                .tx(tx),
                .tx_busy(tx_busy)
        );
```

```
    receiver #(.CLOCKS_PER_PULSE(CLOCKS_PER_PULSE)) uart_rx (
        .clk(clk),
        .rstn(rstn),
        .ready_clr(ready_clr),
        .rx(rx),
        .ready(ready),
        .data_out(data_output)
    );

    binary_to_7seg converter (
        .data_in(data_output[3:0]),
        .data_out(display_out)
    );

    assign data_input = {4'b0, data_in};
    assign led_out = data_output[7:0];


endmodule
```

Conclusiones

B.1. Conclusión

La implementación del mecanismo de ofuscación sobre la plataforma FPGA DE2-115 permitió demostrar que es posible incrementar significativamente la seguridad de un sistema de comunicación serial sin comprometer su rendimiento operativo. A lo largo del desarrollo del proyecto, se integró un esquema de cifrado tipo César reforzado con una capa adicional de ofuscación basada en rotaciones de bits, lo que derivó en una solución hardware capaz de alterar la estructura estadística del mensaje y dificultar sustancialmente cualquier intento de interceptación o análisis por parte de un agente externo.

Las pruebas realizadas tanto en software como en hardware confirmaron la equivalencia funcional del sistema, así como la coherencia entre los flujos de datos cifrados, ofuscados y posteriormente recuperados. Los experimentos de ataque tipo man-in-the-middle demostraron que, aun cuando un tercero pudiera acceder a las líneas físicas de transmisión UART, la información obtenida se reduce a caracteres sin significado, lo que valida la efectividad del mecanismo propuesto. Este resultado evidencia que la ofuscación incorporada actúa como un refuerzo efectivo frente a ataques de análisis de frecuencia y frente a la pérdida potencial de confidencialidad en enlaces expuestos.

Asimismo, la síntesis del diseño en la FPGA mostró que el costo en recursos lógicos fue

reducido, permitiendo que el sistema sea escalable y compatible con arquitecturas de mayor complejidad. Esto confirma que las FPGAs son plataformas idóneas para el desarrollo de mecanismos criptográficos personalizados y orientados a hardware, particularmente cuando se requiere velocidad, seguridad y control total del flujo de datos.

B.2. Futuros Trabajos y Alcances Potenciales

El éxito de la implementación abre la posibilidad de continuar expandiendo la línea de investigación y desarrollo. Entre los futuros alcances destacan:

1. Integración con algoritmos criptográficos modernos

   Extender el mecanismo de ofuscación hacia cifrados más robustos como AES, SHA512, permitiendo validar la eficiencia del método en escenarios de mayor seguridad.

2. Implementación de una arquitectura pipeline

   Optimizar el proceso de cifrado y ofuscación mediante técnicas de paralelismo y segmentación interna para aumentar la tasa de transferencia en sistemas de comunicación de alta velocidad.

3. Inclusión de hardware adicional de autenticación

   Agregar módulos para autenticación de mensajes (HMAC o CMAC), incrementando la protección frente a ataques de suplantación o modificación de datos.

4. Migración a una plataforma SoC (System-on-Chip)

   Implementar el sistema en arquitecturas híbridas como Intel SoC FPGA o Zynq, integrando procesadores ARM para combinar seguridad en hardware y capacidad de gestión en software.

5. Desarrollo de un sistema dinámico de llaves

   Incorporar mecanismos automáticos de actualización o negociación de llaves para evitar ataques de repetición y fortalecer la longevidad del sistema.

Referencias Bibliográficas

- IEEE Standards Association. (1995). IEEE Standard Verilog Hardware Description Language (IEEE Std 1364-1995). Institute of Electrical and Electronics Engineers.
- IEEE Standards Association. (2005). IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language (IEEE Std 1800-2005). Institute of Electrical and Electronics Engineers.
- Palnitkar, S. (2003). Verilog HDL: A Guide to Digital Design and Synthesis (2nd ed.). Prentice Hall.
- Edad del Hierro
- Kahn, D. (1996). The codebreakers: The comprehensive history of secret communication from ancient times to the Internet (2nd ed.). Scribner.
- Schneier, B. (2015). Applied cryptography: Protocols, algorithms, and source code in C (20th anniversary ed.). Wiley.
- Singh, S. (2000). The code book: The science of secrecy from ancient Egypt to quantum cryptography. Anchor Books.
- Stallings, W. (2017). Cryptography and network security: Principles and practice (7th ed.). Pearson.

- Accellera Systems Initiative. (2002). SystemVerilog Language Reference Manual. Accellera. https://www.accellera.org/
- IEEE Standards Association. (2005). IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language (IEEE Std
- 1800-2005). IEEE. https://doi.org/10.1109/IEEESTD.2005.93210
- IEEE Standards Association. (2008). IEEE Standard Verilog Hardware Description Language. IEEE.
- Padrón, L. (1997). Implementación de modelos de circuitos neuronales electrónicos. Laboratorio de Computación Adaptable.
- Padrón, L., et al. (2000). Modelado y simulación de redes neuronales electrónicas usando MATLAB y SIMULINK. Revista de Computación Adaptable, 12(3), 338–349.
- Terasic Technologies. (s.f.). DE2-115 User Manual. Recuperado de https://www.terasic.com.tw
- Intel Corporation. (2020). Cyclone IV Device Handbook (Volume 1 & 2). Retrieved from https://www.intel.com
- Vahid, F., & Givargis, T. (2010). Embedded System Design: A Unified Hardware/Software Introduction (2nd ed.). Wiley.
- Brown, S., & Vranesic, Z. (2013). Fundamentals of Digital Logic with VHDL Design (3rd ed.). McGraw-Hill.