

Máster Universitario en Electrónica y Tecnología Digital Aplicada (METDA)

Trabajo Fin de Máster

**Análisis, diseño y desarrollo de sistemas de
verificación avanzada para implementaciones de
criptografía postcuántica en FPGA**

Autor: RENÉ RODRÍGUEZ PAREDES

**Tutor(es): DR. PEDRO PÉREZ CARBALLO
AGONEY JESÚS MORENO RODRÍGUEZ**

Fecha: agosto de 2026

Máster Universitario en Electrónica y Tecnología Digital Aplicada (METDA)

Trabajo Fin de Máster

**Análisis, diseño y desarrollo de sistemas de
verificación avanzada para implementaciones de
criptografía postcuántica en FPGA**

HOJA DE FIRMAS

Nombre

Firma

Estudiante: RENÉ RODRÍGUEZ PAREDES

Tutor

Académico: DR. PEDRO PÉREZ CARBALLO

Tutor

AGONEY JESÚS MORENO

Colaborador:

RODRÍGUEZ

Fecha: agosto de 2026

Máster Universitario en Electrónica y Tecnología Digital Aplicada (METDA)

Trabajo Fin de Máster

Análisis, diseño y desarrollo de sistemas de
verificación avanzada para implementaciones de
criptografía postcuántica en FPGA

HOJA DE EVALUACIÓN

Calificación:

.....

Presidente:

GUSTAVO MARRERO CALLICÓ

Fdo.:

Secretario:

ROBERTO ESPER-CHAIN
FALCÓN

Fdo.:

Vocal:

FERNANDO DE LA PUENTE
ARRATE

Fdo.:

Fecha: agosto de 2026

Resumen

Los avances en el campo de la física cuántica acercan la posibilidad de que un adversario emplee la computación cuántica para vulnerar los sistemas criptográficos utilizados en la actualidad. Aunque sigue existiendo incertidumbre acerca de cuándo se dispondrá de la capacidad de cómputo necesaria para llevar a cabo ese tipo de ataques, existe el riesgo de ataques de tipo *Harvest Now, Decrypt Later*. En este escenario, un adversario con suficiente capacidad de almacenamiento puede recopilar y conservar información cifrada durante años, para descifrarla en el futuro cuando se disponga de suficientes recursos de computación cuántica.

Como respuesta a la «amenaza cuántica» surge la criptografía postcuántica, un conjunto de algoritmos basados en problemas matemáticos resistentes a ataques cuánticos. Para favorecer su adopción, se ha impulsado un esfuerzo a nivel global para el desarrollo de estándares de criptografía postcuántica, siendo el proceso de estandarización del NIST el referente principal. Hasta la fecha, este proceso ha dado lugar a tres estándares: FIPS 203, FIPS 204 y FIPS 205. Sin embargo, la disponibilidad de estándares basados en principios matemáticos resistentes a ataques cuánticos constituye solo el primer paso hacia su despliegue. El siguiente reto es su implementación eficiente en sistemas reales. Entre los enfoques más prometedores para abordar este desafío destaca el desarrollo de aceleradores *hardware* de propósito específico, cuya necesidad se enmarca en la demanda generalizada de sistemas digitales cada vez más complejos.

El incremento de la complejidad de los sistemas digitales se traduce en un aumento del número de errores que llegan a producción. Para mitigar el impacto de los errores es necesario mejorar los esfuerzos de verificación.

El presente Trabajo de Fin de Máster se centra en la mejora del proceso de verificación orientado al desarrollo de sistemas de criptografía postcuántica implementados sobre FPGA, tomando como caso de estudio un acelerador de *Number-Theoretic Transform* (NTT). Como parte del estudio inicial, se presenta la noción y los conceptos de verificación avanzada que permiten entender las decisiones tomadas durante el desarrollo de un *pipeline* de verificación avanzada. Dicho *pipeline* unifica diferentes enfoques complementarios y orienta su ejecución hacia entornos con control de versiones y CI/CD. En particular, se ha desarrollado un flujo de verificación que aplica diferentes mecanismos de verificación según la naturaleza de los módulos que componen un determinado diseño y su estructura jerárquica. Se ha empleado verificación formal por medio de la herramienta Sby (Yosys), verificación funcional con Cocotb y VUnit junto con OSVVM y verificación de tipo *hardware-in-the-loop* sobre el sistema físico final.

La implementación centraliza la configuración y la ejecución de las tareas de verificación, reduce duplicidades mediante modelos de referencia compartidos y permite integrar diversas herramientas y enfoques de verificación. Los resultados finales dan lugar a una cobertura del 92.75% por unidad de diseño, la ejecución satisfactoria de los tests y la validación sobre la implementación del diseño en la plataforma de desarrollo Arty Z7-20. Además, la combinación de las técnicas ha revelado limitaciones no documentadas en el módulo de reducción modular, código sin uso y un incumplimiento del protocolo AXI4-Stream relacionado con la generación de TLAST, pese a tratarse de un diseño ampliamente probado y utilizado en entornos cercanos a producción. Estos resultados demuestran que la aplicación selectiva de técnicas distintas aporta garantías complementarias y constituye una base reproducible para verificar futuras implementaciones de criptografía postcuántica.

Abstract

Each advancement in quantum physics brings us closer to a reality where an adversary has the means to attack modern cryptosystems. Even though it is uncertain when sufficiently capable quantum computers will be developed, there is still the risk of “Harvest Now, Decrypt Later” attacks. These types of attacks consist of storing large quantities of encrypted data for later decryption once a sufficiently capable quantum computer is developed.

Post-quantum cryptography (PQC) emerges in response to the so-called “quantum threat.” This kind of cryptography consists of cryptographic algorithms designed to be secure against attacks by relying on mathematical problems that are considered hard to solve for both classical and quantum computers. The U.S. National Institute of Standards and Technology (NIST) has led the global effort to develop post-quantum cryptography standards via its standardization processes that until now have resulted in three standards ready to be used: FIPS 203, FIPS 204, and FIPS 205. However, the availability of appropriate standards is only the first step towards their adoption. The next step is their correct implementation. One approach to implementing these cryptographic standards is to develop application-specific hardware accelerators, following the trend of increasingly complex digital designs.

The increase in complexity of modern digital designs necessitates an accompanying verification effort that limits the impact that errors produce during design and decreases the number of errors that reach production.

This master’s thesis proposes a workflow to streamline and simplify the verification flow with a focus on developing post-quantum cryptographic systems on FPGAs with the particular use case of an accelerator for the Number-Theoretic Transform (NTT). As part of the initial study the two main topics of the thesis are presented: advanced verification, which makes it possible to understand the decisions taken during the development of the verification pipeline, and post-quantum cryptography, that granted the knowledge needed to develop a test suite that fits the intricacies of the accelerator and the algorithm that implements. The pipeline unifies different complementary approaches to verification while facilitating its execution both in a local, development context and as part of a CI/CD pipeline. A verification flow has been developed that applies different verification mechanisms depending on the hierarchy and the nature of the modules that make up the design. Formal verification has been used via the Sby (Yosys) tool, functional verification with Cocotb and VUnit with OSVVM, and hardware-in-the-loop verification via a bare-metal application and a custom Python script.

The implementation centralizes the configuration and execution of verification tasks, reduces duplication through shared reference models, and makes it possible to integrate a variety of verification tools and approaches. The final results yield a coverage of 92.75% per design unit, the successful execution of the tests, and the validation of the design implemented on the Arty Z7-20 development platform. In addition, the combination of techniques revealed undocumented limitations in the modular reduction module, unused code, and a violation of the AXI4-Stream protocol related to TLAST generation, despite the design being widely tested and used in near-production environments. These results show that the selective application of different techniques provides complementary guarantees and constitutes a reproducible basis for verifying future post-quantum cryptography implementations.

Índice

Resumen	ii
Abstract	iii
Índice	iv
Índice de figuras	vii
Índice de tablas	ix
Índice de códigos	x
Acrónimos	xi
1. Introducción	1
1.1. Antecedentes	1
1.2. Objetivos	3
1.3. Organización de la memoria	4
2. Verificación orientada a FPGA	5
2.1. Fundamentos de la verificación en sistemas digitales	5
2.2. Enfoque de verificación	5
2.2.1. Verificación dinámica	6
2.2.2. Análisis estático	7
2.2.3. Enfoque híbrido	9
2.3. Dominios de verificación	10
2.4. Nivel de granularidad	10
2.5. Generación de estímulos	11
2.6. Instante en el ciclo de desarrollo	12
2.7. Evidencia aportada por la verificación	12
2.8. Conclusión	13
3. Evaluación de los <i>frameworks</i> y metodologías de verificación	15
3.1. Criterios de evaluación	15
3.2. UVM	16
3.2.1. Aspectos clave	16
3.2.2. Análisis según criterios	18
3.3. OSVVM	19
3.3.1. Aspectos clave	19
3.3.2. Análisis según criterios	21
3.4. UVVM	21
3.4.1. Aspectos clave	22
3.4.2. Análisis según criterios	22
3.5. VUnit	22
3.5.1. Aspectos clave	23
3.5.2. Análisis según criterios	23

3.6. Cocotb	24
3.6.1. Aspectos clave	24
3.6.2. Análisis según criterios	26
3.7. Evaluación	26
3.8. Verificación formal	27
3.8.1. OSS CAD Suite - Sby	27
3.9. Conclusión	28
4. Criptografía postcuántica	29
4.1. Fundamentos de criptografía clásica	29
4.1.1. Propiedades fundamentales	29
4.1.2. Primitivas y esquemas criptográficos	30
4.1.3. Seguridad basada en dificultad computacional	32
4.2. Amenaza cuántica	32
4.2.1. Ataques e impacto sobre los sistemas actuales	33
4.3. Criptografía postcuántica	33
4.3.1. Familias criptográficas postcuánticas	34
4.4. Proceso de estandarización del NIST	34
4.4.1. Estándares publicados	35
4.4.2. Implicaciones sobre la adopción y diversidad criptográfica	35
4.5. Conclusión	36
5. Estudio y análisis de un KEM postcuántico	37
5.1. Mecanismos de Encapsulado de Claves (KEM)	37
5.2. Mecanismos de encapsulado de claves postcuánticas	38
5.3. ML-KEM	38
5.3.1. Principios matemáticos	39
5.3.2. Operaciones principales de los KEM con ML-KEM	40
5.3.3. Operaciones internas relevantes	41
5.4. Estado del arte de las implementaciones	42
5.4.1. «An Area-Time Efficient Hardware Architecture for ML-KEM Post-Quantum Cryptography Standard»	42
5.4.2. «Efficient Hardware Implementation of the Lightweight CRYSTALS-Kyber» . . .	44
5.5. Conclusión	47
6. Diseño del <i>pipeline</i> de verificación avanzada	49
6.1. Análisis del diseño NTT a verificar	49
6.1.1. Fundamento algorítmico	51
6.1.2. Arquitectura <i>hardware</i>	51
6.2. Verificación formal de la aritmética modular	54
6.2.1. Tests con Sby	54
6.3. Verificación funcional con Cocotb	57
6.3.1. Estructura y reglas de diseño de los tests	57
6.3.2. Tests con Cocotb	58
6.4. Verificación de interfaces AXI con VUnit y OSVVM	63
6.4.1. Tests con VUnit y OSVVM	63

6.5. Verificación <i>hardware-in-the-loop</i> (HIL)	67
6.5.1. Plataforma <i>hardware</i>	67
6.5.2. Aplicación <i>bare-metal</i>	68
6.5.3. <i>Script</i> de HIL	69
6.6. Orquestación del <i>Pipeline</i>	70
6.6.1. Casos de uso	70
6.6.2. Estructura de directorios	70
6.6.3. Ficheros principales	71
6.6.4. <i>Runners</i>	73
6.7. Conclusión	74
7. Resultado y evaluación del <i>pipeline</i>	75
7.1. Resultados por técnica de verificación	75
7.1.1. Verificación formal	75
7.1.2. Cocotb	75
7.1.3. VUnit con OSVVM	76
7.1.4. Verificación HIL	76
7.2. Resultados y evaluación del <i>pipeline</i>	77
7.2.1. Tiempo de ejecución del <i>pipeline</i>	77
7.2.2. Cobertura agregada de VUnit y Cocotb	78
7.3. Conclusión	80
8. Conclusiones y trabajo futuro	81
8.1. Conclusión	81
8.2. Trabajos futuros	82
Referencias	84
Objetivos de Desarrollo Sostenible (ODS)	90

Índice de figuras

Figura 1	Errores no triviales que llegan a producción para plataformas basadas en FPGA	1
Figura 2	Comparación del número de operaciones necesarias para factorizar «números grandes» (como los utilizados en los algoritmos criptográficos modernos RSA y ECC) entre algoritmos clásicos y el algoritmo de Shor	3
Figura 3	Ejes que componen el espacio de verificación.	5
Figura 4	Elementos básicos de un <i>testbench</i>	6
Figura 5	Limitaciones de la verificación formal	9
Figura 6	Jerarquía de clases de UVM	16
Figura 7	Fases de ejecución de UVM	17
Figura 8	Elementos básicos de un <i>testbench</i> con UVM.	18
Figura 9	Elementos que componen OSVVM, adaptado de	19
Figura 10	Elementos de un testbench desarrollado con OSVVM.	21
Figura 11	Conceptos básicos de Cocotb	24
Figura 12	Criptografía simétrica	30
Figura 13	Criptografía asimétrica	31
Figura 14	Representación de las tres propiedades del <i>hash</i>	31
Figura 15	Procesos de firma y verificación de firmas digitales	32
Figura 16	Pasos que se toman en un ataque de tipo <i>Harvest Now, Decrypt Later</i> (HNDL). Adaptado de	33
Figura 17	Establecimiento de claves mediante KEM	37
Figura 18	Problema MLWE en términos de retículos. El mensaje con errores (m') se puede entender como el punto verde, cercano a un punto válido	41
Figura 19	Arquitectura general propuesta y organización de la memoria T.-H. Nguyen, T.-K Dang et al.	43
Figura 20	Acelerador NTT, estructura del SHA3 y planificación de operaciones T.-H. Nguyen, T.-K Dang et al.	44
Figura 21	Arquitectura general propuesta y organización de la memoria propuestas por T.-H. Nguyen, D.-T Dam et al.	45

Figura 22 Organización del módulo butterfly doble y SHA3: (a) Construcción de la esponja (b) estructura In-Hash-Out Keccak (c) esquema de padding T.-H. Nguyen, D.-T Dam et al.	45
Figura 23 Arquitectura de la NMI-NTT: (a) nivel superior del diseño (b) unidad de retardo segmentada <code>D0</code> (c) flujo de ejecución de una NTT directa de 16 puntos T.-H. Nguyen, D.-T Dam et al.	46
Figura 24 Orden de ejecución y ciclos de reloj de las operaciones principales del encapsulado para el nivel de seguridad 3 T.-H. Nguyen, D.-T Dam et al.	47
Figura 25 Pipeline de verificación completo.	49
Figura 26 Esquema de los módulos que componen el acelerador <i>hardware</i> , excluyendo los submódulos de <code>modred.v</code>	50
Figura 27 Esquema de los submódulos que componen el módulo <code>modred.v</code>	51
Figura 28 Unidad de reducción modular, <code>modred.v</code>	52
Figura 29 Unidad de <i>butterfly</i> , <code>butterfly.v</code>	53
Figura 30 Organización del elemento de proceso	53
Figura 31 Proceso principal de tests en relación a el DUT y el VC.	64
Figura 32 <i>Block design</i> utilizado en la verificación HIL	67
Figura 33 Sistema de desarrollo Arty Z7-20 con los LED de actividad de la UART activos durante la ejecución de los tests.	68
Figura 34 Estructura de directorios del proyecto.	71
Figura 35 Primer fallo en orden ascendente del módulo <code>modred.v</code>	75
Figura 36 Cambio de <code>TLAST</code> después de <code>TVALID</code>	76
Figura 37 Resultado de la verificación HIL para el último de 1000 vectores de test. El resultado de la NTT está indicado por <code>RESULT NTT</code> y el de la INTT está indicado por <code>RESULT INTT</code>	77
Figura 38 Cobertura agregada por unidad de diseño.	79
Figura 39 Cobertura agregada por instancia.	79

Índice de tablas

Tabla 1	Valoración global de los <i>frameworks</i> con ponderación uniforme	27
Tabla 2	Estándares postcuánticos publicados por el NIST, salvo FIPS 206, en proceso por el momento.	35
Tabla 3	KEMs candidatos considerados durante el proceso de estandarización del NIST	38
Tabla 4	Tamaños, en bytes, de las claves de encapsulado y desencapsulado, el cifrado y el secreto compartido recogidos en el estándar FIPS 203 para ML-KEM.	39
Tabla 5	Parámetros de ML-KEM según los distintos niveles de seguridad.	40
Tabla 6	Registro de control 0 del banco de registros AXI4-Lite (<code>ntt_regs_S00_AXI.vhd</code>)	50
Tabla 7	Organización de la memoria	53
Tabla 8	Relación del proyecto con los Objetivos de Desarrollo Sostenible (ODS)	90

Índice de códigos

Código 1	Script básico de Cocotb con un único test.	25
Código 2	Fichero <code>.sby</code> empleado en la verificación de <code>modred.v</code>	54
Código 3	Fichero <code>.sv</code> que contiene la verificación de <code>modred.v</code>	55
Código 4	Fichero <code>.sv</code> que contiene la verificación de <code>modmul.v</code>	56
Código 5	Fichero de tests de la unidad <i>butterfly</i> , funciones genéricas.	57
Código 6	Fichero de tests de la unidad <i>butterfly</i> , test de la configuración Cooley-Tukey.	58
Código 7	Comparación en lockstep de la planificación completa de una operación contra el modelo de referencia.	61
Código 8	Operación por AXI4-Lite y ejecución de un polinomio a través de las interfaces AXI4-Stream del top.	62
Código 9	Desalineado entre las fases de dirección y datos de la escritura AXI4-Lite, fichero <code>tb_ntt_regs_s00_axi.vhd</code>	65
Código 10	Secuencia de operaciones por cada vector, fichero <code>ntt_dma.c</code>	69
Código 11	Generación del vector de test, transmisión, recepción y comprobación según el modelo, fichero <code>send_ntt_vectors.py</code>	69
Código 12	Estructura general del fichero de configuración <code>project_config.yml</code>	71
Código 13	<code>Makefile</code> con todas las tareas que se pueden llevar a cabo durante el desarrollo. . .	72
Código 14	Estructura genérica del <i>runner</i> de Cocotb.	73
Código 15	Test que establece que <code>din_valid</code> se ignora.	76
Código 16	Generación de <code>m_axis_tlast</code> en <code>kyber_hpm1pe.vhd</code> , la señal está condicionada por <code>m_axis_tready</code>	76
Código 17	Tiempo de ejecución según el tipo de verificación, todos los tests con Cocotb.	78
Código 18	Tiempo de ejecución según el tipo de verificación, Cocotb con simuladores mixtos. .	78

Acrónimos

ARC – *Arquimea Research Center* 49, 81

ASIC – *Application-Specific Integrated Circuit* 1, 12

AXI – *Advanced eXtensible Interface* 26, 51, 61, 63, 67, 75

BMC – *Bounded Model Checking* 54

BRAM – *Block RAM* 47, 50, 53, 53, 53, 53, 60

BROM – *Block ROM* 53, 53, 59, 60, 60

CDC – *Clock Domain Crossing* 8, 12

CI/CD – *Continuous Integration Continuous Deployment* 15, 23, 23, 23, 49, 70, 70, 70, 70, 74, 77, 77, 78, 82

CNSA – *Commercial National Security Algorithm Suite* 35

CPU – *Central Processing Unit* 38

CT – *Cooley Tukey* 51, 52, 52, 59, 59, 59, 59, 60

CVP – *Closest Vector Problem* 34

D-MLWE – *Decision Module Learning With Errors* 40

DMA – *Direct Memory Access* 67, 67, 67, 67, 67, 68, 68, 68, 68, 68, 77

DSP – *Digital Signal Processing* 46, 47, 50, 51

DUT – *Device under test* 6, 7, 13, 17, 17, 17, 17, 19, 20, 20, 20, 24, 24, 24, 24, 25, 47, 54, 57, 60, 61, 64, 65, 65, 66, 66, 66, 66, 67, 69, 83

FF – *Flip Flop* 47, 50

FIFO – *First In, First Out* 8, 19, 43

FIPS – *Federal Information Processing Standards* 34, 34, 34, 34, 35

FPGA – *Field-Programmable Gate Array* 1, 4, 5, 5, 6, 7, 7, 7, 11, 13, 27, 49, 81, 81

FSM – *Finite State Machine* 60, 61

GIC – *Generic Interrupt Controller* 68

GPI – *Generic Procedural Interface* 24

GS – *Gentleman-Sande* 51, 51, 52, 52, 59, 59, 59, 60, 60

HDL – *Hardware description language* 6, 7, 12, 12, 12, 23, 24, 67, 71, 73

HIL – *Hardware-in-the-Loop* 6, 7, 7, 63, 67, 67, 67, 68, 68, 69, 70, 75, 76, 76, 77, 78, 81, 82

HNDL – *Harvest Now, Decrypt Later* vii, 2, 33, 33

HQC – *Hamming Quasi-Cyclic* 35

INTT – Inverse Number Theoretic Transform 42, 44, 44, 46, 46, 46, 47, 47, 50, 50, 50, 51, 51, 51, 52, 52, 53, 59, 60, 60, 61, 62, 68, 74, 77, 79

IP – Intellectual Property 13, 67

IoT – Internet of things 44

KEM – Key encapsulation mechanism 35, 35, 36, 37, 38, 38, 38, 38, 38, 39, 41, 41, 81

LMS – Leighton-Micali Signature 35

LUT – Look-Up-Table 47, 50

ML-KEM – Module-Lattice-Based Key-Encapsulation Mechanism 38, 39, 39, 39, 39, 40, 40, 40, 40, 41, 41, 41, 42, 42, 42, 42, 43, 43, 43, 44, 44, 45, 47, 81, 81, 81

MLWE – Module Learning With Errors 38, 40, 40

NIST – National Institute of Standards and Technology 3, 3, 4, 34, 34, 35, 35, 35, 35, 36, 38, 38, 38, 38, 38

NSA – National Security Agency 35

NTT – Number Theoretic Transform 42, 42, 42, 42, 42, 44, 44, 46, 46, 46, 46, 46, 47, 47, 49, 50, 50, 50, 51, 51, 51, 52, 53, 59, 60, 60, 60, 61, 62, 62, 68, 74, 77, 81, 81

OSVVM – Open Source VHDL Verification Methodology 19, 19, 19, 20, 21, 21, 21, 22, 22, 22, 22, 22, 22, 23, 23, 23, 27, 27, 28, 28, 62, 63, 63, 63, 63, 64, 75, 76, 81

PAU – Polynomial Arithmetic Unit 43, 44, 46

PS – Processing System 67, 67, 68, 77

PSL – Property Specification Language 27

PWM – Point-wise Multiplication 44, 44, 46, 46, 46, 46, 47, 47, 50, 50, 51, 52, 53, 59, 59, 59, 60, 60, 60, 60

RBG – Random Bit Generator 40

RDC – Reset Domain Crossing 8, 12

ROM – Read-only memory 42, 44, 46

RSA – Rivest Shamir Adleman 2, 33

RTL – Register-transfer level 12, 13, 20, 27

SIVP – Shortest Independent Vector Problem 34

SMT – Satisfiability Modulo Theories 27, 54, 54

STA – Static Timing Analysis 8

SVP – Shortest Vector Problem 34

SoC – System-on-Chip 1, 11, 67

TFM – Trabajo Fin de Máster 3, 15, 27, 47, 81

TLS – Transport Layer Security 29

UVM – Universal Verification Methodology 16, 16, 16, 16, 16, 17, 17, 17, 18, 18, 19, 19, 19, 19, 20, 21, 21, 22, 22, 22, 25, 25, 25, 25

UVVM – Universal VHDL Verification Methodology 21, 21, 22, 22, 22, 22, 23, 83

VC – Verification Component 19, 20, 20, 20, 21, 21, 22, 28, 63, 63, 64, 64, 65, 65, 66, 66, 66, 76, 81

VHDL – VHSIC Hardware Description Language 20, 21, 21, 22, 22, 22, 23, 23, 23, 23, 23, 27, 53, 54

VVC – VHDL Verification Component 22, 22, 22, 22, 22

XMSS – eXtended Merkle Signature Scheme 35

Capítulo 1

Introducción

En este capítulo se introducen los aspectos generales del trabajo, abordando los antecedentes del proyecto, su contexto de desarrollo, sus objetivos y la visión general del proyecto.

1.1. Antecedentes

La industria de los semiconductores está en continuo cambio. En la actualidad, existe una creciente demanda de computación avanzada en ámbitos como los centros de datos, los sistemas embebidos basados en *System-on-Chip (SoC)*, comunicaciones de próxima generación e inteligencia artificial, lo que implica una demanda de sistemas altamente complejos e innovadores [1]. En el contexto actual las cadenas de suministro globales están marcadas por la incertidumbre. La oferta de fabricación de *Application-Specific Integrated Circuit (ASIC)* se ha visto gravemente reducida [2], [3] debido a diferentes crisis económicas o geopolíticas. Por tanto, se pone de manifiesto la necesidad de flexibilizar el diseño. Por ello, la capacidad de prototipado rápido y la reducción de costes de desarrollo propias de las *Field-Programmable Gate Array (FPGA)* suponen una ventaja significativa a la hora de implementar arquitecturas digitales complejas.

La demanda de sistemas de alta complejidad conlleva una necesidad cada vez mayor de su verificación. A medida que incrementa el número de bloques, interfaces, funcionalidad y en general las expectativas y el alcance de los sistemas, también aumenta el número de comportamientos que deben comprobarse antes de que el diseño entre en producción. De lo contrario, se espera un incremento del número de errores que llegan a producción como se puede ver en la Figura 1, con su consiguiente impacto en la productividad.

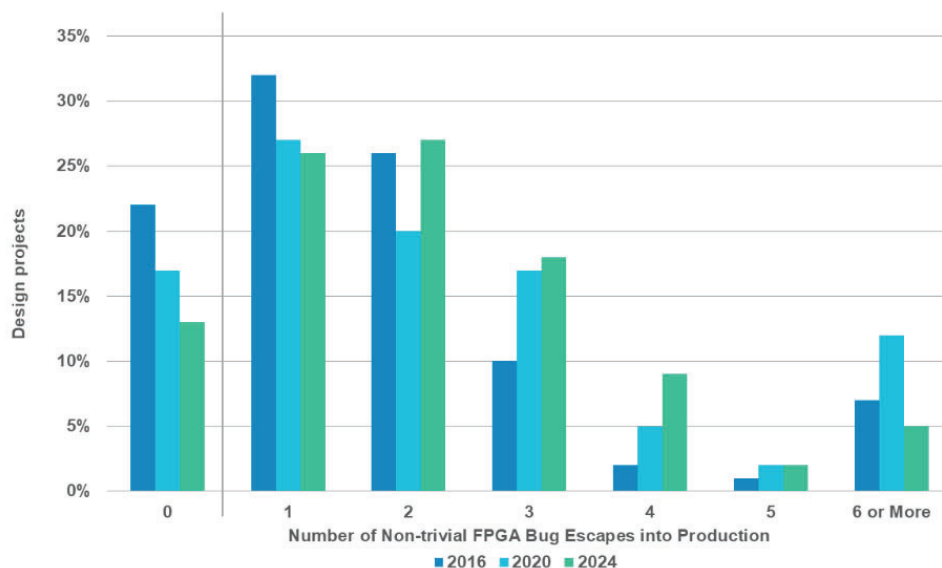


Figura 1: Errores no triviales que llegan a producción para plataformas basadas en FPGA [1].

Uno de los ejemplos más relevantes de cómo un error no detectado puede derivar en grandes costes es el «error de división del Intel Pentium» o «error FDIV del Pentium» [4]. FDIV es la

instrucción de división en coma flotante para la arquitectura x86. El error consistió en que el resultado de algunas operaciones de división resultaba incorrecto. Como consecuencia del problema, Intel estimó en 475 millones de dólares el coste del reemplazo de los microprocesadores [5].

Una de las dificultades que presenta la verificación es que no existe una técnica capaz de cubrir todos los riesgos presentes en un diseño. La emulación y el prototipado permiten observar cómo se comporta el diseño en escenarios cercanos a la realidad con alta velocidad de ejecución a cambio de una reducida visibilidad de los nodos internos. La simulación proporciona un gran control sobre los estímulos aplicados, así como alta visibilidad, pero puede no aportar suficientes garantías sobre la funcionalidad del diseño para todos los escenarios de uso. Por otra parte, la verificación formal permite utilizar métodos matemáticos para explorar los posibles estados de un diseño. En consecuencia, no es posible aplicar una única técnica de verificación de forma general a todos los diseños sin considerar el contexto, criticidad y limitaciones de los propios diseños [6].

La verificación adquiere una relevancia añadida en aplicaciones críticas como sistemas médicos, sistemas aviónicos y/o militares, donde los fallos de funcionamiento pueden tener consecuencias significativas. Uno de los ejemplos clave de sistemas críticos son aquellos que emplean funciones criptográficas. La criptografía representa un caso especialmente sensible en el que, además de producir los resultados correctos, se deben mantener las garantías de seguridad que las aplicaciones y servicios necesitan. Internet, mensajería instantánea, transacciones financieras, en general, todo tipo de intercambios de información son salvaguardados por algoritmos criptográficos. Algoritmos como [Rivest Shamir Adleman \(RSA\)](#), Diffie-Hellman [7] o la criptografía de curva elíptica [8], [9] son los fundamentos de la confidencialidad, autenticación e integridad en sistemas digitales actuales.

La seguridad de los algoritmos criptográficos radica en la dificultad computacional de resolver ciertos problemas matemáticos subyacentes a cada esquema criptográfico. En particular, estos sistemas se sustentan en la existencia de funciones unidireccionales, cuyo cálculo directo es eficiente, mientras que su inversión resulta inviable en la práctica desde el punto de vista computacional [10]. De este modo, la seguridad criptográfica está basada en la inviabilidad de que un adversario disponga de la capacidad computacional necesaria para comprometer dichos sistemas, es decir, la seguridad criptográfica se basa en los llamados «problemas intratables» para los que no se conoce su cálculo eficiente [11].

No obstante, los recientes avances en computación cuántica, combinados con la existencia de algoritmos cuánticos, como el de Shor [12], suponen una amenaza potencial y un riesgo global a todo tipo de sistemas digitales, como se puede ver en la [Figura 2](#). Esto necesariamente conlleva una transición progresiva hacia primitivas criptográficas resistentes a ataques cuánticos [13]. Adicionalmente, existe el riesgo de ataques [Harvest Now, Decrypt Later \(HNDL\)](#) que consisten en que un adversario recolecte y almacene cantidades masivas de datos cifrados con la expectativa de poder descifrarlos en el futuro mediante el uso de computadores cuánticos [14].

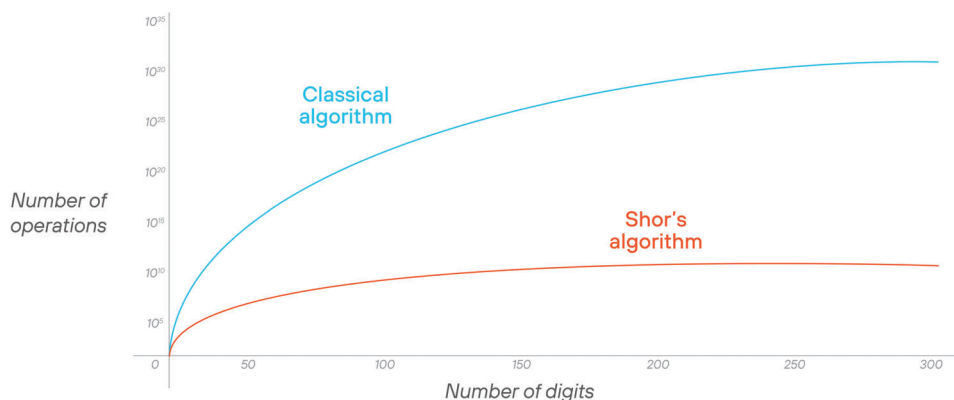


Figura 2: Comparación del número de operaciones necesarias para factorizar «números grandes» (como los utilizados en los algoritmos criptográficos modernos RSA y ECC) entre algoritmos clásicos y el algoritmo de Shor [15].

Ante este escenario, la criptografía postcuántica se presenta como una respuesta ante la amenaza que supone la computación cuántica para los algoritmos criptográficos actuales [16]. La criptografía postcuántica es el conjunto de algoritmos protocolos y primitivas criptográficas resistentes a ataques provenientes tanto de computadores clásicos como cuánticos [11].

En 2016 como parte de la conferencia PQCrypto [17] el *National Institute of Standards and Technology* (NIST) anunció el inicio de un programa con formato de competición ideado para añadir a sus estándares, criptografía postcuántica [18], conocido como «el proceso de estandarización de criptografía del NIST». Tras tres rondas el 13 de agosto de 2024 se presentaron los tres primeros estándares criptográficos FIPS 203, FIPS 204 y FIPS 205, listos para su uso. Sin embargo, el proceso de estandarización continúa con la búsqueda de alternativas en caso de que se encuentren fallos en alguno de los estándares en el futuro [19].

Tras disponer de algoritmos apropiados para un escenario postcuántico se presenta el reto de la adopción. Este reto no está libre de dificultades: los procesos de estandarización [20] de algoritmos postcuánticos no solo plantean un reto matemático, sino también un desafío de implementación eficiente, segura y verificable en *hardware* real.

1.2. Objetivos

El desarrollo de este Trabajo Fin de Máster (TFM) tiene como objetivo general explorar y validar técnicas de verificación avanzada aplicadas a sistemas criptográficos implementados en *hardware*. En particular, se busca lograr un *pipeline* de verificación unificado que reduzca los tiempos de verificación y agilice el desarrollo de sistemas electrónicos mediante la minimización de errores de diseño, reduciendo de forma notable el impacto negativo sobre los usuarios y sobre otros sistemas conectados. Por tanto, el objetivo principal de este proyecto es estudiar, desarrollar e implementar mecanismos de verificación avanzados aplicados a criptografía postcuántica.

Para ello se elige una función a verificar en el contexto de un algoritmo estándar de criptografía postcuántica, se estudia la naturaleza de la función elegida (control, *dataflow*, intensiva en cómputo, etc.), se hace un estudio de los diferentes tipos de verificación, y se define la utilidad de cada tipo de verificación para lograr un enfoque de verificación acorde a la naturaleza de la función. Finalmente, se hace uso del conocimiento extraído para obtener un *pipeline* de verificación avanzada

que elimine duplicidades y reduzca el tiempo invertido para una correcta verificación de la función, que a posteriori se pueda generalizar y adaptar.

Este objetivo general queda desglosado en los siguientes objetivos operativos:

- O1 Analizar el estado del arte de los diferentes enfoques de verificación avanzada orientada a FPGA
- O2 Estudiar algoritmos de criptografía postcuántica y seleccionar un algoritmo para su implementación
- O3 Determinar los enfoques de verificación aptos para criptografía
- O4 Implementar un flujo de trabajo de verificación para criptografía sobre FPGA
- O5 Diseñar un *pipeline* de verificación
- O6 Documentar el trabajo realizado

1.3. Organización de la memoria

En este capítulo se introduce la necesidad de realización del proyecto, definiendo sus antecedentes, su contexto de desarrollo y los objetivos definidos. En el [Capítulo 2](#) se presenta una visión general de la verificación de sistemas digitales orientada a [FPGA](#) y se establecen 6 ejes en los que se organiza la verificación para aportar contexto y definir los términos que se tratan en el resto de la memoria.

En el [Capítulo 3](#) se analizan y comparan diferentes entornos de verificación (que se denominarán como técnicas en el contexto de este proyecto) y las metodologías de verificación asociadas. También se introducirán de forma breve algunos aspectos relacionados con la verificación formal debido a la importancia que está alcanzado en el entorno industrial.

En el [Capítulo 4](#) se dan algunos aspectos relevantes para este trabajo de la criptografía clásica, la criptografía postcuántica y la necesidad de su adopción, así como referencias al proceso de estandarización del [NIST](#) que trae como consecuencia la necesaria adaptación a procesos de verificación estandarizados para poder demostrar el cumplimiento de estándares en el entorno industrial.

En el [Capítulo 5](#) se presenta un ejemplo de diseño relacionado con la criptografía postcuántica. En concreto se presenta en detalle los mecanismos de encapsulado de claves y cómo se materializan en el plano postcuántico con el estándar ML-KEM.

El [Capítulo 6](#) describe el proceso de diseño del *pipeline* de verificación avanzada, el diseño utilizado, y el proceso de verificación utilizado en cada subsistema, aportando los aspectos claves y las principales razones de cada decisión de diseño de los tests utilizados.

El trabajo se completa aportando los resultados obtenidos en el [Capítulo 7](#), incluyendo los hallazgos y limitaciones identificados. Finalmente, en el [Capítulo 8](#) se reúnen las conclusiones y posibles líneas de desarrollo futuro.

Capítulo 2

Verificación orientada a FPGA

En este capítulo se presenta el concepto de verificación en el contexto del diseño de sistemas digitales orientados a **FPGA**. Para ello, se han propuesto seis dimensiones para clasificar las diferentes técnicas de verificación: enfoque empleado, dominio sobre el que se verifica, nivel de granularidad con el que se verifica, naturaleza de los estímulos, fase en la que se verifica y el objetivo final de aplicar un determinado esfuerzo de verificación. Esta clasificación, que no trata de ser una taxonomía completa de la verificación orientada a **FPGA**, sirve como base sobre la que ubicar las ideas tratadas en los capítulos siguientes, principalmente, el **Capítulo 3** que expande el eje de generación de estímulos.

2.1. Fundamentos de la verificación en sistemas digitales

El ecosistema de verificación es amplio y heterogéneo. Las diferentes técnicas de verificación son un reflejo de la complejidad de los sistemas en los que se emplean y, en última instancia, suponen formas complementarias de aportar garantías de que el sistema final se comportará como se ha especificado.

En la **Figura 3** se presentan las diferentes categorías propuestas que permiten clasificar los tipos de verificación que existen. Cada categoría representa un eje independiente ubicado dentro del espacio de verificación. Estos ejes dan lugar a una infinidad de combinaciones distintas para organizar un esfuerzo de verificación determinado. La elección final está condicionada por el coste asumible, la criticidad del diseño o garantías que se deban obtener, entre otros. Debido a la naturaleza del espacio de verificación, los conceptos se pueden ubicar en una categoría u otra. Es por esto que, lejos de ser un fiel reflejo del panorama de verificación, la clasificación que se presenta a continuación tiene como objeto aportar una visión general del ecosistema de verificación.

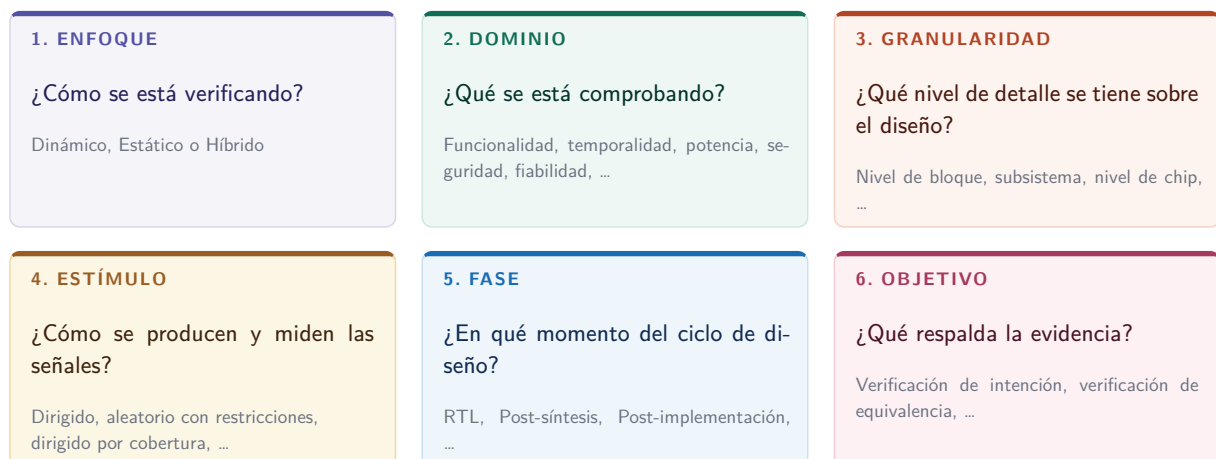


Figura 3: Ejes que componen el espacio de verificación.

2.2. Enfoque de verificación

El enfoque de verificación hace referencia al enfoque principal mediante el que se obtienen evidencias sobre el comportamiento del diseño. Existen dos enfoques principales: **dinámico** y **estático**. Mediante el **enfoque dinámico**, se aporta una serie de estímulos al diseño y se observa cómo responde

a dichos estímulos. En cambio, el **enfoque estático** analiza el diseño sin la presencia de estímulos, es decir, sin simularlo.

Se puede considerar la existencia de un tercer mecanismo alternativo, el **enfoque híbrido**, que consiste en combinar conocimientos obtenidos con uno de estos enfoques para mejorar la implementación del otro.

2.2.1. Verificación dinámica

La verificación dinámica consiste en observar el comportamiento de un determinado diseño ante una serie de estímulos de entrada para tratar de determinar si el comportamiento es el esperado. Existen multitud de técnicas dinámicas, entre las que se encuentran la simulación, la emulación y el prototipado sobre *FPGA* y *Hardware-in-the-Loop (HIL)*, siendo todos enfoques dinámicos.

Simulación

Es el principal enfoque para obtener información sobre el funcionamiento del diseño. Para ello, se emplean programas dedicados, denominados simuladores, que interpretan la descripción *Hardware description language (HDL)* del diseño y la ejecutan como un modelo *software* sobre un ordenador personal o servidor dedicado. El comportamiento del simulador viene determinado por el *testbench*, código adicional que describe cómo ha de llevarse a cabo una prueba determinada. Dicho código instancia el diseño que se desea probar, genera los estímulos y lee las respuestas. El *testbench* tiene asociados tres elementos: el dispositivo bajo prueba o *Device under test (DUT)*, el diseño que se quiere verificar; el entorno de estímulos, cuya función principal es la generación de los vectores de test; y el entorno de observación, que es el responsable de leer los resultados de la simulación una vez que se aplican los estímulos.

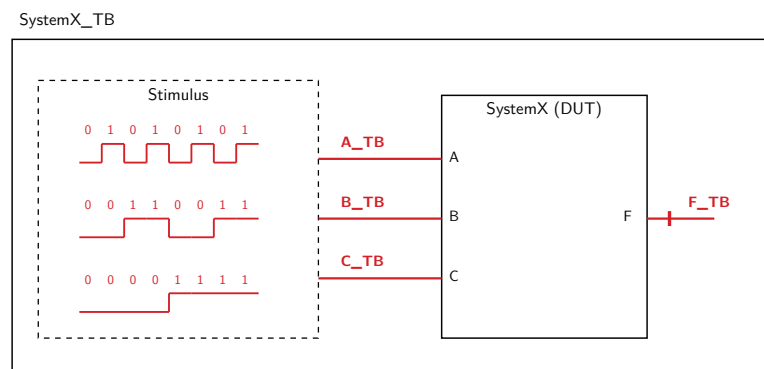


Figura 4: Elementos básicos de un testbench.

Existen dos tipos de simuladores, aquellos basados en eventos o *event-driven* y aquellos basados en ciclos o *cycle-based*. Los simuladores basados en eventos avanzan el tiempo hasta que se produce algún cambio en alguna señal, es entonces cuando se reevalúa la sección del diseño afectada. Esto permite contemplar aquello que ocurre dentro de un mismo ciclo de reloj. Por otra parte, los simuladores basados en ciclos abstraen el tiempo intra-ciclo y evalúan el estado del diseño una vez por ciclo, ignorando aquello que ocurra dentro del ciclo [21]. A cambio de perder la visibilidad dentro del ciclo se obtienen velocidades de simulación mayores, ventaja que cobra importancia en diseños grandes, siempre que el comportamiento intra-ciclo no resulte relevante.

Emulación

La emulación se presenta como respuesta a aquellas situaciones en las que la velocidad de simulación es una limitación. Es un punto intermedio entre la simulación *software* y el prototipado en el que se utiliza la implementación sobre una plataforma de *hardware* programable, obteniendo incrementos de varios órdenes de magnitud en la velocidad de ejecución y aproximando el entorno de pruebas al que tendrá el diseño una vez implementado. Las principales desventajas de la emulación son su elevado coste, determinado por la plataforma *hardware* que se utilice y el posible *software* asociado y la reducción en la observabilidad de los bloques internos del diseño [22]. Cabe destacar que aunque la plataforma final sea una *FPGA*, emular sobre una plataforma independiente sigue siendo útil: se preserva una mayor observabilidad sobre las señales internas sin necesidad de completar el ciclo de síntesis e implementación para la plataforma completa.

Prototipado sobre FPGA

El prototipado sobre *FPGA* sacrifica la observabilidad de los nodos del diseño a cambio de mayor velocidad y de ser representativo respecto de la implementación final. No obstante, a diferencia de la simulación, el prototipado, requiere realizar la síntesis, la implementación, el *place and route* y la generación de artefactos relativos al proceso de prototipado, es decir, un flujo completo con un lento ciclo de preparación y depuración que debe reservarse para las últimas fases del ciclo de diseño.

La observabilidad se reduce no solo por las propias capacidades de la plataforma *hardware* que se utilice, sino también por las limitaciones de observabilidad introducidas por el resto de lógica del diseño. Además, cada ciclo de prototipado produce resultados algo diferentes dependiendo de las decisiones de optimización y *place and route* que tomen las herramientas. En consecuencia, el proceso de prototipado da lugar a resultados que, cercanos a la realidad, deben de interpretarse teniendo en cuenta sus limitaciones.

Hardware-in-the-Loop (HIL)

HIL es una técnica de verificación en la que el *DUT* es *hardware* real y se conecta a un entorno simulado que representa el resto del sistema con el que interactuará. A diferencia del prototipado sobre *FPGA*, el *HIL* tiene un enfoque de funcionamiento en tiempo real con elementos externos al diseño principal. Se establece un lazo en el que las salidas del *hardware* retroalimentan la simulación que genera nuevos estímulos. El diseño se verifica bajo condiciones próximas a las de operación real sin necesidad de disponer del sistema completo.

2.2.2. Análisis estático

La verificación o análisis estático agrupa aquellas técnicas que analizan la descripción de un diseño en lugar de su comportamiento, sin necesidad de estímulos. El enfoque cambia: en lugar de observar cómo responde el diseño al aplicarle una serie de estímulos, se analiza el propio diseño, ya sea su representación *HDL*, una representación intermedia o la *netlist* final. Los programas que realizan análisis estático examinan el diseño tal y como está descrito y obtienen conclusiones sobre su comportamiento.

La principal característica de la mayoría de las técnicas estáticas es que no requieren la elaboración de un *testbench*. Esto permite cubrir escenarios que pueden no ser contemplados en una verificación dinámica, ya que no se depende de los casos que el diseñador haya anticipado. Buena parte del enfoque estático opera utilizando un conjunto de propiedades formuladas con independencia

del diseño concreto. La excepción es la verificación formal, que requiere declarar las propiedades que un diseño debe cumplir, tal y como se verá en la sección correspondiente.

Linting

El *linting* es una técnica de verificación estática que permite detectar posibles problemas utilizando únicamente el código fuente. Se basa en reglas que permiten identificar patrones problemáticos, errores potenciales y desviaciones de los criterios de codificación establecidos. El *linting* actúa como una etapa temprana de filtrado, localizando defectos básicos como elementos declarados no utilizados, asignaciones incompletas, inferencia accidental de *latches* o uso de código no sintetizable, sin necesidad de avanzar con el flujo de desarrollo. Si bien es una técnica de fácil uso, bajo coste y automatización trivial que mejora la calidad del código producido, no trabaja sobre representaciones algorítmicas o funcionales del diseño. Por tanto, un diseño puede superar el *linting* y fallar en fases posteriores.

Análisis temporal estático (STA)

El *Static Timing Analysis* (STA) analiza los caminos temporales del circuito a partir de la *netlist*, los retardos de los elementos lógicos, las interconexiones y las restricciones de diseño definidas, como relojes, límites de retardo y parámetros del entorno externo. Con esta información, la herramienta que realiza el análisis utiliza las características asociadas a los recursos físicos, las restricciones proporcionadas por el diseñador y detalles del propio diseño para comprobar condiciones como tiempos de *setup*, *hold*, *clock skew*, *path delay* y *slack*. La principal ventaja es que la herramienta examina de forma exhaustiva los caminos encontrados, sin depender de estímulos generados por un *testbench* [23].

Análisis de cruces de dominio de reloj (CDC) y reset (RDC)

El análisis de cruces de dominio o *Clock Domain Crossing* (CDC) se centra en asegurar que se produzca una transferencia con garantías de sincronismo de las señales entre diferentes regiones del diseño con relojes distintos, es decir, diferentes dominios de reloj. Se identifican las señales que cruzan entre dominios y se revisa que se estén utilizando estructuras de sincronización apropiadas como sincronizadores de varios *flip-flops*, *handshake* o *First In, First Out* (FIFOs), dependiendo del caso [24].

De forma análoga, las situaciones de *Reset Domain Crossing* (RDC) son aquellas en las que el *reset* cruza de un dominio de reloj a otro o en las que intervienen señales de *reset* asíncronas, en cuyo caso hay que considerar la activación y especialmente la liberación de dicho *reset* sincronizada con el correspondiente reloj. El objetivo final es anticipar posibles problemas de metaestabilidad, pérdida de datos, estados incoherentes o comportamiento no determinista que pueden no manifestarse durante la simulación funcional.

Verificación Formal

La verificación formal agrupa técnicas estáticas basadas en métodos matemáticos que permiten razonar sobre el comportamiento de un diseño. A diferencia de la simulación, que observa la respuesta del diseño ante un conjunto finito de casos, la verificación formal busca demostrar propiedades sobre los estados analizados. La verificación formal se puede utilizar como complemento a la simulación para detectar casos límite difíciles de alcanzar mediante pruebas dirigidas, como se explicará en la [Sección 2.2.3](#).

Dentro de la verificación formal se distinguen principalmente tres enfoques: *equivalence checking*, *theorem proving* y *model checking*. El *equivalence checking* verifica que dos representaciones del diseño preservan el mismo comportamiento funcional. El *theorem proving* permite demostrar propiedades mediante razonamiento matemático, normalmente con mayor intervención del usuario. El *model checking* comprueba si un modelo del diseño satisface un conjunto de propiedades especificadas. Aunque es ampliamente utilizado, este método presenta el problema de la explosión de estados [6].

No todos los diseños son apropiados para aplicar verificación formal. Por ejemplo, el *model checking* no es el mejor candidato en aquellos diseños que presentan operaciones matemáticas complejas o mucha latencia, como se puede ver en la Figura 5, debido a que es difícil de modelar su comportamiento.

Design Block	Difficulty
Arbiter	Easy
Timing Controller	Easy
AHB Bus Bridge	Easy
SRAM Controller	Easy
AXI Bus Bridge	OK
SDRAM Controller	OK (more difficult with data integrity)
DDR Controller	OK (more difficult with data integrity)
DDR2 Controller	Medium (may require tool expertise)
USB Controller	Difficult (long latency)
Cache Controller	More Difficult (with out-of-order trans.)
PCI-Express	Hard (complex & long latency)
ALU (Adders, Multipliers, etc)	Bit-width & function dependent (lower bit widths may be OK)
JPEG/MPEG, DSP, Encryption, Floating-Point Units	Typically not good candidates – heavy complex math operations

Figura 5: Limitaciones de la verificación formal [25].

2.2.3. Enfoque híbrido

La verificación híbrida combina múltiples técnicas con el objetivo de mejorar los resultados de verificación compensando las limitaciones que se presentan al utilizar un único enfoque. Es decir, se combinan diferentes tipos de verificación, tanto estáticos como dinámicos o múltiples técnicas del mismo tipo. Por ejemplo, la verificación dinámica destaca por su capacidad para ser fácilmente aplicada a cualquier nivel de granularidad, es por esto que es el pilar fundamental de la verificación. Sin embargo, a medida que la complejidad de los diseños aumenta, aumenta la dificultad para encontrar los *corner cases*, aquellas situaciones que se salen del flujo típico de uso y que, por tanto, son difíciles de tener en cuenta en el diseño. Las técnicas formales son capaces de analizar exhaustivamente un diseño y demostrar las propiedades declaradas. Sin embargo, en realidad, se produce una explosión de estados, resultando computacionalmente imposible comprobarlos todos. Se pueden utilizar técnicas formales para identificar condiciones o caminos de interés enfocados a una serie de *corner cases* y luego utilizar simulación en dicho conjunto acotado de situaciones [26].

2.3. Dominios de verificación

Mientras que el enfoque de verificación, [Sección 2.2](#), determina el mecanismo mediante el que se obtienen evidencias, los dominios de aplicación se centran en responder a la cuestión de *qué* se verifica. Un diseño puede superar exhaustivas pruebas funcionales y aun así fallar en producción por no cumplir restricciones temporales, de consumo de potencia o presentar fallos de seguridad.

Esta distinción es especialmente relevante en implementaciones criptográficas: un determinado módulo puede cumplir exactamente con la funcionalidad deseada y no ser apto para su uso si es vulnerable a ataques que comprometen su seguridad. Es por esto que es crucial comprender *qué* aspecto concreto se está tratando de verificar con un determinado esfuerzo de verificación.

- **Funcionalidad:** comprueba que el diseño produce resultados de acuerdo a su especificación.
- **Comportamiento temporal (*timing*):** verifica que el diseño cumple las restricciones temporales necesarias para su correcto funcionamiento a la frecuencia deseada, principalmente mediante análisis de tiempos de *setup*, *hold*, *slack* y caminos críticos.
- **Integridad ante asincronía:** comprueba que las señales se transfieren correctamente entre dominios de reloj o de *reset*, evitando problemas de pérdida de datos o metaestabilidad.
- **Potencia:** evalúa que el consumo de potencia del diseño se mantiene dentro de los límites de la plataforma final, mediante *profiling* del consumo y estimaciones de potencia.
- **Seguridad:** analiza si la implementación es vulnerable ante una serie de ataques considerados, por ejemplo, ataques de canal lateral, inyección de fallos o incumplimiento de características de seguridad.
- **Fiabilidad:** estudia el comportamiento del diseño ante los fallos, condiciones adversas y variabilidad. Para ello, principalmente se incluyen en el diseño técnicas de tolerancia a fallos y se realizan pruebas de robustez.

2.4. Nivel de granularidad

El nivel de granularidad establece el nivel de detalle con el que se tiene acceso a un diseño durante la verificación. Una granularidad fina permite poner el foco en un área del diseño en particular, se tiene un mayor control y observabilidad del comportamiento interno de dicha área del diseño. Por el contrario, una granularidad gruesa permite abarcar un área del diseño mayor a costa de tener un menor control y observabilidad del comportamiento interno. El nivel adecuado dependerá de múltiples factores, desde la fase en la que se encuentre el desarrollo del proyecto hasta los fallos de los que se sospeche:

- **Nivel de bloque:** una unidad funcional de forma aislada. Alto grado de control y observabilidad, por lo que resulta adecuado para comprobar operaciones específicas, módulos aritméticos, máquinas de estados, algoritmos concretos o interfaces.
- **Nivel de subsistema:** un conjunto de bloques relacionados que implementan una función mayor. Este nivel permite comprobar la interacción entre módulos, protocolos internos y reparto de responsabilidades entre componentes.
- **Nivel de *top-level* RTL:** diseño completo (desde el punto de vista de lógica programable). Reduce la visibilidad sobre detalles internos, pero permite validar la integración entre subsistemas y el comportamiento global.

- **Nivel de sistema:** engloba todas las partes del diseño y presenta el mayor alcance y la menor visibilidad interna. Surgen nuevos comportamientos a partir de la integración. Según la extensión del sistema, se consideran tres tipos: **SoC**, sistema de desarrollo o múltiples sistemas de desarrollo, por ejemplo, varios sistemas que integren **FPGAs** conectados entre sí.

2.5. Generación de estímulos

La generación de estímulos responde a cómo se explora el espacio de todos los posibles estímulos que se le pueden aplicar a un diseño. Debido a que generar todas las posibles combinaciones no es posible, en la práctica hay que realizar una selección rigurosa. La generación de estímulos está estrechamente relacionada con la metodología y *framework* que se utilice debido a que, al elegir una de estas opciones, se define el futuro desarrollo del proceso de verificación, según el paradigma en el que se base cada una.

A continuación, se introducen los tipos de generación de estímulos más utilizados y en el [Capítulo 3](#) se introducen las metodologías y *frameworks* más conocidos, principalmente UVM, OSVVM, UVVM, Cocotb y sus extensiones.

- **Dirigido:** los estímulos se definen manualmente para comprobar diferentes escenarios, conocidos de antemano. Aunque es un enfoque sencillo, resulta de gran utilidad en fases iniciales para validar casos básicos, reproducir errores específicos o conocer si el enfoque del diseño es el correcto. Cabe destacar que depende del criterio del diseñador y puede dejar sin explorar casos no previstos.
- **Aleatorio con restricciones:** los estímulos se generan de forma aleatoria aplicando un conjunto determinado de restricciones. Esto permite explorar más combinaciones que una prueba dirigida a la vez que se evita la explosión de estados.
- **Dirigido por cobertura:** la generación de estímulos se guía mediante métricas de cobertura. Las métricas de cobertura indican qué partes del diseño, estados, transiciones o combinaciones, dependiendo del tipo, aún no han sido recorridos en simulación. El objetivo es acotar los *gaps* de verificación informados por las métricas de cobertura. Es importante recalcar que es posible diseñar estímulos que cumplan con la cobertura sin cubrir los casos límite, de modo que un diseño con una cobertura del 100% no es necesariamente un diseño sin fallos [21]. Existen principalmente dos tipos de cobertura usados en la industria:
 - **Cobertura de código:** permite identificar qué parte del código se ha ejecutado. A su vez, hay distintos tipos según la naturaleza del código: sentencias/ramas, expresiones/condiciones, caminos de ejecución, estados y transiciones de máquinas de estados finitos o aserciones [27].
 - **Cobertura funcional:** se registran las métricas relevantes durante el funcionamiento de un diseño para estudiar qué valores se han empleado, teniendo en cuenta la especificación. Pueden identificarse métricas como instrucciones ejecutadas, tamaños de paquete o uso de *buffers*, entre otras.
- **Fuzzing:** se aplican entradas modificadas de las existentes con el objetivo de encontrar comportamientos inesperados o casos límite. Aunque es más habitual en *software*, puede aplicarse a diseños *hardware* en simulación o mediante bancos de prueba automatizados.

2.6. Instante en el ciclo de desarrollo

El momento en que se aplica la verificación dentro del ciclo de desarrollo condiciona directamente el coste de los fallos que se detectan. Un error identificado a nivel *Register-transfer level (RTL)* tiene un coste de corrección notablemente inferior al del mismo error detectado tras la síntesis o, peor aún, sobre *hardware* real. Esta asimetría justifica anticipar la verificación tanto como sea posible, sin que ello implique renunciar a los niveles posteriores, que aportan garantías que no son alcanzables en etapas tempranas. Aunque a continuación se enuncian diferentes instantes de forma aislada, estos solo cobran sentido real cuando son parte de un proceso iterativo centrado en la verificación de un diseño. Con cada iteración se obtienen diferentes evidencias que dan lugar a un diseño más refinado en cada paso.

- **Pre-RTL:** se trabaja sobre la especificación, modelos algorítmicos y decisiones acerca de la arquitectura antes de escribir la descripción HDL. En esta etapa se utilizan modelos y algoritmos de referencia, se analizan requisitos y se hace *profiling* de los programas en caso de que existan para identificar las áreas de especial interés.
- **RTL:** se cuenta con una descripción HDL del diseño. Es la etapa en la que se centra gran parte de los esfuerzos de verificación: simulación funcional, *linting*, verificación formal, simulación dirigida por cobertura, entre otros.
- **Post-síntesis:** se utiliza el diseño una vez transformado a una representación estructural por la herramienta de síntesis. Se puede comprobar la equivalencia funcional con el código HDL, detectar problemas introducidos en la síntesis y realizar análisis temporales, de utilización de recursos, CDC/RDC y de potencia, principalmente. Su uso está más extendido en el desarrollo orientado a ASICs.
- **Post-implementación:** se trabaja con el diseño una vez concretadas la ubicación, interconexiones y retardos físicos de la plataforma objetivo. En esta fase se realizan análisis similares a la post-síntesis teniendo en cuenta los recursos físicos de la plataforma objetivo.
- **Sobre plataforma hardware:** el diseño se encuentra implementado en una plataforma concreta haciendo uso de los recursos y configuración asociados, estimados en la post-implementación. La verificación y análisis en esta fase son muy representativos del desempeño final del diseño a costa de una reducida observabilidad y capacidad de depuración.
- **En operación:** el diseño está desplegado y en funcionamiento en su entorno final. El enfoque principal está en la obtención de datos y métricas que permitan el mantenimiento y diagnóstico del sistema, así como su actualización en caso de posibles fallos en su desempeño.

2.7. Evidencia aportada por la verificación

La verificación tiene como último objetivo aportar evidencias de que el diseño satisface su especificación. Sin embargo, esta demostración puede articularse de formas muy distintas:

- verificando que la especificación se ha trasladado correctamente al código,
- comprobando la equivalencia entre dos representaciones alternativas del mismo diseño,
- validando que un componente se comporta como el resto del sistema espera, o
- validando un componente para su reutilización en futuros proyectos.

Cada uno de estos enfoques aporta un tipo de evidencia distinta y complementaria. Es clave tener en cuenta el objetivo de cada uno de los esfuerzos de verificación:

- **Verificación de intención:** se trata de determinar si el diseño cumple con las intenciones establecidas en la especificación. Un posible enfoque para un bloque criptográfico, puede ser contrastar la salida de un determinado **DUT** con un modelo de referencia o frente a un conjunto de vectores de prueba.
- **Verificación de equivalencia:** busca comprobar que tras transformar el diseño de una representación a otra se mantiene su comportamiento, por ejemplo, entre su representación **RTL** y la *netlist* obtenida tras la síntesis o entre una versión optimizada del diseño y la original. Permite acotar la fuente de un error: si procede de la especificación o del flujo de desarrollo.
- **Verificación de IP:** busca aportar garantías a un bloque *Intellectual Property (IP)* para su futura reutilización. No se limita a la funcionalidad del propio bloque, además se incluyen sus restricciones y compatibilidad con diferentes entornos de integración.
- **Verificación de integración:** se centra en comprobar que el bloque tiene el comportamiento esperado en un entorno concreto, es decir, con el resto de bloques que componen un sistema y sus interacciones.

2.8. Conclusión

En este capítulo se ha establecido una clasificación de las principales técnicas de verificación empleadas en diseños digitales orientados a **FPGA**. La verificación es multidimensional, cada uno de los ejes planteados contribuye de forma diferente para una correcta verificación de los diseños: cómo se quiere verificar, qué se está verificando, con qué nivel de granularidad, mediante qué tipo de estímulos, cuando se verifica y cuál es la evidencia generada. La toma de estas decisiones condiciona directamente los costes de verificación, la capacidad de detectar qué errores y de qué manera se puede confiar en el correcto funcionamiento de un diseño. Una vez definido el marco conceptual, el siguiente paso es estudiar los aspectos prácticos de la verificación, es decir, cómo se materializa en metodologías y *frameworks*, como se presenta en el **Capítulo 3**.

Capítulo 3

Evaluación de los *frameworks* y metodologías de verificación

En este capítulo se presentan los diferentes *frameworks* o entornos de verificación utilizados durante el desarrollo del trabajo. Para determinar cuáles de los *frameworks* estudiados son más apropiados para el desarrollo del presente TFM, se ha realizado una evaluación comparativa, atendiendo a criterios técnicos, tanto teóricos como prácticos. Estos criterios permiten valorar las cualidades teóricas de cada alternativa, así como la utilidad y aplicabilidad en un entorno de desarrollo centrado en la criptografía.

3.1. Criterios de evaluación

La evaluación se ha centrado en estudiar los siguientes aspectos: curva de aprendizaje, capacidades, reutilización, productividad, flexibilidad y adecuación a diseños criptográficos. A continuación se define y concreta el alcance de cada uno de estos criterios.

- **Curva de aprendizaje:** se valora la dificultad de adopción de cada *framework*, considerando el lenguaje en el que se basa, la complejidad de su flujo de trabajo, la calidad de la documentación y el grado de familiaridad con conceptos de verificación avanzada necesarios. Un *framework* de verificación puede ser de gran utilidad, pero si se introduce una elevada fricción en el diseño de los tests se corre el riesgo de apartar el foco de la utilidad final de la verificación y del diseño en sí.
- **Capacidades:** se estudia la batería de utilidades que incluye cada *framework*, como tipos de cobertura, aserciones, aleatorización, generación de estímulos y soporte para *Continuous Integration Continuous Deployment (CI/CD)*¹.
- **Reutilización:** se analiza la facilidad para elaborar unidades reutilizables entre distintos diseños o tests. Se valora la capacidad para crear abstracciones, el uso de modelos de referencia y la facilidad para verificar interfaces estándar.
- **Productividad:** se valora el impacto en el tiempo de desarrollo, ejecución, depuración y mantenimiento de los tests. Aquellos *frameworks* que favorecen la productividad son capaces de escalar su propia complejidad según la complejidad del diseño, de modo que se observe una dificultad apropiada tanto para desarrollar pequeños prototipos como para grandes tests de integración.
- **Flexibilidad:** se estudia la capacidad de cada *framework* para adaptarse a diferentes escenarios: variedad de lenguajes de diseño, simuladores, sistemas operativos, automatizaciones y otras herramientas externas. Todo con el fin de que el *framework* utilizado se adapte al modelo de trabajo existente de los diseñadores.
- **Adecuación a diseños criptográficos:** se evalúa el grado de adecuación de cada *framework* para verificar diseños criptográficos, teniendo en cuenta sus peculiaridades. Aunque este

¹Práctica que busca automatizar la integración, la verificación y el despliegue de cambios realizados durante el desarrollo de *software* con garantías y de forma rápida.

es un criterio específico de la naturaleza de este trabajo, es posible generalizar a aquellos módulos que guarden similitudes, principalmente, el uso de operaciones matemáticas complejas estándar.

3.2. UVM

Universal Verification Methodology (UVM) es un entorno de verificación recogido en el estándar IEEE 1800.2, implementada como una biblioteca de clases sobre SystemVerilog [28]. Su objetivo es aportar estructura al desarrollo de *testbenches* utilizando capas de abstracción para permitir la reutilización de los diferentes componentes. Aporta una metodología de referencia de la industria de la verificación. Es por esto que muchos de los conceptos propios de *UVM* están presentes en otros entornos, como se presentará más adelante.

3.2.1. Aspectos clave

Debido a la extensión de *UVM*, se detallan únicamente los aspectos generales que permitan comprender su naturaleza y que sirvan como base para dar contexto a conceptos similares en el resto de los entornos presentados en este capítulo. A continuación, se describen los conceptos más importantes de *UVM*.

- **Jerarquía de clases:** *UVM* está organizado en una biblioteca de clases con diferentes responsabilidades cuyo origen es el `uvm_object`. En la Figura 6 se ilustra la jerarquía. Gracias a los mecanismos de herencia soportados, es posible reutilizar y especializar el comportamiento de cada componente sin reescribirlo desde cero.

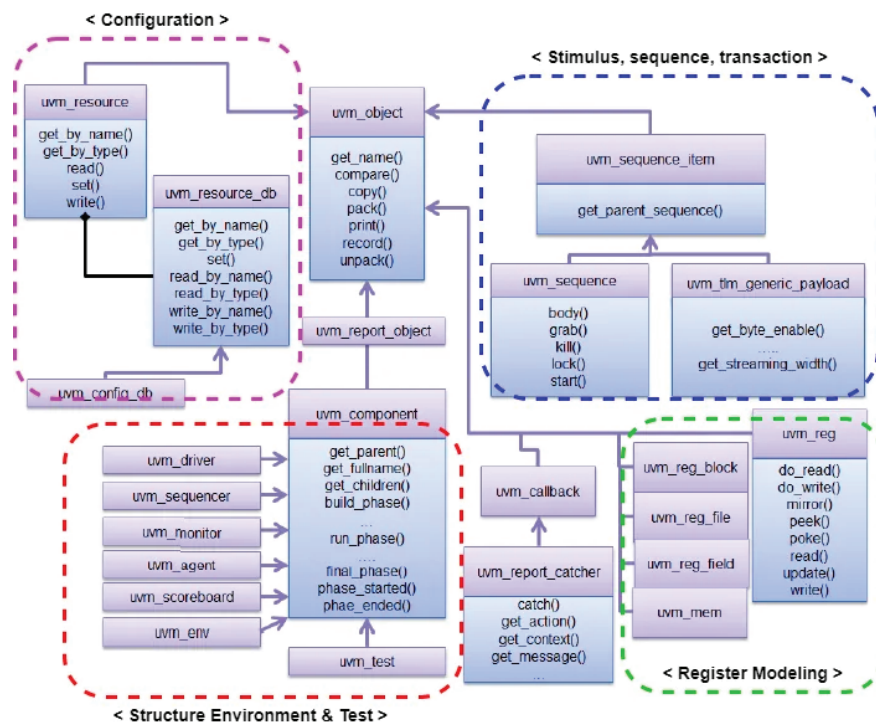


Figura 6: Jerarquía de clases de UVM [29].

- **Transacciones:** con [UVM](#) se abstraen las señales para dar lugar a las transacciones. Estas transacciones son la unidad de información que intercambian los componentes. En lugar de lidiar con flancos y buses, los componentes operan sobre objetos que representan operaciones completas.
- **Fases:** para coordinar la ejecución del conjunto de la jerarquía, [UVM](#) emplea un conjunto predefinido de fases de ejecución según se indica en la [Figura 7](#).

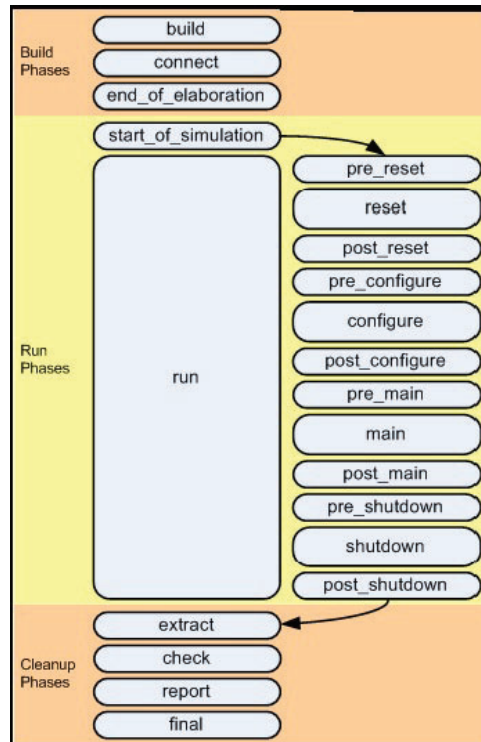


Figura 7: Fases de ejecución de UVM [30].

Los tres conceptos anteriores son comunes a cualquier *testbench* de [UVM](#) y adquieren su sentido práctico al materializarse en componentes concretos. En la [Figura 8](#) se muestra la jerarquía de los componentes que forman un *testbench* básico. A continuación, se describe la función de cada uno de los bloques:

- **sequences:** describe una secuencia de transacciones que serán aplicadas al [DUT](#).
- **sequencer:** organiza las transacciones antes de entregarlas al *driver*, esto permite crear distintos escenarios de prueba.
- **Driver:** traduce las transacciones a señales concretas para el [DUT](#).
- **Monitor:** actúa como el *driver*, pero en sentido contrario y sin modificar las señales. Observa las señales del [DUT](#) y las convierte en transacciones.
- **Agent:** agrupa los componentes anteriores para su reutilización en una interfaz o grupo de señales del mismo diseño o en otros diseños.
- **Scoreboard:** realiza la comparación entre las transacciones que provienen del [DUT](#) y la referencia.
- **Environment:** agrupa configuraciones de los componentes anteriores para verificar desde bloques hasta sistemas completos.

- **Test:** contiene la configuración del entorno según este lo permita (estímulos a utilizar, escenarios, estrategias de generación de estímulos, etc.), posibilitando la ejecución de tests en secuencia con configuraciones distintas.

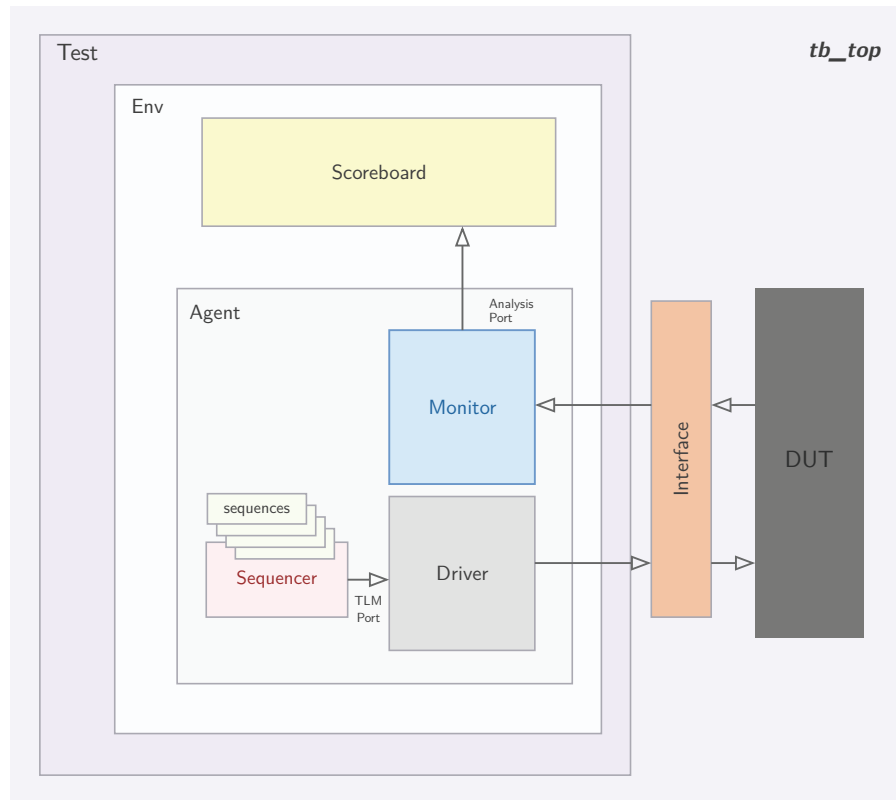


Figura 8: Elementos básicos de un testbench con UVM.

3.2.2. Análisis según criterios

1. **Curva de aprendizaje:** la más pronunciada de los entornos estudiados. Requiere dominar SystemVerilog y tener un alto grado de familiaridad con las herramientas que están disponibles para construir un *testbench*.
2. **Capacidades:** UVM proporciona una gran variedad de mecanismos de verificación avanzada: transacciones, aleatorización de estímulos, mecanismos de cobertura, diferentes esquemas de verificación, *scoreboards* (vectores de test, modelos de referencia, etc.), jerarquías y muchos otros.
3. **Reutilización:** es uno de sus principios. Presenta facilidades para reutilizar al nivel que se desee, desde *sequences* hasta *environments* completos. Esto permite elaborar un conjunto de componentes probados y utilizarlos en múltiples proyectos con garantías.
4. **Productividad:** en un principio, tiende a ser baja debido a su propia complejidad y a la dificultad frente a otras opciones de integrar de forma progresiva elementos en un *testbench* según se necesiten.
5. **Flexibilidad:** dentro del ecosistema SystemVerilog y herramientas profesionales es estándar. Sin embargo, no existen simuladores no comerciales que permitan ejecutar UVM (a excepción del soporte en desarrollo de Verilator [31]), limitando las opciones en cuanto a aprendizaje e integración en entornos existentes.

6. **Adecuación a diseños criptográficos:** **UVM** permite verificar todo tipo de diseños, especialmente si forman parte de un sistema mayor o con múltiples niveles de integración. Para módulos o diseños acotados o aquellos que requieran de rápida iteración, puede introducir complejidad innecesaria.

3.3. OSVVM

Open Source VHDL Verification Methodology (OSVVM) es un *framework* de verificación basado en VHDL que permite construir *testbenches* estructurados [32] que se presenta como alternativa a **UVM**. Surge en 2011 para dar solución tanto a la complejidad de **UVM** como a la necesidad de escribir SystemVerilog para realizar verificación. Está compuesto por una serie de bibliotecas y herramientas: bibliotecas de utilidades, biblioteca de componentes de verificación, conjunto de *scripts*, herramientas para la cosimulación y herramientas para la generación de informes. El *framework* está distribuido como un conjunto, *OsvvmLibraries* [33], como se muestra en la **Figura 9**.

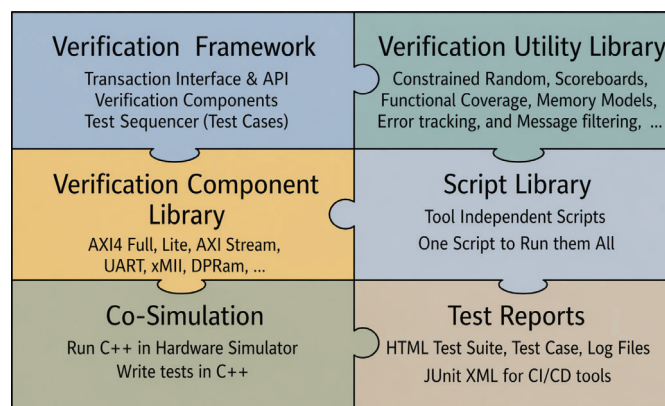


Figura 9: Elementos que componen OSVVM, adaptado de [34].

3.3.1. Aspectos clave

- **Ecosistema modular:** **OSVVM** se distribuye con un conjunto de bibliotecas y herramientas que se pueden adoptar o no, en función de las necesidades del proyecto. Esto permite incorporar capacidades de verificación de forma escalonada sin la imposición desde el inicio de un esquema completo rígido de verificación como con **UVM**. A continuación se listan los diferentes elementos que componen **OSVVM** (**Figura 9**):
 - *Verification framework*: define la estructura de los *testbenches*, mecanismos de sincronización entre elementos y las conexiones entre el **DUT** y el resto de componentes del *testbench*.
 - *Verification utility library*: proporciona utilidades reutilizables para la verificación: generación de estímulos aleatorios, cobertura funcional, modelos para **FIFOs** y memorias, *scoreboards* y mecanismos para *logging*².
 - *Verification component library*: es una biblioteca de componentes de verificación (*Verification Components (VCs)*). Permite generar, observar y comprobar transacciones sobre protocolos e interfaces estándar.

²Registro de eventos y generación de mensajes de estado o diagnóstico.

- *Script library*: API estándar sobre Tcl [35] para interactuar con OSVVM y con diferentes simuladores para compilar, controlar la ejecución, generar informes y organizar los tests.
- *Co-Simulation*: posibilita integrar modelos externos mediante C/C++ con los *testbenches*.
- *Test reports*: sistema que permite recopilar información sobre la compilación, simulación, cobertura, alertas y resultados de simulación.
- **Transacciones**: de un modo similar a UVM las transacciones permiten abstraer los cambios de señal. El diseñador trabaja con conceptos como leer, escribir, enviar o recibir datos y esas operaciones actúan sobre el secuenciador, que es el responsable de traducir las operaciones a los protocolos concretos. Así, se desacoplan los detalles de los protocolos y se facilita la lectura y escritura de los *testbenches*.
- **Componentes de verificación**: los VCs implementan interfaces o protocolos concretos para verificación. Son entidades y arquitecturas *VHSIC Hardware Description Language (VHDL)*, por lo que pueden adoptar una estructura similar a la del propio diseño RTL. Un VC recibe las transacciones generadas por el *TestCtrl*, las convierte en señales y ciclos de reloj, observa las respuestas del DUT y genera las comprobaciones necesarias. De este modo, el mismo componente puede reutilizarse en pruebas unitarias, de integración y de sistema.

Los elementos anteriores son la base que fundamenta los *testbenches*. En la Figura 10 se puede ver un ejemplo de *testbench* con los elementos típicos para llevar a cabo la verificación de un sistema. Dichos elementos son:

- **TestCtrl**: la entidad `TestCtrl_e` contiene la infraestructura común de control, mientras que las arquitecturas asociadas tienen distintos escenarios de test. Esto elimina duplicidades y permite utilizar distintos conjuntos de estímulos dependiendo del test.
- **Componentes de verificación**: los VCs, como `AXI Manager`, `UART Rx` y `UART Tx`, implementan las interfaces externas del DUT. Reciben las operaciones del *TestCtrl* y las convierten en señales y ciclos de reloj.
- **Modelos auxiliares**: elementos como `SRAM Model` representan memorias o dispositivos externos necesarios para reproducir el entorno de operación del diseño.
- **Elementos estándar**: aquellos presentes en cualquier *testbench* independiente del *framework* de verificación empleado: **Generador de reloj y reset** y **DUT (MemIO)**. Este último se conecta con los VCs y modelos auxiliares, que responde a los estímulos y estos se registran.

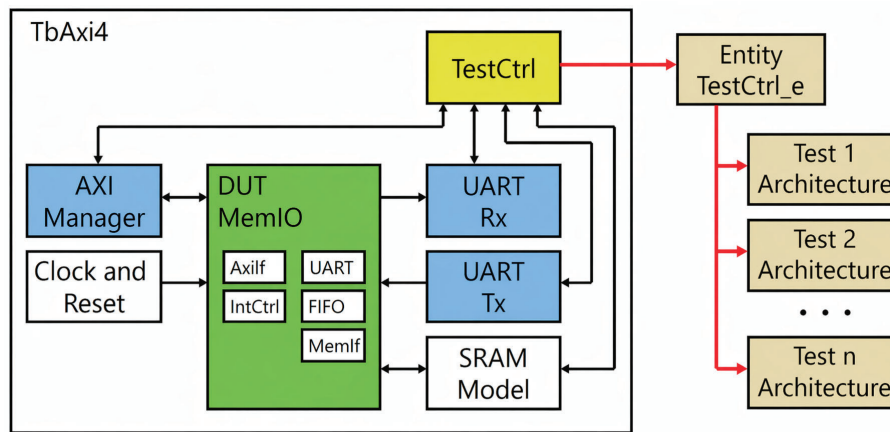


Figura 10: Elementos de un testbench desarrollado con OSVVM.

3.3.2. Análisis según criterios

1. **Curva de aprendizaje:** moderada dependiendo de los módulos de los que hagan uso. Como se ha mencionado antes, es posible adoptar solo aquello que se necesite y extender las capacidades de [VHDL](#) según lo necesite el proyecto.
2. **Capacidades:** amplio abanico de capacidades entre las que se encuentran: generación aleatoria con restricciones, *scripts* para automatizaciones, cobertura funcional y herramientas para cosimulación. Permiten verificar desde bloques simples hasta sistemas completos.
3. **Reutilización:** es elevada y forma parte de la filosofía del *framework*. Todos los módulos que componen un *testbench* se pueden reutilizar.
4. **Productividad:** [OSVVM](#) es más productivo que [UVM](#) para verificar [VHDL](#), pues se elimina la interacción con SystemVerilog. Sin embargo, puede reducirse en aquellos diseños que utilicen interfaces que no estén en su biblioteca de [VCs](#). De un modo similar, en diseños reducidos puede introducir más complejidad de la necesaria.
5. **Flexibilidad:** tiene integraciones directas con multitud de simuladores, aunque la compatibilidad dependerá del simulador. Por ejemplo, la compatibilidad con el simulador xsim [36] de Vivado sigue en desarrollo [37]. Por otra parte, las capacidades de cosimulación permiten integrar modelos en C/C++. Finalmente, tiene soporte directo por parte de otro de los *frameworks* estudiados en este capítulo, VUnit.
6. **Adecuación a diseños criptográficos:** el grueso de la utilidad de [OSVVM](#) para diseños criptográficos reside en su biblioteca de [VCs](#) para verificar las interfaces estándar del diseño con garantías. Sin embargo, el uso de VHDL o C/C++ complica la verificación de módulos que realicen operaciones matemáticas o requieran de modelos complejos.

3.4. UVVM

Universal VHDL Verification Methodology (UVVM) es un entorno de verificación basado en VHDL que permite construir *testbenches* estructurados [38] que se presenta como alternativa a [UVM](#) y [OSVVM](#) con una filosofía similar a esta última. [UVVM](#) está desarrollado con énfasis en crear tests uniformes y fáciles de leer incluso para aquellos proyectos que tienen multitud de interfaces.

3.4.1. Aspectos clave

- **Componentes de verificación VHDL:** los *VHDL Verification Components* (VVCs), análogos a los VCs de OSVVM, son componentes de verificación que permiten generar estímulos y observar y ejecutar transacciones sobre interfaces determinadas.
- **Orquestación centralizada de interfaces:** UVVM ha sido desarrollado con especial cuidado para mantener una forma fácil de interactuar con multitud de interfaces. Un único proceso es capaz de enviar comandos a múltiples VVCs. Se describen las operaciones de forma secuencial o concurrente sin necesidad de mecanismos de sincronización explícitos como es común con OSVVM [39].
- **Trazabilidad de requisitos:** UVVM incluye múltiples mecanismos de cobertura funcional (transacciones, valores/rangos/vectores, zonas, etc.). Sin embargo, también tiene mecanismos de cobertura de especificación. Dicha cobertura permite relacionar requisitos y tests de modo que se puede generar una matriz de trazabilidad, especialmente útil en sistemas críticos en los que hay una especificación bien definida.

3.4.2. Análisis según criterios

1. **Curva de aprendizaje:** moderada, de forma similar a OSVVM, representa una mejora respecto de UVM sobre todo para aquellos proyectos que utilicen VHDL. Frente a OSVVM puede suponer una mayor facilidad para manejar multitud de interfaces.
2. **Capacidades:** comparte capacidades con UVM y OSVVM, componentes de verificación reutilizables (VVCs), transacciones, mecanismos de alerta y registro, cobertura funcional. Sin embargo, también tiene cobertura de especificación, como se ha mencionado.
3. **Reutilización:** a nivel de reutilización, una vez más, similar a UVM y OSVVM. UVVM tiene una arquitectura modular que permite reutilizar escenarios de prueba, modelos de referencia, VVCs, mecanismos de comprobación, entre otros.
4. **Productividad:** principalmente en entornos con múltiples interfaces, concurrencia y requisitos estrictos de trazabilidad. Es posible orquestar los diferentes VVCs sin atender a la sincronización entre diferentes procesos. Además, los mecanismos de *logging* son más uniformes. Sin embargo, para módulos de reducido tamaño puede introducir complejidad innecesaria.
5. **Flexibilidad:** tiene soporte tanto en simuladores libres (GHDL [40], NVC [41]) como en simuladores comerciales (Questa SIM [42], Riviera-PRO [43], etc.). No proporciona soporte para cosimulación equivalente a la de OSVVM. La posibilidad de incluir modelos externos dependerá de los mecanismos específicos del simulador utilizado.
6. **Adecuación a diseños criptográficos:** es apropiado para verificar las interfaces de un acelerador criptográfico y puede tener ventajas dependiendo de cómo sea la especificación del diseño y sus requisitos de trazabilidad.

3.5. VUnit

VUnit es un *framework* orientado a realizar *unit tests* sobre diseños VHDL o Verilog. La idea central de VUnit es no imponer una metodología para el desarrollo de los *testbenches*. Es posible utilizar únicamente VHDL o Verilog estándar o añadir metodologías externas como UVVM

u [OSVVM](#). Está compuesto por dos partes principales: el *runner* o coordinador de tests basado en Python y las bibliotecas [HDL](#) que añaden diversas capacidades de verificación avanzada [44].

3.5.1. Aspectos clave

- **Python runner:** gestiona las dependencias, la elaboración, la ejecución selectiva de los tests, la simulación y configuración para todos los pasos. Centraliza la información acerca de los tests y permite la fácil integración del proyecto en entornos automatizados y de [CI/CD](#).
- **Bibliotecas de verificación HDL:** VUnit incluye bibliotecas para *logging*, comunicación entre procesos y componentes de verificación para interfaces comunes, entre muchas otras utilidades. Esto permite tener una progresión natural desde *testbenches* basados únicamente en [VHDL](#) estándar hasta la integración de metodologías externas como [OSVVM](#) y [UVVM](#).
- **Exploración paramétrica:** permite configurar los parámetros, *generics* ([VHDL](#)) o *parameters* (Verilog), configuraciones de bibliotecas u opciones de simulación. De este modo se pueden probar multitud de casos sin la necesidad de escribir de forma explícita diferentes tests y no solo para un conjunto acotado de parámetros.

3.5.2. Análisis según criterios

1. **Curva de aprendizaje:** muy reducida y adaptada a las necesidades del diseñador. Como se ha mencionado, es posible adoptar de forma incremental capacidades de verificación. Los *testbenches* se escriben en [VHDL](#) o Verilog, lo que puede ser una ventaja dependiendo del equipo si no se desea introducir un nuevo lenguaje para verificación, como se verá en la siguiente sección con Cocotb.
2. **Capacidades:** alta, permite descubrimiento y ejecución selectiva de tests, parametrización y generación de informes. Además, tiene bibliotecas para comprobación, *logging* y mecanismos de sincronización y registro.
3. **Reutilización:** moderada, ya que no existe una jerarquía metodológica estricta, de una forma similar a [VHDL](#) o Verilog estándar, aunque es posible desarrollar una propia metodología o conjunto de reglas de diseño o incluso utilizar [OSVVM](#) u otra metodología junto con VUnit.
4. **Productividad:** elevada, aunque una vez más dependerá de las capacidades de aquellos que adopten el *framework* y del proyecto que se desarrolle. Destaca en entornos en los que se desea mejorar las capacidades de [VHDL](#) o Verilog sin demasiados cambios.
5. **Flexibilidad:** elevada, debido principalmente al uso de Python por parte del *runner*, posee fácil integración con entornos automatizados y/o [CI/CD](#). Además, tiene soporte para simuladores libres y comerciales.
6. **Adecuación a diseños criptográficos:** aunque no es especialmente útil en diseños criptográficos, es adecuado para la verificación de interfaces o la integración con conjuntos de tests existentes, pues es posible integrar los *testbenches* que se hayan desarrollado incluso si se ha utilizado alguna metodología en específico. El objetivo en ese caso puede ser una mejor integración con otras metodologías, herramientas o [CI/CD](#).

3.6. Cocotb

Cocotb (*coroutine-based cosimulation testbench*) es un *framework* de verificación de código abierto con el que se escriben los *testbenches* en Python, mientras el **DUT** permanece escrito en VHDL o Verilog, con soporte para una gran variedad de simuladores [45], [46]. La comunicación entre ambas partes se realiza a través de una capa de abstracción propia de Cocotb, la *Generic Procedural Interface (GPI)*, que se apoya en las interfaces estándar de los simuladores: VPI para Verilog, VHPI y FLI para VHDL, dependiendo del simulador utilizado. Este modelo permite que el *testbench* sea independiente tanto del lenguaje del diseño como del simulador empleado y que no sea necesario escribir **HDL** para verificar.

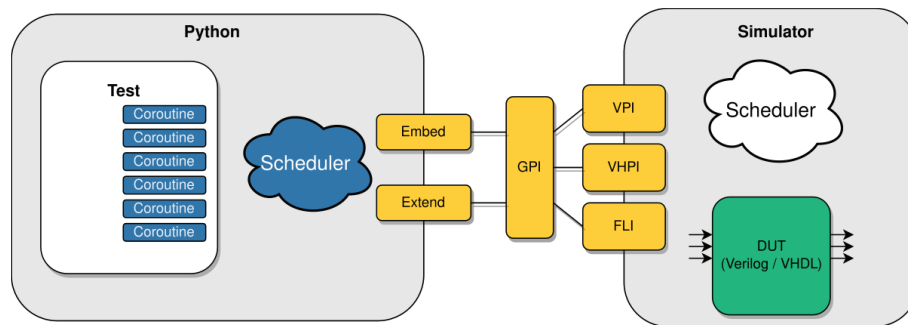


Figura 11: Conceptos básicos de Cocotb [47].

3.6.1. Aspectos clave

Los elementos principales de Cocotb son el uso de decoradores³ y corutinas⁴, el concepto de *trigger*, el acceso al **DUT** y las transiciones entre Cocotb y el simulador. Estos son los elementos que componen el ciclo de vida de los tests: cómo se señalizan los tests, cómo avanza el tiempo de simulación, la interacción con el diseño y el control del simulador:

- **Decoradores y corutinas:** para indicar a Cocotb qué corutinas son tests se utiliza el decorador `@cocotb.test()` sobre una determinada corutina. Cada test es una función asíncrona que permite describir el comportamiento temporal de un *testbench*. Es posible ejecutar corutinas concurrentes mediante `cocotb.start_soon()`.
- **Acceso al **DUT**:** una de las funciones del decorador `@cocotb.test()` es la inyección del *handler* `dut`, que permite interactuar con las señales y submódulos del **DUT** según la jerarquía del diseño. Dentro de un test se utiliza la notación de punto (`dut.señal`) para lecturas y asignaciones.
- **Transiciones entre Cocotb y el simulador:** cuando se lanzan a ejecución los tests, el *script* de Python que define los tests comienza su ejecución. Una vez el *script* se encuentra con un *await* se cede el control al simulador y avanza el tiempo de simulación hasta que se lleva a cabo la operación indicada por dicho *await*. Al concluir la operación, Cocotb retoma el control y se prosigue con la ejecución del *script*.

³Los decoradores son construcciones que afectan el comportamiento de funciones, métodos o clases sin alterar el código de las mismas, se indican mediante la construcción `@decorador`.

⁴Una corutina es una función, definida usando `async def`, cuya ejecución puede pararse mediante el uso de `await` y reanudarse tras la ejecución de lo que acompañe dicho `await`. Permite expresar concurrencia.

- **Triggers:** los *triggers* permiten controlar el simulador. Al esperar un *trigger* mediante un *await* se pasa el control al simulador hasta que se cumple con la tarea indicada. Existen *triggers* de diferente naturaleza: de flanco (`RisingEdge`, `FallingEdge`), temporales (`Timer`), número de ciclos (`ClockCycles`) y de sincronización dentro del paso de la simulación (`ReadOnly`, `NextTimeStep`).

En el [Código 1](#) se presenta un *script* de ejemplo en el que se puede ver un test y los elementos detallados con anterioridad: en las líneas 5 y 6 se pueden ver el decorador que utiliza Cocotb y la declaración del test haciendo uso de las corutinas, respectivamente. En la línea 8, un ejemplo de cómo interactuar con el DUT. En la línea 9, el uso de `await` y el *trigger* `Timer`. Incluidas en el ejemplo están funciones estándar de los *frameworks* y metodologías como las funciones `log` y `assert`, líneas 13 y 14.

```

1 import cocotb
2 from cocotb.triggers import Timer
3
4 @cocotb.test()
5 async def my_first_test(dut):
6     for _ in range(10):
7         dut.clk.value = 0
8         await Timer(1, unit="ns")
9         dut.clk.value = 1
10        await Timer(1, unit="ns")
11
12    cocotb.log.info("my_signal_1 is %s", dut.my_signal_1.value)
13    assert dut.my_signal_2.value == 0

```

Código 1: Script básico de Cocotb con un único test.

Los mecanismos anteriores cubren los aspectos clave de Cocotb, permiten escribir *testbenches* para abordar entornos complejos y permiten la reutilización de los elementos de diseño de tests. Sin embargo, Cocotb está centrado en la funcionalidad que más se utiliza. Aquellos casos de uso que no encajan con Cocotb es posible cubrirlos mediante el uso de extensiones. Existen multitud de extensiones. A continuación, se describen dos de las más importantes: pyuvvm y Cocotbext-*

pyuvvm

pyuvvm [48] es una biblioteca que implementa UVM sobre Cocotb, basándose en las secciones relevantes para Python del estándar IEEE 1800.2 [49]. Su finalidad es trasladar al ecosistema de Cocotb elementos como *monitors*, *scoreboards*, *drivers*, etc. para su utilización con los mismos fines planteados en la [Sección 3.2.1](#).

Su utilidad principal es la reutilización siguiendo los principios de UVM, de modo que los *testbenches* dejan de ser una colección de tests independientes y pasan a ser estructuras estándar. El código que compone los tests sigue una filosofía similar a la de UVM, pero con las facilidades y abanico de bibliotecas de Python a disposición del diseñador, facilitando la integración de modelos y vectores de referencia con dependencias externas o procesamiento matemático complejo.

No obstante, pyuvvm, de forma similar a UVM, introduce una capa adicional de abstracción. Para módulos reducidos o pequeños tests dirigidos, puede suponer una complejidad innecesaria frente a utilizar Cocotb estándar.

Cocotbext-*

Destacan de igual modo las extensiones del autor Alex Forenchich, que utiliza, en el nombre de sus extensiones, el prefijo recomendado según la documentación de Cocotb, `cocotbext-:` `cocotbext-{eth,i2c,axi,pcie,uart}` [50], que introducen elementos de verificación para Ethernet, I²C, *Advanced eXtensible Interface (AXI)*, PCIe y UART, respectivamente.

3.6.2. Análisis según criterios

1. **Curva de aprendizaje:** presenta una curva de aprendizaje reducida. Cocotb está basado en Python, un lenguaje de programación de uso general conocido por su facilidad, amplia biblioteca estándar e infinidad de bibliotecas de terceros de fácil uso. Además, presenta una documentación accesible en la que se detalla cómo dar los primeros pasos y cómo aplicar Cocotb a los casos de uso más comunes.
2. **Capacidades:** permite desarrollar *testbenches* avanzados con aleatorización, modelos complejos, cosimulación, señal mixta (dependiendo del simulador), entre otros, y pone a disposición del diseñador la fácil ampliación de sus capacidades mediante las extensiones.
3. **Reutilización:** permite organizar los tests en funciones, clases y módulos reutilizables siguiendo el enfoque orientado a objetos. De forma alternativa, se pueden utilizar enfoques funcionales o procedimentales, propios de Python. Cocotb no impone un flujo de trabajo aunque es posible utilizar extensiones para adoptar un esquema de organización rígido, como con pyuvvm.
4. **Productividad:** es elevada sobre todo en fases de desarrollo iniciales o prototipado, aunque puede disminuir en caso de que se trate de recrear esquemas de organización complejos.
5. **Flexibilidad:** permite trabajar con diseños escritos tanto en VHDL como en Verilog y con una amplia lista de simuladores tanto comerciales como de uso libre [45]. Aparte, queda a disposición del diseñador todo el ecosistema de Python y la capacidad de interoperabilidad que este conlleva.
6. **Adecuación a diseños criptográficos:** es especialmente idóneo para diseños criptográficos. Permite modelar fácilmente algoritmos complejos o utilizar implementaciones de terceros verificadas de forma independiente.

3.7. Evaluación

En la [Tabla 1](#) se sintetiza el análisis desarrollado en el capítulo mediante una valoración numérica de las propiedades de cada *framework*. Cada propiedad se ha valorado en una escala del 1 al 5, donde el 5 es la puntuación más favorable. Por tanto, para la curva de aprendizaje una puntuación alta significa una adopción más sencilla. El total se ha calculado con ponderación uniforme. Cabe mencionar que en la evaluación no se ha incluido la verificación formal, [Sección 3.8](#), porque en el contexto de este capítulo no se considera como alternativa a los *frameworks* que emplean simulación funcional.

Tabla 1: Valoración global de los frameworks con ponderación uniforme

Dimensión	UVM	UVVM	OSVVM	VUnit	Cocotb
Curva de aprendizaje	1	3	3	5	4
Capacidades	5	4	4	4	4
Reutilización	5	4	4	2	2
Productividad	2	3	3	4	5
Flexibilidad	1	3	3	5	5
Adecuación criptográfica	3	3	3	3	5
Total (sobre 30)	17	20	20	23	25

La valoración no trata de seleccionar un único *framework* para todos los bloques del diseño. Cocotb alcanza la mayor puntuación y se considera la opción más apta para verificar bloques genéricos por su alta flexibilidad y la posibilidad de utilizar un modelo de referencia en Python o incluso integrar el *testbench* con otras herramientas y lenguajes. Por otra parte, VUnit junto con OSVVM proporcionan a este trabajo la capacidad de explorar la adopción de *frameworks* de verificación avanzados mediante un camino de baja fricción: permite partir de un *testbench* básico e incorporar VUnit, o VUnit con OSVVM, a medida que las necesidades de verificación crecen.

3.8. Verificación formal

La herramienta de verificación considerada para verificación formal ha sido Sby que forma parte de la distribución de *software* «OSS CAD Suite» de YosysHQ. Las razones para su empleo radican en su extendido uso y buen soporte por parte de las herramientas de simulación de uso libre para SystemVerilog Assertions frente al equivalente en *VHDL Property Specification Language (PSL)*.

3.8.1. OSS CAD Suite - Sby

OSS CAD Suite [51] de YosysHQ es una distribución de herramientas de código abierto que forma parte, a su vez, de Tabby CAD [52], distribución comercial que expande las capacidades de la primera. OSS CAD Suite contiene diferentes utilidades para el diseño digital: síntesis RTL, verificación formal, *place & route* y programación FPGA, entre otras. A diferencia del resto de *frameworks* anteriores, no constituye un *framework* o metodología para la verificación, sino un entorno completo que permite desarrollar diferentes flujos de trabajo para el desarrollo de diseños digitales. Su utilidad como parte de este TFM reside en una de sus herramientas, Sby, que se utilizará para realizar verificación formal.

Sby utiliza los ficheros con extensión `.sby` para describir las tareas de verificación formal que se han de llevar a cabo. Se especifican las fuentes del diseño, el modo de verificación y los motores *Satisfiability Modulo Theories (SMT)* utilizados [53]. Es posible configurar diferentes modos (`mode`) de funcionamiento: `bmc`, `prove`, `cover` y `live`. Los modos `bmc` y `prove` se utilizan para realizar verificación acotada o no acotada respectivamente mediante las sentencias `assert()` de SystemVerilog. El modo `live` permite realizar verificación no acotada para verificar propiedades de tipo *liveness*, mediante las sentencias `assert(s_eventually ...)`. El último modo, `cover`, permite generar las trazas más cortas necesarias para alcanzar las sentencias `cover()`, es decir, comprobar si es posible que el diseño llegue a una situación concreta.

3.9. Conclusión

Tanto la evaluación realizada en este capítulo como el análisis de los diferentes ejes planteados en el [Capítulo 2](#) muestran que no existe un único *framework* aplicable a todos los bloques y niveles de verificación.

Por este motivo, la selección no se basa de forma exclusiva en la puntuación obtenida en la [Tabla 1](#), sino que incluye las necesidades y limitaciones que existen en los diseños reales. Cocotb se ha utilizado como *framework* principal para la verificación funcional de los bloques genéricos que no se benefician de forma directa de otras facilidades del resto de los *frameworks*.

[OSVVM](#) se ha empleado en aquellos casos en los que los [VCs](#) faciliten la verificación de interfaces estándar, junto a VUnit para explorar la integración de ambos y un camino de fácil adopción de *frameworks* de verificación avanzadas a partir de *testbenches* comunes.

Finalmente, la verificación formal se ha seleccionado para aquellos bloques de baja latencia y baja complejidad aritmética y, en concreto, [Sby](#) por su uso extendido y amplio soporte por parte de las herramientas de uso libre. La combinación de Cocotb, VUnit, [OSVVM](#) y [Sby](#) establece la base de *frameworks* sobre la que se elabora el *pipeline* de verificación.

Capítulo 4

Criptografía postcuántica

4.1. Fundamentos de criptografía clásica

La criptografía moderna permite proteger la información frente a posibles adversarios que buscan modificar u observar las comunicaciones en un sistema digital. Si bien las técnicas han pasado por un proceso de evolución continuo desde los cifrados clásicos hasta los protocolos actuales, el objetivo final se mantiene: diseñar construcciones matemáticas cuya vulneración resulte inviable para un modelo de adversario muy concreto que no cuenta con capacidad computacional inagotable.

En esta sección se pretende contextualizar cómo se ven afectadas las primitivas y esquemas criptográficos actuales frente a la computación cuántica y por qué surge la necesidad de algoritmos postcuánticos. En particular, son relevantes las propiedades de seguridad necesarias, las primitivas utilizadas para establecer los sistemas criptográficos y la noción de seguridad basada en dificultad computacional.

4.1.1. Propiedades fundamentales

Los sistemas criptográficos están diseñados de modo que puedan proveer de diferentes garantías, dependiendo de los escenarios y tipos de amenazas planteados, los más habituales son: confidencialidad, integridad de los datos, autenticación y la autenticación de origen [54, sec. 1.1].

La **confidencialidad** es la propiedad que establece que la información no debe acabar en manos de terceros no autorizados. Para lograr esto, un determinado mensaje se cifra de modo que el adversario que haya interceptado la comunicación no sea capaz de recuperar el contenido de dicho mensaje. Esta propiedad fundamenta las comunicaciones privadas y el almacenamiento de datos seguro.

La **integridad** garantiza que una determinada información no ha sido modificada por terceros no autorizados, posibilitando la detección de alteraciones. Para esta propiedad se utilizan *hashes* y firmas digitales, principalmente. Esta propiedad es vital para la distribución de *software*, la mensajería segura y los intercambios de documentos firmados.

La **autenticación** permite comprobar la identidad de una entidad que interactúa con un sistema para evitar que un adversario suplante una entidad legítima. Es imprescindible en procesos de inicio de sesión y protocolos como el *Transport Layer Security (TLS)*.

La **autenticación de origen** capacita a una entidad receptora para confiar en el origen de la información, es decir, garantías acerca de la fuente. Un caso particular es el **no repudio**, con el que el servicio de **autenticación de origen** es prestado por un tercero.

En los sistemas reales rara vez las propiedades se materializan de forma aislada. Por ejemplo, un protocolo de mensajería segura ha de estar diseñado teniendo en cuenta múltiples vectores de ataque, confidencialidad para proteger el contenido, integridad para detectar modificaciones y autenticación de origen para tener garantías acerca de la identidad de los participantes y la fuente de los mensajes.

4.1.2. Primitivas y esquemas criptográficos

Las primitivas criptográficas son algoritmos criptográficos de bajo nivel a partir de los cuales se elaboran esquemas y protocolos más complejos. Las primitivas de mayor relevancia son: las funciones unidireccionales, la criptografía simétrica, la criptografía asimétrica, las firmas digitales, mecanismos de establecimiento de claves, generadores de números aleatorios, redes de mezcla, entre muchas otras.

Cifrados de bloque y de flujo

Los cifrados transforman el texto plano en texto cifrado. Los cifrados de flujo cifran la información unidad por unidad, normalmente *bytes*, mientras que los cifrados de bloque toman un bloque de bits y lo transforman añadiendo relleno al texto plano en caso de que sea necesario. El tamaño del resultado es múltiplo del tamaño del bloque utilizado.

Criptografía simétrica

La criptografía simétrica o criptografía de clave simétrica usa una única clave compartida tanto para cifrar el texto plano como para descifrar el cifrado. La principal desventaja que presenta es la necesidad de que todas las partes posean el secreto compartido, a diferencia de la criptografía asimétrica. Sin embargo, es más apta para cifrar grandes volúmenes de información debido a que generalmente es más rápida y presenta un tamaño de clave menor. La criptografía simétrica se puede implementar mediante unidades de cifrado de flujo o de bloque.

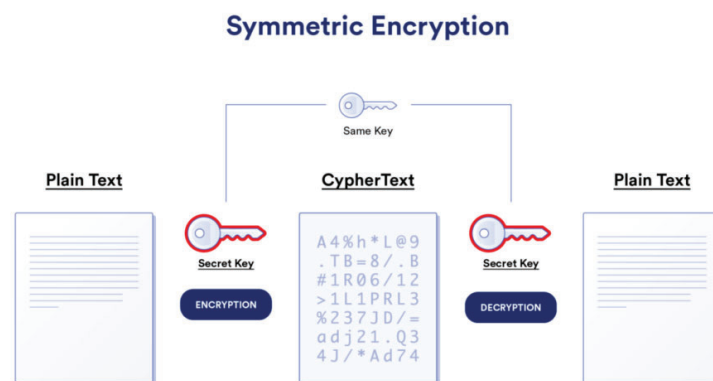


Figura 12: Criptografía simétrica [55].

Criptografía asimétrica

La criptografía asimétrica o de clave pública emplea un par de claves relacionadas matemáticamente que tienen unas propiedades particulares: la conocida como clave pública que puede distribuirse libremente y una clave privada que ha de mantenerse secreta. A partir de esta primitiva se desarrollan criptosistemas como las firmas digitales y el protocolo Diffie-Hellman.

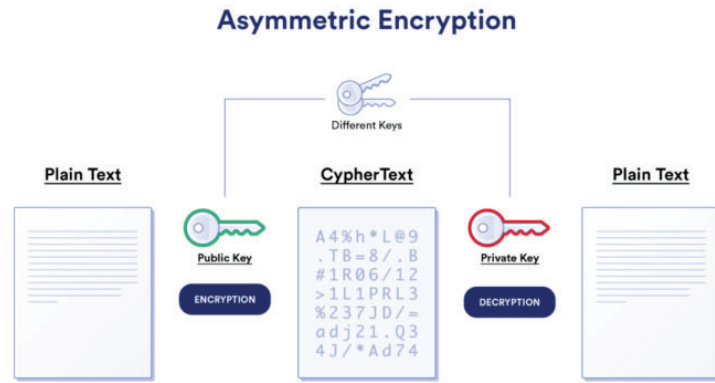


Figura 13: Criptografía asimétrica [55].

Funciones *hash*

Las funciones *hash* son algoritmos que cumplen con tres propiedades: primero, ha de ser no reversible; segundo, ante una entrada de tamaño arbitrario produce una salida de longitud fija; y por último, ha de ser resistente a las colisiones, es decir, tiene que ser difícil de encontrar dos entradas que den la misma salida [56, Sec. 9]. Tres propiedades que se pueden entender de forma visual mediante la Figura 14.

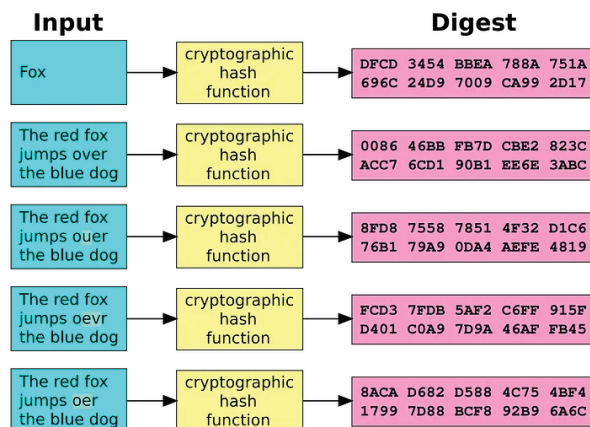


Figura 14: Representación de las tres propiedades del hash [57].

Firmas digitales

El concepto de firma digital se basa en la criptografía asimétrica. Partiendo de un par de claves, un algoritmo de firma toma el mensaje y la clave privada y produce una firma. Por otra parte, el algoritmo de verificación de firma toma un mensaje, una clave pública y la firma y comprueba la autenticidad del mensaje. En la Figura 15 se presentan los procesos de firma y verificación de firmas.

La integridad de los documentos firmados queda establecida debido a que la firma está asociada de forma inequívoca con el documento firmado. Finalmente, el no repudio, es decir, que ni el receptor ni el emisor pueden negar su participación a posteriori, se garantiza gracias a que las claves

identifican de forma inequívoca a ambas partes. De esta forma quedan establecidas las propiedades de las firmas digitales: autenticidad, integridad y no repudio.

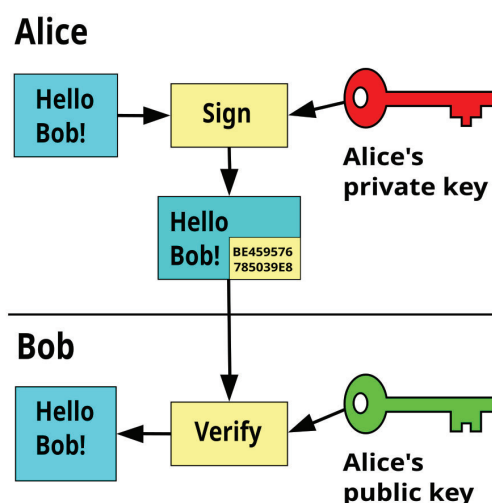


Figura 15: Procesos de firma y verificación de firmas digitales [58].

4.1.3. Seguridad basada en dificultad computacional

En la práctica, la seguridad en los sistemas criptográficos no se basa en la imposibilidad absoluta de quebrantarlos debido a que, en caso de imponer este requisito, es necesario implementar el concepto de **seguridad perfecta** que requiere de claves del mismo tamaño que los mensajes que se tratan de compartir: para compartir 1 GB de información, la llave ha de ser de 1 GB, esquema que resulta de poca utilidad. [59, Sec. 2.1.2]

Es en este marco, donde las construcciones criptográficas se apoyan para desarrollar un concepto de seguridad que no siendo perfecto resulta adecuado debido a los costes computacionales asociados, de modo que se busca imponer costes prohibitivos en términos de computación a los posibles atacantes manteniendo los procesos eficientes para los usuarios legítimos. Es por esto que, ante el desarrollo de las capacidades de computación, los algoritmos pueden quedar inservibles dependiendo del adversario considerado. Un algoritmo puede ser seguro ante adversarios clásicos y no serlo ante un modelo de computación capaz de resolver el problema matemático en el que se basa. Esta es la base de la amenaza que representa la computación cuántica y la motivación para realizar una transición hacia criptosistemas para los que no se conocen ataques cuánticos eficientes.

4.2. Amenaza cuántica

Como se ha mencionado con anterioridad, la seguridad computacional moderna se ha basado en un modelo de computación que no es capaz de resolver de forma eficiente los problemas matemáticos que sustentan los criptosistemas de los que hacemos uso en la actualidad. La computación cuántica surge en 1980 con la idea de Paul Benioff de aprovechar las leyes cuánticas para describir un ordenador simplificado [60]. Pese a que su aplicación práctica sigue limitada en la actualidad, se continúan haciendo avances que mejoran las capacidades de los «ordenadores cuánticos».

Sobre la idea de computación cuántica se han desarrollado algoritmos como los planteados por Shor en 1994 que permiten resolver los problemas de cálculo de logaritmos discretos y la factorización de enteros en tiempo polinómico [12]. Otro ejemplo de estos algoritmos es el de Grover [61] (1996), con el que se puede realizar la búsqueda en una secuencia de datos N en un tiempo $O(N^{\frac{1}{2}})$ en lugar de $O(N)$.

4.2.1. Ataques e impacto sobre los sistemas actuales

El impacto sobre los sistemas criptográficos no es uniforme. Los algoritmos propuestos por Shor afectan a la criptografía asimétrica clásica, utilizada en [RSA](#) y Diffie-Hellman y la criptografía de curva elíptica. Es por esto que los mecanismos utilizados para establecimiento de claves, intercambio seguro de información y firmas digitales, frente a un adversario cuántico, dejan de ofrecer garantías suficientes.

Por otra parte, el algoritmo de Grover tiene un efecto en la criptografía simétrica y las funciones *hash*. Al reducirse el coste de las búsquedas exhaustivas, disminuye el nivel de seguridad efectivo de las claves utilizadas y la utilidad de los *hashes*, aunque no las invalida por completo.

Finalmente, es importante tener en cuenta que aunque en la actualidad no exista un computador cuántico con las capacidades necesarias para implementar los algoritmos anteriores, los ataques [HNDL](#) recuerdan la necesidad y urgencia de realizar la migración hacia sistemas postcuánticos. Un adversario, con las capacidades de almacenamiento necesarias, puede guardar comunicaciones cifradas actualmente para descifrarlas con posterioridad en el momento en que se disponga de capacidad cuántica necesaria siguiendo los pasos detallados en la [Figura 16](#).

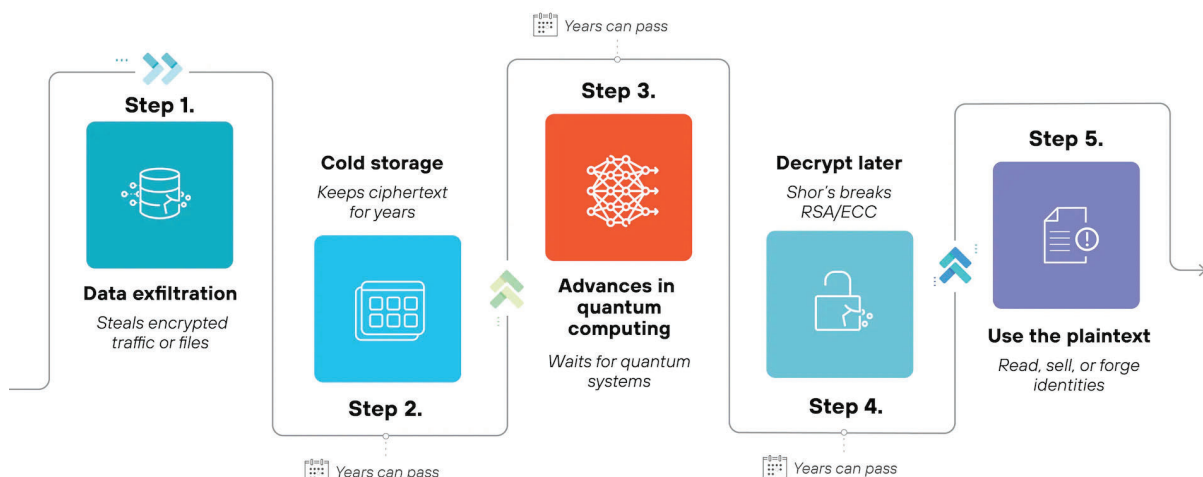


Figura 16: Pasos que se toman en un ataque de tipo [Harvest Now, Decrypt Later \(HNDL\)](#). Adaptado de [62].

4.3. Criptografía postcuántica

La criptografía postcuántica es el conjunto de protocolos, esquemas y primitivas criptográficas diseñados teniendo en cuenta tanto los ataques posibles con computación clásica como con computación cuántica para su implementación sobre sistemas digitales convencionales. Es clave la distinción respecto de la criptografía cuántica, que se basa en el uso de fenómenos físicos cuánticos para implementar mecanismos criptográficos.

4.3.1. Familias criptográficas postcuánticas

De un modo similar a la criptografía clásica, existen diferentes aproximaciones a la hora de construir criptosistemas postcuánticos. Estas aproximaciones se han desarrollado partiendo de diferentes problemas considerados «difíciles» tanto para computadores clásicos como cuánticos. A continuación, se presentan las diferentes familias de problemas utilizados.

- **Basados en retículos:** son problemas de elevada complejidad basados en el concepto de retículo [63] como el problema del vector más corto o *Shortest Vector Problem (SVP)*, el problema del vector más cercano a otro o *Closest Vector Problem (CVP)* o el problema del cálculo de la base generadora de retículo de longitud mínima *Shortest Independent Vector Problem (SIVP)*.
- **Basados en códigos:** están centrados en el concepto de códigos de corrección de errores y la dificultad de recuperar un mensaje sin conocer el código que corrige los errores que se han introducido. El ejemplo clásico es el criptosistema propuesto por McEliece en 1978 [64].
- **Basados en problemas multivariables:** son problemas basados en la dificultad de resolución de sistemas de ecuaciones no lineales, generalmente sobre cuerpos finitos.
- **Basados en isogenias:** son problemas similares al logaritmo discreto, pero definidos sobre grupos de isogenias asociadas a ciertas estructuras algebraico-geométricas cuya complejidad se estima elevada.
- **Basados en funciones *hash*:** basados en la seguridad de las funciones *hash*, estructuras ampliamente estudiadas y consideradas seguras ante ataques clásicos y postcuánticos. Su principal uso ha sido la implementación de esquemas de firmas.

4.4. Proceso de estandarización del NIST

El **NIST** es una agencia que forma parte del Departamento de Comercio de los Estados Unidos. Está dedicada al desarrollo de estándares, referencias y guías técnicas que afectan a los ámbitos científicos y tecnológicos. Sus estándares tienen influencia a nivel global, tanto en la industria como en entornos académicos y son utilizados como referencia por fabricantes, organismos públicos, proveedores de servicios y desarrolladores de protocolos.

Dentro del ámbito de la seguridad de la información, el **NIST** ha publicado diferentes estándares junto con recomendaciones, algoritmos y parámetros de seguridad para su correcta implementación. Es por esto que en el proceso de estandarización de la criptografía postcuántica han sido claves con publicación de los conocidos *Federal Information Processing Standards (FIPS)*, en concreto, **FIPS 203**, **204**, **205** y en un futuro **FIPS 206**.

Previo a la publicación de los **FIPS** mencionados, se planteó un proceso público e internacional para evaluar posibles candidatos, enfocado a ser un esfuerzo colaborativo que tuviera en cuenta tanto la selección de algoritmos matemáticamente seguros como los aspectos prácticos de cada uno: rendimiento, facilidad de implementación, tamaños de claves y firmas y madurez de las primitivas criptográficas utilizadas, tratando de diversificar los fundamentos matemáticos utilizados. El proceso se organizó en varias rondas de selección de algoritmos divididos por las diferentes familias de problemas matemáticos en los que se basaron, concluyendo con la estandarización de aquellos más aptos.

4.4.1. Estándares publicados

El 13 de agosto de 2024, el [NIST](#) publicó los primeros tres estándares para criptografía postcuántica, [Tabla 2](#). Estos estándares son la base para la migración hacia algoritmos resistentes a adversarios con capacidades cuánticas. Cubren dos necesidades clave: las firmas digitales y el establecimiento de claves mediante mecanismos de encapsulado de claves.

Tabla 2: Estándares postcuánticos publicados por el NIST, salvo FIPS 206, en proceso por el momento.

Estándar	Algoritmo	Uso	Familia
FIPS 203	ML-KEM	KEM	Retículos
FIPS 204	ML-DSA	Firma	Retículos
FIPS 205	SLH-DSA	Firma	Funciones <i>hash</i>
FIPS 206	FN-DSA	Firma	Retículos

- **FIPS 203 (ML-KEM):** es un mecanismo de encapsulado de claves o *Key encapsulation mechanism (KEM)*, basado en retículos. Su finalidad es permitir que dos partes establezcan un secreto compartido sobre un canal público. Este, posteriormente, puede utilizarse con algoritmos de clave simétrica para cifrado o autenticación.
- **FIPS 204 (ML-DSA):** es un algoritmo de firma digital basado, una vez más, en retículos. Permite proporcionar autenticación e integridad mediante la generación y verificación de firmas digitales.
- **FIPS 205 (SLH-DSA):** es un algoritmo de firma digital sin estado, basado en funciones *hash*. Parte de su finalidad es aportar diversidad respecto a los esquemas basados en retículos, porque emplea *hashes*, primitivas ampliamente estudiadas. Es una alternativa a los algoritmos de firma digital con estado basados en funciones *hash*, que pese a no formar parte del proceso de estandarización para criptografía postcuántica, se indican en la recomendación SP 800-208 del [NIST](#): los algoritmos *Leighton-Micali Signature (LMS)* y *eXtended Merkle Signature Scheme (XMSS)* requieren que se mantenga un control estricto sobre el estado de la clave privada, debido a que la reutilización de dicho estado compromete la seguridad del esquema. No se recomienda su uso generalizado aunque ofrecen un nivel de seguridad postcuántica alto en entornos controlados. La [National Security Agency \(NSA\)](#) los recomienda para la firma y la verificación de *firmware* dentro de su suite de algoritmos *Commercial National Security Algorithm Suite (CNSA)* 2.0 [65].

Además de los estándares publicados, el [NIST](#) ha seleccionado el algoritmo Falcon para su futura estandarización como FN-DSA en [FIPS 206](#), como se muestra en la [Tabla 2](#). Se presenta como una alternativa adicional para las firmas digitales y ofrece compromisos en términos de tamaño, rendimiento y complejidad de implementación respecto a ML-DSA y SLH-DSA.

4.4.2. Implicaciones sobre la adopción y diversidad criptográfica

La transición hacia la criptografía postcuántica ha de contemplarse como un proceso continuo, no como la sustitución puntual de los algoritmos clásicos. Para ello, el [NIST](#) continúa impulsando alternativas, como se ha mencionado, que aportan diversidad criptográfica, como es el caso de FN-DSA para la firma digital y [Hamming Quasi-Cyclic \(HQC\)](#) como [KEM](#) de respaldo basado en códigos. El objetivo de esta estrategia es reducir los posibles riesgos que futuros avances en el criptoanálisis puedan revelar.

4.5. Conclusión

La seguridad de los sistemas digitales modernos se apoya en las primitivas criptográficas que los componen. A su vez, la utilidad de cada una de estas primitivas depende de la robustez de ciertos problemas matemáticos frente a las capacidades computacionales de un determinado adversario considerado. La aparición de algoritmos cuánticos y los avances en computación cuántica obligan a replantear los criptosistemas desplegados en la actualidad.

A lo largo del capítulo se han presentado tanto la criptografía clásica como los esquemas que son necesarios para combatir el futuro cada vez más cercano de la computación cuántica, es decir, la criptografía postcuántica. Es aquí donde el proceso de estandarización del [NIST](#) ha sido clave para generar una serie de estándares que presenten tanto garantías ante los ataques clásicos conocidos como aquellos que requieren de computación cuántica, principalmente, ML-KEM y ML-DSA.

El presente trabajo pone el foco en que no solo es necesario tener estándares aptos, sino que también es necesario realizar las implementaciones de dichos estándares con garantías. Los esquemas postcuánticos están compuestos por algoritmos, estructuras de datos y operaciones aritméticas que deben verificarse con rigor. En el siguiente capítulo, [Capítulo 5](#) se presentan los [KEM](#) postcuánticos relevantes y, en concreto, ML-KEM como caso de aplicación para el diseño de un *pipeline* de verificación avanzada.

Capítulo 5

Estudio y análisis de un KEM postcuántico

5.1. Mecanismos de Encapsulado de Claves (KEM)

Los mecanismos de encapsulado de claves son esquemas criptográficos que permiten establecer un secreto compartido entre dos o más participantes a través de un canal de comunicación no seguro. Guarda similitudes con los esquemas de clave simétrica. Sin embargo, en este caso no se cifra un mensaje arbitrario. En su lugar, se cifra una clave simétrica que podrá emplearse en otros esquemas criptográficos, como protocolos de comunicación segura o cifrados autenticados [66].

Los KEM son tipos de esquemas de establecimiento de claves. Es común que involucren a dos partes (por ejemplo, Alice y Bob en el caso de la Figura 17). Los esquemas de establecimiento de claves están compuestos por tres operaciones principales: generación de claves, encapsulado y desencapsulado.

- Generación de claves: Alice genera un par de claves, la clave de desencapsulado (privada) y la clave de encapsulado (pública).
- Encapsulado: Bob utiliza la clave de encapsulado de Alice para generar una copia de la clave compartida y su texto cifrado asociado. Bob transmite el texto cifrado a Alice.
- Desencapsulado: Alice utiliza el texto cifrado y su clave de desencapsulado para generar una copia del secreto compartido.

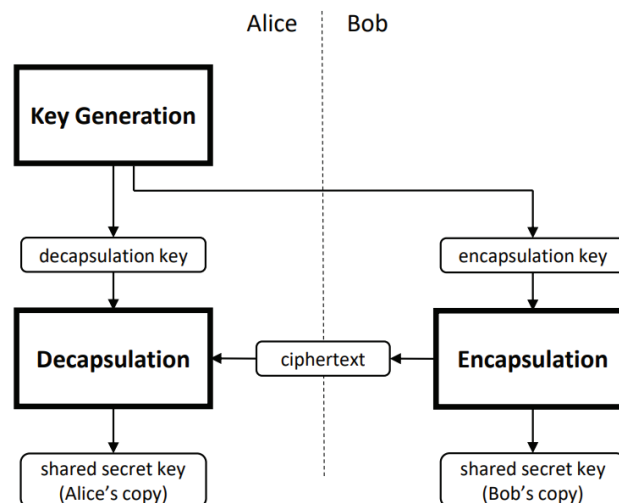


Figura 17: Establecimiento de claves mediante KEM [66].

El resultado principal de estas operaciones es que ambas partes terminan con un secreto compartido que es aleatorio, desconocido para los adversarios y con alta probabilidad idéntico para las partes. No obstante, para que se mantenga la seguridad del esquema, se han de cumplir tres condiciones:

1. Seguridad del **KEM** empleado: el **KEM** seleccionado ha de estar especificado sin ambigüedades en los algoritmos que lo componen y ha de satisfacer la noción matemática de seguridad según el contexto en el que se utiliza.
2. Seguridad en la implementación: tanto los algoritmos que componen el **KEM** como el sistema en su totalidad han de ser llevados a la práctica, es decir, implementados sin fallos de acuerdo con la especificación.
3. Seguridad de despliegue: se han de tener en cuenta los diferentes vectores de ataque para un entorno concreto, de modo que el sistema no presente vulnerabilidades específicas del mismo.

De no cumplirse estas condiciones, no se asegura la seguridad del sistema. Por ejemplo, un esquema de establecimiento de claves puede ser seguro de acuerdo a sus principios matemáticos, pero si es vulnerable a ataques de canal lateral⁵, el sistema pierde garantías de seguridad.

5.2. Mecanismos de encapsulado de claves postcuánticas

Durante el proceso de estandarización del **NIST** se evaluaron los **KEMs** postcuánticos de la [Tabla 3](#). De la tercera ronda resultó un único candidato a estandarización, ML-KEM. Continuaron a la cuarta ronda BIKE, Classic McEliece, HQC y SIKE. La cuarta ronda finalizó con HQC como segundo candidato a estandarización. En el caso de SIKE, además de no ser seleccionado, se encontraron vulnerabilidades criptográficas.

Tabla 3: KEMs candidatos considerados durante el proceso de estandarización del NIST

KEM	Familia	Estado	Conclusión principal
ML-KEM	Retículos (<i>Module-LWE</i>)	Estandarizado: FIPS 203	Primer KEM en ser estandarizado, equilibrio entre tamaños de clave, rendimiento y madurez del criptoanálisis.
HQC	Códigos (QCSD)	Futura estandarización (2027)	Elegido como alternativa a ML-KEM. Mayor tamaño de clave y esfuerzo computacional que BIKE y ML-KEM.
BIKE	Códigos (QC-MDPC)	No seleccionado por el NIST	Más compacto que HQC, no seleccionado en favor de HQC. Tiene menos recorrido en cuanto a criptoanálisis.
Classic McEliece	Códigos (binarios de Goppa)	No seleccionado por el NIST	Alternativa criptográfica conservadora con textos cifrados pequeños, pero claves muy extensas. Su utilidad principal es en situaciones en las que las claves se puedan reutilizar o distribuir con poca frecuencia.
SIKE	Isogenias (SIDH)	No seleccionado por el NIST (fallas demostradas)	Los ataques de tipo Castryck-Decru rompen la seguridad de SIKE/SIDH.

5.3. ML-KEM

El origen de *Module-Lattice-Based Key-Encapsulation Mechanism* (ML-KEM) está en CRYSTALS-Kyber [67]. Fue seleccionado por el **NIST** por su equilibrio entre seguridad, tamaños de clave, rendimiento y madurez desde el punto de vista del criptoanálisis. Está basado en retículos, en particular, su seguridad se apoya en la dificultad del problema *Module Learning With Errors* (MLWE) y soporta tres niveles de seguridad como se puede ver en la [Tabla 4](#). Una de las ventajas de esta base matemática es que permite su eficiente implementación tanto en *software* como en *hardware* debido a que son estructuras algebraicas regulares.

⁵Los ataques de canal lateral o *side-channel attacks* se basan en las fugas de información de un sistema informático debidas a la implementación física del propio sistema. Un ejemplo de esto es el análisis de potencia que puede derivar en la lectura de los bits que componen claves privadas en el momento en que sean utilizados por la *Central Processing Unit* (CPU).

Tabla 4: Tamaños, en bytes, de las claves de encapsulado y desencapsulado, el cifrado y el secreto compartido recogidos en el estándar FIPS 203 para ML-KEM.

	clave de encapsulado	clave de desencapsulado	cifrado	secreto compartido
ML-KEM-512	800	1632	768	32
ML-KEM-768	1184	2400	1088	32
ML-KEM-1024	1568	3168	1568	32

5.3.1. Principios matemáticos

Para entender ML-KEM es necesario conocer la serie de conceptos matemáticos en los que se basan las diferentes operaciones que son necesarias para llevar a cabo los tres mecanismos principales de los KEM desarrollados en la Sección 5.1. Estos conceptos son: la aritmética modular, los anillos polinómicos, el módulo R_q^k , y los problemas basados en retículos.

Aritmética modular

La aritmética modular se basa en el estudio del resto de la división (división euclidiana). En términos generales, se trata de la operación $\boxed{\text{mod}}$. Una de las ideas clave de la aritmética modular es la de «congruente». Una forma de entenderlo es utilizar el reloj de 12 horas, que divide el día en dos. Cada hora tiene dos congruentes, de forma intuitiva, cada posición en el reloj tiene dos horas asociadas, para las 7:00: las 7:00 y las 19:00. Esto se puede escribir $7 \equiv 19(\text{mod } 12)$.

Dos enteros son congruentes módulo q si su diferencia es divisible por q . El conjunto de representantes puede expresarse como $Z_q = \{0, 1, \dots, q-1\}$, y las operaciones de suma, resta y multiplicación se realizan reduciendo el resultado módulo q . En ML-KEM se utiliza $q = 3329$. Por tanto, cualquier coeficiente producido durante una operación interna debe reducirse a un representante válido módulo q .

Anillos polinómicos

ML-KEM no opera de forma directa con números enteros módulo q sino con polinomios cuyos coeficientes pertenecen al anillo Z_q . El conjunto de los polinomios cuyos coeficientes pertenecen a Z_q se denota como $Z_q[x]$. Se dice que ML-KEM utiliza el anillo $R_q = Z_q[x]/(x^{256} + 1)$, esto quiere decir que los elementos de R_q son polinomios de grado hasta 255 y sus coeficientes son módulo q o 3329. Cuando una operación entre dos polinomios da lugar a coeficientes o grados superiores a los «rangos» comentados, estos se mantienen en dichos «rangos» mediante las reducciones, por ejemplo, $5x^{257} = 5xx^{256} = -5x$ en el caso de los grados o, en el caso de los coeficientes, $671 \equiv 4000 \text{ mod } 3329$.

Módulo R_q^k

Las operaciones que componen ML-KEM, además de emplear polinomios individuales, también emplean vectores de polinomios. El conjunto R_q^k representa vectores de longitud k donde cada componente es un polinomio de R_q . Por tanto, un elemento de R_q^k es una columna de k polinomios, cada uno con 256 coeficientes módulo q . El parámetro k depende del nivel de seguridad seleccionado, a mayor k , mayor dimensión de los vectores y matrices internas, mayor coste computacional y mayor nivel de seguridad. Los diferentes niveles de seguridad tienen asociadas cinco constantes, siendo k la principal, como se puede ver en la Tabla 5.

Tabla 5: Parámetros de ML-KEM según los distintos niveles de seguridad.

	n	q	k	η_1	η_2	d_u	d_v	nivel de seguridad requerido del <i>Random Bit Generator (RBG)</i> (bits)
ML-KEM-512	256	3329	2	3	2	10	4	128
ML-KEM-768	256	3329	3	2	2	10	4	192
ML-KEM-1024	256	3329	4	2	2	11	5	256

Problemas basados en retículos

La seguridad de ML-KEM se fundamenta en el problema [MLWE](#). Se parte de una matriz A con elementos en R_q , un vector «pequeño»⁶ s , y un vector de error «pequeño» e . A partir de ellos se calcula: $t = As + e$. El problema consiste en recuperar s conociendo únicamente A y t . Si el error e no existiera, el sistema podría resolverse mediante álgebra lineal. Sin embargo, la introducción del vector ruido dificulta el problema hasta hacerlo inviable.

La variante *decisional*, [Decision Module Learning With Errors \(D-MLWE\)](#), no busca recuperar el secreto, sino distinguir si un valor ha sido generado como $As + e$ o si es un valor aleatorio. Esta propiedad es fundamental para que los «datos públicos» (A, t) no revelen información útil que permita recuperar el secreto. Aunque esta descripción se expresa mediante álgebra sobre polinomios, [ML-KEM](#) pertenece a la familia de criptografía basada en retículos porque existen reducciones formales entre estos problemas y problemas sobre retículos. Además, los ataques contra este tipo de esquemas suelen formularse mediante técnicas de teoría de retículos [\[68\]](#).

5.3.2. Operaciones principales de los KEM con ML-KEM

A continuación, se detallan de forma simplificada los tres procesos principales de [ML-KEM](#) de modo que sea aparente la relación entre [ML-KEM](#) y el problema [MLWE](#). El estándar completo incluye más pasos, como la generación de semillas, la generación de A , muestreos, procesos de compresión/descompresión, funciones *hash*, entre otros.

Generación de claves

El proceso de generación de claves genera el par de claves, encapsulado (pública) y desencapsulado (privada). Para ello, se selecciona una matriz pública A , un vector secreto s y vector de error e , ambos «pequeños»:

$$A \in R_q^{k \times k} \quad s, e \in R_q^k \quad 1.$$

A partir de estos valores se calcula $t = As + e$, la clave de encapsulado es (A, t) y la clave de desencapsulado es s .

Encapsulado

El encapsulado utiliza la clave pública y produce el cifrado c junto con un secreto compartido. Los algoritmos internos de [ML-KEM](#) generan otro vector «pequeño» de ruido e_1 y realizan las siguientes operaciones con la matriz A y la información contenida en la clave pública, donde r, e_1

⁶Se entiende por polinomio «pequeño» aquel cuyos coeficientes son menores que η , donde η es un entero positivo «pequeño» en comparación con $q/2$. S_η es el conjunto de todos los polinomios «pequeños». Por ejemplo: si $q = 31$, entonces $1 + 30x + 29x^2 + x^4 + 2x^5 \in S_2$. Debido a que el rango de los coeficientes tras realizar mod 31 (simétrico) es $[-2, 2]$.

y e_2 son vectores «pequeños», y m representa el valor interno asociado al secreto que se encapsula. El cifrado está compuesto por (u, v) , es decir, $c = (u, v)$.

$$u = A^T r + e_1 \quad v = t^T r + e_2 + m \quad 2.$$

Desencapsulado

En el proceso de desencapsulado se utiliza la clave de desencapsulado y el cifrado para obtener algo similar al mensaje que fue cifrado. El resultado de la operación no es m . En su lugar, es m más algo de ruido. El paso de m' a m se puede entender como un redondeo hacia valores válidos dentro del espacio reticular (Figura 18) y es la razón por la que es posible fallar al recuperar el mensaje si el ruido es superior al tolerable.

$$m' = v - s^T u \quad 3.$$

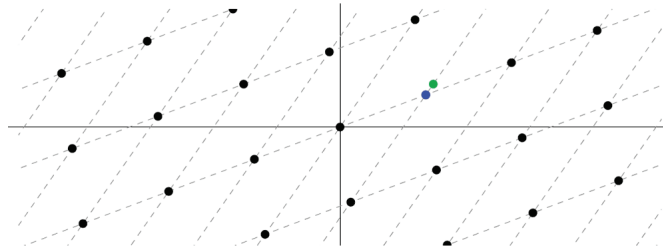


Figura 18: Problema MLWE en términos de retículos. El mensaje con errores (m') se puede entender como el punto verde, cercano a un punto válido [69].

5.3.3. Operaciones internas relevantes

Hasta este punto, la descripción que se ha presentado de ML-KEM oculta el conjunto de operaciones internas que concentran las dificultades a la hora de implementar dicho KEM. Estas operaciones internas se pueden agrupar en tres categorías: la compresión y serialización de la información, la aritmética polinómica y la generación de aleatoriedad. Como se verá en el estado del arte, Sección 5.4, las implementaciones de ML-KEM se presentan en torno a estas tres áreas. A continuación, se presenta el contexto necesario.

Muestreo y compresión

Los polinomios aleatorios que emplea ML-KEM se derivan de semillas mediante algoritmos de la familia SHA3, recogidos en el estándar FIPS 202: las funciones *hash* SHA3-512, SHA3-256 y las funciones de salida variable SHAKE128 y SHAKE256. Es por esto que el módulo SHA3 se utiliza de forma intensiva en las tres operaciones del KEM. A partir de los datos pseudoaleatorios producidos por estas funciones se realizan dos tipos de muestreo: uniforme por rechazo y binomial centrado.

- **Uniforme por rechazo:** de la salida del SHAKE128 se utilizan aquellos valores menores que q y se descarta el resto. Este tipo de muestreo se utiliza para la generación de la matriz A .
- **Binomial centrado:** produce polinomios «pequeños» para s , e , r , e_1 y e_2 con coeficientes acotados por los parámetros η_1 y η_2 de la Tabla 5.

Las funciones de compresión y descompresión reducen el tamaño del cifrado descartando los bits menos significativos de cada coeficiente. En este caso, los parámetros d_u y d_v de la [Tabla 5](#) determinan cuántos bits se conservan para las componentes u y v , respectivamente.

NTT

Es posible implementar [ML-KEM](#) sin utilizar la [Number Theoretic Transform \(NTT\)](#). Sin embargo, la multiplicación de polinomios presente en los productos matriz-vector (As , $A^T r$) y vector-vector ($t^T r$, $s^T u$) de las tres operaciones del esquema y su cálculo directo mediante convolución tienen una complejidad $O(n^2)$. La [NTT](#) permite transformar los polinomios a una representación en la que la multiplicación se realiza coeficiente a coeficiente, reduciendo la complejidad total a $O(n \log n)$, de un modo similar a la transformada discreta de Fourier:

$$h = f \cdot g = \text{INTT}(\text{NTT}(f) \circ \text{NTT}(g)) \quad 4.$$

Para realizar las multiplicaciones entre f y g de la ecuación anterior, se transforman los polinomios. Es aquí donde se ve el impacto del parámetro $q = 3329$ de [ML-KEM](#). La consecuencia clave de q es que Z_q contiene una raíz 256-ésima primitiva de la unidad ($\zeta = 17$), pero no una raíz 512-ésima. Es decir, no es posible descomponer los polinomios en conjuntos de 256 valores independientes, sino 128 polinomios de grado uno, formados por los pares de coeficientes de índice par e impar $(\hat{f}_{2i}, \hat{f}_{2i+1})$.

El algoritmo de referencia del estándar computa la [NTT](#) en 7 etapas de 128 operaciones *butterfly* cada una. La *butterfly* es la célula básica de la transformada: toma dos coeficientes y un *twiddle factor*⁷ y produce dos resultados. De forma común se utiliza la configuración Cooley-Tukey para la [NTT](#) y Gentleman-Sande para la [Inverse Number Theoretic Transform \(INTT\)](#).

Organización de memoria

Durante la ejecución de [ML-KEM](#) se consumen y generan multitud de valores intermedios: semillas, la matriz A expandida, los vectores de ruido, los operandos en el dominio de la [NTT](#). Cada polinomio necesita de $12 \text{ bits} \times 256$ coeficientes. Las operaciones manejan vectores de k polinomios con una matriz de $k \times k$, de modo que es necesario prestar atención al acceso y distribución de la memoria.

5.4. Estado del arte de las implementaciones

En las secciones a continuación se presentan las dos implementaciones de [ML-KEM](#) más novedosas y en las que se pueden ver las consideraciones que han de tomarse para una correcta implementación y las áreas que son el foco de atención de las optimizaciones.

5.4.1. «An Area-Time Efficient Hardware Architecture for ML-KEM Post-Quantum Cryptography Standard»

T.-H. Nguyen, T.-K Dang et al. [70] proponen una arquitectura orientada a tener un buen compromiso entre área y tiempo de ejecución, [Figura 19](#). No se centra en optimizar un bloque en particular, sino que presenta mejoras en varias áreas para reducir esperas, limitar el uso de memorias y reutilizar recursos. Dichas mejoras se localizan en el uso particular de los *twiddle factors* en el

⁷Son potencias de $\zeta = 17$ precalculadas, normalmente almacenadas en [Read-only memory \(ROM\)](#).

acelerador NTT, la adaptación del SHA3 para el flujo de ML-KEM, la organización de la memoria y una correcta planificación de todas las operaciones.

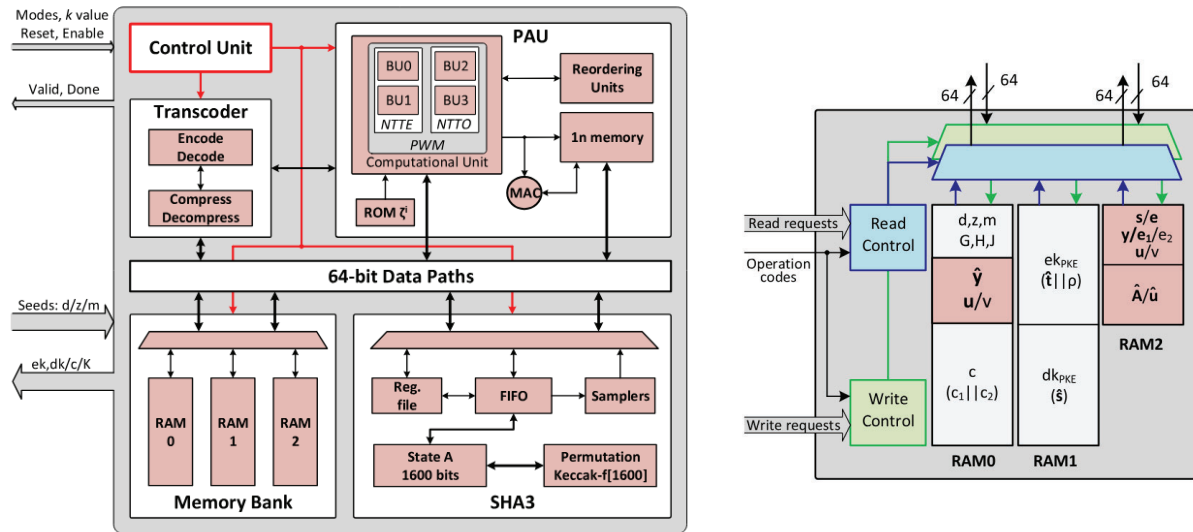


Figura 19: Arquitectura general propuesta y organización de la memoria T.-H. Nguyen, T.-K Dang et al. [70].

Una de las partes clave del diseño es la organización de la memoria, durante la ejecución de ML-KEM se generan y consumen diversos valores intermedios: semillas, claves, cifrados, polinomios, y resultados intermedios. Para evitar el uso de buffers intermedios, se emplea un módulo de memoria particionado (RAM0, RAM1 y RAM2) junto con una memoria interna en la Polynomial Arithmetic Unit (PAU). Aparte de la reducción del uso de recursos, se cumple con la necesidad de ML-KEM de destruir los valores intermedios cuando dejan de tener utilidad.

A cambio de la reducción del uso de recursos tanto en términos de buffers como de memoria, es necesario tener un mayor control de la planificación. Las operaciones han de ejecutarse en orden de modo que no ocurran esperas innecesarias y se cumpla con el acceso secuencial de la memoria.

Por otra parte, el módulo SHA3, Figura 20, está adaptado al flujo de operaciones de ML-KEM. Una vez más, se evita el uso de buffers intermedios que resultan costosos debido al elevado ancho de bits necesario para funciones como el SHAKE128. La propuesta para el SHA3 está centrada en el uso de una FIFO multifunción compuesta por registros de 64 bits y varios modos de operación para la carga de datos de entrada, el guardado del estado interno del módulo y flujos internos. Este diseño permite solapar el muestreo con la siguiente permutación de SHA3, sin perjudicar el flujo de datos. Esto tiene especial relevancia debido a que el SHA3 se emplea tanto para funciones hash como para generar los polinomios aleatorios para las operaciones de generación de claves, encapsulado y desencapsulado.

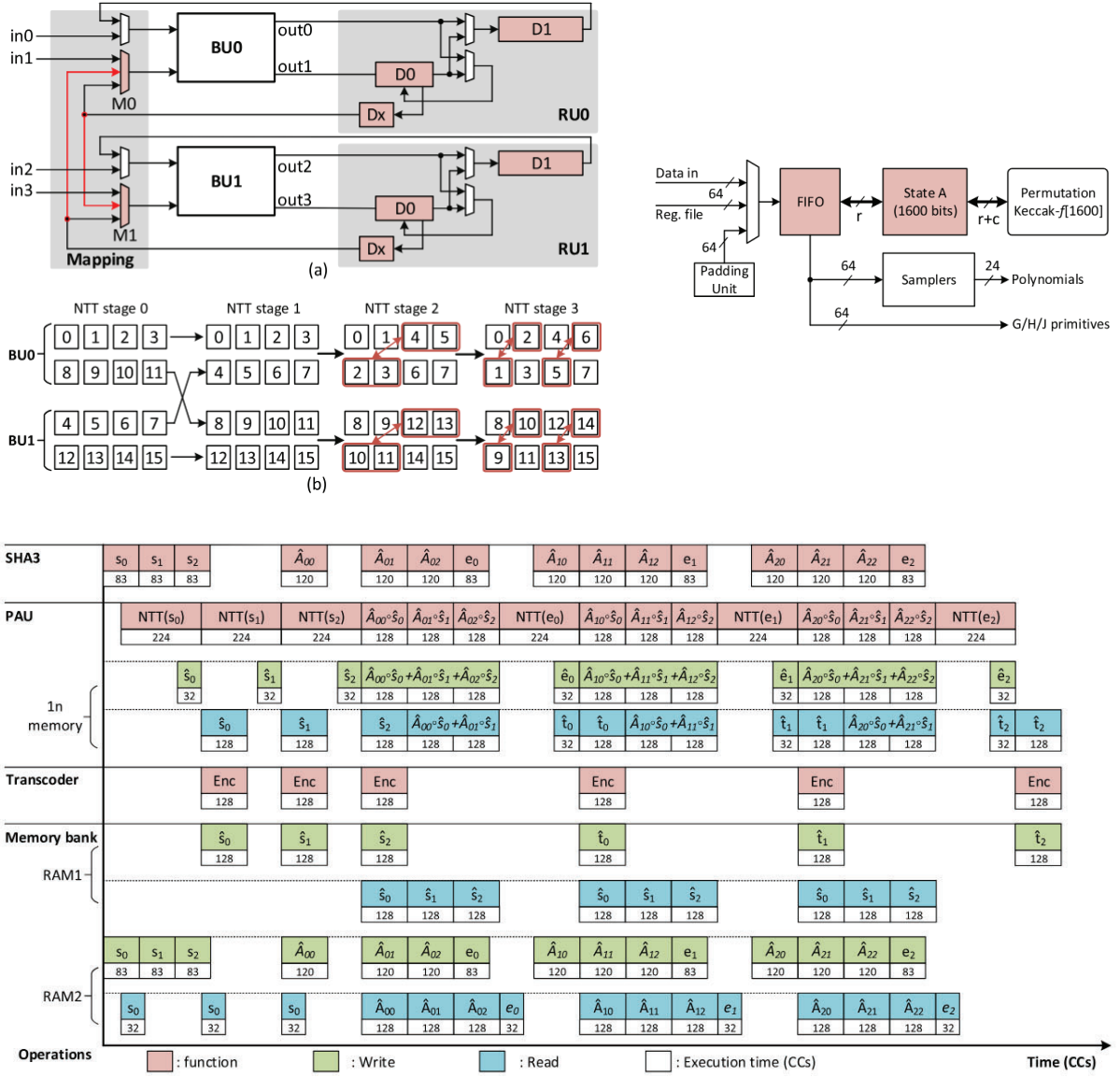


Figura 20: Acelerador NTT, estructura del SHA3 y planificación de operaciones T.-H. Nguyen, T.-K Dang et al. [70].

La PAU está compuesta principalmente por cuatro unidades *butterfly*, dos para los coeficientes impares y dos para los pares, estructura que sigue de forma cercana el estándar de ML-KEM. Este bloque permite realizar la NTT, INTT y la *Point-wise Multiplication* (PWM) e incluye una unidad que permite derivar los *twiddle factors* para las operaciones de INTT y PWM a partir de los *twiddle factors* para la NTT guardados en ROM.

5.4.2. «Efficient Hardware Implementation of the Lightweight CRYSTALS-Kyber»

T.-H. Nguyen, D.-T Dam et al. [71] proponen una arquitectura orientada a dispositivos *Internet of things* (IoT) con restricciones de recursos, con el objetivo principal de reducir el uso de recursos y dar soporte a los tres niveles de seguridad de ML-KEM. El diseño tiene tres aportaciones significativas:

- el módulo de SHA3,

- el módulo NTT (denominado NMI-NTT en este diseño), y
- la planificación de las operaciones [71].

En la Figura 21 se presentan la arquitectura general y la organización de la memoria principal.

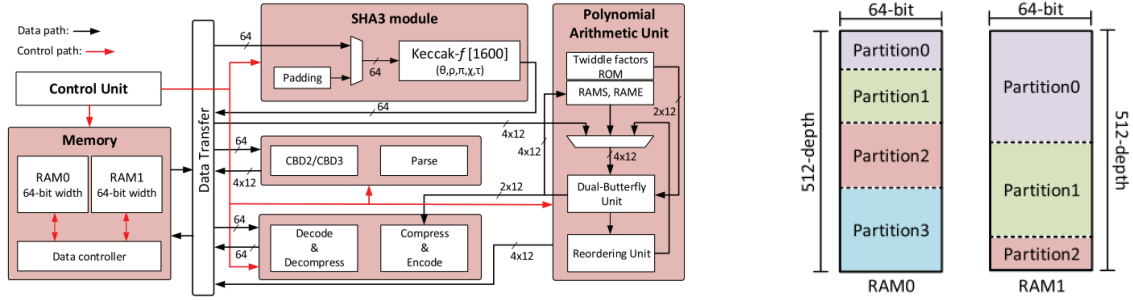


Figura 21: Arquitectura general propuesta y organización de la memoria propuestas por T.-H. Nguyen, D.-T Dam et al. [71].

El diseño en su conjunto opera con un flujo de datos de 64 bits, que coincide con el ancho de palabra del módulo SHA3 y con las longitudes de datos de ML-KEM que son múltiplos de 64 bits. La memoria principal tiene un ancho, una vez más, de 64 bits y una profundidad de 512, dividida en dos particiones RAM0 y RAM1: la primera almacena las semillas, el cifrado y los bits pseudoaleatorios producidos por las funciones XOF y PRF. La segunda contiene las claves de encapsulado y desencapsulado y una sección de *buffer*.

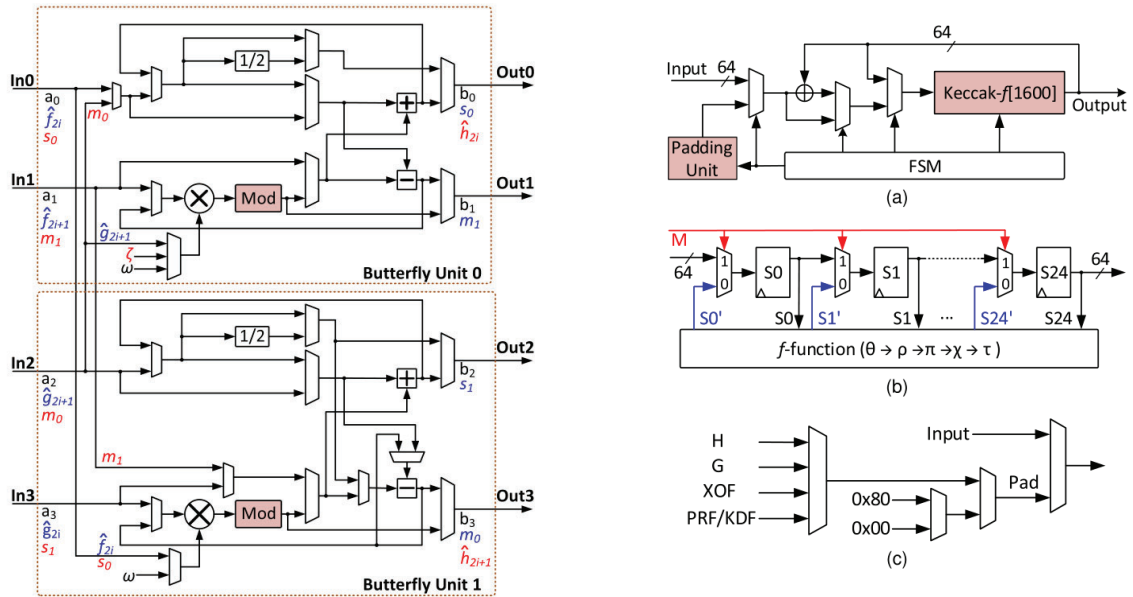


Figura 22: Organización del módulo butterfly doble y SHA3: (a) Construcción de la esponja (b) estructura In-Hash-Out Keccak (c) esquema de padding T.-H. Nguyen, D.-T Dam et al. [71].

El módulo SHA3 guarda el estado de 1600 bits de la permutación Keccak- $f[1600]$ mediante 25 registros de 64 bits a modo de *buffer* multifunción: se recibe el mensaje con el *padding* aplicado, se ejecutan las 24 rondas de permutación y se extraen los bits pseudoaleatorios. De este modo se eliminan *buffers* de entrada y salida para dar lugar a un diseño más compacto. La contrapartida son cálculos de los *hashes* más lentos.

La PAU ejecuta las operaciones de NTT, INTT y PWM. Está compuesta principalmente por la unidad de *butterfly* doble, Figura 22, las memorias internas RAMS y RAME, la ROM para los *twiddle factors* y la unidad de reorganización. Como en el diseño de la sección anterior, se utiliza una configuración unificada Cooley-Tukey (NTT)/Gentleman-Sande (INTT), que evita el preprocesado, el postprocesado y la reordenación por inversión de bits.

La reducción modular se realiza mediante un bloque *Digital Signal Processing (DSP)* para la multiplicación entera y tablas precomputadas para las reducciones. Para la operación PWM, se emplea el algoritmo de Karatsuba, que reduce el cómputo a cuatro multiplicaciones modulares organizadas en dos etapas cuyos resultados intermedios se guardan en RAMS y RAME. La selección entre los modos de operación se realiza controlando los multiplexores del módulo: la PWM se completa en 256 ciclos y las operaciones de NTT/INTT en 448 ciclos.

La aportación principal del trabajo es la adopción de la NMI-NTT, mostrada en la Figura 23. Las NTTs convencionales requieren de memoria adicional para almacenar los coeficientes entre etapas. En este caso, la NMI-NTT combina la unidad de *butterfly* con una unidad de reorganización diseñada a base de retardos que mantiene los coeficientes reordenados y listos para la siguiente etapa. Los coeficientes se mantienen en circulación dentro del módulo durante las transformadas directa e inversa.

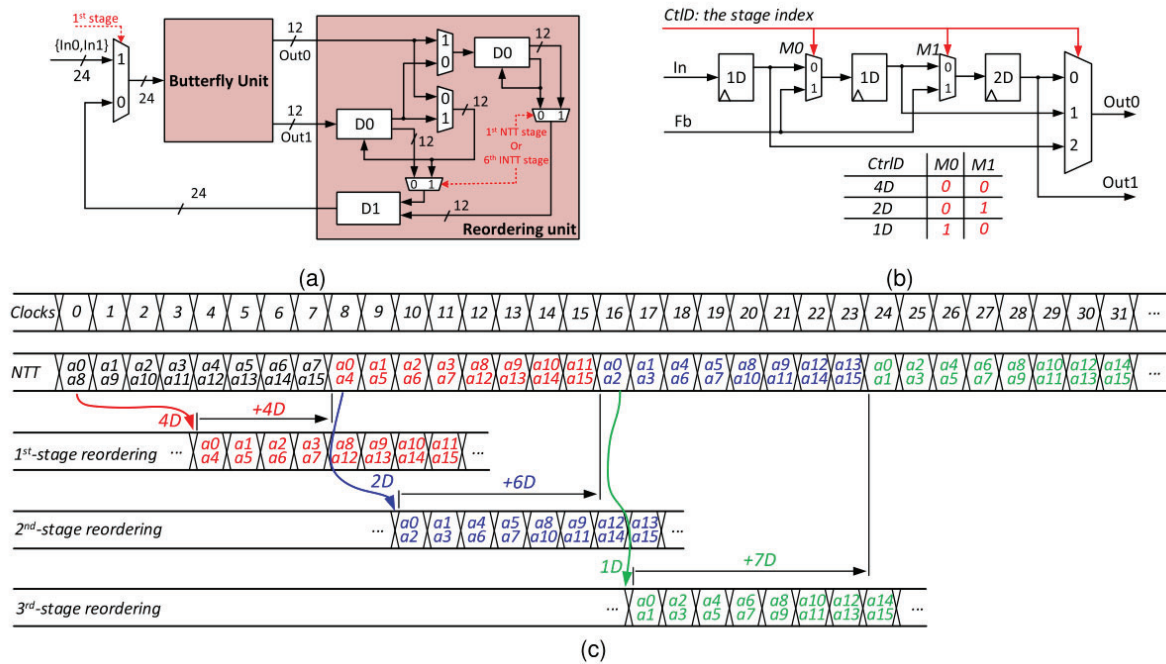


Figura 23: Arquitectura de la NMI-NTT: (a) nivel superior del diseño (b) unidad de retardo segmentada (c) flujo de ejecución de una NTT directa de 16 puntos T.-H. Nguyen, D.-T Dam et al. [71]

La última de las aportaciones consiste en una eficiente planificación de las operaciones, Figura 24, que compensa la baja velocidad de generación del módulo SHA3. Debido a que toda la información se ubica en las secciones de memoria RAM0 y RAM1, las operaciones que hagan uso de dicha memoria han de ejecutarse de forma secuencial, mientras que las operaciones del SHA3 se pueden solapar con la NTT y PWM. Para ello, solo se comienza a generar un segundo polinomio una vez que los coeficientes del primero comienzan a consumirse. De forma similar, las acumulaciones

se ejecutan durante las operaciones de **PWM** o **INTT** y la compresión del cifrado se solapa con la generación de los polinomios siguientes. La latencia resultante de las decisiones de diseño del SHA3 queda compensada por la ejecución de la NMI-NTT.

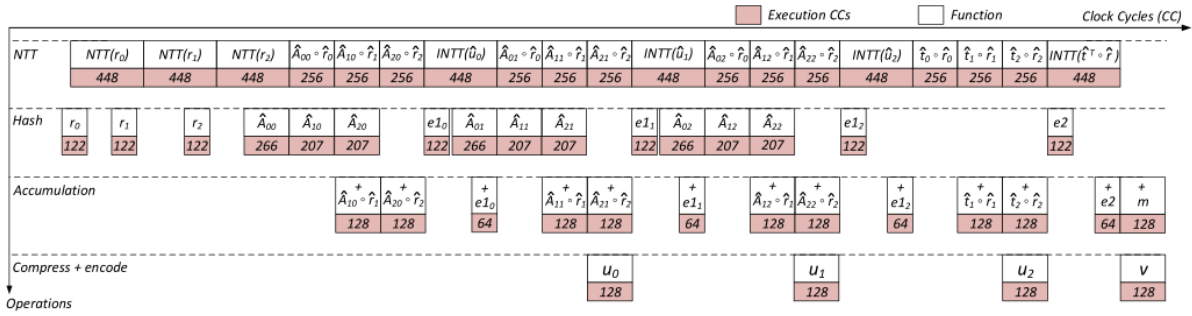


Figura 24: Orden de ejecución y ciclos de reloj de las operaciones principales del encapsulado para el nivel de seguridad 3 T.-H. Nguyen, D.-T Dam et al. [71]

El diseño ha sido implementado sobre una FPGA Artix-7 con una frecuencia objetivo de 185 MHz, con un uso de recursos de 5487 *Look-Up-Table* (LUT), 3426 *Flip Flop* (FF), 1548 SLICES, 3.5 *Block RAM* (BRAM) y 1 DSP. Se completan las operaciones de generación de claves, encapsulado y desencapsulado en 30.3, 38.4 y 49.7 μs , respectivamente (nivel 3 de seguridad).

5.5. Conclusión

De las operaciones presentadas, el «módulo NTT», encargado por lo general de realizar las operaciones de **NTT**, **INTT** y **PWM**, es el candidato idóneo sobre el que desarrollar el *pipeline* de verificación de este **TFM**. Es el núcleo computacional, gran parte de los algoritmos de **ML-KEM** lo utilizan, domina en flujo de datos y tiempo de ejecución. Por ello, se convierte en un foco importante de optimizaciones y, por tanto, es propenso a ser modificado durante el desarrollo.

Además, su estructura jerárquica suele estar compuesta por unidades representativas de los módulos criptográficos y de diferente índole, con unidades aritméticas modulares, unidades *butterfly*, memorias internas y *buffers*, entre otros, junto a una estrecha integración y dependencia del resto de unidades mediante diferentes tipos de interfaces. Todo esto permite aplicar diferentes tipos de verificación para cada contexto. Por último, la naturaleza de sus errores (los casos límite de la reducción modular, el indexado de los *twiddle factors* o la reordenación de coeficientes) puede inducir fallos difíciles de detectar mediante simples pruebas dirigidas.

Por todo ello, en los dos capítulos siguientes la **NTT** y las unidades que la componen serán el **DUT** sobre el que se implementarán las técnicas de verificación seleccionadas y se construirá el *pipeline* de verificación.

Capítulo 6

Diseño del *pipeline* de verificación avanzada

En este capítulo se presenta la implementación del flujo de verificación para criptografía sobre **FPGA** integrado dentro de un *pipeline* unificado. Se parte del diseño a verificar, luego se presenta cómo y por qué se han aplicado los diferentes enfoques de verificación y se finaliza con la unificación del *pipeline* y cómo se pueden ejecutar diferentes conjuntos de tests. Durante el desarrollo han seguido los pasos de la [Figura 25](#).

1. **Diseño:** según la jerarquía del diseño y la naturaleza de los módulos que lo componen se plantean los enfoques de verificación para cada área.
2. **Verificación:** se aplican los diferentes enfoques de verificación a las áreas planteadas.
3. **Pipeline:** se lleva a cabo la ejecución del *pipeline*, desarrollado para que sea posible ejecutarlo en un entorno local o como parte de **CI/CD**, y se obtienen los resultados que permiten guiar el desarrollo del diseño.

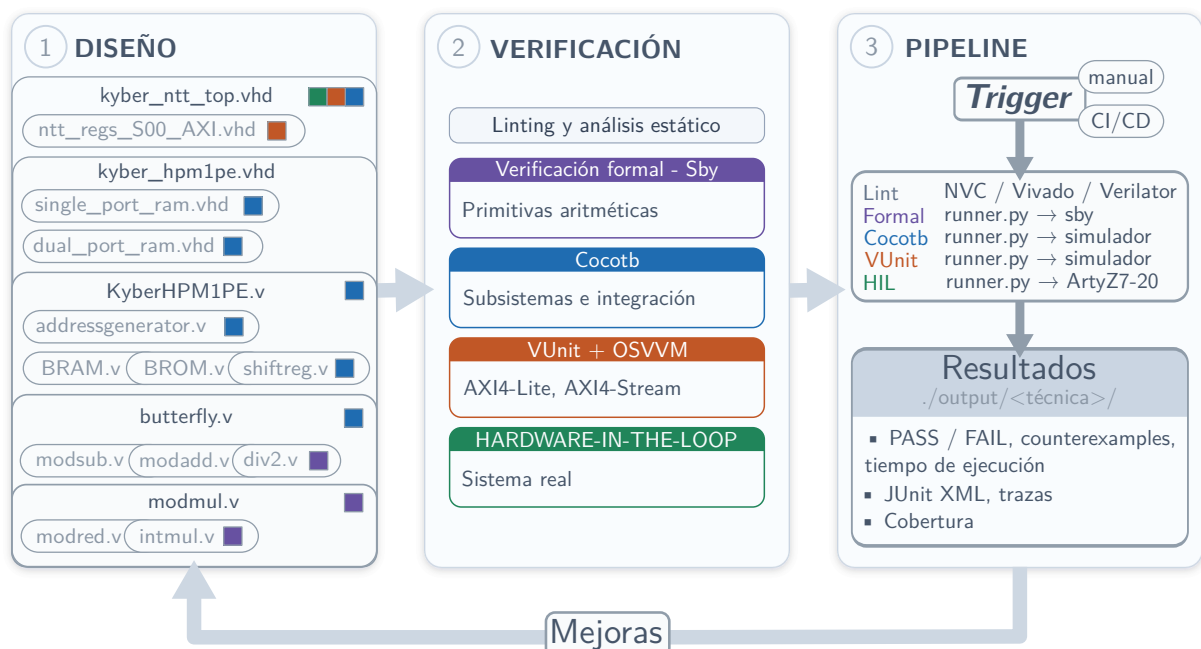


Figura 25: Pipeline de verificación completo.

6.1. Análisis del diseño NTT a verificar

El diseño que se ha utilizado para realizar el *pipeline* de verificación es un «acelerador *hardware NTT*». Este diseño está basado en el trabajo de Yaman et al. [72], implementado en el lenguaje Verilog, sobre el que se ha empleado un *wrapper* VHDL que añade la interfaz AXI4-Stream para transmitir los polinomios o recibir su transformación y la interfaz AXI4-Lite para configurar la operación a realizar según la [Tabla 6](#). Dicho *wrapper* es aportado por la empresa colaboradora, [Arquimea Research Center \(ARC\)](#).

Tabla 6: Registro de control 0 del banco de registros AXI4-Lite (`ntt_regs_S00_AXI.vhd`)

Bits	Señal	Descripción
31:6	–	Sin uso
5	<code>out_mux_sel</code>	No empleada en el trabajo
4	<code>input_mux_sel</code>	No empleada en el trabajo
3:2	–	Sin uso
1:0	<code>opmode</code>	Operación a realizar por el acelerador: <code>00</code> para NTT, <code>01</code> para INTT

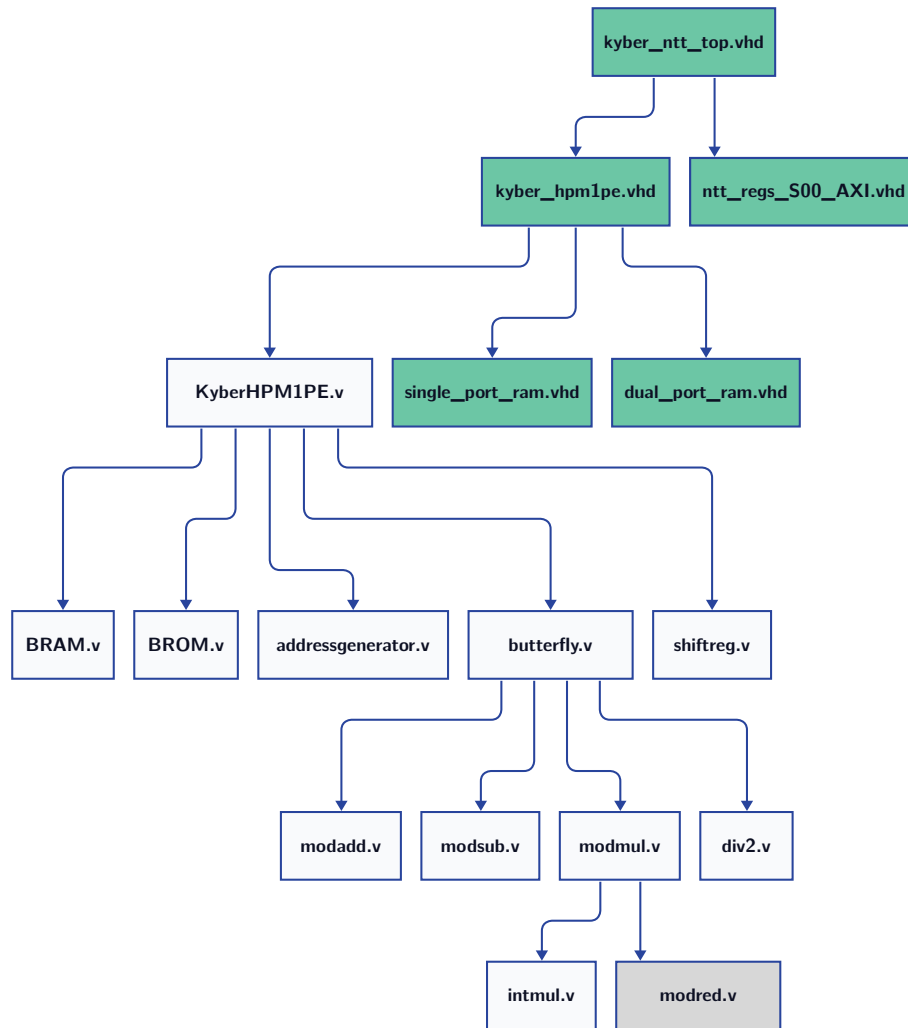


Figura 26: Esquema de los módulos que componen el acelerador hardware, excluyendo los submódulos de `modred.v`.

El trabajo de Yaman et al. presenta tres arquitecturas (*lightweight*, *balanced* y *high-performance*) que implementan las operaciones de NTT, INTT y PWM con 1, 4 y 16 unidades de *butterfly*, respectivamente. La variante *lightweight* presenta una latencia de 904 ciclos para la NTT y la INTT y de 647 ciclos para la PWM. Su uso de recursos sobre una FPGA Artix-7 es de 948 LUTs, 352 FFs, 1 bloque DSP y 2.5 BRAMs para una frecuencia objetivo de 190 MHz. El elemento novedoso de este

trabajo de 2021 es una unidad *butterfly* unificada, [Figura 29](#), con soporte para las tres operaciones que comparte un único bloque *DSP*.

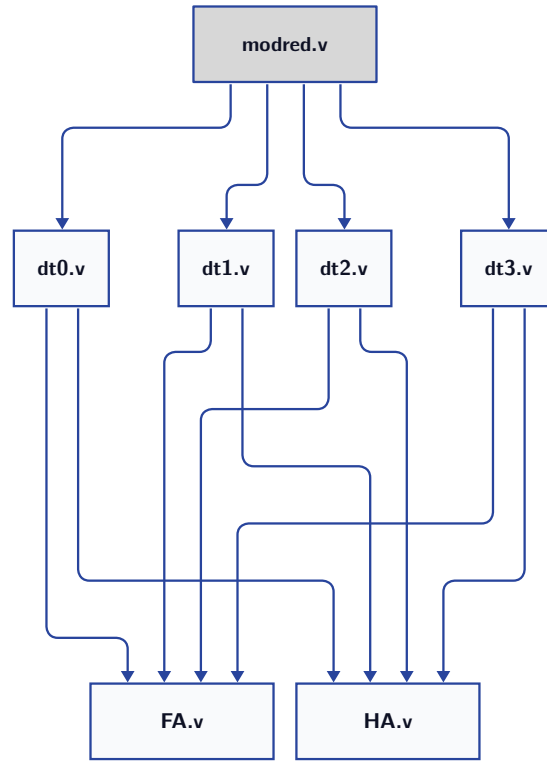


Figura 27: Esquema de los submódulos que componen el módulo `modred.v`.

El diseño sobre el que se ha desarrollado este trabajo se basa en la variante *lightweight* del diseño original y expone las operaciones de *NTT* y *INTT*, excluyendo la *PWM*, mediante las interfaces *AXI* mencionadas. Es un diseño que concreta en un número reducido de módulos la complejidad aritmética y de control del acelerador, como se puede ver en la [Figura 26](#) y en la [Figura 27](#), de modo que resulta un caso de estudio idóneo para contrastar los diferentes tipos de verificación.

6.1.1. Fundamento algorítmico

El diseño adopta la variante de la *NTT* propuesta por [73], donde la transformada numérica se computa en 7 etapas de 128 operaciones *butterfly* y los polinomios se descomponen en 128 polinomios de grado uno, como se detalló en el [Capítulo 5](#). La *NTT* directa emplea la configuración *Cooley Tukey (CT)* que recibe los coeficientes en orden natural y los entrega con el orden de bits invertido. La *INTT* emplea la configuración *Gentleman-Sande (GS)*, con la ordenación inversa. El diseño utiliza el factor de escala $128^{-1} = 2^{-7}$ de la *INTT* dentro de las etapas *GS* mediante divisiones sucesivas [74] de modo que para modelar esta operación es necesario reproducir este escalado distribuido.

6.1.2. Arquitectura hardware

A continuación, se describe la arquitectura del diseño a verificar incluyendo los bloques más significativos, siguiendo la organización del propio diseño.

- **Reducción y aritmética modular (`modred.v`):** es una unidad de reducción modular combinacional, especializada para el número primo $q = 3329 = 2^{12} - 2^9 - 2^8 + 1$. Recibe

un número de hasta 24 bits de una multiplicación entera $12 \cdot 12$ y se calcula $\text{mod } 3329$ de forma eficiente. Los bits que están por encima de la posición 12 se «reescriben» como bits por debajo de 12 de forma recursiva (*folding*) hasta que no se producen más acarreo que den lugar a bits ≥ 12 , según la [Ecuación 5](#). Una vez se completa este proceso, se utiliza el árbol de Dadda para realizar la suma paralela de los términos parciales que resultan del proceso. A partir del resultado del árbol de Dadda se calculan cuatro candidatos: $(C + S + q)$, $(C + S)$, $(C + S - q)$, $(C + S - 2q)$ y se elige el que queda en el rango $[0, q)$. Mediante la reducción modular, `modred.v` y el multiplicador de enteros `intmul.v` se implementa la multiplicación modular, `modmul.v`. La suma, la resta y la división modular entre dos se implementan de forma independiente.

$$2^{12} \equiv 2^9 + 2^8 - 1 \pmod{3329} \quad 5.$$

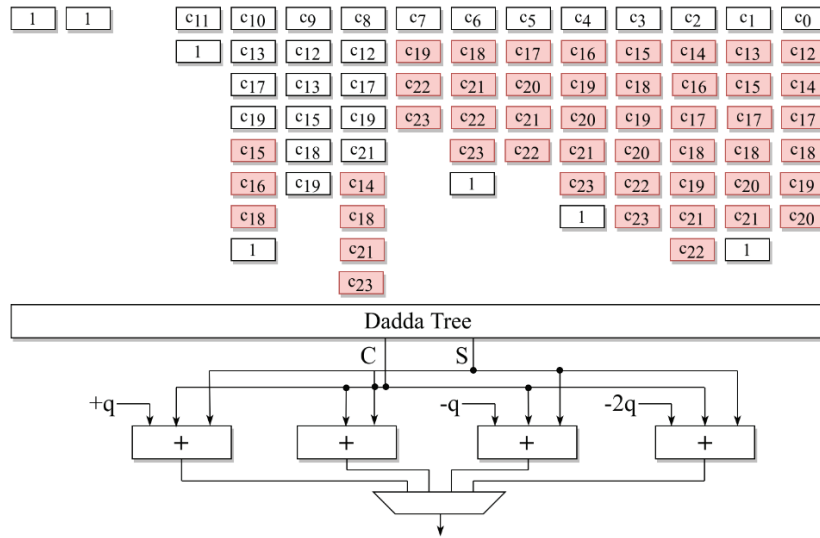


Figura 28: Unidad de reducción modular, `modred.v` [72].

- **Butterfly unificada (`butterfly.v`):** integra un multiplicador modular, un sumador y un restador modulares, además de los multiplexores de configuración y registros de segmentación (Figura 29). Es la unidad que posibilita realizar la NTT que requiere la configuración CT para las operaciones $A + B \cdot \omega \pmod{q}$ y $A - B \cdot \omega \pmod{q}$ y la INTT que requiere la configuración GS para las operaciones $A + B \pmod{q}$ y $(A - B) \cdot \omega \pmod{q}$ con una latencia de 3 ciclos en ambos casos. Este módulo presenta tres optimizaciones frente a trabajos anteriores:
 - Se utiliza un único multiplicador para ambas configuraciones (CT y GS).
 - Para evitar la normalización final de la INTT (multiplicar los 256 coeficientes resultantes por n^{-1}) se divide entre 2 en el interior de cada etapa.
 - Es capaz de configurarse para realizar la PWM, el producto de dos polinomios de grado 1, mediante 5 ciclos de operaciones *multiply-accumulate*.

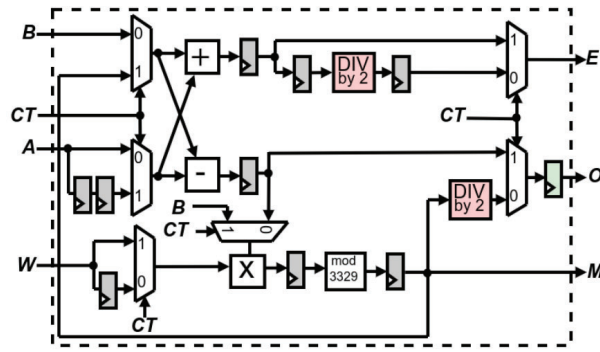


Figura 29: Unidad de butterfly, `butterfly.v` [72].

- **Memoria y generación de direcciones:** el elemento de proceso (`KyberHPM1PE.v`) está compuesto por 1 módulo *butterfly*, 4 *BRAMs* de doble puerto (`BRAM.v`) y 1 *Block ROM* (*BROM*) (`BROM.v`), además de la lógica de control. Las *BRAMs* almacenan los polinomios de entrada y salida y la *BROM* almacena los *twiddle factors*. El generador de direcciones (`addressgenerator.v`) es el encargado del patrón de acceso alternado entre las mitades de las *BRAMs* que permite equilibrar los accesos a memoria para permitir 1 operación *butterfly* por ciclo. Para lograr la latencia de 1 ciclo se organiza la memoria de modo que hasta dos polinomios (2 para la *PWM* y 1 para la *NTT/INTT*) se ubican en las *BRAMs* 1-4 y el resultante polinomio se ubica en las *BRAMs* 1-2 siguiendo la distribución de la [Tabla 7](#).

Tabla 7: Organización de la memoria

BRAM	Polinomio	Coefficientes
1	<i>a</i> (parte baja)	<i>a</i> [0...127]
2	<i>a</i> (parte alta)	<i>a</i> [128...255]
3	<i>b</i> (parte baja)	<i>b</i> [0...127]
4	<i>b</i> (parte alta)	<i>b</i> [128...255]

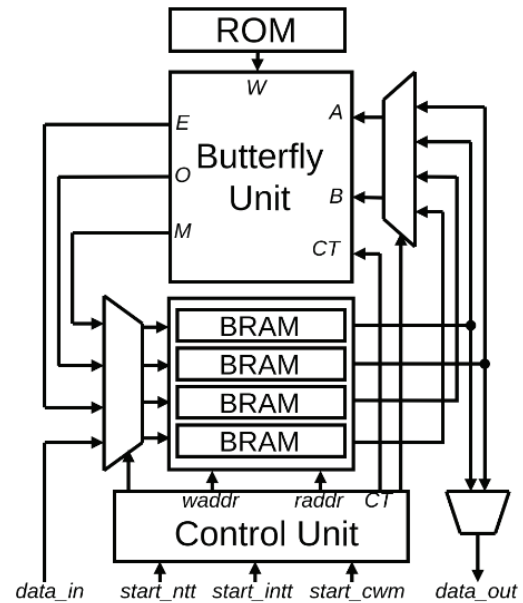


Figura 30: Organización del elemento de proceso [72].

- **Control, subsistema y wrapper.** La unidad de control organiza las 7 etapas de la transformada, gobierna los multiplexores de carga y lectura de los polinomios y la gestión del *pipeline* de operaciones. Sobre el núcleo (`KyberHPM1PE`), descrito íntegramente en Verilog, se ubica el *wrapper* escrito en *VHDL*, que integra el elemento de proceso (`kyber_hpm1pe.vhd`) con sus memorias (`single_port_ram.vhd`, `dual_port_ram.vhd`) y

las interfaces AXI4-Lite y AXI4-Stream y da lugar a la integración a nivel de sistema (`kyber_ntt_top.vhd`). El diseño resultante es, por tanto, de lenguaje mixto: un núcleo Verilog envuelto por una capa VHDL, lo que impone el entorno de verificación con soporte para ambos lenguajes.

6.2. Verificación formal de la aritmética modular

Como se mencionó en el [Capítulo 2](#) y el [Capítulo 3](#), la verificación formal es apropiada en bloques del diseño con baja latencia y baja complejidad aritmética. Pese a que este segundo punto puede no aplicarse a todos los bloques verificados con `Sby`, ha sido posible plantear modelos compactos que describen dichos bloques, que es el principal reto con los módulos aritméticos.

El foco de la verificación formal utilizada en este trabajo se centra en las operaciones aritméticas sobre las que se construyen el bloque *butterfly* y partes del resto del acelerador: suma, resta, multiplicación y reducción módulo ($q = 3329$), división modular entre 2 y multiplicación entera.

Se ha tomado la decisión de no verificar los submódulos aritméticos de los que dependen los anteriores (`dt0.v`, `dt1.v`, `dt2.v`, `dt3.v`, `FA.v` y `HA.v`), debido a que el comportamiento de estos módulos queda restringido al cono lógico de los módulos que las emplean y, por tanto, se aportan garantías sobre su funcionamiento mediante los módulos que los emplean.

6.2.1. Tests con Sby

A continuación se presenta el planteamiento de la verificación formal de cada uno de los módulos, empezando por aquellos que son más representativos del conjunto.

La verificación de cada módulo se organiza en diferentes directorios, cada uno contiene dos ficheros: el fichero `.sby` que describe la tarea de verificación formal ([Código 2](#)) y un fichero de SystemVerilog que instancia el DUT, declara las entradas simbólicas⁸, formula las suposiciones en relación con la entrada y contiene las aserciones. Si se encuentra un caso en el que no se cumplan las suposiciones o aserciones, se genera una traza con el caso. Todas las pruebas emplean el motor `smtbmc` en modo `bmc` con el solucionador Z3 y activan la generación de trazas en formato FST.

- Motor `smtbmc`: en modo `bmc`, *Bounded Model Checking (BMC)*, es el encargado de convertir el diseño y las propiedades especificadas para una profundidad o *depth* determinada a su representación matemática (*SMT*) que los solucionadores pueden interpretar.
- Solucionador Z3: los solucionadores son herramientas que analizan las fórmulas *SMT* generadas por `smtbmc` y determinan si existe una combinación de entradas y estados que no cumpla con las condiciones establecidas, por ejemplo, que un número se encuentre en un determinado rango. Z3 es el solucionador por defecto.

```
1 [options]
2 mode bmc
3 depth 1
4 fst on
5
6 [engines]
7 smtbmc z3
8
9 [script] ini
```

⁸Las entradas simbólicas (`anyseq` o `anyconst`) indican al solucionador que una determinada señal o registro no está limitado a un valor concreto, es decir, la prueba se realiza con todos los valores posibles.

```

10 read -formal HA.v
11 read -formal FA.v
12 read -formal dt0.v
13 read -formal dt1.v
14 read -formal dt2.v
15 read -formal dt3.v
16 read -formal modred.v
17 read -formal -sv modred_formal.sv
18 prep -top modred_formal
19
20 [files]
21 modred_formal.sv
22 ../../src/hdl/HA.v
23 ../../src/hdl/FA.v
24 ../../src/hdl/dt0.v
25 ../../src/hdl/dt1.v
26 ../../src/hdl/dt2.v
27 ../../src/hdl/dt3.v
28 ../../src/hdl/modred.v

```

Código 2: Fichero `.sby` empleado en la verificación de `modred.v`.

Reducción modular

La verificación de `modred.v` es la más representativa del conjunto por su complejidad y diferencia entre su especificación e implementación. La operación que realiza puede describirse como una «reducción módulo q », mientras que la implementación es más compleja, como se ha mencionado en la [Sección 6.1](#).

```

1  module modred_formal;
2      localparam [23:0] KYBER_Q    = 24'd3329;
3      localparam [23:0] MAX_PRODUCT = 24'd11075584;
4
5      (* anyconst *) reg [23:0] C;
6
7      wire [11:0] R;
8      wire [23:0] expected;
9
10     modred dut (
11         .C(C),
12         .R(R)
13     );
14
15     assign expected = C % KYBER_Q;
16
17     always @* begin
18         assume(C <= MAX_PRODUCT);
19
20         assert({12'd0, R} == expected);
21         assert({12'd0, R} < KYBER_Q);
22     end
23 endmodule

```

Código 3: Fichero `.sv` que contiene la verificación de `modred.v`.

Como se puede ver en el [Código 3](#), la entrada `C` es simbólica y se restringe al intervalo $[0, (q - 1)^2]$. Sobre la salida `R` se comprueban dos condiciones: que sea congruente con `C` módulo q y que permanezca dentro del rango $[0, q]$.

Mantener ambas propiedades separadas permite distinguir entre un error en el valor modular y la obtención de un representante no comprendido en el rango canónico. Al permanecer simbólica la entrada y comprobarse todas las posibilidades, la prueba incluye los extremos y los valores próximos a los múltiplos de q .

Multipliación modular segmentada

La verificación de `modmul.v` introduce un matiz temporal y combina `intmul.v` y `modred.v`. Se registra el resultado, de modo que la salida observada en un ciclo debe compararse con los operandos capturados en el ciclo anterior.

```
1 module modmul_formal (input wire clk);
2     localparam [23:0] KYBER_Q = 24'd3329;
3
4     (* anyseq *) reg [11:0] A;
5     (* anyseq *) reg [11:0] B;
6
7     reg [11:0] A_q;
8     reg [11:0] B_q;
9
10    wire [11:0] R;
11
12    modmul dut (
13        .clk (clk),
14        .rst (1'b0),
15        .A (A),
16        .B (B),
17        .R (R)
18    );
19
20    wire [23:0] sampled_product = {12'd0, A_q} * {12'd0, B_q};
21    wire [23:0] expected       = sampled_product % KYBER_Q;
22
23    reg loaded = 1'b0;
24    always @(posedge clk) begin
25        A_q <= A;
26        B_q <= B;
27        loaded <= 1'b1;
28    end
29
30    always @* begin
31        assume({12'd0, A} < KYBER_Q);
32        assume({12'd0, B} < KYBER_Q);
33
34        if (loaded) begin
35            assert({12'd0, R} == expected);
36            assert({12'd0, R} < KYBER_Q);
37        end
38    end
39 endmodule
```

Código 4: Fichero `.sv` que contiene la verificación de `modmul.v`.

Como se puede ver en el Código 4, se restringen los operandos a su rango canónico y se establece que el módulo no esté en *reset* y exista una muestra anterior válida. Para cada ciclo se establece que R_{k+1} debe ser congruente con $A_k \cdot B_k$ módulo q y permanecer en el intervalo $[0, q)$. Se comprueba así tanto el valor resultante como la temporalidad del módulo.

Operaciones combinacionales con un patrón común

La verificación de los módulos `modadd.v`, `modsub.v`, `div2.v` e `intmul.v` comparten la misma estructura: las entradas son `anyconst`, se aplican las restricciones y se compara la salida combinacional con la expresión matemática correspondiente, debido a su similitud se presentan agrupadas:

- **Suma y resta modular (`modadd.v` y `modsub.v`):** los operandos se extienden hasta 13 bits antes de realizar la operación. Para la suma se calcula el valor intermedio $A + B$ y para la resta $A + q - B$, para evitar que se produzca un *underflow*. Debido a que ambos operandos están restringidos al intervalo $[0, q)$, para obtener el representante canónico se

realiza una resta en el caso que el valor sea mayor o igual a q . Las aserciones comprueban que la salida coincide con el valor esperado y permanece en el intervalo $[0, q)$.

- **División modular entre 2 (`div2.v`):** se restringe el operando x a su valor canónico y se distinguen entre valores pares e impares. Para valores pares de x , el resultado esperado es $\frac{x}{2}$, para valores impares es $\frac{x+q}{2}$. La salida se compara con los resultados esperados, se comprueba su rango y además se comprueba que al multiplicar la salida por 2 y reducirla módulo q debe recuperarse el valor original.
- **Multipliación entera (`intmul.v`):** en este caso no es una operación modular, por ello, no se restringen los operandos al rango canónico y se considera todo el espacio que resulta de las entradas de 12 bits. El modelo es el producto entero de los operandos extendidos a 24 bits, se verifica que la salida `P` coincide con el valor del modelo.

6.3. Verificación funcional con Cocotb

Cocotb se utiliza como entorno de verificación de propósito general. Los tests se plantean en 3 niveles de granularidad: bloque básico, subsistema y nivel de *top-level*. En el nivel de bloque se verificaron componentes acotados en extensión; en el nivel de subsistema, el núcleo KyberHPM1PE con las transformadas completas; y en el nivel de integración se añade el *wrapper* con las interfaces AXI4-Stream y AXI4-Lite para verificar el conjunto del sistema sin verificar las propias interfaces, objeto de los test con VUnit y OSVVM, [Sección 6.4](#). La idea central con Cocotb es verificar aquellos módulos cuya verificación no se beneficie de los demás enfoques de verificación.

6.3.1. Estructura y reglas de diseño de los tests

Debido a que Cocotb no impone un patrón de diseño rígido, se ha optado por desarrollar los tests siguiendo una serie de reglas que dan lugar a una estructura general a través de todos los ficheros. Dicha estructura general se basa en los siguientes pasos:

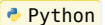
1. generación de estímulos,
2. muestreo, y
3. comparación con el modelo de referencia.

De este modo, se facilita la lectura y mantenimiento de los tests y se siguen las buenas prácticas del *framework*. El patrón tiene 6 pasos que se concretan o adaptan en cada test:

1. Inicialización del reloj, entradas y *reset*.
2. Aplicación de estímulos en el flanco de bajada.
3. Avance hasta el flanco activo.
4. Actualización del modelo de referencia.
5. Espera con `ReadOnly` antes de observar el `DUT`.
6. Comparación y generación de mensajes que incluyen: operación, ciclo, operandos y otra información relevante.

Estas operaciones se encapsulan en funciones como `setup_clock()`, `init_dut()`, `drive_read_assert()`, `tick()` y `compare_outputs()`. En el [Código 5](#) se puede ver un ejemplo de la declaración de este tipo de funciones, y en el [Código 6](#), su aplicación.

```
...  
31 def setup_clock(dut):  
...
```



```

32 dut.clk.value = 0
33 cocotb.start_soon(Clock(dut.clk, CLK_PERIOD_NS, unit="ns").start())
34
35
36 async def tick(dut, cycles=1):
37     for _ in range(cycles):
38         await RisingEdge(dut.clk)
39
40
41 def compare_outputs(dut, expected, context):
42     for name, value in expected.items():
43         got = int(getattr(dut, name).value)
44         assert got == value, f"{context}: {name} got {got}, expected {value}"
45
46
47 async def drive_read_assert(dut, *, drives, expected, context, latency):
48     await FallingEdge(dut.clk)
49     for name, value in drives.items():
50         setattr(dut, name).value = value
51
52     await tick(dut, latency)
53     await ReadOnly()
54     compare_outputs(dut, expected, context)
55
56 ...

```

Código 5: Fichero de tests de la unidad butterfly, funciones genéricas.

```

1 # CT butterfly
2 @cocotb.test()
3 async def test_ct_butterfly_outputs(dut):
4     """CT=1, PWM=0 --> CT butterfly on A,B,W."""
5     await init_dut(dut)
6
7     random.seed(2)
8
9     for _ in range(200):
10         a = random.randrange(Q)
11         b = random.randrange(Q)
12         w = random.randrange(Q)
13         expected_e, expected_o = ct_butterfly(a, b, w)
14
15         await drive_read_assert(
16             dut,
17             drives={"CT": 1, "PWM": 0, "A": a, "B": b, "W": w},
18             expected={"E": expected_e, "O": expected_o},
19             context=f"CT butterfly A={a}, B={b}, W={w}",
20             latency=CT_BUTTERFLY_LATENCY,
21         )

```

Código 6: Fichero de tests de la unidad butterfly, test de la configuración Cooley-Tukey.

6.3.2. Tests con Cocotb

A continuación se presentan los tests más significativos desde el punto de vista del *pipeline* en su conjunto hechos con Cocotb.

Registro de desplazamiento (`shiftreg.v`)

Mediante la clase `deque` (*double-ended queue*) se verifica el registro de desplazamiento. Una cola con dos extremos que permite insertar y extraer elementos por ambos lados. Al instanciar `deque` se crea una cola de longitud indicada por el parámetro `SHIFT`, en cada ciclo se añade un elemento y se compara la salida de la cola con la salida del módulo, reproduciendo así la latencia del módulo. Los estímulos incluyen transiciones de todos los bits, patrones, secuencias de vaciado. Además, estos se han probado con diferentes parámetros `SHIFT`.

Memorias

- RAM (`BRAM.v` y `*_port_ram.vhd`): se comprueban mediante estímulos dirigidos a casos límite y estímulos aleatorios. Los casos dirigidos utilizan las primeras y últimas posiciones de la memoria y valores representativos del ancho de bit. Para la `BRAM.v` se comprueban las latencias y el comportamiento ante lecturas y escrituras simultáneas en la misma dirección. Para ello se hace uso de los modelos `SinglePortRamModel` y `DualPortRamModel`. En el primer caso se modela una memoria en la que la escritura modifica la dirección dada sin actualizar la salida. En el segundo se contempla puertos de entrada y salida independientes, incluyendo accesos simultáneos a direcciones distintas y el caso de lectura y escritura sobre una misma posición.
- ROM (`BROM.v`): la `BROM` almacena los *twiddle factors* utilizados en las operaciones. El modelo genera los valores esperados a partir de la raíz primitiva $\zeta = 17$, las operaciones de inversión de bits y el orden de utilización correspondiente a cada operación. Se recorren las 512 posiciones y se tiene en cuenta que las direcciones 0 a 126 contienen los factores empleados para la `NTT`, las direcciones 127 a 253 los factores empleados en la `INTT` y las direcciones 254 a 381 los utilizados durante la `PWM`, de modo que el valor de cada posición es el esperado.

Unidad *butterfly* (`butterfly.v`)

Para la verificación de la unidad *butterfly* se utilizan dos tipos de técnicas: verificación formal de las primitivas (Sección 6.2) y verificación de los módulos de alto nivel que computan las operaciones de `CT`, `GS` y `PWM`. Los tests se centran en la composición de las operaciones, los multiplexores para cada configuración (`CT`, `GS` y `PWM`) y los registros de segmentación que definen la latencia de cada modo.

El modelo de referencia, `butterfly_model.py`, se construye directamente sobre la definición matemática de las operaciones: las primitivas (`mod_add`, `mod_sub`, `mod_mul` y `mod_div2`) y las funciones `ct_butterfly` y `gs_butterfly` que las componen. La configuración `CT` produce el par $(A + B \cdot \omega, A - B \cdot \omega) \pmod{q}$ y la configuración `GS` produce $(\frac{A+B}{2}, (A - B) \cdot \omega/2) \pmod{q}$. El modelo no reproduce ningún detalle de la implementación (*folding*, árbol de Dadda, estructura de multiplexores).

Los tests siguen el patrón general presentado en el Código 6 y hacen explícita la profundidad del *pipeline* verificada mediante constantes de latencia con nombre (`ADD_SUB_LATENCY`, `MUL_LATENCY`, `CT_BUTTERFLY_LATENCY`, `GS_BUTTERFLY_LATENCY` y `PWM_STEADY_LATENCY`): cada estímulo se aplica en el flanco de bajada y la salida se muestrea tras el número de ciclos declarado, de forma que un cambio en la segmentación del módulo provoca un fallo inmediato del test. Se han definido 5 tests, cada uno de los cuales aplica 200 estímulos aleatorios uniformes en el rango $[0, q)$, uno por cada operación:

- `test_mod_add_sub_outputs`: con `CT=0` se observan las salidas intermedias `ADD` y `SUB`. Se verifica el sumador modular y restador modular.
- `test_mod_mul_output`: con `CT=1` observa la salida `MUL` $= \omega \cdot B \pmod{q}$, el multiplicador modular.
- `test_ct_butterfly_outputs`: con `CT=1` y `PWM=0` comprueba la operación *butterfly* `CT` completa sobre las salidas `E` y `O`.

- `test_gs_butterfly_outputs`: con `CT=0` y `PWM=0` comprueba la operación `GS`, incluyendo las divisiones entre dos de ambas salidas.
- `test_pwm_outputs`: con `CT=1` y `PWM=1` comprueba el modo de realimentación en el que el resultado del multiplicador se redirige al sumador y al restador, que constituye el bloque elemental de las operaciones *multiply-accumulate* de la `PWM`. Ambos operandos del sumador y restador toman el mismo valor P , Ecuación 6.

$$\begin{aligned} P &= \omega \cdot B(\bmod q) \\ E &= P + P(\bmod q) \\ O &= P - P \equiv 0(\bmod q) \end{aligned} \quad 6.$$

Generador de direcciones (`addressgenerator.v`)

El generador de direcciones es el módulo de control principal del acelerador está compuesto por una *Finite State Machine* (FSM) con cuatro estados (reposo, `NTT`, `PWM` y `INTT`) que produce la planificación completa de cada operación que requiere de:

- Direcciones de lectura y escritura de las `BRAMs`
- Habilitaciones de escritura
- Señales de selección de banco que materializan el patrón de acceso alternado descrito en la Sección 6.1
- Señales de configuración de la unidad *butterfly* (`CT/GS` y `PWM`)
- Dirección de la `BROM` con los *twiddle factors*.

Además, el módulo retrasa internamente cada señal entre 1 y 6 ciclos mediante registros de desplazamiento (`shiftreg.v`) para alinearla con la latencia del *pipeline* de la unidad *butterfly*, de modo que las escrituras se producen cuando el resultado correspondiente está disponible.

Es apto para su verificación con Cocotb, debido a que presenta una latencia considerable y no contiene aritmética justifique verificación formal. Su salida es una secuencia determinista de cientos de ciclos (7 etapas de 128 operaciones para la `NTT` y la `INTT`, 128 pares de coeficientes con 5 fases para la `PWM`) en la que un fallo de temporización invalida el resultado de la operación en curso. Los tests comparan ciclo a ciclo el modelo de referencia. Se comprueba el estado de las 17 salidas del `DUT` de forma que, si se produce una divergencia, se presenta por pantalla.

El modelo, `addressgenerator_model.py`, es una descripción de la planificación. Cada ciclo activo se describe como una tupla (operación, etapa, iteración) a partir de la cual se derivan las direcciones según la estructura de la etapa (distancia entre operandos de la unidad *butterfly* y agrupación de coeficientes), mientras que los índices de *twiddle factors* se obtienen del mismo modelo compartido que emplea el test de la `BROM`.

Se han definido tres tests, uno por operación: `test_addressgenerator_fntt_schedule`, `test_addressgenerator_intt_schedule` y `test_addressgenerator_pwm2_schedule`. Cada test aplica el *reset*, indica el inicio de una de las operaciones y recorre la planificación completa. La comparación se extiende más allá de los ciclos activos, para comprobar el vaciado de los desplazadores internos que generan retardos (`shiftreg.v`) y el retorno al estado de reposo. En el Código 7 se presenta el desarrollo del test para la planificación de la `NTT` `test_addressgenerator_fntt_schedule`.

```

...
72 async def run_operation(dut, model, start_name, model_start_name, operation):
73     starts = {"start_fntt": 0, "start_pwm2": 0, "start_intt": 0}
74     starts[model_start_name] = 1
75
76     await drive_read_assert(
77         dut,
78         model,
79         f"{start_name} start cycle",
80         model_inputs=starts,
81         drives={start_name: 1},
82         clear_after_edge=(start_name,),
83     )
84
85     for cycle in range(operation_check_cycles(operation)):
86         await drive_read_assert(dut, model, f"{start_name} cycle {cycle}")
87
88
89 @cocotb.test()
90 async def test_addressgenerator_fntt_schedule(dut):
91     model = await init_dut(dut)
92     await run_operation(dut, model, "start_fntt", "start_fntt", Operation.FNTT)
93
...

```

Código 7: Comparación en lockstep de la planificación completa de una operación contra el modelo de referencia.

Núcleo NTT (KyberHPM1PE.v)

El núcleo `KyberHPM1PE.v` es el primer punto del conjunto de tests en el que se computa la transformada de principio a fin, sin intervención de las interfaces AXI. Los módulos que lo componen son:

- Módulo *butterfly*
- Módulo generador de direcciones
- Módulos de memorias
- Módulo de control: FSM para las operaciones de *load/exec/read*

El modelo (`ntt_model.py`) implementa la transformada, descrita en la Sección 6.1: la NTT directa recorre las 7 etapas y entrega los coeficientes con el orden de bits invertido, y la INTT aplica el factor de escala $2^{-7} = 128^{-1}$ mediante una única multiplicación final, a diferencia de la implementación *hardware* que distribuye la multiplicación final en sucesivas divisiones entre 2.

Al finalizar una operación, el DUT emite un *strobe* de un ciclo de duración en la señal `done` y, durante la lectura, entrega 256 muestras acompañadas de `dout_valid` y de la dirección `addr_dout`. La función `trace_to_logical_poly` reconstruye el polinomio y comprueba que la secuencia de direcciones es la esperada, que no existen coeficientes duplicados ni ausentes y que todos los valores permanecen en el intervalo $[0, q)$. La correspondencia entre el orden físico de lectura y el orden lógico de los coeficientes depende de la operación y se obtiene del mismo modelo que emplean los tests del generador de direcciones (`addressgenerator_model.py`).

Verificación del top (kyber_ntt_top.vhd)

Los test sobre el conjunto `kyber_ntt_top.vhd` permiten verificar la integración del núcleo `KyberHPM1PE.v`, de la sección anterior, con el *wrapper* que contiene las interfaces AXI4-Stream y AXI4-Lite, es decir, verificación a nivel de *top*. El fichero `kyber_ntt_top.vhd` contiene:

- Elemento de proceso (`kyber_hpm1pe.vhd`)

- El banco de registros AXI4-Lite `ntt_regs_S00_AXI.vhd`
- La interfaz AXI4-Stream

El objetivo en este nivel es comprobar que los datos que atraviesan las interfaces reales producen las transformadas correctas, sin verificar el comportamiento de las propias interfaces, que es objeto de los tests con VUnit y OSVVM de la [Sección 6.4](#).

Para interactuar con las interfaces se emplea la extensión `cocotbext-axi` [50], presentada en el [Capítulo 3](#): los polinomios se envían y reciben como tramas de 256 coeficientes mediante los componentes `AxiStreamSource` y `AxiStreamSink` y, en el nivel de sistema, un `AxiLiteMaster` programa la operación a realizar.

Los estímulos son tres polinomios deterministas (el polinomio nulo y dos rampas) y la referencia vuelve a ser el modelo compartido `ntt_model.py`: para la [NTT](#) se comparan las tramas recibidas con la transformada del modelo. Para la [INTT](#) los estímulos se generan aplicando primero la [NTT](#) del modelo y luego se utiliza ese resultado como vector. Al igual que en el nivel de subsistema, el camino inverso verifica con entradas que son transformadas reales. Ante una discrepancia, se presenta por pantalla el número de coeficientes erróneos y los primeros pares (índice, obtenido, esperado), lo que facilita distinguir un error puntual de un error sistemático. La verificación se organiza en dos escalones:

1. `kyber_hpmlpe_axi_ntt` y `kyber_hpmlpe_axi_intt`: se selecciona la operación mediante el puerto `opmode` de forma directa.
2. `kyber_ntt_top_axi_ntt` y `kyber_ntt_top_axi_intt`: la operación se programa escribiendo el registro de control 0 a través del bus AXI4-Lite, se reproduce así el uso que hace el sistema de procesamiento del ZYNQ sobre la placa, sección [Sección 6.5](#).

Tras cada escritura se relee el registro para confirmar el valor programado y se comprueba que las señales de selección de multiplexores (`input_mux_sel`, `out_mux_sel`) y el destino del *stream* (`m_axis_tdest`) permanecen en sus valores por defecto, [Código 8](#).

```

1  async def configure_opmode(dut, axi_master, opmode):
2      await axi_write(axi_master, CONTROL_REG, opmode)
3      got = await axi_read(axi_master, CONTROL_REG)
4      assert (got & 0x3) == opmode
5
6      await ClockCycles(dut.clk, 1)
7      await ReadOnly()
8      assert int(dut.input_mux_sel.value) == 0
9      assert int(dut.out_mux_sel.value) == 0
10     assert int(dut.m_axis_tdest.value) == 0
11
12
13  async def run_polynomial(dut, stream_source, stream_sink, axi_master, opmode, stimulus):
14      await reset_dut(dut)
15      await configure_opmode(dut, axi_master, opmode)
16      await stream_source.send(AxiStreamFrame(stimulus))
17
18      frame = await with_timeout(stream_sink.recv(), 120000, "ns")
19      received = list(frame.tdata)
20      assert len(received) == N
21      return received

```

Código 8: Operación por AXI4-Lite y ejecución de un polinomio a través de las interfaces AXI4-Stream del top.

El último de los tests, `kyber_ntt_top_axi_register_sweep`, tiene un fin distinto. En el sistema real, el puerto AXI4-Lite se utiliza para una única función: escribir el modo de operación en el registro de control 0, que es el tráfico que modelan los dos tests anteriores.

Este test no representa, por tanto, un modelo de uso estándar del bloque, sino que se utiliza para verificar el conexionado del top. En esencia hace pasar patrones de palabra completa complementarios (`0xFFFFFFFF`, `0x00000000`, `0xAAAAAAAA`, `0x55555555`) por los cuatro registros para conmutar cada línea de datos en ambos sentidos. Además, realiza escrituras de un solo byte sobre cada carril de un registro para ejercitar las señales de *strobe* individualmente junto a una lectura de byte no alineada.

Por último, comprueba la decodificación del registro de control, verificando que sus bits 4 y 5 gobiernan las salidas `input_mux_sel` y `out_mux_sel`. Este tráfico cubre las líneas que el uso normal del sistema nunca cambia debido al estado de escritura y lectura de datos, y que constituyen la mayor parte del residuo de cobertura de conmutación que se analiza en la [Sección 6.4](#).

6.4. Verificación de interfaces AXI con VUnit y OSVVM

A continuación se presenta la verificación de las interfaces [AXI](#) del diseño: AXI4-Lite y AXI4-Stream. Existen dos razones por las cuales no se ha optado por verificar estas interfaces con la extensión `cocotbext-axi` empleada en el nivel de integración:

1. Se ha decidido depositar un mayor grado de confianza en los [VCs](#) de [OSVVM](#),
2. La integración de VUnit con [OSVVM](#) forma parte de las labores exploratorias del trabajo en las que se busca encontrar un flujo de trabajo de baja fricción, a partir de un *testbench* básico hasta VUnit y VUnit con [OSVVM](#), para adoptar nociones de verificación en proyectos existentes, como se estableció en el [Capítulo 3](#).

Los tests de esta sección se centran en las interfaces y no en el contenido o validez de las tramas: se verifica el *handshake*, el comportamiento ante *backpressure* y el acceso a los registros. El *testbench* de AXI4-Stream no compara `TDATA` con un polinomio esperado y el *testbench* de AXI4-Lite no comprueba qué hace el resto del diseño con `opmode`. La correcta operación de las transformadas queda garantizada por los tests de Cocotb, la verificación formal y la ejecución [HIL](#) ([Sección 6.5](#)).

6.4.1. Tests con VUnit y OSVVM

Las responsabilidades de VUnit y OSVVM en ambos tests son complementarias:

- VUnit: actúa como *runner*, proporciona las utilidades de selección de casos mediante `run(...)`, las aserciones `check`/`check_equal` y el `_watchdog_`.
- [OSVVM](#): aporta los [VCs](#) que implementan los protocolos: `Axi4LiteManager` para el banco de registros y `AxiStreamTransmitter`/`AxiStreamReceiver` para la interfaz de *stream*.

AXI4-Lite

El *testbench* (`tb_ntt_regs_s00_axi.vhd`) verifica el banco de registros `ntt_regs_S00_AXI.vhd`, que está formado por:

- cuatro registros de 32 bits con un espacio de direcciones de 4 bits,
- el registro de control 0 controla, en el top del diseño:
 - el `opmode` (bits 1:0),

- las señales `input_mux_sel` (bit 4) y `out_mux_sel` (bit 5).

Estas dos últimas no se han utilizado en el desarrollo del trabajo porque son utilizadas en una configuración del acelerador que queda fuera del alcance del trabajo.

El *testbench* emplea el VC `Axi4LiteManager` de la biblioteca AXI4 de OSVVM que actúa como maestro del bus. Se instancia como `axi_manager_map` y tiene dos puertos que enlazan el proceso principal de test con el DUT, como se puede ver en la Figura 31.

- `axibus`: se conecta al *record* de tipo `Axi4LiteRecType` que agrupa los cinco canales del protocolo que a su vez se conecta con la interfaz `s_axi` del banco de registros.
- `transrec`: se conecta al *record* de tipo `AddressBusRecType` que es la interfaz que utiliza el proceso principal de tests para comunicar las transacciones que se han de llevar a cabo.

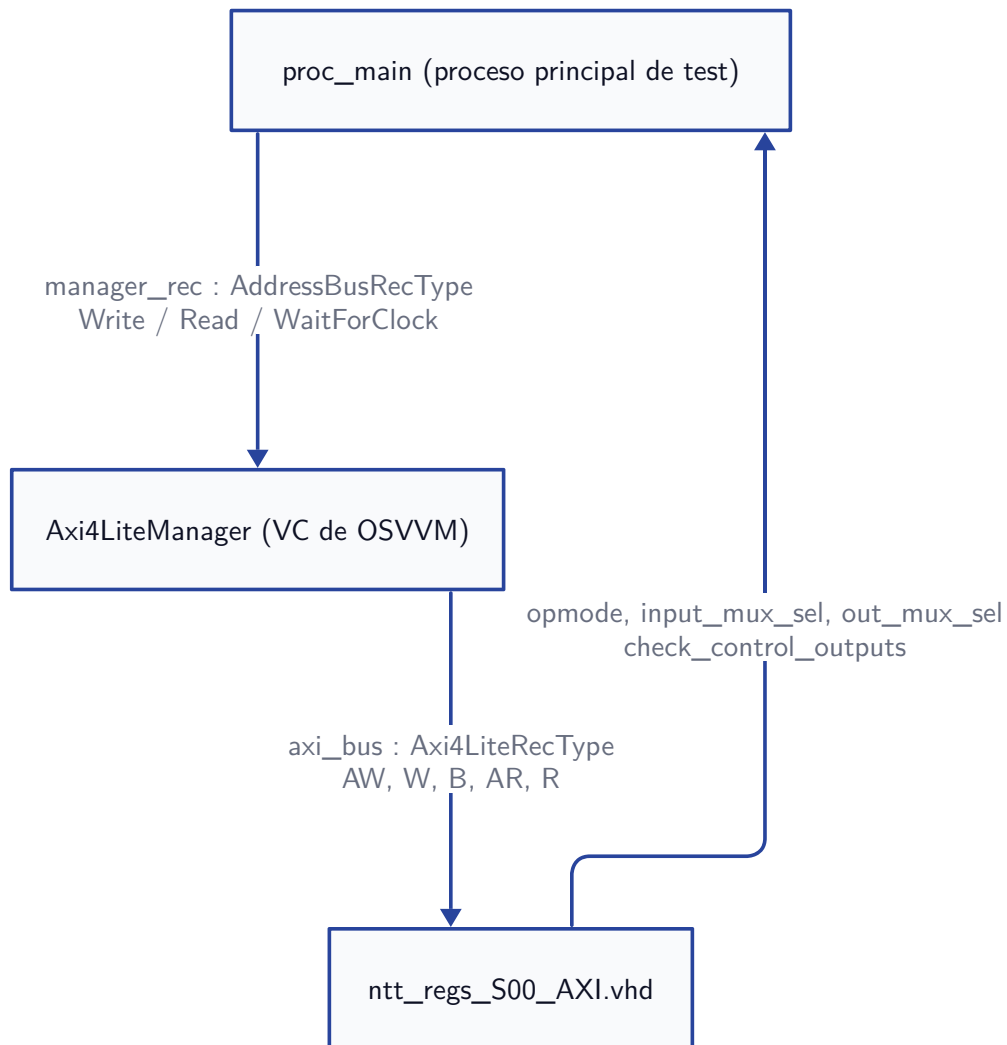


Figura 31: Proceso principal de tests en relación a el DUT y el VC.

De este modo el proceso de test indica las operaciones mediante las transacciones (`Write`, `Read`, `WaitForClock`) y el VC es el responsable de convertir dichas transacciones a niveles de señal

concretos. En el caso de que se desee modificar el comportamiento temporal se configura el **VC** sin necesidad de reescribir estímulos, como se verá en el caso de los tests de *backpressure*.

En base a las transacciones se se definen ciertos procedimientos:

- `write_reg`: escritura del registro.
- `read_reg`: lectura del registro.
- `check_reg`: lectura y comprobación mediante el `check_equal` de VUnit.
- `check_control_outputs`: comprueba que las tres salidas de control decodifican el valor presente en el registro.

Cada uno de los tests está contenido en un procedimiento propio, mediante la construcción `run()` de VUnit se seleccionan los que se han de llevar a cabo. Los 8 tests definidos son:

- **Funcionalidad del *reset*** (`Reset_defaults`): con el **DUT** en *reset* se comprueba que `BVALID`, `RVALID` y `RDATA` permanecen inactivos. Tras liberar el *reset*, los cuatro registros han de estar a cero.
- **Escritura** (`Full_write_readback`, `Data_toggle_patterns`): escrituras de palabras sobre los cuatro registros. Primero con valores representativos para decodificación de las salidas de control, después, con patrones complementarios (`0xFFFFFFFF`, `0x00000000`, `0xAAAAAAAA`, etc.).
- **Strobes al byte** (`Byte_strobes`, `Byte_strobes_all_regs`): escrituras de *bytes* y de media palabra. Se comprueba que se modifican los bytes habilitados por `WSTRB` y que el resto del registro conserva su valor.
- **Desalineado de canales y *backpressure*** (`Write_address_data_skew`, `Write_response_backpressure`, `Read_backpressure`): casos en los que el tráfico se aparta de la secuencia normal de los canales, descritos a continuación en el [Código 9](#).

```
1 -- Address phase before the data phase
2 WriteAddressAsync(manager_rec, f_addr(C_REG1));
3 WaitForClock(manager_rec, 3);
4 WriteDataAsync(manager_rec, f_addr(C_REG1), C_REG1_VALUE);
5 WaitForClock(manager_rec, 6);
6 check_reg(C_REG1, C_REG1_VALUE, "slv_reg1 readback after address-before-data write");
7
8 -- Data phase before address phase
9 WriteDataAsync(manager_rec, f_addr(C_REG2), C_REG2_VALUE);
10 WaitForClock(manager_rec, 3);
11 WriteAddressAsync(manager_rec, f_addr(C_REG2));
12 WaitForClock(manager_rec, 6);
13 check_reg(C_REG2, C_REG2_VALUE, "slv_reg2 readback after data-before-address write");
```

Código 9: Desalineado entre las fases de dirección y datos de la escritura AXI4-Lite, fichero `tb_ntt_regs_s00_axi.vhd`.

Para comprobar el comportamiento ante *backpressure* solo es necesario ajustar la configuración del **VC** con `SetAxi4Options`:

- `WRITE_RESPONSE_READY_BEFORE_VALID` a falso y `WRITE_RESPONSE_READY_DELAY_CYCLES` a 3 ciclos: el *manager* retiene `BREADY` tras cada escritura, de modo que el **DUT** debe mantener `BVALID` y la respuesta estable hasta que se completa el *handshake*.
- `READ_DATA_READY_BEFORE_VALID` y `READ_DATA_READY_DELAY_CYCLES` retienen `RREADY` y comprueban el mismo comportamiento sobre `RVALID` y `RDATA`. En ambos casos las

relecturas posteriores confirman que ninguna escritura se pierde ni se duplica bajo estas condiciones.

AXI4-Stream

El **DUT** de este *testbench* (`tb_kyber_hpm1pe_axi.vhd`) es el subsistema `kyber_hpm1pe.vhd` en el que el modo de operación se establece mediante un puerto directo (sin el uso de AXI4-Lite) y los polinomios de entrada y salida se corresponden con dos AXI4-Stream. El *testbench* instancia dos **VCs** conectados mediante *records* `StreamRecType`:

- `AxiStreamTransmitter`: gobierna la interfaz esclava de entrada (`s_axis`).
- `AxiStreamReceiver`: gobierna el `TREADY` de la interfaz maestra de salida (`m_axis`).

En la entrada, la señal `s_axis_tlast` no está implementada por el **DUT**, por lo que una trama se define como exactamente 256 transferencias. El **VC** transmisor mantiene `TLAST` inactiva (`SetAxiStreamOptions(stream_tx_rec, DEFAULT_LAST, 0)`) así se hace explícito que `s_axis_tlast` no se usa. En la salida, en cambio, `m_axis_tlast` es el delimitador de la trama y se comprueba que permanece baja en las muestras 0 a 254 y se activa exactamente en la última.

En cada test se manejan una o dos tramas mediante la secuencia `run_axi_frame`, compuesta por cuatro fases:

1. envío de la trama de entrada,
2. comprobación de que el esclavo retira `s_axis_tready` mientras procesa la trama,
3. recepción de la trama de salida con la comprobación de `TLAST` descrita y, por último,
4. comprobación de la recuperación tras la trama (no aparecen muestras y `s_axis_tready` vuelve a activarse, dejando el **DUT** listo para la siguiente operación).

El comportamiento temporal de los **VCs** se controla mediante:

- `SetUseRandomDelays` introduce esperas aleatorias para `TVALID` y `TREADY`.
- La operación `RECEIVE_READY_WAIT_FOR_GET`: el receptor mantiene `TREADY` hasta que se consume la transacción, esto permite forzar a que se mantenga la última muestra y observar el comportamiento en ese caso. El **DUT** debe presentar la muestra final con `TLAST` ya activa y mantenerla estable durante varios ciclos de *backpressure* antes del *handshake* definitivo.

Los tests definidos son:

- `Simple_reset`: se comprueban los valores de las señales de ambas interfaces durante y tras el *reset*.
- `ONE_frame`: una trama con tráfico normal, sin esperas en ninguno de los extremos.
- `ONE_frame_random_delays`: una trama con retardos aleatorios de `TVALID` y `TREADY`, que desplaza los *handshakes* de ambas interfaces respecto al caso normal.
- `ONE_frame_output_backpressure_VIA_final_stall`: se fuerza la parada en la última muestra de la trama de salida, el punto en el que coinciden el delimitador `TLAST`, el *backpressure* y el retorno al estado de reposo.
- `TWO_frames_back_to_back`: dos tramas consecutivas sin *reset* intermedio, que comprueba que el acelerador esté en condiciones de aceptar la siguiente.

6.5. Verificación *hardware-in-the-loop* (HIL)

En la verificación **HIL**, a diferencia de los niveles anteriores, el **DUT** deja de ser una descripción **HDL** interpretada por un simulador. El objetivo de esta técnica de verificación no es comprobar la funcionalidad de la transformada, que queda garantizada por el resto de técnicas tratadas con anterioridad, sino cubrir aquellos casos en que realizar los procesos de verificación mediante simulación son costosos por diversos motivos: el comportamiento del diseño tras la síntesis e implementación sobre un sistema real, que opera con los relojes y restricciones temporales propios de un dispositivo físico y que se integra con el resto de la infraestructura necesaria para hacer uso del mismo.

La verificación **HIL** está implementada mediante tres componentes:

1. La **plataforma hardware**: un *block design* de Vivado que integra el acelerador con el sistema de procesado (PS) de Zynq mediante la utilización de un bloque de **Direct Memory Access (DMA)**.
2. La **aplicación C/C++ bare-metal** (`ntt_dma.c`): se ejecuta en el sistema de procesado y realiza una transacción **DMA** por cada vector recibido a través del puerto serie.
3. El **script de HIL** (`send_ntt_vectors.py`): genera los vectores de prueba, realiza la conexión serie y contrasta los resultados con el modelo de referencia.

6.5.1. Plataforma *hardware*

El top del diseño (`kyber_ntt_top.vhd`) se convierte en un bloque **IP** que, a su vez, se utiliza en el diseño de bloques con Vivado IP Integrator, **Figura 32**. Una vez se desarrolla la plataforma en Vivado se utiliza Vitis para programar el **Processing System (PS)** del **SoC** de la Artys Z7-20, **Figura 33**.

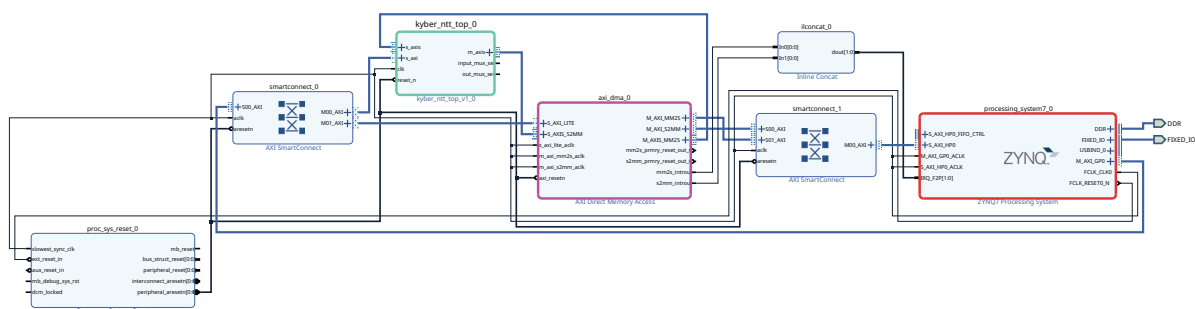


Figura 32: Block design utilizado en la verificación HIL

Los principales componentes del *block design* son el **PS** (rojo), el acelerador (cian) y un **AXI DMA** (magenta), según la **Figura 32**. Además se incluyen el generador de *resets*, el bloque **Inline Concat** que agrupa las interrupciones del **DMA** y dos **SmartConnect**. Las conexiones respecto a los puertos del acelerador son:

- **AXI4-Stream**, flujo de datos o vectores:
 - Lectura (`s_axis`): el canal `M_AXIS_MM2S` del **DMA** lee el polinomio de la memoria **DDR** y lo entrega al acelerador. Debido a que el acelerador no hace uso de la señal `s_axis_tlast`, la operación comienza cuando se han aceptado exactamente 256 muestras.

- Escritura (`m_axis`): la salida del acelerador va al canal `S_AXIS_S2MM` del `DMA`, que escribe el polinomio resultante en la memoria DDR. La señal `m_axis_tlast`, que el acelerador genera al final de la secuencia de 256 muestras, indica el final del paquete.
- AXI4-Lite (`s_axi`), interfaz de control mapeada en la dirección `0x4000_0000` del espacio de direcciones del `PS` y accedido desde `M_AXI_GP0`:
 - Escritura: selecciona la operación a realizar (`NTT` o `INTT`) mediante el registro de control 0, el mismo acceso realizado en simulación.
 - Lectura: tras cada escritura, la aplicación *bare-metal* relee el registro para confirmar el valor programado antes de iniciar las transferencias.

El `DMA` expone una interfaz AXI4-Lite propia mediante la que la aplicación *bare-metal* programa las transferencias de ambos canales.

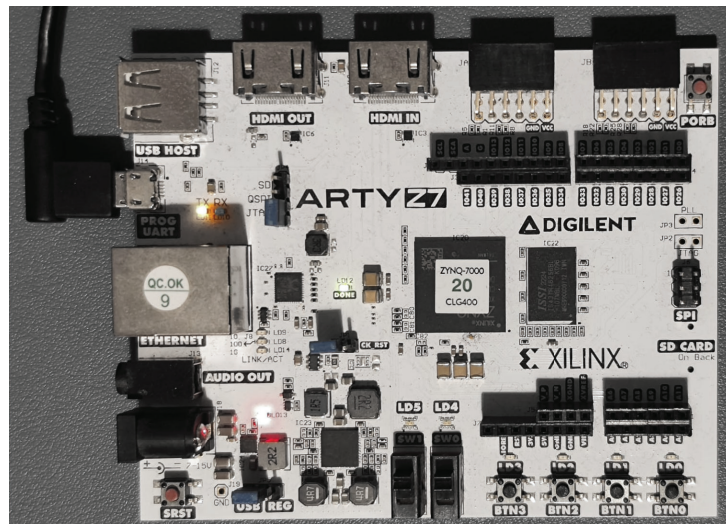


Figura 33: Sistema de desarrollo Arty Z7-20 con los LED de actividad de la UART activos durante la ejecución de los tests.

El resultado de la implementación se exporta como plataforma *hardware* (`*.xsa`), que contiene el *bitstream* y la descripción del sistema, y es el punto de inicio del proyecto de Vitis con el que se construye el *software* de la parte de procesamiento.

6.5.2. Aplicación *bare-metal*

La aplicación *bare-metal* diseñada recibe los vectores de test a través del puerto serie y lleva a cabo las transacciones `DMA` para hacer llegar el vector de test al acelerador y transmitir el resultado para su comprobación mediante el *script* de `HIL`.

En la fase de inicialización se configuran el `DMA` y el controlador de interrupciones, *Generic Interrupt Controller* (`GIC`), al que se conectan las interrupciones de recepción y envío del `DMA`. Para sincronizar ambas partes de la comunicación a través del puerto serie se utiliza un protocolo simple: el *script* de `HIL` espera hasta que la plataforma de desarrollo transmita `READY`. Tras esto se transmiten dos líneas: la primera, la operación a realizar `NTT` o `INTT`, y la segunda, 256 coeficientes. La plataforma de desarrollo responde con `RUN`, `RESULT` con los 256 coeficientes resultantes y `DONE` al finalizar. Tras esta secuencia regresa a `READY`.

Por cada vector recibido se ejecuta la secuencia incluida en el [Código 10](#):

1. Se selecciona el modo de operación con el registro de control 0 del acelerador, es decir, se utiliza la interfaz AXI4-Lite del mismo modo que en las etapas de simulación.
2. Se vacía el caché de los *buffers* de envío y recepción.
3. Se prepara el canal de recepción y de envío.
4. Se espera hasta la finalización de ambas transferencias con un *timeout*.
5. Se vacía la caché de *buffer* de recepción y se transmite por el puerto serie `RESULT`.

```
1
2 Xil_Out32(KYBER_BASEADDR + KYBER_REG0_OFFSET, opmode);
3
4 Xil_DCacheFlushRange((UINTPTR)tx_buf, NTT_BYTES);
5 Xil_DCacheFlushRange((UINTPTR)rx_buf, NTT_BYTES);
6
7 status = XAxiDma_SimpleTransfer(&AxiDma, (UINTPTR)rx_buf, NTT_BYTES,
8                                XAXIDMA_DEVICE_TO_DMA);
9 status = XAxiDma_SimpleTransfer(&AxiDma, (UINTPTR)tx_buf, NTT_BYTES,
10                               XAXIDMA_DMA_TO_DEVICE);
11
12 timeout = DMA_TIMEOUT;
13 while ((!TxDone || !RxDone) && !DmaError) {
14     if (--timeout == 0) {
15         xil_printf("ERROR: DMA interrupt timeout\r\n");
16         return XST_FAILURE;
17     }
18 }
19
20 Xil_DCacheInvalidateRange((UINTPTR)rx_buf, NTT_BYTES);
```

Código 10: Secuencia de operaciones por cada vector, fichero `ntt_dma.c`.

6.5.3. Script de HIL

El *script* de [HIL](#), equivalente a los *testbenches* utilizados en el resto de técnicas, genera los vectores de test, los aplica sobre el [DUT](#) a través del puerto serie y contrasta el resultado mediante el mismo modelo de referencia ya utilizado, `ntt_model.py`. Por cada vector se comprueba que la operación recibida es la esperada, que se reciben 256 coeficientes y que estos coinciden con el resultado generado por el modelo. En caso de discrepancia, se genera un mensaje de error. En el [Código 11](#) se muestra cómo se lleva a cabo la generación (`expected`), transmisión al acelerador (`send_vector`) y la obtención del resultado de ejecución para cada vector de test (`read_result`).

```
1 def run_case(port, op, stimulus, timeout_s, quiet):
2     expected = run_operation(op, stimulus)
3
4     send_vector(port, op, stimulus)
5     result_op, got = read_result(port, timeout_s, quiet)
6     read_done(port, timeout_s, quiet)
7     read_ready(port, timeout_s, quiet)
8
9     if result_op != op:
10         raise AssertionError(f"board returned {result_op}, expected {op}")
11
12     if len(got) != N:
13         raise AssertionError(f"board returned {len(got)} coefficients, expected {N}")
14
15     mismatch = first_mismatch(got, expected)
16     if mismatch is not None:
17         index, received, wanted = mismatch
18         raise AssertionError(
19             f"{op} mismatch at coefficient {index}: got {received}, expected {wanted}")
```

Código 11: Generación del vector de test, transmisión, recepción y comprobación según el modelo, fichero `send_ntt_vectors.py`.

6.6. Orquestación del *Pipeline*

En las secciones anteriores se ha presentado cada técnica de verificación y se ha mostrado su utilización para cada caso de uso. En esta sección se describen los componentes desarrollados que realiza la integración en un *pipeline* de verificación. En particular, en la [Sección 6.6.1](#) se indican las formas de interactuar con el mismo, es decir, cuál es el flujo para desarrollar los tests necesarios y cómo se pueden ejecutar en cada caso. En la [Sección 6.6.2](#) y la [Sección 6.6.3](#) se presentan la estructura de archivos que se ha seguido y los ficheros que gobiernan todo el *pipeline*, respectivamente.

6.6.1. Casos de uso

El *pipeline* se ha desarrollado teniendo en cuenta dos casos de uso: durante el **desarrollo**, para realizar el proceso iterativo de implementar los tests y confirmar que el *pipeline* se comporta como se espera, y para **CI/CD**, es decir, la ejecución y obtención de métricas automatizadas propias del desarrollo *software* y que está asociada al sistema de control de versiones distribuido, git [75]. En este trabajo no se ha desplegado la infraestructura necesaria para ejecutar **CI/CD** mediante el uso de un servidor *on-premises* o un servicio externo de git, como Gitlab, Bitbucket, Github u otros.

Se ha desarrollado el *pipeline* enfocado a su ejecución en los entornos indicados, minimizando el esfuerzo necesario para lograr su integración. Para facilitar la ejecución del *pipeline* en el contexto de **CI/CD** se han incluido dos estrategias de ejecución:

- Modo `light`: selecciona un subconjunto de tests pensado para ser rápido y limitar el uso de licencias para que se pueda ejecutar con cada git *commit* sin entorpecer el desarrollo del diseño ni acaparar un posible servidor en el que se ejecutan los flujos de trabajo **CI/CD**.
- Modo `full`: ejecuta todo el conjunto de tests e incluye los tests de **HIL**. Está pensado para ser ejecutado una vez por día cuando se estima que no se va a hacer uso de las licencias ni de los recursos del servidor en el que se ejecuten los *runners*, que son los encargados, en última instancia, de interactuar con los tests y el *software* de verificación de forma automatizada y reproducible.

6.6.2. Estructura de directorios

La estructura organizativa del proyecto está planteada tanto para continuar con el desarrollo del diseño a verificar como para desarrollar y ejecutar los diferentes tests. El proyecto sigue la organización de la [Figura 34](#), donde `project_config.yml` y `Makefile` son los ficheros principales.

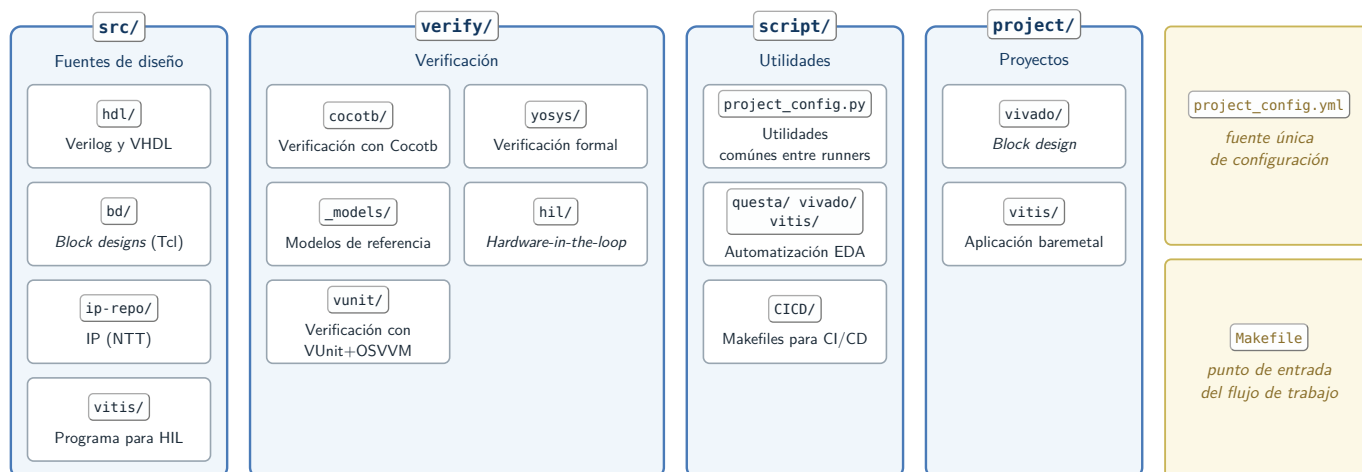


Figura 34: Estructura de directorios del proyecto.

6.6.3. Ficheros principales

El fichero `project_config.yml` guarda toda la configuración del proyecto de forma centralizada y es la referencia que utiliza todo el resto de componentes, desde herramientas de desarrollo como Vivado y Vitis hasta las herramientas de *linting*, pasando por todos los *runners* de cada una de las técnicas de verificación. Contiene parámetros de diseño generales que permiten recrear los proyectos mediante *scripts* Tcl, rutas de ficheros de código HDL junto a sus dependencias, los tests que se han de realizar y si un determinado test pertenece al conjunto de tests `light` o `full` (Código 12). El objetivo de utilizar un archivo centralizado es permitir que todos los componentes del *pipeline* permanezcan en sincronía entre sí y los cambios se acoten al menor número de ficheros posible.

```

1  #> project_config.yml: single source of truth for the project.
2  #><-----General values----->
3  TOP: kyber_ntt_top
4  #TB_TOP: tb_mult
5  TARGET_LANGUAGE: Verilog
6  SIMULATOR_LANGUAGE: Verilog
7  TARGET_FPGA: "xc7z020clg400-1"
8  TARGET_BOARD: "digilentinc.com:arty-z7-20:part0:1.1"
9  PHYSICAL_DEVICE: "xc7z020_1"
10
11 #><-----Cocotb tests----->
12 cocotb:
13   paths:
14     output_dir: "./output/cocotb"
15     sim_build_dir: "./output/cocotb/sim_build"
16     pythonpath: "../verify/cocotb"
17   runs:
18     addressgenerator_schedule:
19       module_name: test_addressgenerator
20       source_set: addressgenerator
21       simulator: verilator
22       mode: light
23       generics: {}
24     KyberHPM1PE_fntt_intt:
25       module_name: test_KyberHPM1PE
26       source_set: KyberHPM1PE
27       simulator: questa
28       mode: full
29       generics: {}
...
101 #-----VUnit tests----->
...

```

```

120 #-----Yosys tests----->
...
143 #-----Sources----->
144 source_sets:
145     ntt_regs_s00_axi:
146         files:
147             - ./src/hdl/ntt_regs_S00_AXI.vhd
148
149     shiftreg:
150         files:
151             - ./src/hdl/shiftreg.v
152
153     div2:
154         files:
155             - ./src/hdl/div2.v
...
228 #-----Libraries----->
229 sources:
230     hdl: "./src/hdl/"
231     libraries:
232         osvvm:
233             path: "./verify/osvvm/new/2026.01"
234             files:
235                 - osvvm/IfElsePkg.vhd
236                 - osvvm/OsvvmTypesPkg.vhd
237                 - osvvm/OsvvmScriptSettingsPkg.vhd
238                 - osvvm/OsvvmSettingsPkg.vhd
239                 - osvvm/OsvvmScriptSettingsPkg_default.vhd
240                 - osvvm/OsvvmSettingsPkg_default.vhd
...

```

Código 12: Estructura general del fichero de configuración `project_config.yml`.

El fichero de reglas `Makefile` permite, mediante el comando `make`, llevar a cabo de forma sencilla diferentes tareas como ejecutar los tests (todos o un solo tipo), abrir las trazas generadas o regenerar los proyectos con Vivado o Vitis (Código 13).

```

...
38 $(VIVADO)-create $(VIVADO)-sim $(VIVADO)-lint:
39     export TCL_FLOW_TARGET_PROC=$@ && cd output && \
40         $(COLOR_START) vivado -mode batch -notrace -source ../script/vivado/utills.tcl $(TCL_ARGS) 2>&1 | \
41         grep --line-buffered --invert-match 'Board 49-26' | \
42         $(COLOR_END)
43
44 vivado:
45     cd project/vivado && setsid -f vivado ./*.xpr && cd ../..
46
47 # --- Vitis ---
48 VITIS := vitis
49 $(VITIS):
50     cd project/vitis && vitis
51
52 $(VITIS)-create:
53     vitis -s script/vitis/recreate_vitis_project.py ./src/vitis/design_1_wrapper_arty_v1.xsa --clean
54
55 $(VITIS)-build:
56     xsdb script/vitis/run_hw.tcl -build
57
58 $(VITIS)-run:
59     xsdb script/vitis/run_hw.tcl -run
60
61 # --- verification ---
62
63 VUNIT_OUTPUT_DIR := output/vunit
64 COCOTB_OUTPUT_DIR := output/cocotb
65
66 include ./script/questa/questa.mk

```

```

67
68 include ./verify/cocotb/cocotb.mk
69 include ./verify/yosys/yosys.mk
70 include ./verify/vunit/vunit.mk
71 include ./verify/hil/hil.mk
72
73 include ./script/CICD/cicd.mk
74
75 # --- Utils ---
76 wave:
77 @file=$(find . \( -name "*.ghw" -o -name "*.vcd" -o -name "*.fst" \) | fzf); \
78 [ -n "$$file" ] || exit 0; \
79 viewer=$(printf "surfer\ngtkwave\n" | fzf --prompt="viewer> "); \
80 [ -n "$$viewer" ] || exit 0; \
81 setsid -f "$$viewer" "$$file" > /dev/null 2>&1
82
83 nvc-lint: $(GENERATED_CONFIG)
84 mkdir -p ./output/nvc && \
85 nvc --std=08 --work=./output/nvc/work \
86 --map=unisim:$(UNISIM_PATH) \
87 -a $(WORK_SOURCES)
88
89 # --- Other ---
90 clean:
91 rm -rf ./project/vitis/* ./project/vivado/* ./output/*

```

Código 13: `Makefile` con todas las tareas que se pueden llevar a cabo durante el desarrollo.

6.6.4. Runners

El concepto de *runner* es propio de algunas técnicas de verificación. Dependiendo de la técnica concreta, se trata de un programa encargado de compilar las fuentes HDL, configurar e interactuar con el simulador, recopilar y configurar los tests que se llevan a cabo. Por ejemplo, el *runner* de Cocotb [76] es un *script* de Python con la estructura general del Código 14, en el que se selecciona Icarus Verilog como simulador, las fuentes SystemVerilog (`my_design.sv`), el *top* (`my_design`), el test a ejecutar (`test_my_design`) y se encarga de iniciar los tests.

```

1 # test_runner.py
2
3 import os
4 from pathlib import Path
5
6 from cocotb_tools.runner import get_runner
7
8
9 def test_my_design_runner():
10     sim = os.getenv("SIM", "icarus")
11
12     proj_path = Path(__file__).resolve().parent
13
14     sources = [proj_path / "my_design.sv"]
15
16     runner = get_runner(sim)
17     runner.build(
18         sources=sources,
19         hdl_toplevel="my_design",
20     )
21
22     runner.test(hdl_toplevel="my_design", test_module="test_my_design")
23
24
25 if __name__ == "__main__":
26     test_my_design_runner()

```

Código 14: Estructura genérica del runner de Cocotb.

En el trabajo desarrollado se ha expandido la funcionalidad de los *runners* de Cocotb y VUnit y se ha añadido un equivalente para la verificación formal, con el fin de que hagan uso del fichero `project_config.yml`, [Código 12](#), para determinar la configuración a utilizar.

Runner de Cocotb

El *runner* de Cocotb desarrollado (`verify/cocotb/runner.py`) extiende la estructura general del [Código 14](#): cada test declarado en `cocotb.runs` que pertenezca al modo seleccionado (`full` o `light`) se convierte en test independiente, identificado por el nombre del test en la configuración. Cada test compila las fuentes indicadas en el fichero de configuración y ejecuta el test correspondiente. El resultado de los tests está formado por un informe JUnit XML y las trazas de ejecución, en el caso de que se haya activado su generación.

Runner de VUnit

VUnit también tiene un *runner* propio similar (`verify/vunit/run.py`) al que tiene Cocotb. En este caso utiliza `project_config.yml` para filtrar los tests de `vunit.runs` según el modo, registrar las bibliotecas de verificación indicadas en la configuración (OSVVM y sus componentes de verificación de AXI4) y añade a las fuentes de cada `source_set` junto con sus *testbenches*.

Runner de Yosys

El *runner* de verificación formal (`verify/yosys/runner.py`) es el más simple. Cada test de `yosys.runs` se corresponde con un fichero `.sby` ([Sección 6.2](#)). El *runner* se limita a seleccionar los tests según el modo y a llamar `sby` sobre cada uno (`sby -f -d <directorio>`). Los resultados, como en el resto de técnicas, se localizan en `output/yosys/`.

6.7. Conclusión

En este capítulo se ha descrito cómo se han empleado los conceptos planteados en los capítulos anteriores para desarrollar los tres pasos que dan lugar al *pipeline* de verificación aplicado al acelerador criptográfico de NTT/INTT. Se empieza por dividir el diseño por áreas a las que aplicar los diferentes tipos de verificación, según la propia jerarquía y las particularidades de los módulos. A continuación se busca un enfoque apropiado para cada test, se desarrollan los modelos de referencia y se desarrolla cada uno de los test. Finalmente se plantea la ejecución del *pipeline* de modo que se puedan ejecutar tanto en modo local o como parte de un *pipeline* de CI/CD. El conjunto de test debe estar diseñados para contribuir al desarrollo de los bloques del diseño y a que ayuden a encontrar errores en distintos casos de verificación.

Capítulo 7

Resultado y evaluación del *pipeline*

La verificación desarrollada combina 4 técnicas: verificación formal para los bloques aritméticos primitivos, verificación funcional con VUnit y OSVVM para las interfaces AXI, verificación funcional con Cocotb a tres niveles de granularidad con modelos de referencia compartidos y la verificación HIL sobre la implementación del acelerador en la plataforma de desarrollo Arty Z7-20 que ha reutilizado el modelo `ntt_model.py`. En esta sección se presentan los resultados obtenidos con cada técnica.

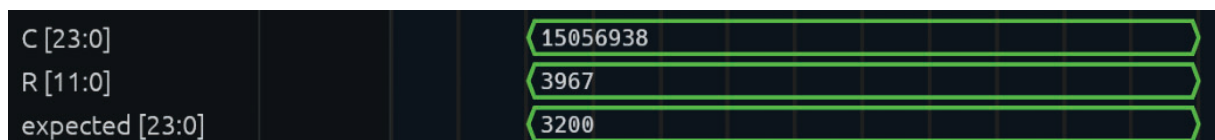
7.1. Resultados por técnica de verificación

7.1.1. Verificación formal

Los seis tests de verificación formal planteados en la Sección 6.2 se completan de forma satisfactoria. Debido a que se mantienen las entradas simbólicas se explora de forma exhaustiva el espacio de estados por ejemplo con `intmul.v` se cubren las 2^{24} combinaciones posibles de operados de 12 bits. Dicha exhaustividad es la principal aportación frente a las técnicas que emplean simulación, sobre todo en módulos cuyos valores no están restringidos por el resto del sistema, en los que, aún existiendo un error, se puede no llegar a manifestar debido a un uso final acotado.

Hallazgos con verificación formal

Durante el desarrollo del test para `modred.v` se encontró que el módulo, aunque acepta una entrada de 24 bits, presenta huecos en los que la reducción no se produce de forma correcta como se puede ver en la Figura 35 para el valor de entrada **15056938**. Esto no representa un fallo del módulo porque no afecta a su funcionamiento, debido a que la entrada es la multiplicación de dos coeficientes en el rango canónico, como máximo, $(3329 - 1) \cdot (3329 - 1) = 11075584$. A partir de este caso se examinó con Cocotb el resto de reducciones hasta 24 bits, dando lugar a 500 reducciones erróneas.



C [23:0]	15056938
R [11:0]	3967
expected [23:0]	3200

Figura 35: Primer fallo en orden ascendente del módulo `modred.v`.

7.1.2. Cocotb

Durante el desarrollo de los tests de Cocotb se plantearon 13 ficheros de test, uno por cada uno de los módulos que componen el diseño, incluyendo los de integración y excluyendo los módulos aritméticos básicos y las primitivas que los componen. Los principales beneficios obtenidos han sido, por una parte, la facilidad para integrar modelos complejos como los de la unidad *butterfly* o el generador de direcciones, y la posibilidad de generar tests acotados para hacer diferentes pruebas, como en el caso del módulo `modred.v` planteado en la sección anterior.

Hallazgos con Cocotb

El test `test_kyber_hpm1pe_load_ignores_din_valid` establece que los contadores de carga del núcleo ignoran `din_valid`, una señal que por su nombre y el código comentado dentro del módulo podría servir para validar la entrada. Como se puede ver en el [Código 15](#) con la entrada `din_valid = 0` en el test se computa la transformada de forma favorable.

```
1 @cocotb.test()
2 async def test_kyber_hpm1pe_load_ignores_din_valid(dut):
3     ...
23     for coeff in poly:
24         dut.din.value = coeff
25         dut.din_valid.value = 0
26         await RisingEdge(dut.clk)
27     ...
47     await pulse_start_fntt(dut)
48     await wait_done(dut)
49     got = await collect_output_poly(dut)
50     expected = fntt_model(poly)
51     assert got == expected, first_mismatch_message(got, expected)
```

Código 15: Test que establece que `din_valid` se ignora.

7.1.3. VUnit con OSVVM

Se desarrollaron 2 *testbenches* de VUnit uno para AXI4-Lite y uno para AXI4-Stream. La principal ventaja que ofrece OSVVM es la certeza de que los VCs están implementados de forma correcta y han sido probados de forma extensiva.

Hallazgos con VUnit

- **TLAST condicionado por TREADY**: en el caso en el que el acelerador entrega en 255 coeficientes resultantes de la computación y el receptor de esos coeficientes no está preparado para recibir el último (mantiene `TREADY=0`) se produce un cambio en la señal de control `TLAST` debido a que la implementación la genera de forma combinacional, [Código 16](#), incluso después de haber indicado que la transacción es válida (`TREADY=1`), como se puede ver en la [Figura 36](#). El comportamiento esperado es que no se cambien las señales de control una vez se da por válida la transacción, según el estándar AXI4-Stream [77].

```
1 m_axis_tlast <= '1' when (tlast_int = '1'
2                         and m_axis_tready = '1'
3                         and m_axis_tvalid = '1') else '0';
```

Código 16: Generación de `m_axis_tlast` en `kyber_hpm1pe.vhd`, la señal está condicionada por `m_axis_tready`.



Figura 36: Cambio de `TLAST` después de `TVALID`

7.1.4. Verificación HIL

El flujo de verificación HIL permite utilizar un solo comando para lanzar los tests sobre la plataforma de desarrollo Arty Z7-20, facilitando probar sobre *hardware* real el acelerador. Comienza con recrear el proyecto con Vitis a partir del fichero `.xsa` de la plataforma creada con Vivado, carga el *bitstream*, compila y carga la aplicación *bare-metal* y ejecuta la *script* HIL que se conecta con

el puerto serie e indica al acelerador la operación a realizar con los vectores provistos. Tal y como está configurado el comando `make hil`, se computan 1000 veces las operaciones `NTT` e `INTT`. El terminal presenta un resumen de las operaciones según se llevan a cabo como se puede ver en la Figura 37.

```

RUN NTT
KYBER REG0 = 0x00000000
DMA transfers started
TX DMA done
RX DMA done
RESULT NTT 238 961 525 2698 3146 1001 849 1843 2238 1776 1292 2763 1755 2877 126 597 1061 543 2147 1263 2501 1374 515 2900 2495 3283 682
15 2726 2116 1906 2998 1597 3023 1545 382 268 568 2170 2560 1643 1922 1881 1555 1822 518 1250 1260 1638 1617 1553 1940 2610 210 1461 10
31 2465 1206 169 2870 613 998 827 3277 1627 1700 2302 1991 1474 1597 2397 1598 681 1050 674 1296 904 3205 2146 2334 950 1262 2341 475 21
04 2659 830 1447 2721 1108 2428 3312 873 1061 1717 1972 2238 1476 2390 1684 2012 3080 1358 1847 3062 469 3146 1276 3072 1878 2718 3282 1
999 1944 1219 3297 579 3266 640 338 1565 2358 548 304 814 2327 1967 2813 2325 355 375 1195 2044 1375 1586 1673 2678 1130 173 2141 2427 1
744 65 2644 960 2558 1869 3263 2879 2000 2380 1541 54 391 1242 1515 2146 2354 310 1043 626 1678 2442 651 2936 1608 2298 1885 2864 758 28
15 1726 1877 2008 1181 1095 1716 1550 957 1599 2558 2112 684 1512 1089 1189 2820 2734 3090 2218 248 2020 1839 1415 2101 2605 2565 2398 3
195 2153 1086 1987 432 2111 7 1565 2551 3138 2215 2069 105 3148 1980 1965 140 1621 829 1854 3036 525 362 1939 3098 2978 789 2172 1059 18
76 1226 433 209 2153 1813 2419 1445 2908 1591 2890 3162 2829 785 833 3160 2773 1951 2482 3055 2544 3235 1536 176 2455 2777 1748 305 2114
DONE
READY format: NTT|INTT c0 c1 ... c255
PASS NTT vector=999
case 2000/2000: INTT pattern=random seed=1 vector=999
RUN INTT
KYBER REG0 = 0x00000001
DMA transfers started
TX DMA done
RX DMA done
RESULT INTT 2494 916 2616 2144 1778 2014 1842 912 2791 2811 1777 1666 533 2917 1161 924 1644 2315 2718 1110 2865 1308 2698 2275 1730 61
108 828 1065 1749 2462 634 1801 1452 2760 710 2931 2242 2803 1603 2233 1271 993 229 2739 3307 1598 1752 3245 54 2699 2940 2981 808 92 15
45 24 1364 2454 1367 1929 1705 2924 1436 843 2578 2446 7 1542 2660 1818 2315 1228 935 3168 2228 449 54 1657 2941 1899 1768 1912 1624 316
5 2184 554 1289 2533 115 2759 1461 947 995 274 878 1097 246 795 177 1421 1555 2473 923 2314 2734 2899 3178 2206 1785 1174 2310 2158 2096
1601 1324 1142 2432 1726 1885 2436 1089 2937 1855 778 1964 2897 2774 361 438 593 2431 3206 644 445 1401 1697 2017 714 398 1327 2335 138
1648 894 1035 14 3312 2841 2943 169 1417 2006 3315 3163 2447 1740 2990 2025 2661 1990 445 2394 298 200 2738 1314 352 1975 921 2811 3117
296 6 3153 1921 773 1482 3210 2654 512 2780 1732 551 3091 705 16 608 3113 2331 2062 1674 664 810 2182 2094 469 467 442 3322 601 1139 12
22 2293 718 1742 573 2154 466 3007 1564 19 592 399 1142 1366 1821 1575 3223 8 574 2267 3060 1369 1272 1887 1558 990 52 3072 1349 2934 67
3 1259 1763 1101 2485 2081 706 1293 476 3248 911 981 1593 1910 1712 2724 1037 197 2735 1154 2735 959 2652 571
DONE
READY format: NTT|INTT c0 c1 ... c255
PASS INTT vector=999
PASS 2000/2000 cases

```

Figura 37: Resultado de la verificación `HIL` para el último de 1000 vectores de test. El resultado de la `NTT` está indicado por `RESULT NTT` y el de la `INTT` está indicado por `RESULT INTT`.

La verificación `HIL` permite así comprobar la correcta integración de todo el sistema, se lleva a cabo la síntesis, la implementación, se cumplen con las restricciones temporales y los componentes de la plataforma (`PS`, `DMA` y `software`) se comportan como se espera.

Hallazgos con la verificación `HIL`

El hecho de que el acelerador no implemente la señal (no obligatoria según el estándar) `TLAST`, en el `AXI4-Stream` de entrada del polinomio no tiene un impacto significativo en el funcionamiento del sistema siempre que se tenga en cuenta y se cuenten 256 transferencias en la aplicación *bare-metal* para determinar el final de la trama.

7.2. Resultados y evaluación del *pipeline*

En esta sección se presentan los resultados del *pipeline* como conjunto.

7.2.1. Tiempo de ejecución del *pipeline*

La idea detrás del *pipeline* de verificación es automatizar la ejecución de los tests mediante en un *pipeline* de `CI/CD` en un contexto de desarrollo del diseño que un *software* de control de versiones como Git, aunque esto queda fuera del alcance de este trabajo. Sin embargo, se ha tenido en consideración que en un *pipeline* de `CI/CD` se busca la granularidad de modo que, se pueda

ejecutar un subconjunto de los tests frecuentemente por ejemplo con cada *commit* y se ejecute el conjunto de forma esporádica, por ejemplo, cada noche (*nightly*). Para dar soporte a esta finalidad se han incluido dos *targets*: `make ci-light` y `make ci-full` el primero ejecuta los tests marcados como `light` el segundo ejecuta todos los test e incluye la verificación [HIL](#).

La métrica principal en el contexto de los *pipelines* de [CI/CD](#) es su tiempo de ejecución. Limita la utilización de los recursos de cómputo en que se ejecuta el *workflow*, la velocidad de desarrollo y el uso de licencias en uso (recurso crítico que debe estar disponible cuando sean requeridas por el diseñador). En el [Código 17](#) se muestra el tiempo de ejecución para cada una de las técnicas de verificación utilizando Questa Sim [\[42\]](#).

Es posible optimizar el tiempo de ejecución mediante la utilización de distinto tipos de simuladores. En el [Código 18](#) se muestran los tiempos de ejecución para los tests de Cocotb, que son los únicos que lo permiten debido al limitado soporte para lenguajes mixtos por parte de los simuladores de uso libre. En este caso se ha utilizado NVC [\[41\]](#), Verilator [\[31\]](#) y Questa Sim [\[42\]](#).

```
make yosys MODE=full: 22:14.61
make cocotb MODE=full: 5:29.92
make vunit MODE=full: 2:50.23
make hil MODE=full: 8:06.32
```

Código 17: Tiempo de ejecución según el tipo de verificación, todos los tests con Cocotb.

```
make yosys MODE=full: 22:14.61
make cocotb MODE=full: 2:46.92
make vunit MODE=full: 2:50.23
make hil MODE=full: 8:05.51
```

Código 18: Tiempo de ejecución según el tipo de verificación, Cocotb con simuladores mixtos.

7.2.2. Cobertura agregada de VUnit y Cocotb

Las métricas de cobertura presentadas a continuación han sido agregadas en dos pasos: se han agregado las bases de datos UCDB generadas por Questa Sim de los tests de Cocotb y los tests de VUnit, ambas bases de datos se han agregado una vez más, el resultado es la cobertura por unidad de diseño o módulo, [Figura 38](#) y la cobertura por instancia, [Figura 39](#).

Design Units Coverage Summary (92.75%)					
					☰
Coverage Type ↑	Bins	Hits	Misses	Coverage	
Search... ▼	Search... ▼	Search... ▼	Search... ▼	Search... ▼	
Assertions	1	1	0	100%	
Branches	950	904	46	95.15%	
Conditions	188	176	12	93.61%	
Expressions	24	21	3	87.5%	
FSM States	14	13	1	92.85%	
FSM Transitions	23	19	4	82.6%	
Statements	1227	1174	53	95.68%	
Toggles	5926	5607	319	94.61%	

Figura 38: Cobertura agregada por unidad de diseño.

Instance Coverage Summary (80.71%)					
					☰
Coverage Type ↑	Bins	Hits	Misses	Coverage	
Search... ▼	Search... ▼	Search... ▼	Search... ▼	Search... ▼	
Assertions	1	1	0	100%	
Branches	4625	3622	1003	78.31%	
Conditions	718	503	215	70.05%	
Expressions	72	49	23	68.05%	
FSM States	56	46	10	82.14%	
FSM Transitions	91	65	26	71.42%	
Statements	6031	5096	935	84.49%	
Toggles	19432	17728	1704	91.23%	

Figura 39: Cobertura agregada por instancia.

La cobertura agregada por unidad de diseño, [Figura 38](#), alcanza un 92.75% en comparación a la cobertura por instancia, [Figura 39](#), 80.71%. Esto es debido a cómo se ha planteado la verificación, por ejemplo, los tests que de la interfaz AXI4-Stream, que utilizan el acelerador completo en lugar de un módulo independiente, tienen una cobertura muy reducida y al agregar esa cobertura al resto, baja la media. Tienen una cobertura reducida porque no se centran en verificar la *INTT* que sí ha sido verificada con otra parte de los tests con Cocotb. Es por esto que hay tests que ofrecen poca cobertura a un determinado módulo, sin embargo, si se agrega dicha cobertura por unidad de diseño (archivo de diseño) se puede ver que sí alcanza la cobertura deseada, [Figura 38](#).

7.3. Conclusión

El *pipeline* en su conjunto se comporta de acuerdo a sus expectativas: las diferentes técnicas de verificación coexisten entre sí a partir del planteamiento por área inicial y es posible integrarlas en un *pipeline* unificado.

Los productos resultantes de su ejecución: resultado de los tests, trazas e información de cobertura, permiten obtener garantías e información relevante, tal como: limitaciones, errores y código sin uso.

En concreto se han encontrado ciertas áreas en las que se puede mejorar el diseño como la conformidad con el estándar de la interfaz AXI4-Stream o el uso de la señal `din_valid` y documentación del módulo `modred.v`. Indicar que estos factores de mejora son deseables, incluso cuando este diseño ha sido probado y utilizado de forma extensa en entornos cercanos a la producción.

Capítulo 8

Conclusiones y trabajo futuro

8.1. Conclusión

En este TFM se ha estudiado, diseñado e implementado un *pipeline* unificado de verificación avanzada orientado a FPGA aplicado a un diseño centrado en la criptografía postcuántica cumpliendo así con el objetivo general planteado al inicio del trabajo.

La fase de estudio inicial se ha centrado en las áreas fundamentales para el trabajo: la verificación y la criptografía. En cuanto a la verificación se ha comenzado con un estudio de la organización de la verificación avanzada que ha permitido orientar los diferentes enfoques en 6 ejes. En cuanto a la criptografía, se ha partido de las bases, las primitivas y esquemas básicos en los que se basa la criptografía moderna, para concluir con el estudio de la amenaza cuántica, la necesidad de algoritmos resistentes a ataques postcuánticos y los estándares que han surgido como respuesta.

Tras haber adquirido una visión clara del panorama de estándares postcuánticos se ha decidido utilizar ML-KEM como estándar para KEM sobre el que centrar los esfuerzos de verificación. Con el objetivo de acotar la extensión de los esfuerzos de verificación se ha buscado el componente de ML-KEM que más representativo es del estándar y de los módulos de criptografía postcuánticos. Se ha concluido que ese elemento es el módulo NTT por su complejidad aritmética, arquitectura estrechamente relacionada con los parámetros del estándar y la diversa naturaleza de sus módulos. Para ello ha sido necesario un estudio de los algoritmos que componen el estándar que ha permitido entender las decisiones de diseño tomadas en las diferentes implementaciones presentadas en el estado del arte de implementaciones de ML-KEM sobre FPGA.

El desarrollo se ha centrado en la implementación de NTT provista por la empresa colaboradora ARC que añade al trabajo de Yaman et al. las interfaces AXI4-Stream y AXI4-Lite. Sobre este trabajo se han seguido de forma general los 3 pasos planteados en la Figura 25:

- la asignación de técnicas de verificación por área del diseño,
- el desarrollo de la infraestructura de tests (*runners* y los propios tests) y
- la ejecución del *pipeline* de verificación.

La asignación de las diferentes técnicas de verificación se ha realizado en función de la jerarquía del diseño y la naturaleza de los módulos que lo componen y ha permitido obtener el mejor rendimiento de las fortalezas de cada técnica:

- el *runner* de VUnit,
- los VCs de OSVVM,
- la versatilidad y la facilidad de añadir modelos de referencia de Cocotb,
- la exhaustividad de la verificación formal con Sby y
- las garantías que aporta la verificación HIL.

El desarrollo de la infraestructura de tests ha presentado distintos retos debido a la diversidad de herramientas que se han integrado entre sí. Como resultado se ha obtenido el beneficio de

normalizar las interacciones con todas las técnicas mediante Python. En consonancia con el fichero `project_config.yml`, que contiene la configuración del *pipeline* en un solo lugar, se han facilitado las tareas de añadir o quitar tests y ejecutar tests con configuraciones alternativas, como simuladores alternativos.

De la ejecución del *pipeline* se obtienen distintos productos:

- resultado de los tests,
- trazas e información de cobertura,

todos localizados en el directorio (`./output`) que permiten recopilar información acerca del diseño para su posterior mejora.

En concreto, se han encontrado limitaciones en la unidad de reducción de módulo (`modred.v`), no conformidad con el estándar por parte del AXI4-Stream, código sin uso con la señal `din_valid` y se ha adquirido la certeza de que el diseño funciona sobre *hardware* real bajo las condiciones específicas probadas en la fase [HIL](#).

Se han obtenido los tiempos de ejecución del *pipeline* para cada una de las técnicas utilizadas, en los que la verificación formal y el uso de diferentes simuladores presentan el mayor impacto, métrica que es útil por la necesidad de optimizar los recursos de los que se disponga en cada momento y la planificación del equipos de diseño.

Se puede concluir que este trabajo ha contribuido de forma significativa a la comprensión tanto de la criptografía postcuántica como de la verificación avanzada. En el caso de la criptografía postcuántica su obligatoriedad de cara al futuro y los estándares que hacen posible su adopción. En el caso de la verificación avanzada, los aspectos necesarios para la correcta implementación y aplicación de diferentes técnicas a las áreas del diseño que más se benefician en un tipo concreto.

La adopción de este *pipeline* de verificación avanzado permite agilizar la verificación de los diseños criptográficos y adquirir garantías a la vez que se eliminan duplicidades, se centraliza su configuración y se simplifican las interacciones entre sus componentes, garantizando la fiabilidad de un componente crítico en las comunicaciones futuras.

8.2. Trabajos futuros

Una vez completado el trabajo realizado, se han identificado distintos trabajos futuros a considerar. A continuación se incluye una lista, necesariamente incompleta, de posibles ampliaciones del trabajo presentado:

1. Generalizar el *pipeline* para ampliar el soporte de entornos de ejecución alternativos como Windows y otras herramientas como simuladores, *linters* y *toolchains* que permitan dar garantías sobre las particularidades de cada entorno y diseño.
2. Mejorar la cohesión de los diferentes componentes del *pipeline* por medio de interfaces robustas que incluyan errores descriptivos en caso de una configuración incorrecta.
3. Extender la ejecución del *pipeline* de verificación a un entorno de [CI/CD](#) real como un despliegue de Gitlab local que haga uso de contenedores y máquinas virtuales para distribuir las utilidades necesarias para llevar a cabo tareas de verificación más complejas, por ejemplo, incluir emulación de sistemas completos.

4. Extender el *pipeline* de verificación para dar soporte a una mayor diversidad de diseños criptográficos.
5. Extender el *pipeline* para añadir trazabilidad hacia la especificación de funcionalidad del DUT, de modo que se preserven las garantías hasta la llegada del diseño a producción por medio de enfoques de verificación como la cobertura funcional o *frameworks* como UVVM.
6. Extender el uso de la verificación formal por dos vías: el uso de herramientas que introduzcan alteraciones transitorias del diseño como MCY de Yosys para asegurar que se ha realizado un buen planteamiento de los tests y la aplicación de técnicas formales a módulos de mayor complejidad, es decir, extensión o profundidad en ciclos.

Referencias

- [1] Harry D. Foster, «2024 Wilson Research Group FPGA functional verification trend report», 2024, [En línea]. Disponible en: <https://verificationacademy.com/topics/planning-measurement-and-analysis/wrg-industry-data-and-trends/2024-siemens-eda-and-wilson-research-group-ic-asic-functional-verification-trend-report/>
- [2] Christian de Looper, «TSMC has hit maximum capacity». Accedido: 12 de abril de 2026. [En línea]. Disponible en: <https://rcrtech.com/semiconductor-news/tsmc-maximum-capacity/>
- [3] W.-Y. Lee, «Broadcom flags supply constraints, says TSMC capacity a bottleneck», *Reuters*, mar. 2026, Accedido: 12 de abril de 2026. [En línea]. Disponible en: <https://www.reuters.com/world/asia-pacific/broadcom-flags-supply-constraints-says-tsmc-capacity-bottleneck-2026-03-24/>
- [4] A. Edelman, «The Mathematics of the Pentium Division Bug», *SIAM Review*, vol. 39, n.º 1, pp. 54-67, ene. 1997, doi: [10.1137/S0036144595293959](https://doi.org/10.1137/S0036144595293959).
- [5] «Intel Historical Annual Reports». Accedido: 1 de julio de 2026. [En línea]. Disponible en: <https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2025-05/history-1994-annual-report.pdf>
- [6] T. Grimm, D. Lettnin, y M. Hübner, «A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip», *Electronics*, vol. 7, n.º 6, p. 81, jun. 2018, doi: [10.3390/electronics7060081](https://doi.org/10.3390/electronics7060081).
- [7] W. Diffie y M. Hellman, «New directions in cryptography», *IEEE Transactions on Information Theory*, vol. 22, n.º 6, pp. 644-654, nov. 1976, doi: [10.1109/TIT.1976.1055638](https://doi.org/10.1109/TIT.1976.1055638).
- [8] V. S. Miller, «Use of Elliptic Curves in Cryptography», en *Advances in Cryptology*, en CRYPTO '85. Berlin, Heidelberg: Springer-Verlag, ago. 1985, pp. 417-426.
- [9] C. Easttom, «RSA, Diffie–Hellman, Basic Operations on Elliptic Curves», en *Modern cryptography: applied mathematics for encryption and information security*, Cham: Springer, 2021, pp. 227231248.
- [10] D. Boneh y V. Shoup, «2.2 Computational ciphers and semantic security», en *A Graduate Course in Applied Cryptography*, 2023. Accedido: 19 de abril de 2026. [En línea]. Disponible en: <https://toc.cryptobook.us/>
- [11] L. Chen *et al.*, «Report on Post-Quantum Cryptography», NIST IR 8105, abr. 2016. doi: [10.6028/NIST.IR.8105](https://doi.org/10.6028/NIST.IR.8105).
- [12] P. W. Shor, «Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer», *SIAM Journal on Computing*, vol. 26, n.º 5, pp. 1484-1509, oct. 1997, doi: [10.1137/S0097539795293172](https://doi.org/10.1137/S0097539795293172).
- [13] Bas Westerbaan, «Cloudflare targets 2029 for full post-quantum security». Accedido: 12 de abril de 2026. [En línea]. Disponible en: <https://blog.cloudflare.com/post-quantum-roadmap/>

- [14] J. Mascelli y M. Rodden, «“Harvest Now Decrypt Later”: Examining Post-Quantum Cryptography and the Data Privacy Risks for Distributed Ledger Networks», *Finance and Economics Discussion Series*, n.º 2025, sep. 2025, doi: [10.17016/FEDS.2025.093](https://doi.org/10.17016/FEDS.2025.093).
- [15] «What Is Quantum Security? Preparing for the Post-Quantum Era». Accedido: 21 de julio de 2026. [En línea]. Disponible en: <https://www.paloaltonetworks.com/cyberpedia/what-is-quantum-security>
- [16] M. Cain *et al.*, «Shor's algorithm is possible with as few as 10,000 reconfigurable atomic qubits». Accedido: 19 de abril de 2026. [En línea]. Disponible en: <http://arxiv.org/abs/2603.28627>
- [17] «PQCrypto 2016». Accedido: 21 de julio de 2026. [En línea]. Disponible en: <https://pqcrypto2016.jp/>
- [18] D. Moody, «The Future Is Now: Spreading the Word About Post-Quantum Cryptography», *NIST*, dic. 2020, Accedido: 21 de julio de 2026. [En línea]. Disponible en: <https://www.nist.gov/blogs/taking-measure/future-now-spreading-word-about-post-quantum-cryptography>
- [19] I. T. L. Computer Security Division, «Post-Quantum Cryptography | CSRC | CSRC». Accedido: 21 de julio de 2026. [En línea]. Disponible en: <https://csrc.nist.gov/projects/post-quantum-cryptography>
- [20] I. T. L. Computer Security Division, «Post-Quantum Cryptography | CSRC | CSRC». Accedido: 12 de abril de 2026. [En línea]. Disponible en: <https://csrc.nist.gov/projects/post-quantum-cryptography>
- [21] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*. Boston, MA: Springer US, 2003. doi: [10.1007/978-1-4615-0302-6](https://doi.org/10.1007/978-1-4615-0302-6).
- [22] C.-L. Chuang, W.-H. Cheng, D.-J. Lu, y C.-N. J. Liu, «Hybrid Approach to Faster Functional Verification with Full Visibility», *IEEE Design & Test of Computers*, vol. 24, n.º 2, pp. 154-162, mar. 2007, doi: [10.1109/MDT.2007.46](https://doi.org/10.1109/MDT.2007.46).
- [23] «Static Timing Analysis (STA)». Accedido: 26 de mayo de 2026. [En línea]. Disponible en: https://www.cadence.com/en_US/home/explore/static-timing-analysis.html
- [24] «Some Simple Clock-Domain Crossing Solutions». Accedido: 26 de mayo de 2026. [En línea]. Disponible en: <https://zipcpu.com/blog/2017/10/20/cdc.html>
- [25] «Formal Verification for DO-254 (and other Safety-Critical) Designs | Siemens Verification Academy». Accedido: 26 de mayo de 2026. [En línea]. Disponible en: <https://verificationacademy.com/topics/functional-safety/formal-verification-for-do-254-and-other-safety-critical-designs/>
- [26] J. Bhadra, M. S. Abadir, L.-C. Wang, y S. Ray, «A Survey of Hybrid Techniques for Functional Verification», *IEEE Design & Test of Computers*, vol. 24, n.º 2, pp. 112-122, mar. 2007, doi: [10.1109/MDT.2007.30](https://doi.org/10.1109/MDT.2007.30).
- [27] «Functional Verification». Accedido: 7 de junio de 2026. [En línea]. Disponible en: https://www.aldec.com/en/products/functional_verification/riviera-pro/feature/381

- [28] «IEEE Standard for Universal Verification Methodology Language Reference Manual», *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, pp. 1-458, sep. 2020, doi: [10.1109/IEEESTD.2020.9195920](https://doi.org/10.1109/IEEESTD.2020.9195920).
- [29] «02장 UVM for Testbench». Accedido: 14 de junio de 2026. [En línea]. Disponible en: <https://wikidocs.net/170314>
- [30] Siemens, *Universal Verification Methodology (UVM) Cookbook*. Siemens Verification Academy, 2018.
- [31] «Veripool». Accedido: 10 de julio de 2026. [En línea]. Disponible en: <https://www.veripool.org/verilator/>
- [32] «Open Source VHDL Verification Methodology – VHDL Made Better!». Accedido: 20 de junio de 2026. [En línea]. Disponible en: <https://osvvm.org/>
- [33] «OSVVM/OsvvmLibraries». Accedido: 20 de junio de 2026. [En línea]. Disponible en: <https://github.com/OSVVM/OsvvmLibraries>
- [34] Jim Lewis, «SynthWorks Design Inc. - Why Your Team Should be using VHDL + OSVVM for Verification». Accedido: 20 de junio de 2026. [En línea]. Disponible en: <https://indico.esa.int/event/531/contributions/10602/>
- [35] «Tcl Developer Site». Accedido: 20 de junio de 2026. [En línea]. Disponible en: <https://www.tcl-lang.org/>
- [36] «xsim ■ Vivado Design Suite Tcl Command Reference Guide (UG835) ■ AMD Technical Information Portal». Accedido: 10 de julio de 2026. [En línea]. Disponible en: <https://docs.amd.com/r/en-US/ug835-vivado-tcl-commands/xsim>
- [37] «OSVVM-Scripts/README.rst at main · OSVVM/OSVVM-Scripts». Accedido: 21 de junio de 2026. [En línea]. Disponible en: <https://github.com/OSVVM/OSVVM-Scripts/blob/main/README.rst>
- [38] «Introduction — UVVM documentation». Accedido: 20 de junio de 2026. [En línea]. Disponible en: https://uvvm.github.io/uvvm_intro.html
- [39] «Using Multiple OSVVM Verification Components – Open Source VHDL Verification Methodology». Accedido: 23 de junio de 2026. [En línea]. Disponible en: <https://osvvm.org/archives/2092>
- [40] «ghdl/ghdl». Accedido: 10 de julio de 2026. [En línea]. Disponible en: <https://github.com/ghdl/ghdl>
- [41] N. Gasson, «nickg/nvc». Accedido: 10 de julio de 2026. [En línea]. Disponible en: <https://github.com/nickg/nvc>
- [42] «Questa One Sim». Accedido: 10 de julio de 2026. [En línea]. Disponible en: <https://www.siemens.com/en-us/products/ic/questa-one/simulation/questa-one-sim/>
- [43] «Functional Verification». Accedido: 10 de julio de 2026. [En línea]. Disponible en: https://www.aldec.com/en/products/functional_verification/riviera-pro

- [44] «VUnit documentation». Accedido: 24 de junio de 2026. [En línea]. Disponible en: <https://vunit.github.io/>
- [45] «Simulator Support — cocotb 2.0.1 documentation». Accedido: 11 de junio de 2026. [En línea]. Disponible en: https://docs.cocotb.org/en/stable/simulator_support.html
- [46] «cocotb». Accedido: 11 de junio de 2026. [En línea]. Disponible en: <https://www.cocotb.org/>
- [47] «Welcome to cocotb's Documentation! — cocotb 2.0.1 documentation». Accedido: 11 de junio de 2026. [En línea]. Disponible en: <https://docs.cocotb.org/en/stable/index.html#how-does-cocotb-work>
- [48] «Introduction — pyuvvm v4.0.1 documentation». Accedido: 16 de junio de 2026. [En línea]. Disponible en: <https://pyuvvm.readthedocs.io/en/latest/docsources/README.html>
- [49] «IEEE 1800.2 and pyuvvm». Accedido: 16 de junio de 2026. [En línea]. Disponible en: <https://github.com/pyuvvm/pyuvvm#ieee-18002-and-pyuvvm>
- [50] «alexforencich - Overview». Accedido: 16 de junio de 2026. [En línea]. Disponible en: <https://github.com/alexforencich>
- [51] «YosysHQ/oss-cad-suite-build». Accedido: 26 de junio de 2026. [En línea]. Disponible en: <https://github.com/YosysHQ/oss-cad-suite-build>
- [52] «Tabby CAD Datasheet». Accedido: 26 de junio de 2026. [En línea]. Disponible en: <https://www.yosyshq.com/tabby-cad-datasheet>
- [53] «YosysHQ Tool Documentation - YosysHQ Docs documentation». Accedido: 26 de junio de 2026. [En línea]. Disponible en: <https://yosyshq.readthedocs.io/en/latest/tools.html>
- [54] E. B. Barker, «Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms», NIST SP 800–175B, ago. 2016. doi: [10.6028/NIST.SP.800-175B](https://doi.org/10.6028/NIST.SP.800-175B).
- [55] Uniqkey, «Zero-Knowledge Encryption vs Other Methods of Encryption». Accedido: 6 de junio de 2026. [En línea]. Disponible en: <https://blog.uniqkey.eu/zero-knowledge-encryption-vs-others/>
- [56] W. Easttom, *Modern Cryptography: Applied Mathematics for Encryption and Information Security*. Cham: Springer International Publishing, 2021. doi: [10.1007/978-3-030-63115-4](https://doi.org/10.1007/978-3-030-63115-4).
- [57] N. P, «Hashing & SHA in Information Security». Accedido: 6 de junio de 2026. [En línea]. Disponible en: <https://medium.com/@pnandhiniofficial/hashing-sha-in-information-security-cbabe57762ef>
- [58] «File:Private key signing.svg - Wikipedia». Accedido: 6 de junio de 2026. [En línea]. Disponible en: https://commons.wikimedia.org/wiki/File:Private_key_signing.svg
- [59] D. Boneh y V. Shoup, *A Graduate Course in Applied Cryptography*. 2023. Accedido: 19 de abril de 2026. [En línea]. Disponible en: <https://toc.cryptobook.us/>
- [60] P. Benioff, «The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines», *Journal of Statistical Physics*, vol. 22, pp. 563-591, may 1980, doi: [10.1007/BF01011339](https://doi.org/10.1007/BF01011339).

- [61] L. K. Grover, «A fast quantum mechanical algorithm for database search», en *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*, Philadelphia, Pennsylvania, United States: ACM Press, 1996, pp. 212-219. doi: [10.1145/237814.237866](https://doi.org/10.1145/237814.237866).
- [62] «What Is Post-Quantum Cryptography (PQC)? A Complete Guide». Accedido: 7 de junio de 2026. [En línea]. Disponible en: <https://www.paloaltonetworks.com/cyberpedia/what-is-post-quantum-cryptography-pqc>
- [63] «Prepping for post-quantum: a beginner's guide to lattice cryptography». Accedido: 7 de junio de 2026. [En línea]. Disponible en: <https://blog.cloudflare.com/lattice-crypto-primer/>
- [64] R. J. McEliece, «A Public-Key Cryptosystem Based On Algebraic Coding Theory», *Deep Space Network Progress Report*, vol. 44, pp. 114-116, ene. 1978, Accedido: 7 de junio de 2026. [En línea]. Disponible en: https://tda.jpl.nasa.gov/progress_report/42-44/44N.PDF
- [65] National Security Agency y Cybersecurity and Infrastructure Security Agency, «Commercial national security algorithm suite 2.0», may 2025. [En línea]. Disponible en: https://media.defense.gov/2025/May/30/2003728741/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS.PDF
- [66] G. Alagic *et al.*, «Recommendations for Key-Encapsulation Mechanisms», NIST Special Publication (SP) 800–227, sep. 2025. doi: [10.6028/NIST.SP.800-227](https://doi.org/10.6028/NIST.SP.800-227).
- [67] R. Avanzi *et al.*, «Algorithm Specifications And Supporting Documentation».
- [68] «V1b: Mathematical prerequisites (Kyber and Dilithium short course)». Accedido: 29 de junio de 2026. [En línea]. Disponible en: <https://www.youtube.com/watch?v=h5pfTIE6slU>
- [69] I. Corporation, «Lattice-Based Cryptography». Accedido: 30 de junio de 2026. [En línea]. Disponible en: <https://www.isara.com/blog-posts/lattice-based-cryptography.html>
- [70] T.-H. Nguyen *et al.*, «An Area-Time Efficient Hardware Architecture for ML-KEM Post-Quantum Cryptography Standard», *IEEE Access*, vol. 13, pp. 103834-103847, 2025, doi: [10.1109/ACCESS.2025.3579379](https://doi.org/10.1109/ACCESS.2025.3579379).
- [71] T.-H. Nguyen, D.-T. Dam, P.-P. Duong, B. Kieu-Do-Nguyen, C.-K. Pham, y T.-T. Hoang, «Efficient Hardware Implementation of the Lightweight CRYSTALS-Kyber», *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 72, n.º 2, pp. 610-622, feb. 2025, doi: [10.1109/TCSI.2024.3443238](https://doi.org/10.1109/TCSI.2024.3443238).
- [72] F. Yaman, A. C. Mert, E. Öztürk, y E. Savaş, «A Hardware Accelerator for Polynomial Multiplication Operation of CRYSTALS-KYBER PQC Scheme». Accedido: 8 de julio de 2026. [En línea]. Disponible en: <https://eprint.iacr.org/2021/485>
- [73] V. Lyubashevsky y G. Seiler, «NTTRU: Truly Fast NTRU Using NTT». Accedido: 8 de julio de 2026. [En línea]. Disponible en: <https://eprint.iacr.org/2019/040>
- [74] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, y L. Liu, «Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT», *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 49-72, mar. 2020, doi: [10.13154/tches.v2020.i2.49-72](https://doi.org/10.13154/tches.v2020.i2.49-72).
- [75] «Git». Accedido: 9 de julio de 2026. [En línea]. Disponible en: <https://git-scm.com/>

- [76] «Quickstart Guide — cocotb 2.0.1 documentation». Accedido: 15 de julio de 2026. [En línea]. Disponible en: <https://docs.cocotb.org/en/stable/quickstart.html#using-runners>
- [77] «AMBA AXI-Stream Protocol Specification». Accedido: 16 de julio de 2026. [En línea]. Disponible en: <https://developer.arm.com/documentation/ih0051/b/>

Objetivos de Desarrollo Sostenible (ODS)

Tabla 8: Relación del proyecto con los Objetivos de Desarrollo Sostenible (ODS)

Objetivos de Desarrollo Sostenible	Grado de relación con el ODS			
	0 No procede	1 Bajo	2 Medio	3 Alto
ODS 1 Fin de la Pobreza	☒	☐	☐	☐
ODS 2 Hambre cero	☒	☐	☐	☐
ODS 3 Salud y Bienestar	☒	☐	☐	☐
ODS 4 Educación de calidad	☐	☒	☐	☐
ODS 5 Igualdad de género	☒	☐	☐	☐
ODS 6 Agua limpia y saneamiento	☒	☐	☐	☐
ODS 7 Energía Asequible y no contaminante	☒	☐	☐	☐
ODS 8 Trabajo decente y crecimiento económico	☐	☐	☒	☐
ODS 9 Industria, Innovación e Infraestructuras	☐	☐	☐	☒
ODS 10 Reducción de las desigualdades	☒	☐	☐	☐
ODS 11 Ciudades y comunidades sostenibles	☒	☐	☐	☐
ODS 12 Producción y consumo sostenibles	☒	☐	☐	☐
ODS 13 Acción por el clima	☒	☐	☐	☐
ODS 14 Vida submarina	☒	☐	☐	☐
ODS 15 Vida de ecosistemas terrestres	☒	☐	☐	☐
ODS 16 Paz, justicia e instituciones sólidas	☒	☐	☐	☐
ODS 17 Alianzas para lograr objetivos	☒	☐	☐	☐

La realización de este TFM está alineado con los Objetivos de Desarrollo Sostenible indicados en la [Tabla 8](#) a través de los objetivos mencionados a continuación y justificado mediante las siguientes metas concretas indicadas:

- Objetivo 4, “Educación de calidad” (ODS 4): metas 4.3 y 4.4
- Objetivo 8, “Trabajo decente y crecimiento económico” (ODS 8): meta 8.2
- Objetivo 9, “Industria, innovación e infraestructuras” (ODS 9): metas 9.1, 9.5 y 9.b